Modular Verification of Security Protocol Implementations

Linard Arquint

DISS. ETH NO. 31494

DISS. ETH NO. 31494

Modular Verification of Security Protocol Implementations

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES

(Dr. sc. ETH Zurich)

presented by

LINARD ARQUINT

born on September 8, 1993

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner
Prof. Dr. David A. Basin, co-examiner
Prof. Dr. Frank Piessens, co-examiner
Dr. Jonathan Protzenko, co-examiner

2025

Abstract

Security protocols such as TLS or Signal ensure security and privacy for browsing the web, sending private messages, and using cloud services. It is, thus, crucial that these ubiquitous and critical protocols are designed *and* implemented correctly. Protocol model verifiers such as Tamarin and ProVerif make it viable to formally verify protocol *models*. However, proving protocol *models* secure is insufficient to guarantee secure *implementations*. Coding errors such as missing bounds checks (e.g., causing the Heartbleed bug), omitted protocol steps (as in the Matrix SDK), or ignored errors (e.g., returned by a TLS library) may invalidate all security properties proven for the corresponding models.

This dissertation is centered around proving security properties in the symbolic model of cryptography for protocol *implementations*. This faces three key challenges. First, security properties like secrecy and authentication are *global properties*, which depend on the collective behavior of all protocol participants and the attacker. Accounting for this entire behavior in a proof presents a significant obstacle as implementation-level proof techniques rely heavily on local reasoning, which proves each method in isolation. Second, practically deployed protocol implementations are written in languages such as C, Go, Java or Rust and utilize *complex programming language features* like side effects, mutable state, and concurrency to achieve high performance. Despite complicating the reasoning about implementations and, thus, their security, it is crucial to support these features and programming languages to verify real-world protocol implementations. Third, protocol implementations are often embedded in *large software systems* to provide secure communication as a building block to application logic. Requiring a laborious proof for the entire software system is prohibitively expensive and, thus, impractical. This dissertation addresses all three key challenges.

To address the first and second key challenge, this dissertation develops two novel verification methodologies. If an abstract model of a security protocol preexists, the first methodology exploits this abstract model to provide the required global view for proving security properties. In a second step, this methodology extracts proof obligations from the abstract model for implementations. Successfully discharging these proof obligations guarantees that an implementation refines an abstract model and, thus, inherits the security properties proven for the abstract model. The second methodology does not require a preexisting, accurate abstract model; instead, it uses invariants to establish a global view on the behavior of protocol participants and the attacker. By verifying each implementation against these invariants, we soundly consider the collective behavior in an implementation's proof. Our evaluations on different security protocol implementations in C, Go, and Java demonstrate that both methodologies are applicable to a wide range of security protocols and programming languages.

To address the third key challenge, this dissertation introduces Diodon, a novel and provably sound methodology to symbiotically combine proof systems of different expressive power, significantly reducing the proof effort and, thus, scaling security property verification to large, production codebases. We partition a codebase into its, typically small, security-critical part and the rest of the codebase for tailoring the employed proof system to each partition. Mandated by the security properties we want to prove, we apply a highly expressive but laborious program verifier to the security-critical part as we have to reason, e.g., about the content of messages and their cryptographic protection. We accomplish this task by using the aforementioned verification methodologies addressing the first and second key challenges and, thus, we directly benefit from their advances. Although the remaining partition is less security-critical, we cannot simply ignore it. A priori, there are ample opportunities for vulnerabilities in this partition because it is generally impossible to isolate the partitions for two main reasons. First, widely adopted programming languages like C, Go, and Java do not provide sufficiently strong isolation guarantees because such guarantees conflict with their features enabling high performance implementations like mutable state, aliasing, and concurrency. Second, refactoring a codebase to ensure—without relying on the programming language—that only the security-critical partition has access to sensitive data such as cryptographic keys can be prohibitively expensive and unacceptable from a performance perspective, as is the case for the production codebase on which we evaluate Diodon. Therefore, we apply lightweight, fully-automatic static analyses to ensure that the partitions soundly compose, i.e., without violating proven security properties. Our evaluation demonstrates that Diodon supports different coding styles and allows us to prove security properties for a production codebase of more than 100k LOC.

In summary, this dissertation achieves sound and modular verification of security protocols implemented in real-world codebases, regardless of programming language, coding style or program verifier.

Resumaziun

Protocols da segirtad sco TLS ni Signal garanteschan segirtad e sfera privata per navigar sin il web, tarmetter messadis e duvrar survetschs da cloud. Igl ei perquei elementar che quels protocols tutpresents e critics vegnien concepi ed implementai correctamein. Ils verificaders da models da protocol sco Tamarin e ProVerif possibliteschan da verificar formalmein *models* da protocol. Igl ei denton buc avunda da cumprovar *models* segirs per garantir *implementaziuns* segiras. Sbagls da programms sco la munconza da controllas da cunfins (tgei che ha p. ex. caschunau il bug Heartbleed), pass da protocol emblidai (sco el SDK da Matrix) ni sbagls ignorai (che vegnan returnai p. ex. d'ina biblioteca da TLS) san invalidar tut las qualitads da segirtad ch'eran vegnidas cumprovidas per ils models corrispundents.

Quella dissertaziun seconcentrescha sin la cumprova da qualitads da segirtad el model simbolic da criptografia per *implementaziuns* da protocols. Ella explica treis sfidas principalas. Sco emprem, qualitads da segirtad sco manteniment dil secret ed autentificaziun ein *qualitads globalas* che dependan dil cumportament collectiv da tut ils participonts d'in protocol e digl attaccader. Risguardar il secuntener entir el process da cumprova presenta in obstachel considerabel, oramai che las tecnicas da cumprova sin nivel d'implementaziun dependan fermamein dad arguments locals, tier las quallas mintga metoda vegn cumprovida isoladamein. Sco secund, implementaziuns da protocols duvradas en la pratica vegnan screttas en lungatgs sco C, Go, Java ni Rust, e drovan *caracteristicas cumplicadas da lungatgs da programmaziun* sco effects laterals, stadis midabels e parallelitads per contonscher aulta performanza. Malgrad che quei engreviescha ils arguments davart ina implementaziun e consequentamein sia segirtad, eis ei essenzial da sustener quellas caracteristicas e quels lungatgs per verificar implementaziuns realas. Sco tierz, implementaziuns da protocols vegnan savens integradas en *gronds sistems da software* per porscher communicaziun segira sco in element da construcziun per logica d'applicaziun. Pretender ina cumprova custeivla per igl entir sistem ei insupportablamein car e perquei impraticabel. Questa dissertaziun adressescha tut las treis sfidas principalas.

Per adressar l'emprema e la secunda sfida principala, sviluppa quella dissertaziun duas metodicas novas da verificaziun. Sch'ei exista gia in model abstract d'in protocol da segirtad, lu seprofitescha l'emprema metodica da quei model per porscher ina vesta globala necessaria per cumprovar qualitads da segirtad. En in proxim pass, extrai quella metodica obligaziuns da cumprova per implementaziuns dil model abstract. Il reussi dallas obligaziuns da cumprova garantescha che in'implementaziun refineschi il model abstract e perquei possedi las qualitads da segirtad cumprovadas per quei model. La secunda metodica sebasa buca sin in model abstract accurat, mobein sin invariantas che stabilischeschan ina vesta globala sin il cumportament dils participonts d'in protocol e digl attaccader. Verifitgond mintga implementaziun encunter quellas invariantas, risguardein nus correctamein il cumportament collectiv en la cumprova d'ina implementaziun. Nus evaluein las duas metodicas sin implementaziuns differentas da protocols da segirtad en C, Go e Java e mussein che quellas ein applicablas ad in vast spectrum da protocols da segirtad e lungatgs da programmaziun.

Per adressar la tiarza sfida principala, presenta quella dissertaziun Diodon, ina metodica nova e cumprovadamain correcta per cumbinar simbioticamein sistems da cumprova cun expressivitads differentas, tgei che reducescha significativamein igl impundiment da cumprova ed engrondescha perquei la cumprova da qualitads da segirtad a gronds codebases da producziun. Nus partiziunein ina codebase en sia part critica per segirtad – savens pintga – e la part restonta dalla codebase, per adaptar il sistem da cumprova tenor mintga partiziun. Tenend en consideraziun las qualitads da segirtad che nus vulein cumprovar, duvrein nus in verificader da programs fetg expressiv, dentont pretensiv, per la part critica per segirtad, perquei ch'ei basegna argumentaziun detagliada p. ex. davart cuntegn da messadis e lur protecziun criptografica. Nus contonschin quell'incumbensa cun las metodicas da verificaziun menziunadas avon che adressar l'emprima e la secunda sfida principala e profitein perquei directamein da lur svilup. Malgrad che la partiziun restonta ei meins critica per segirtad, sa ella buc semplamein vegnir ignorada. A priori dat ei numerusas pusseivladads da vulnerabilitads en quella part, perquei ch'igl ei en general nunpusseivel per duas raschuns primaras d'isolar partiziuns. Sco emprem, lungatgs da programmaziun sco C, Go e Java porschan buc garanzias d'isolaziun suffizientas, perquei che quellas garanzias stattan en conflict cun lur caracteristicas da performanza sco p. ex. stadi midabels, aliasing e parallelitad. Sco secundo, refar ina codebase per garantir – senza seschar sil lungatg da programmaziun – che mo la part critica hagi access a datas sensitivas sco clavs criptograficas po esser insupportablamein car ed inacceptabel ord perspectiva da performanza, sco igl ei il cass per la codebase da producziun nua che nus evaluain Diodon. Perquei druvein nus analisas staticas levas e cumplettamein automaticas per garantir che las partiziuns secumbineschien correctamein, qvd. senza donnegiar las qualitads da segirtad cumprovadas. Nossa evaluaziun muossa che Diodon sustegn diversas stilisticas da programmaziun e possiblitescha da cumprovar qualitads da segirtad per ina codebase da producziun cun dapli che 100k LOC.

En resumaziun, quella dissertaziun contonscha ina verificaziun correcta e modulara da protocols da segirtad implementai en codebases realas, independentamein dil lungatg da programmaziun, stilistica da programmaziun ni verificader da programs.

Zusammenfassung

Sicherheitsprotokolle wie TLS oder Signal gewährleisten Sicherheit und Privatsphäre beim Surfen im Web, Versenden von privaten Nachrichten und Nutzen von Cloud-Diensten. Daher ist es essenziell, dass diese allgegenwärtigen und kritischen Protokolle korrekt entworfen *und* implementiert werden. Protokollmodellverifizierer wie etwa Tamarin und ProVerif ermöglichen die formale Verifikation von Protokoll*modellen*. Sicherheitsbeweise für Protokoll*modelle* sind jedoch unzureichend, um die Sicherheit von *Implementierungen* zu garantieren. Programmfehler wie fehlende Grenzüberprüfungen (die z. B. den Heartbleed-Bug verursacht haben), vergessene Protokollschritte (wie im Matrix-SDK) oder ignorierte Fehler (welche z. B. von einer TLS-Library zurückgegeben werden) können alle Sicherheitseigenschaften, welche für das entsprechende Modell bewiesenen wurden, invalidieren.

Diese Dissertation konzentriert sich auf das Beweisen von Sicherheitseigenschaften im symbolischen Modell der Kryptographie für Protokoll*implementierungen*. Damit sind drei Hauptherausforderungen verbunden. Erstens sind Sicherheitseigenschaften wie Geheimhaltung und Authentifizierung *globale Eigenschaften*, welche vom kollektiven Verhalten aller Protokollteilnehmer und des Angreifers abhängen. Die Berücksichtigung dieses gesamten Verhaltens in einem Beweis stellt ein erhebliches Hindernis dar, da sich Beweistechniken auf Implementierungsebene stark auf lokale Argumentationen stützen, bei denen jede Methode isoliert bewiesen wird. Zweitens sind in der Praxis eingesetzte Protokollimplementierungen in Sprachen wie C, Go, Java oder Rust verfasst und nutzen *komplexe Programmiersprachenbesonderheiten* wie etwa Seiteneffekte, veränderbare Zustände und Nebenläufigkeiten, um einen hohen Datendurchsatz und gute Laufzeit zu erzielen. Obwohl diese Besonderheiten die Argumentation über Implementierungen und damit deren Sicherheit erschweren, ist es essenziell, diese Besonderheiten und Programmiersprachen zu unterstützen, um reale Protokollimplementierungen zu verifizieren. Drittens sind Protokollimplementierungen oft in *grosse Softwaresysteme* eingebettet, um sichere Kommunikation als Baustein für Anwendungslogik bereitzustellen. Einen aufwändigen Beweis für das gesamte Softwaresystem zu erfordern, ist unerschwinglich teuer und deshalb praxisfern. Diese Dissertation adressiert alle drei Hauptherausforderungen.

Um die ersten beiden Hauptherausforderungen zu adressieren, entwickelt diese Dissertation zwei neuartige Verifikationsmethodiken. Falls ein abstraktes Modell eines Sicherheitsprotokolls bereits existiert, nutzt die erste Methodik dieses abstrakte Modell, um die erforderliche globale Sicht zum Beweisen von Sicherheitseigenschaften bereitzustellen. In einem zweiten Schritt extrahiert diese Methodik Beweisverpflichtungen für Implementierungen aus dem abstrakten Modell. Das erfolgreiche Erfüllen dieser Beweisverpflichtungen garantiert, dass eine Implementierung ein abstraktes Modell verfeinert und somit die für das abstrakte Modell bewiesenen Sicherheitseigenschaften erbt. Die zweite Methodik erfordert kein bestehendes, genaues abstraktes Modell; stattdessen verwendet sie Invarianten, um eine globale Sicht auf das Verhalten der Protokollteilnehmer und des Angreifers zu etablieren. Indem jede Implementierung gegen diese Invarianten verifiziert wird, berücksichtigen wir das kollektive Verhalten korrekt im Beweis einer Implementierung. Unsere Evaluationen auf verschiedenen Sicherheitsprotokollimplementierungen in C, Go und Java zeigen, dass beide Methodiken auf ein breites Spektrum von Sicherheitsprotokollen und Programmiersprachen anwendbar sind.

Um die dritte Hauptherausforderung zu adressieren, führt diese Dissertation Diodon ein, eine neuartige und beweisbar korrekte Methodik, um Beweissysteme unterschiedlicher Ausdrucksstärke symbiotisch zu kombinieren, was den Beweisaufwand erheblich reduziert und somit die Verifikation von Sicherheitseigenschaften auf grosse Produktionscodebasen hochskaliert. Wir partitionieren eine Codebasis in ihren, typischerweise kleinen, sicherheitskritischen Teil und den Rest der Codebasis, um das eingesetzte Beweissystem auf jede Partition zuzuschneiden. Aufgrund der zu beweisenden Sicherheitseigenschaften wenden wir einen ausdrucksstarken, aber aufwändigen Programmverifizierer auf den sicherheitskritischen Teil an, da wir über den Inhalt von Nachrichten und deren kryptografischen Schutz argumentieren müssen. Wir lösen diese Aufgabe, indem wir eine der zuvor genannten Verifikationsmethodiken anwenden, welche die ersten beiden Hauptherausforderung adressieren, und profitieren daher direkt von deren Fortschritten. Obwohl die verbleibende Partition weniger sicherheitskritisch ist, können wir sie nicht einfach ignorieren. A priori gibt es unzählige Möglichkeiten für Schwachstellen in dieser Partition, da es im Allgemeinen unmöglich ist, die Partitionen aus zwei primären Gründen zu isolieren. Erstens bieten weit verbreitete Programmiersprachen wie C, Go und Java keine hinreichend starken Isolationsgarantien, da solche Garantien mit ihren Besonderheiten – wie beispielsweise veränderbare Zustände, Aliasing und Nebenläufigkeiten – in Konflikt stehen, die leistungsfähige

Implementierungen ermöglichen. Zweitens kann die Restrukturierung einer Codebasis, um sicherzustellen, dass nur die sicherheitskritische Partition auf sensible Daten wie kryptografische Schlüssel zugreifen kann – ohne sich dabei auf die Programmiersprache zu verlassen – unerschwinglich aufwändig sein und inakzeptable Leistungseinbussen mit sich bringen, wie dies bei der Produktionscodebasis der Fall ist, auf der wir DIODON evaluieren. Daher wenden wir leichtgewichtige und vollautomatische statische Analysen an, um sicherzustellen, dass die Partitionen korrekt zusammenspielen, d. h. ohne bewiesene Sicherheitseigenschaften zu verletzen. Unsere Evaluation zeigt, dass DIODON verschiedene Programmierstile unterstützt und es uns ermöglicht, Sicherheitseigenschaften für eine Produktionscodebasis mit mehr als 100k LOC zu beweisen.

Zusammenfassend erzielt diese Dissertation eine korrekte und modulare Verifikation von Sicherheitsprotokollen, die in realen Codebasen implementiert sind, unabhängig von Programmiersprache, Programmierstil oder Programmverifizierer.

Acknowledgments

Pursuing a doctorate over the past six years has been an incredible journey filled with challenges, rewarding experiences, and countless fun moments. Since this journey would have been completely different without all the support, interactions, and collaborations I have had along the way, I would like to take this opportunity to express my sincere gratitude to everyone who has contributed to making this experience so memorable.

First and foremost, I would like to express my deepest gratitude to my advisor, Peter Müller. Thank you, Peter, for sparking the idea of pursuing a doctorate during your engaging and funny COOP lectures, for believing in me from the very first meeting onwards, for giving me the opportunity to pursue a doctorate under your supervision, and for your continuous support and all the insightful and productive technical meetings throughout these years. Experiencing your dedication to research and willingness to understand every detail no matter how small, your passion for teaching, and your care for your students in always making time for a meeting has been truly inspiring. I particularly appreciate the freedom you have given me to explore my own research ideas and to venture into security-related topics, all of which have significantly contributed to making this journey in general, and our collaboration in particular, both delightful and fulfilling.

I am thankful to my second advisor and co-examiner, David Basin, for his critical questions, insightful input, and security-focused perspective on my research over the years; and to Frank Piessens and Jonathan Protzenko for agreeing to join David in serving as my co-examiners, reviewing this dissertation, asking thoughtful questions during the defense, and providing valuable feedback.

Over the course of my doctorate, I had the privilege of working closely with outstanding researchers and learning from their expertise, which has significantly helped me grow as a researcher.

What began already with my Master's thesis and outlasts to this day is my collaboration with Malte Schwerhoff. Malte, thank you for being a role model throughout all these years, initially as my Master's thesis advisor and later as my closest collaborator. I appreciate all the time that you committed to our collaboration, every single technical discussion we had, and all your input, which helped me in fostering my understanding, making the arguments more convincing, and unblocking me whenever I was stuck. I also enjoyed working closely with you as head teaching assistant for several years, and I am grateful that you trusted me to be up to the challenge of managing such a large course.

Throughout my doctorate, I had the pleasure of collaborating with bright minds on Gobra. Working on the same tool together, rushing to support some concurrency features in time for the CAV deadline, all the technical conversations about new features and how to implement them best, and the numerous code reviews were extremely insightful and enjoyable. I am grateful to every contributor, particularly to Felix and João, for all their work and always finding time for discussions no matter how busy their schedules were.

Being part of the collaboration between Peter's and David's groups provided me with a great introduction to applying formal methods to security protocols and allowed me to learn a lot about Tamarin. Thank you, David, Joseph, Peter, Ralf, Christoph, and Felix, for the interesting discussions, for all the expertise you shared with me over the years, and for helping me develop my research skills early in my doctorate by showing me how established researchers work.

In 2023 and 2024, I had the privilege to intern at Amazon Web Services (AWS) in Portland, OR and apply my research to their problems. I am thankful for experiencing how research is conducted in industry and learning a lot about static analyses. Thank you to my collaborators Joey, Samarth, Jason, and Daniel for sharing your knowledge during countless technical discussions and for helping shape Diodon. Beyond the scientific contributions, I am thankful for Joey's personal support and understanding when needed and for all the fellow interns who made both internships unforgettable. Without these internships, I would not have met Margarida in particular, whose dedication to research was truly inspiring and who made both stays such great experiences. Thank you, Margarida, for all the wonderful moments outside of work, from exploring Portland's food and beer scenes to discovering the natural beauty of the Pacific Northwest, for offering a lovely place to stay in Boston when I needed it, and for your kindness, empathy, and friendship.

My deepest gratitude goes to current and former members of the PM group as this enabled me to base my work on Viper and benefit from all its improvements. Moreover, the friendly and supportive atmosphere in the group made it a pleasure to come to the office every day and delay graduation as long as possible. Vytautas, your dedication to teaching deeply impressed me, and I thank you for providing exceptionally well-maintained, high-quality teaching materials for the computer science course, which made my life much easier when I became head teaching assistant. Aurea, it is impressive how you manage to monitor repositories you are not even contributing to and create pull requests for them – thank you!; Thank you, Lea, for taking initiatives to organize social events and even pre-empt me in offering tea at my place after a cold plunge in the Limmat; Alexandra, for keeping up the Christmas spirit; Thibault, for bringing so much joy with your positive attitude no matter the activity, from swimming in the Rhein and attending events that *others* organized to discussions about formalizations; Xavier, for leaving your lease in the best possible hands and supplying the office with French deliciousness; Jérôme, for teaching drivers about pedestrians' right of way and your Swiss charm; Marco, for all your hard work on Viper and being the best oracle I could have hoped for when discussing implementation ideas; Jonáš, for enabling numerous awesome activities outside of work, from visiting Olympic National Park over feeling guilty at Rheinfall (in a fun way!) to showing your home country; Andrea, for bringing more Swissness to the group; Nick, for making our office the best one, being there to listen to my rants about broken implementations, organizing a beer tasting, taking over the worst codebase, and all the fun conversations we had; Andrew, for being such an independent student; Anqi, for baking by far the most stunning cakes; Hongyi, for being my applied cryptography TA; Christoph, for the time we shared an office and for your amusing misuse of the gender of "tram"; Thomas, for the impressive number of initiatives you have taken since starting your doctorate; João, for all the great memories we created together all over the world, from situations in Kandersteg over OPLSS in Boston to exploring Seattle; Wytse, for implementing mathematical data structures in Gobra; Fábio, for the impressive number of fixed issues; Gaurav, for your Swiss politeness, high work and expense ethics, and quality standards, which I find deeply impressive; Federico, for migrating Mercurial repositories with me early on in my doctorate; Michael, for teaching us what proper mulled wine is; Sandra, for being a great help throughout my doctorate and for perfectly organizing our retreats; Dionisios, for being our hope for better performance and for your recommendations for Greek food; Alex, for discussing research ideas over a cup of coffee and for showing me UBC; Arshavir, for all the time you took to explain the ViperServer codebase to me; Felix, for gently introducing me to the Gobra codebase and for our many technical discussions about Gobra, bytes, and security properties; and Yushuo, for your inspiring work ethic and great attention to detail.

I additionally would like to thank the entire PLF group, Isaac, Alessio, Johannes, Ralf, and Max, for not only making the J floor meeting room even more cramped but also for all the interesting conversations over lunch and coffee, and for the fun activities outside of work (thanks for pushing for more cold plunges!). I would also like to thank Matthias for the entertaining and sometimes utopian discussions over dinner.

A very rewarding part of my doctorate has been supervising students, and I am grateful to so many talented students I had the pleasure of working with over the years: Fabio Aliberti, Nico Berling, Dennis Buitendijk, Johannes Gasser, Andrew Lee, Kwok Wai Lui, Eva Charlotte Mayer, Lasse Meinen, Hugo Queinnec, Silas Walker, Dina Weiersmüller, and Cheng Xuan.

I hold a special place in my heart for all the amazing people who served with me on the board of the association of scientific staff of the Department of Computer Science at ETH Zurich (VMI). I truly enjoyed my time as a board member mostly because of you all, which made every meeting, every event, every welcome bag, and every response to a change of regulations so much more fun. I overlapped with three board members the longest, and I am grateful that our commitment to the department turned into lasting (and culinary) friendships. Matilda, your drive for the mental health initiative impressed me from day one, and I admire your dedication and attention to detail also for all other things you do. Thank you for being such a caring and supportive friend and for all created memories throughout these years. Felix, your talent for bringing people together is outstanding, and I am grateful for getting to know you early on, our friendship, and, thanks to you, meeting many great minds in the information security institute. Anu, thank you for brightening up a long day in the office with your joyful attitude and always having an idea for a fun activity (is pancake day really as big as you made it sound?). Furthermore, I want to thank Graciana, for your remarkable kindness; Nabil, for your insights as a postdoc; Alexandre, for all your banter in Zermatt; Yutong, for organizing a running dinner; Giovanni, for spreading so much joy with your Italian charm; Francesca, for your constant stream of great event ideas; Tobias, for convincing the department to value VMI more; Karel, for having us at your wedding; Florian, for conducting the best outdoor events; Sverrir, for taking care of the finances so well; Alexander, for all your knowledge that you collected and shared over the years; and Duong, for your empathy and delicious dinner.

Throughout my doctorate, I always enjoyed going to Super Kondi. While the "Hey-Ho"s are not too bad, Super Kondi sessions in Portland confirmed that it is all about the people you do Super Kondi with. In that sense, a big thank you to everyone who joined me for Super Kondi sessions and dinners afterwards over all these years – you contributed a lot to making the doctorate fun. From the people I have not mentioned yet, I would especially like to thank (ordered by the number of sessions I believe we attended together) Friederike, for always being up for a session no matter how early in the morning; Srđan, for taking the front row and sometimes even outperforming the instructors; and Simon, for being the Duracell Bunny and the super-charming Swiss guy, who is always full of excitement for all kinds of things, as I learned over the years.

My doctorate would not have been the same without the amazing colleagues and the welcoming and supportive atmosphere in the department. A big thank you to everyone in the department, to Denise and Bernadette from the studies administration for always having an open ear and answering every single of my questions; to the entire information security institute for the countless moments we shared at work (mostly while eating cake), outside of work, and at conferences around the world (riding rollercoasters at subzero temperatures in Copenhagen, trying hard to taste the 30 different flavors in a single cup of coffee in Salt Lake City, and sampling beers in Sofia, to name just a few); to my fellow ridge hikers, Matilda, Sofia, Friederike, Martin, Srđan, and Matteo, for being brave enough to join me on ridge hikes despite knowing that weather is doomed to turn bad (does the hike even count if we did not experience at least a thunderstorm, downpour, or hail?); to Daniele, for our great conversations about research, life, and everything in between; and to Matteo, for being exceptionally kind, always spreading joy, and for introducing me to the Calabrian cuisine.

My time in Zurich would not have been the same without the amazing people I met and had the pleasure of spending time with during my entire studies. Thank you, Jana, Fadri, Oliver, and Valentin for being great flatmates and making the start at ETH Zurich much easier; my meh als WG flatmates and Hageholz crew, for enjoying countless evenings of fun, cooking together, and getting through Covid together; the entire Burgeros crew, Julia, David, Franziska, Jonas, Aurel, Jonathan, Xenia, Pascal J. & L., Felix, Vincent, Katja, Andrina, Robert, Tim, and many more, for creating such a welcoming atmosphere, for always having far too little brie, far too large burgers, and "Foto-Pommes", and for all the fun evenings we shared over the years; my ınfıx cofounders, for being able to practically apply so many concepts that we learned during our studies all while having a great time together (sharks!, Gaucho in London, the mistake in Stresa, and many more amazing memories) and earning some money on the side; and my former partners, Kristina and Cheyenne, for your unconditional support, for always believing in me, and for being there during ups and downs of this journey. I am thankful for the time we have spent together and all the memories we created.

My journey started long before my studies at ETH Zurich. I am grateful for the amazing six years I spent in Disentis, all the memories created, and the friendships formed there. In particular, I want to thank Pater Pirmin, Paul Kocian, and Jörg Schmuki for showing me how fascinating STEM subjects are; Giusep Simonet, for sparking my interest in programming; Samuel and Fadri, for struggling through the spring equations in PAM together and for all the great moments we have shared since; Belinda, for all the train rides we shared to and from Disentis, for being a great role model then and now, for every delicious dinner we had together (which became even more special after you introduced Philip), and for being such a kind and supportive friend throughout the years; Rita, for being a great skiing buddy and the best reviser of my Romansh abstract that I could have wished for; Alea, for the unforgettable memories we made together and your support in so many ways over all these years; and Anita and Roman, for a friendship dating back to elementary school, and for introducing me to great music and many memorable concerts.

Last but not least, I want to express my deepest gratitude to my family for their unwavering support, encouragement, and love throughout my entire life. No words can express how much I appreciate everything you have done for me. Thank you to my siblings, Angelina and Flurin, for your care and support, for making time for me despite your busy schedules, and for all the wonderful memories we have created together. Thank you to my parents, Helen and Philipp, for being my most important role models, for your unconditional love, for always believing in me, for visiting me no matter where I was, for the sacrifices you made to provide me with the best opportunities, and ultimately for making *all* of this possible. Without you, I would not be the person I am, nor would I be where I am today. I hope this achievement and all future ones make you proud.

Singapore, December 2025

Contents

Introduction 1

Security protocols are central to securing communication and distributed computation and, by nature, they are often employed in critical applications. These critical applications are numerous. E.g., in online banking, we use Transport Layer Security (TLS) to securely communicate with our bank's server and desire at least authentication, meaning that no one can perform transactions on our behalf, and replay protection, which ensures that legitimate transactions cannot be duplicated. Likewise, we employ secure messaging services to ensure that our private conversations remain confidential and are not tampered with. This is relevant for exchanging sensitive information, such as medical records, intellectual property or political and religious discussions; disclosing the latter in particular can result in life-threatening consequences in certain jurisdictions.

Unfortunately, as history amply demonstrates, security protocols and their implementations are notoriously difficult to get right, and their flaws can be a source of devastating attacks. Thus, it is crucial to prove that a security protocol actually provides the security properties it is intended to guarantee to an application, such as secrecy and authentication. However, reasoning about security protocols and their implementations is challenging.

First, proving that a security protocol achieves a particular security property must consider *all* possible attacks that an attacker can perform in every possible state of a distributed system and show that neither attack breaks this security property. Enumerating all these attacks is impossible as we not only consider distributed systems with infinitely many states but also the attacker can perform an infinite number of actions, such as constructing messages, sending them, and observing responses by protocol participants. For example, an attacker performing the TLS Triple Handshake Attack [1] engages in two parallel TLS connections with different entities, such as a bank client and a bank's server, and selects particular connection parameters. After several session resumptions, the attacker manages to connect the bank client with the bank's server, which they do not intend. This attack potentially enables the attacker to inject data before the final session resumption that ultimately appears to the bank's server as coming from the bank client, which means that the attacker can impersonate the bank client and, thus, break authentication.

[1]: Bhargavan et al. (2014), *Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS*

Second, implementations of security protocols pose significant risks for coding errors. As they are typically employed at a lower layer in a software stack, which establishes security for higher layers, they interface with the untrusted, adversarial network. Carefully crafted messages by the attacker can thus exploit coding errors in this layer. E.g., the famous Heartbleed vulnerability [2] allowed attackers to read sensitive data from the memory of servers and clients, including private keys and passwords, due to missing bounds checks in the OpenSSL library, which is widely used for establishing TLS connections. In addition, security protocol implementations are complex due to their intricate combination of cryptographic primitives and interactions with other protocol participants making them error-prone. Apple's TLS library was susceptible to impersonation attacks due to a superfluous goto statement [3] causing the library to skip checking TLS certificate signatures. Ignoring errors, e.g., returned by a TLS library [4, 5], or omitting protocol

[2]: CVE (2013), *CVE-2014-0160*

[3]: CVE (2014), *CVE-2014-1266*

[4]: CVE (2022), *CVE-2022-22805*

[5]: CVE (2022), *CVE-2022-22806*

[6]: CVE (2021), *CVE-2021-40823*

steps as in the Matrix software development kit (SDK) [6] enable attacks even if the protocol design might be secure.

Faulty security protocols and their implementations can have devastating consequences, as they can lead to unauthorized access to sensitive data, financial loss, and even physical harm. Additionally, the widespread deployment of, e.g., the OpenSSL library means that a single vulnerability can affect millions of users and systems worldwide.

This highlights the importance of formal verification of security protocols and their implementations to ensure their correctness. This dissertation tackles this problem by developing verification methodologies to prove security properties for implementations, facing the complexities of security protocols and modern, widely-used programming languages. Additionally, we present a solution to soundly scale verification to large codebases while allowing for a high degree of automation. This is crucial for production codebases because application logic on higher levels of a software stack can forfeit security properties established by a security protocol implementation on a lower level.

[7]: Schmidt et al. (2012), *Automated Analysis of Diffie–Hellman Protocols and Advanced Security Properties*
[8]: Meier et al. (2013), *The TAMARIN Prover for the Symbolic Analysis of Security Protocols*
[9]: Blanchet (2001), *An Efficient Cryptographic Protocol Verifier Based on Prolog Rules*
[10]: Cremers et al. (2016), *Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication*
[11]: Cremers et al. (2017), *A Comprehensive Symbolic Analysis of TLS 1.3*
[12]: Bhargavan et al. (2017), *Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate*
[13]: Basin et al. (2018), *A Formal Analysis of 5G Authentication*
[14]: Basin et al. (2021), *The EMV Standard: Break, Fix, Verify*
[15]: Basin et al. (2021), *Card Brand Mixup Attack: Bypassing the PIN in non-Visa Cards by Using Them for Visa Transactions*
[16]: Basin et al. (2024), *Getting Chip Card Payments Right*
[17]: Linker et al. (2025), *A Formal Analysis of Apple's iMessage PQ3 Protocol*
[18]: Dolev et al. (1983), *On the Security of Public Key Protocols*
[19]: Barbosa et al. (2021), *SoK: Computer-Aided Cryptography*
[20]: Blanchet (2006), *A Computationally Sound Mechanized Prover for Security Protocols*
[21]: Barthe et al. (2011), *Computer-Aided Security Proofs for the Working Cryptographer*
[22]: Abate et al. (2021), *SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq*
[23]: Blanchet (2012), *Security Protocol Verification: Symbolic and Computational Models*

1.1 State of the Art

This section overviews the state of the art in reasoning about security protocol models, programs, and implementations of security protocols. Sec. 2.8, Sec. 3.11, and Sec. 4.7 provide more detailed comparisons of this dissertation's contributions with related work.

1.1.1 Protocol Model Verification

A *security protocol model* captures the essence of a security protocol, abstracting from low-level details like memory management, bit-level descriptions of protocol messages, and the algorithm details of the employed cryptographic primitives. Therefore, such models focus on the interactions between protocol participants and the high-level components of protocol messages. Their abstract nature makes them amenable to formal reasoning as the proof search is not hindered by low-level details. Over the past decades, expressive and highly automated *security protocol model verifiers* have been developed, including the two state-of-the-art tools Tamarin [7, 8] and ProVerif [9], which have been used to analyze real-world protocol models such as TLS [10–12], 5G [13], EMV [14–16], and Apple's iMessage PQ3 [17]. These tools build on a model of cryptographic protocols called the *symbolic* or *Dolev–Yao (DY) model* [18], where cryptographic primitives are idealized, protocols are modeled by process algebras or rewriting systems, and the attacker is an abstract entity controlling the network and manipulating messages represented as terms. The symbolic model is powerful as it abstracts from low-level details of cryptographic primitives, such as the probabilities for generating an already existing key, and hash collisions; thus, the symbolic model reduces the proof effort and enables automation, i.e., to automatically find a proof that a model satisfies certain security properties [19]. In contrast, approaches adopting the *computational* model (e.g., CryptoVerif [20], EasyCrypt [21], and SSProve [22]) are more precise and, thus, give stronger guarantees than symbolic ones. However, their proofs are very difficult to automate, as surveyed by Blanchet [23] and Barbosa *et al.* [19].

While these protocol model verifiers enable proving that a security protocol achieves desired security properties, they do *a priori* not provide any guarantees about implementations of the protocols. For example, omitted protocol steps, incorrect certificate checks, missing error handling, and other coding errors like buffer overflows can lead to security vulnerabilities despite the protocol model being secure.

1.1.2 Program Verification

Traditionally, developers apply *testing* to ensure that their implementations are correct by checking for a small set of possible inputs whether an implementation demonstrates the desired behavior. However, the main limitation of testing is that only the presence of bugs for the tested inputs can be shown, not providing any guarantees about their absence, as a bug might get triggered by an untested but nevertheless possible input. Alternatives to testing exist; E.g., *runtime monitoring* augments an implementation with additional runtime checks that abort the execution as soon as a bug, i.e., a violation of a desired property, is detected. While runtime monitoring provides guarantees that the monitored execution is free of bugs, it does not provide any guarantees about all other possible executions. *Static* approaches fill this gap by reasoning about an implementation's source code without executing it or adding any runtime checks. Model checking is such an approach that typically starts at the entry point of an implementation and tries to exhaustively explore all reachable program states, which, however, quickly becomes intractable. A common remedy is to *bound* a model checker to a finite number of program states, e.g., by limiting the number of considered loop iterations. However, this bound trades soundness for practicality as a model checker misses bugs that lie in a program state that is outside the chosen bound.

Program verification, on the other hand, aims to prove that an implementation achieves safety and functional properties for *all* possible inputs without limiting the number of considered program states. In this context, the term *safety* expresses that an implementation neither causes runtime exceptions nor undefined behavior. In particular, it covers the absence of memory errors, buffer overflows, and data races. *Functional properties* are implementation-specific and express the desired behavior, e.g., that a sorting algorithm's result is a sorted permutation of the input. To scale to large, real-world programs, the proof for an entire implementation is decomposed into smaller proofs for individual components, typically functions and methods, to make the proof search tractable for proof engineers and dedicated verification tools. This decomposition relies on a *modular* program logic that allows verifying components individually and then combining the per-component proofs to constitute a proof for the entire implementation. To do this, we equip every method (and function) with a *specification* that consists of a pre- and postcondition. A method's precondition is a logical formula specifying all valid program states in which this method can be called, and a method's postcondition specifies properties that hold for all valid program states after executing the method's body.

Verifying programs with side effects, mutable state, and concurrency poses additional challenges for modular verification. While purely functional programming languages do not offer these features, they are commonly found in widely used programming languages such as C, C++, Java, Python, and Go. Mutable state in a fully sequential program not only causes aliasing, where multiple variables refer to the same memory location, but also side effects, i.e., calling another method might modify arbitrary memory locations. Concurrency adds further complications, as multiple threads can access and modify the same memory locations at the same time, leading to data races.

To enable modular verification of programs with mutable state and concurrency, specialized program logics have been developed, foremost among them separation logic [24, 25], the de facto standard. Separation logic achieves modular reasoning by associating a so-called *permission* with each memory location, which is required to read or write a memory location and conceptually represents ownership of a memory location. Thus, every method includes in its precondition the permissions required to perform its memory accesses, and its postcondition specifies the permissions that are returned to the caller. Since permissions are non-duplicable, separation logic provides a powerful way to reason about side effects and concurrency. More specifically, separation logic allows us to express precisely in a method's precondition which heap fragment f this method operates on. Using separation logic's connectives that split and combine heap fragments, e.g., when calling another method that operates only on a subfragment $f' \subseteq f$, we know that the method does not modify any heap location in the *frame* $f \setminus f'$.

[24]: O'Hearn et al. (2001), *Local Reasoning about Programs that Alter Data Structures*
[25]: Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*

Proving that each method has sufficient permissions for each heap access guarantees safety. E.g., a buffer overflow corresponds to accessing an array element out of bounds; this is prevented since allocating an array creates permissions only for in-bound elements. Similarly, data races are prevented since two threads simultaneously writing the same heap location would require that both threads have permission for writing this heap location, which is impossible as there is only a single permission for writing any given heap location.

Several program verifiers for different programming languages exist that are based on separation logic. For example, VeriFast [26], VST [27], and RefinedC [28] for C, VeriFast and VerCors [29] for Java, Nagini [30] for Python, and Verus [31] and Prusti [32] for Rust. Most of these verifiers are auto-active [33], i.e., encode the proof obligations as an unsatisfiability query to a satisfiability modulo theories (SMT) solver, which automates the proof search. Despite the automation, verifying programs remains a high effort task as proof engineers have to provide pre- and postconditions for each method and loop invariants. Additionally, proof engineers may have to assert intermediate properties to guide the SMT solver's proof search.

[26]: Jacobs et al. (2011), *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*

[27]: Cao et al. (2018), *VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs*

[28]: Sammler et al. (2021), *RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types*

[29]: Blom et al. (2014), *The VerCors Tool for Verification of Concurrent Programs*

[30]: Eilers et al. (2018), *Nagini: A Static Verifier for Python*

[31]: Lattuada et al. (2023), *Verus: Verifying Rust Programs using Linear Ghost Types*

[32]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*

[33]: Leino et al. (2010), *Usable Auto-Active Verification*

Gobra. Despite not forming one of the technical chapters of this thesis, I[1] have significantly contributed to the development of Gobra [34], a program verifier for the Go programming language. Gobra is a translational verifier translating Go programs into Viper [35]. Viper is an infrastructure for program verification offering the Viper language, a sequential programming language with mutable state, and verifiers that prove the correctness of Viper programs. Hence, Gobra encodes a potentially concurrent Go program using, e.g., structural subtyping, goroutines, which are lightweight threads, and message-passing communication into a Viper program, such that the correctness of this Viper program implies the correctness of the original Go program.

1: Here and in Sec. 1.4, I use first-person singular to distinguish my contributions

[34]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*

[35]: Müller et al. (2016), *Viper: A Verification Infrastructure for Permission-Based Reasoning*

Over the years, Gobra has been extended to not only support additional Go features but also to support more verification techniques that make reasoning easier and more concise. For example, my contributions to Gobra include adding support for ghost struct fields and ghost memory locations. The former enable co-locating additional information used exclusively for verification purposes with a struct. The latter allow us to share potentially complex data structures, such as the one in Chapter 3, with other goroutines without requiring the data structure to be part of the program's state at runtime. All these advancements made it possible to apply Gobra to real-world Go programs such as a next-generation internet router [36] and the case studies in this thesis, which include an implementation of the WireGuard [37] Virtual Private Network (VPN) protocol.

[36]: Pereira et al. (2025), *Protocols to Code: Formal Verification of a Secure Next-Generation Internet Router*

[37]: Donenfeld (2017), *WireGuard: Next Generation Kernel Network Tunnel*

1.1.3 Protocol Implementation Verification

Reasoning about security properties on the implementation level is challenging as on the one hand the same challenges as for program verification apply and on the other hand, security properties are typically global properties, i.e., hold only if all protocol participants behave correctly. E.g., confidentiality of some key material requires that every participant in a distributed system having access to this key material does not leak it to an attacker. Since protocol participants often run different implementations, depending on their role in the protocol, reasoning about security properties requires reasoning about the interactions of these different implementations.

Related work faces this challenge using one of three approaches, which we discuss in detail next: generating an implementation from a verified protocol model, extracting a protocol model from an implementation, or proving security properties directly on an implementation—possibly in conjunction with a protocol model. Avalle *et al.* [38] survey early work using one of the first two approaches.

[38]: Avalle et al. (2014), *Formal Verification of Security Protocol Implementations: a Survey*

Code Generation. The first approach avoids reasoning about implementations by *generating* secure-by-construction implementations from an abstract protocol model. For instance, Bhargavan *et al.*'s DY* framework [39–41] takes functional specifications and implementations in F* [42] as input and generates OCaml code. Besides proving that an implementation refines a specification, F* additionally supports generating C code if an implementation is written in Low* [43] (a particular subset of F*). The generated OCaml and C code is secure by construction (provided the code generator is correct). However, changing the code manually (e.g., to optimize performance) forfeits any security guarantees. To achieve modular verification, DY* relies on a specific coding discipline (at most one protocol step per F* function), which must be enforced manually, and is in general not adhered to by existing implementations. A violation of this discipline unwittingly restricts the capabilities of the attacker and, thus, may cause DY* to miss attacks.

[39]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

[40]: Bhargavan et al. (2021), *An In-Depth Symbolic Security Analysis of the ACME Standard*

[41]: Ho et al. (2022), *Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations*

[42]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

[43]: Protzenko et al. (2017), *Verified Low-Level Programming Embedded in F*

The DY* framework has been applied to several protocols. Bhargavan *et al.* [39] take an existing Signal implementation (in Low*) and its functional protocol specification (in F*) [44] and show that its functional protocol specification (also in F*) satisfies forward and post-compromise security. The same authors implement and verify ACME in F* [40], which extracts to OCaml. Ho *et al.* [41] provide a framework for generating implementations for Noise protocols. Their framework consists of Low* code (extracting to C), which refines functional F* specifications (that would extract to OCaml). To apply DY*, the authors replace in their F* specification calls to concrete cryptographic primitives by calls to their symbolic counterpart. The resulting symbolic specification is not

[44]: Protzenko et al. (2019), *Formally Verified Cryptographic Web Applications in WebAssembly*

executable, and is manually kept in sync with the original, executable specification. The authors prove secrecy and authentication properties as stated by Noise for the symbolic specification. To avoid manually keeping a symbolic and concrete specification in sync, Wallez *et al.* [45] parameterize their F⋆ specification for Messaging Layer Security (MLS) by a type class, which can be instantiated either by concrete bytes and concrete cryptographic primitives (to generate an executable specification in OCaml) or symbolic terms and symbolic cryptographic primitives (to apply DY⋆).

[45]: Wallez et al. (2023), *TreeSync: Authenticated Group Management for Messaging Layer Security*

While DY⋆ and Pozza *et al.* [46] obtain guarantees in the symbolic model, Cadé *et al.* [47, 48], Delignat-Lavaud *et al.* [49, 50] and, developed in parallel to this dissertation, Owl [51] provide computational guarantees. As is the case for DY⋆, implementations generated by these works cannot be optimized by hand or integrated into a larger codebase without forfeiting proven security properties. OwlC [52], which extends Owl, partially addresses this limitation by generating not only a library implementing a security protocol in Rust but also a specification to verify the generated library against this specification using Verus. Composing the generated library with handwritten code is particularly crucial for Owl and OwlC to support realistic applications as their domain-specific protocol modeling language supports neither recursion nor loop constructs. Thus, any looping behavior can occur only in the handwritten code. Since OwlC treats all code except for the generated and verified library as being attacker-controlled, the generated library can share only public values with the handwritten code—a constraint enforced via the Rust type system. To support more sophisticated applications that require access to non-public values, one would have to insert declassifications of these values into the generated specification. However, modifying the generated specification forfeits the security guarantees as Verus does not currently support reasoning about security properties.

[46]: Pozza et al. (2004), *Spi2Java: Automatic Cryptographic Protocol Java Code Generation from Spi Calculus*

[47]: Cadé et al. (2012), *From Computationally-Proved Protocol Specifications to Implementations*

[48]: Cadé et al. (2013), *Proved Generation of Implementations from Computationally Secure Protocol Specifications*

[49]: Delignat-Lavaud et al. (2017), *Implementing and Proving the TLS 1.3 Record Layer*

[50]: Delignat-Lavaud et al. (2021), *A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer*

[51]: Gancher et al. (2023), *Owl: Compositional Verification of Security Protocols via an Information-Flow Type System*

[52]: Singh et al. (2025), *OwlC: Compiling Security Protocols to Verified, Secure, High-Performance Libraries*

Model Extraction. The second approach *extracts* an abstract protocol model from an implementation, such that protocol model verifiers can be used to prove security properties about the extracted model. Over the years, numerous approaches have been developed targeting different programming languages and protocol model verifiers. Bhargavan *et al.* [53] extract a ProVerif model from code written in a subset of F#, which is used in [54] to verify TLS 1.0. Recent work by Bhargavan *et al.* [55] targets a subset of Rust to extract a symbolic protocol model for ProVerif, an implementation in F⋆ for safety verification, and packages for a computational proof using SSProve. They apply their methodology to their own post-quantum implementation of TLS 1.3. Elyjah [56] extracts a model from Java, which requires implementing all protocol roles and the network connecting instances of these roles as different Java classes within the same file. While Aizatulin *et al.* [57] target C code and obtain guarantees in the computational model by targeting CryptoVerif, their tool considers only a single execution of the input code, i.e., ignores untaken branches and unrolls loops as many times as taken in the considered execution. Bhargavan *et al.* [12] and Kobeissi *et al.* [58] extract a model from a subset of JavaScript that disallows, e.g., recursion and loops, and generate a ProVerif model. [12] and [58] additionally use CryptoVerif to obtain computational guarantees but must modify the generated model to match CryptoVerif's input language and to make them easier to verify. CryptoBap [59] analyzes ARMv8 and RISC-V machine code and extracts to ProVerif and CryptoVerif. Follow-up work [60] extends the extraction to Tamarin and enables the combination of components implemented in different languages, e.g., to manually model the high-level behavior

[53]: Bhargavan et al. (2008), *Verified Interoperable Implementations of Security Protocols*

[54]: Bhargavan et al. (2012), *Verified Cryptographic Implementations for TLS*

[55]: Bhargavan et al. (2025), *Formal Security and Functional Verification of Cryptographic Protocol Implementations in Rust*

[56]: O'Shea (2008), *Using Elyjah to Analyse Java Implementations of Cryptographic Protocols*

[57]: Aizatulin et al. (2012), *Computational Verification of C Protocol Implementations by Symbolic Execution*

[12]: Bhargavan et al. (2017), *Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate*

[58]: Kobeissi et al. (2017), *Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach*

[59]: Nasrabadi et al. (2023), *CryptoBap: A Binary Analysis Platform for Cryptographic Protocols*

[60]: Nasrabadi et al. (2025), *Symbolic Parallel Composition for Multi-Language Protocol Verification*

of other protocol roles instead of considering their binaries. However, both works [59, 60] translate a binary to a sequential intermediate representation, thus, suggesting that they do not support concurrency within a protocol role.

Several works generate both models for verification and executable code from abstract protocol descriptions. In [61, 62], Alice&Bob-style protocol specifications are translated into ProVerif models and into JavaScript or Java implementations. Sisto *et al.* [63] generate a ProVerif model and a refined Java implementation from an abstract Java protocol specification.

[61]: Modesti (2015), *AnBx: Automatic Generation and Verification of Security Protocols Implementations*
[62]: Almousa et al. (2015), *Alice and Bob: Reconciling Formal Models and Implementation*
[63]: Sisto et al. (2018), *Formally Sound Implementations of Security Protocols with JavaSPI*

Common to all approaches based on model extraction is that this extraction typically requires that an implementation follows restrictive coding disciplines, such that relevant protocol steps can be identified and extracted. E.g., protocol steps that could happen concurrently must be identified and represented accordingly in the extracted model.

Code-Level Verification. The third approach, which this dissertation advances, takes implementations and proves security properties either directly or in combination with an abstract protocol model. Dupressoir *et al.* [64, 65] combine an interactive theorem prover and the program verifier VCC [66] to reason about security properties of C code. Bhargavan *et al.* [67] modularly verify protocol code written in F# using the F7 refinement type checker [68]. They rely on protocol-specific invariants for cryptographic structures, e.g., stating which messages are public. To state authentication properties, they use a combination of (trusted) assume and assert statements. The more recent work [39], which we already covered as an approach for generating implementations, overcomes this limitation by incorporating a global trace; we explain this idea in detail in Chapter 3. Vanspauwen and Jacobs [69, 70] use a similar approach for protocols implemented in C and verified using VeriFast. They extend the symbolic model of cryptography to enable the attacker to directly manipulate byte strings, which they over-approximate by considering the set of symbolic terms that possibly influences a byte string. The Igloo framework [71] provides a series of generic steps that gradually transform an abstract model into a specification. While program verifiers can establish that an implementation meets this specification, this framework requires establishing a refinement relation between each successive pair of steps.

[64]: Dupressoir et al. (2011), *Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols*
[65]: Dupressoir et al. (2014), *Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols*
[66]: Cohen et al. (2009), *VCC: A Practical System for Verifying Concurrent C*
[67]: Bhargavan et al. (2010), *Modular Verification of Security Protocol Code by Typing*
[68]: Bengtson et al. (2008), *Refinement Types for Secure Implementations*
[39]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*
[69]: Vanspauwen et al. (2015), *Verifying Protocol Implementations by Augmenting Existing Cryptographic Libraries with Specifications*
[70]: Vanspauwen et al. (2017), *Verifying Cryptographic Protocol Implementations that use Industrial Cryptographic APIs*
[71]: Sprenger et al. (2020), *Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification*

1.2 Challenges

This dissertation advances the state of the art in verifying strong security properties for protocols implemented in programming languages offering mutable state and concurrency. To achieve this goal, this dissertation tackles the following three main challenges.

Challenge 1: Leveraging Verified Protocol Models. Security protocol model verifiers, such as Tamarin, have been successfully applied to a wide range of protocols to ensure that these protocol models are secure. Several such applications focused on protocol candidates and had a positive impact on the standardization process by first uncovering security flaws, often suggesting fixes, and ultimately verifying the fixed protocol, such as for TLS [10–12], 5G [13], and MLS [45, 72]. As a natural next step in a standardization process, a protocol is implemented potentially multiple times in different programming languages to provide, e.g., a reference

[10]: Cremers et al. (2016), *Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication*
[11]: Cremers et al. (2017), *A Comprehensive Symbolic Analysis of TLS 1.3*
[12]: Bhargavan et al. (2017), *Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate*
[13]: Basin et al. (2018), *A Formal Analysis of 5G Authentication*
[45]: Wallez et al. (2023), *TreeSync: Authenticated Group Management for Messaging Layer Security*
[72]: Wallez et al. (2025), *TreeKEM: A Modular Machine-Checked Symbolic Security Analysis of Group Key Agreement in Messaging Layer Security*

implementation, an optimized high-performance implementation or an implementation for a specific platform. Iterating over protocol models and gradually moving towards implementations catches security flaws as early as possible, before potentially spending days or weeks implementing a flawed protocol. Instead of ignoring an existing, verified protocol model and reproving all security properties for an implementation, it is highly desirable to leverage this model. While the state of the art offers several different approaches, these approaches are generally limited to a single programming language. Approaches based on code generation do not allow optimizing and adapting a generated implementation for performance and to other platforms, respectively. Other approaches do not use the verified protocol model as they either extract a *new* model from the implementation or do the entire proof on the implementation level.

To flexibly support various use cases, we need an approach that is programming language-agnostic, allows for optimized implementations, and does not impose coding disciplines such that existing implementations can be verified. Additionally, such an approach should exploit the automation offered by protocol model verifiers to additionally reduce the overall proof effort for the protocol model *and* the implementation compared to directly proving the same security properties on the implementation level. Achieving this goal is challenging as it requires bridging the distinct formalisms and abstraction levels used by a protocol model verifier and a program verifier. Furthermore, it requires dealing with the full details of prevalent programming languages as reference implementations optimize for readability (thus preferring widely used, imperative languages over, e.g., purely functional ones) and fast implementations exploit mutable state and concurrency, neither of which make verification easier.

Challenge 2: Unifying Reasoning in a Program Verifier. There are several cases in which a protocol model verifier cannot be leveraged. E.g., if a protocol model does not exist, is outdated, the protocol model's proof turns out to be very challenging for the protocol model verifier or if the trust assumptions and required expertise have to be minimized to make it more accessible to proof engineers. These cases demand for proving ideally the same security properties for implementations as one would prove for protocol models, and for using a single formalism, namely the one of a program verifier. This is challenging for mainly two reasons. First, security properties are typically expressed in a different formalism than found in program verifiers and, second, are global properties, i.e., require co-operation by multiple protocol participants. In particular, a security property often refers to the existence or absence of certain actions not only in the past but also performed by other protocol participants. This global view is a priori not available when verifying an implementation, which usually contains only the code for a single protocol role. In addition, a program verifier has to cope with the complexity arising from both, the programming patterns employed by developers, and the advanced features provided by programming languages. Since a protocol, or more precisely its protocol roles, are potentially implemented in different programming languages, there is the additional challenge of developing a language-agnostic approach.

Challenge 3: Scaling Verification to Real-World Codebases. The final challenge that this dissertation must address such that real-world codebases can be verified is scalability. Many approaches focus on a particular protocol or even just a particular phase therein without considering the codebase in which a protocol is implemented. Existing, real-world codebases go against the spirit of approaches based on code generation and are typically too large and complex to extract a protocol model from them. Our chosen approach, i.e., verifying security properties on the level of implementations using a program verifier, gives in principle a story line how to verify an entire codebase. However, verifying an entire codebase is often financially infeasible since program verification is labor-intensive. Our observation is that security protocols are typically employed at a lower protocol stack layer such that the application layer can focus on implementing the actual business logic, ideally without worrying about how messages are protected against eavesdropping or tampering. Thus, security risks vary for different parts of a codebase, e.g., parts implementing a security protocol pose higher security risks than others. To maximize the guarantees one can obtain with bounded financial resources, it is desirable to focus most of the verification effort on these high-risk parts by applying a program verifier there and use more lightweight and scalable techniques for all other parts. However, combining the program verifier's reasoning about the high-risk parts with more these lightweight techniques is challenging for codebases implemented in programming languages with mutable state and concurrency, as in principle any memory location accessed by the high-risk parts can be concurrently modified by all other parts. This issue surfaces in separation logic as the proof obligation that the entire codebase respects the permissions, i.e., performs memory accesses only if a method currently possesses the required permissions. While this proof obligation is discharged for high-risk parts of the codebase by applying a program verifier, all other parts do not *a priori* respect these permissions. Thus, to verify security properties for large, real-world codebases, this dissertation must address the challenge of ensuring in a scalable way that the parts of a codebase on which we do not apply a program verifier do not interfere with the high-risk, verified parts.

1.3 This Dissertation

This dissertation enables verifying strong security properties for protocols implemented in (a) different programming languages offering mutable state and concurrency without presuming particular coding disciplines, thus, avoiding the need to reimplement protocols, (b) utilizing abstract protocol models if they exist, and (c) large, real-world codebases. To achieve this goal, this dissertation successfully develops novel methodologies overcoming the three challenges in Sec. 1.2 and applies them to various case studies ranging from WireGuard, a state of the art VPN protocol, to a large AWS codebase.

Chapter 2 presents a methodology to bridge the gap between abstract protocol models and concrete implementations. This methodology establishes a refinement relation showing that all trace-based security properties proven for an abstract protocol model hold for a concrete system. Since this concrete system is composed of potentially unboundedly many instances, each executing a specific protocol role, a key ingredient of this methodology is decomposing the (global) abstract protocol model into the different protocol roles and proving that each role's implementation refines the corresponding protocol role. We extend Tamarin to automate this process. I.e., Tamarin automatically produces specification for each protocol role from a protocol model that can be used by different program verifiers. Verifying that an implementation satisfies such a specification guarantees that the implementation refines the corresponding protocol role. On a high level, this refinement proof shows that every I/O operation performed by an implementation is justified by the abstract model, which encompasses not only the correct ordering of I/O operations but also the correct payloads for these operations. Establishing the correctness of these payloads requires bridging the different abstraction levels of protocol models and implementations as the former operates on symbolic terms and the latter on concrete byte arrays stored in memory.

While the first methodology shifts all reasoning about global security properties such as secrecy and agreement to the abstract protocol model and its verifier, our second methodology in Chapter 3 integrates this reasoning into off-the-shelf program verifiers. To obtain a language-agnostic methodology, we build on separation logic and well understood techniques from concurrency reasoning to capture all possible interleavings of operations performed by instances of protocol roles and the attacker. Furthermore, separation logic allows us to go beyond state of the art in this area, i.e., prove stronger security properties than related work, namely injective agreement.

Chapter 4 introduces Diodon, a novel methodology for scaling verification to large, production codebases with an emphasis on security properties. Diodon's central idea is to focus the verification effort on the most security-critical parts of a codebase, which requires precise reasoning about, e.g., payloads of I/O operations. However, all other parts of a codebase cannot be ignored as they still pose a security risk. Thus, Diodon employs lightweight static analyses that scale to the size of these codebases to ensure that these parts are not only free of security risks but also use the critical parts correctly. To achieve this goal, we bridge the gap between the different formalisms used in program verifiers and static analyses, and obtain a provably sound technique to construct a proof for the entire codebase, where all side conditions are discharged by static analyses.

In summary, this dissertation makes the following high-level contributions:

- A methodology for automatically extracting refinement proof obligations from an abstract protocol model for implementations. Discharging these proof obligations guarantees that implementations satisfy the same security properties as the abstract protocol model.
- A language-agnostic methodology for separation logic-based program verifiers to prove strong security properties including forward secrecy and injective agreement without relying on protocol model verifiers.
- A methodology for combining separation logic-based verification with lightweight static analyses to scale verification to large, production codebases. Security-critical code parts are verified using one of the first two methodologies, while the rest of the codebase is checked for security risks using lightweight static analyses.
- We apply all three methodologies to case studies of varying sizes and different protocols implemented in multiple programming languages, demonstrating that our methodologies are language-agnostic, applicable to real-world codebases, and scalable.

Outline. This dissertation presents the three methodologies one by one in the following technical chapters. Particularly noteworthy are the sections 3.10 and 4.6 therein. The former compares the first two methodologies and discusses their applicability. While Chapter 4 focuses on using the first methodology for verifying security-critical code parts in the context of Diodon, Sec. 4.6 discusses alternatively applying the second methodology to these code parts. Finally, Chapter 5 concludes and discusses future work.

Throughout this thesis, various stylistic conventions are employed to enhance readability. Supplementary information that is not essential for following the main narrative, but is nevertheless relevant, is presented in green boxes. In code listings, lines consisting exclusively of ghost code are highlighted with a gray background, except when another background color is already used elsewhere in the same listing.

1.4 Publications and Collaborations

The main results of this dissertation have been presented in the following publications.

The main results of Chapter 2 appeared in:

> *Sound Verification of Security Protocols: From Design to Interoperable Implementations*
> Linard Arquint, Felix A. Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N. Wiesner, David A. Basin, and Peter Müller.
> In IEEE Symposium on Security and Privacy (S&P), 2023. [73]

In comparison, this dissertation considers equational theories when deriving the pattern requirement in Sec. 2.4.3. Furthermore, Sec. 2.7 discusses an additional case study which is not part of the above publication.

The main results of Chapter 3 appeared in:

> *A Generic Methodology for the Modular Verification of Security Protocol Implementations*
> Linard Arquint, Malte Schwerhoff, Vaibhav Mehta, and Peter Müller.
> In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), 2023. [74]

Compared to the above publication, this dissertation additionally describes mechanisms to simplify the verification of concurrent implementations (in Sec. 3.6.3) and to enforce deletion of key material (in Sec. 3.9).

The main results of Chapter 4 are part of a US patent application [75] and are going to appear in:

[75]: Arquint et al. (2024), *A Method for Extending Safety Proofs of Protocols to Industrial Applications*

> *The Secrets Must Not Flow: Scaling Security Verification to Large Codebases*
> Linard Arquint, Samarth Kishor, Jason R. Koenig, Joey Dodds, Daniel Kroening, and Peter Müller.
> In IEEE Symposium on Security and Privacy (S&P), 2026. [76]

The work in this dissertation has been conducted in close collaboration with several researchers and students. The following paragraphs describe these collaborations.

Chapter 2 is the outcome of a collaboration with the Information Security Group at ETH Zurich. Besides pushing for a solution that supports message formats as they appear in real-world protocols, evaluating early versions of I/O specifications, and contributing to the technical discussions, I created and verified the Diffie–Hellman (DH) case study and, together with Felix A. Wolf, adapted and verified the WireGuard case study. The additional case study in Sec. 2.7 is the result of Lasse Meinen's Master's Thesis [77], which I co-led with Felix Linker. While Felix Linker provided the technical guidance on the Authentic Digital EMblem (ADEM) and its implementation, I advised on the application and adaptation of our verification methodology and the use of Gobra.

[77]: Meinen (2024), *Verification^2 of the Authentic Digital EMblem*

In close collaboration with my supervisor Peter Müller and the senior scientist Malte Schwerhoff, I led the work on Chapter 3. The work in Chapter 3 was led by me. Malte Schwerhoff and Peter Müller contributed with technical discussions and to the writing of the corresponding publication [74]. In addition to designing the entire methodology, I implemented the verification framework for Go, implemented all Go case studies that did not already exist, and verified this verification framework and all Go case studies. Additionally, I proved the methodology sound. Vaibhav Mehta contributed the framework's prototype in C and the Needham–Schroeder–Lowe (NSL) case study in C, primarily during a summer internship under my close supervision. The mechanism for secure deletion of key material (Sec. 3.9), resulted from Hugo Queinnec's Master's Thesis [78], which I closely led.

[74]: Arquint et al. (2023), *A Generic Methodology for the Modular Verification of Security Protocol Implementations*

[78]: Queinnec (2023), *Secure Deletion of Sensitive Data in Protocol Implementations*

Diodon (Chapter 4) is the result of two internships at AWS in Portland, OR. While I provided the expertise on Gobra, Tamarin, and their combination for verification of security protocol implementations, my collaborators at AWS contributed to technical discussions, provided the expertise on static analyses, and adapted the existing static analyses. I modeled the security protocol of the corresponding case study in Tamarin and implemented this protocol in an existing AWS codebase, building on an initial prototype implementation of an earlier protocol draft by Samarth Kishor. Afterwards, I performed the code-level verification with Gobra independently. In addition, I carried out the soundness proof after brainstorming sessions with my AWS collaborators. Finally, I provided the vast majority of the technical writing in the publication. My AWS collaborators mostly helped with editing and discussions on the story line, and Peter Müller contributed to the framing and by revising the paper.

Contributions Beyond This Dissertation. In the time working on this dissertation, I supervised numerous students working on Gobra and Gobra's ecosystem and contributed to the following publication:

> *Gobra: Modular Specification and Verification of Go Programs*
> Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller.
> In International Conference on Computer Aided Verification (CAV), 2021. [34]

Refinement-Based Verification of Security Protocol Implementations 2

2.1 Introduction

In this chapter, we approach the verification of security protocol implementations with the premise that an abstract protocol model exists—a premise we eschew in Chapter 3 when a model does not exist and is too difficult to construct. Leveraging abstract protocol models is attractive as dedicated tools like Tamarin and ProVerif with specialized proof search engines were developed in the past decades to reason about these models (cf. Sec. 1.1). Since implementation-level details, like memory accesses, are not present in these models, these tools can focus on proving security properties, such as secrecy and authentication, without getting sidetracked. At the same time, this absence of implementation-level details is a drawback as the protocol verified is a highly abstract version of the actual protocol executed and there is *a priori* no formal link between the model and implementation.

Existing approaches to verified security protocol implementations, as covered in Sec. 1.1, are usually tied to a specific implementation language, like ML, F*, or Java dialects, and they are difficult to extend to other languages. They are therefore ill-suited to verifying pre-existing implementations, especially when used for code extraction. In addition, the extraction mechanisms used are not always proved correct or even formalized, which weakens the guarantees for the resulting code. Moreover, in many cases, the security proof is tailored to the implementation considered rather than being performed at an abstract level by a standard security protocol verifier such as Tamarin or ProVerif. Hence, one can neither leverage these tools' automation capabilities nor the substantial prior work invested in security protocol proofs using them.

Our Approach. We propose a novel approach to end-to-end verified security protocol implementations. Our approach leverages the combined power of state-of-the-art security protocol verifiers and source code verifiers. This provides abstract, concise, and expressive security protocol specifications on the modeling side and flexibility and versatility on the implementation side.

More precisely, our approach bridges abstract security protocol models expressed in Tamarin as multi-set rewriting systems—one of the most advanced and widely used security protocol verifiers—with concrete program specifications expressed as I/O specifications (in a dialect of separation logic [25]), against which implementations can be verified. Its technical core is a procedure, and the associated tool implementation, that translates Tamarin models into I/O specifications along with a soundness proof, stating that an implementation satisfying the I/O specifications refines the abstract model in terms of trace inclusion. As a result, any *trace property* proved for the abstract model using Tamarin, including standard security protocol properties such as secrecy and authentication, also holds for the implementation.

[25]: Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*

Our approach provides a modular and flexible way to verify security protocol implementations. On the model verification side, we can leverage Tamarin's proof automation capabilities to prove protocols secure.

Moreover, we can prove a given protocol's security once in Tamarin, and reuse this proof to verify multiple implementations of this protocol, rather than having to produce a custom security proof for each implementation. In fact, numerous complex, real-world protocols have been analyzed using Tamarin over the years. Using our method, this substantial body of prior work can be exploited to verify implementations.

On the code verification side, the I/O specifications we produce can be encoded in many existing verifiers that support separation logic. Our tool currently generates I/O specifications for the Go code verifier Gobra [34] and for the Java code verifier VeriFast [26], which we respectively use for our case study and for our running example. It could easily be extended to other verifiers supporting I/O specifications such as Nagini [30] for Python code. In addition, the requirements for adding other code verifiers based on separation logic to our arsenal are low: they need to only support abstract predicates to encode I/O specifications and to guarantee that successful verification implies trace inclusion between the I/O traces of the program and those of its I/O specification.

[34]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*

[26]: Jacobs et al. (2011), *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*

[30]: Eilers et al. (2018), *Nagini: A Static Verifier for Python*

We establish our central soundness result relating Tamarin models via I/O specifications to implementations. This result follows a methodology inspired by the Igloo framework [71], which provides a series of generic steps that gradually transform an abstract model into an I/O specification, and requires establishing a refinement relation between each successive pair of steps. We take similar steps and prove these refinements *once and for all* starting from a generic Tamarin system, so that our method can be applied to obtain an I/O specification from any Tamarin protocol model (under some mild syntactic assumptions) without any additional proof.

[71]: Sprenger et al. (2020), *Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification*

Our Contributions. We summarize our contributions as follows:

- We design a framework for the end-to-end verification of security protocol implementations. This consists of a procedure and an associated tool to extract I/O specifications from a Tamarin model, which can be verified on implementation code. Our soundness result ensures that the implementation inherits all properties proven in Tamarin.
- We propose a novel approach to relate the I/O specifications' symbolic terms to the code's byte string messages. We parametrize the code verifiers' semantics with an abstraction function, instantiate it to a message abstraction function, and identify assumptions and proof obligations to verify that the code correctly implements terms as byte strings.
- To validate our approach, we perform a substantial case study on a complex, real-world protocol: the WireGuard key exchange, which is part of the widely-used WireGuard VPN in the Linux kernel. We verify a part of the official Go implementation of WireGuard, which is interoperable with the full version. Using our method, we thereby obtain an end-to-end symbolically verified WireGuard implementation.

2.2 Background

We present background on tools and methodology.

2.2.1 Tamarin and Multiset Rewriting

In the Tamarin prover [7, 8], protocols are represented as *multiset rewriting (MSR) systems*, where each rewrite rule represents a step or action taken by a protocol participant or the attacker. We present the building blocks in order: messages, facts, and rules.

[7]: Schmidt et al. (2012), *Automated Analysis of Diffie–Hellman Protocols and Advanced Security Properties*
[8]: Meier et al. (2013), *The TAMARIN Prover for the Symbolic Analysis of Security Protocols*

Message *terms* are elements of a *term algebra* $\mathcal{T} = \mathcal{T}_\Sigma(\mathcal{N} \cup \mathcal{V})$. These are built over a signature Σ of function symbols and a set of names $\mathcal{N} = \mathit{fresh} \cup \mathit{pub}$ consisting of a set of fresh names *fresh* (for secret values, generated by parties, unguessable by the attacker), a countably infinite set of public names *pub* (for globally known values), and a set of variables $\mathcal{V}$. Cryptographic *messages* $\mathcal{M}$ are modeled as ground terms, i.e., terms without variables. The term algebra is equipped with an equational theory E, which is a set of equations, and we denote by $=_\mathsf{E}$ the equality modulo E.

Example 2.2.1 (Diffie–Hellman Equational Theory) The signed Diffie–Hellman (DH) protocol is a well-known key exchange protocol, where two agents A and B exchange two DH public keys, g^x and g^y, to establish the shared key g^{xy} (where g is a group generator). For the Tamarin model, we use a term signature containing symbols $\hat{}$, g, *sign*, *verify*, *pk* modeling respectively exponentiation, the group generator, signature, verification, and public keys. We use the simplified theory:

$$(g^x)^y = (g^y)^x \qquad \mathit{verify}(\mathit{sign}(x,k), \mathit{pk}(k)) = \mathit{true}$$

Tamarin's actual model includes further equations. The protocol's informal description is as follows:

$$\begin{array}{lll} A \to B: & g^x & x \text{ fresh} \\ B \to A: & \mathit{sign}(\langle 0, B, A, g^x, g^y\rangle, k_B) & y \text{ fresh} \\ A \to B: & \mathit{sign}(\langle 1, A, B, g^y, g^x\rangle, k_A) & \text{agree on } (g^x)^y =_\mathsf{E} (g^y)^x \end{array}$$

The tags 0 and 1 are used to distinguish the last two messages.

All parties, including the attacker, can use the equational theory. The attacker also has its own set of rewriting rules, expressing that it can intercept, modify, block, and recombine all network messages, following the classic DY model [18]. These rules are generated automatically from the equational theory, but users may formalize additional rules giving the attacker further scenario-specific capabilities.

[18]: Dolev et al. (1983), *On the Security of Public Key Protocols*

Facts are simply atomic predicates applied to message terms, constructed over a signature $\Sigma_{\mathsf{facts}} = \Sigma_{\mathsf{lin}} \uplus \Sigma_{\mathsf{per}}$ of *fact symbols*, partitioned into *linear* (Σ_{lin}), i.e., single-use, and *persistent* (Σ_{per}) facts, which encode the state of agents and the network. We write $\mathcal{F} = \{F(t_1, \ldots, t_k) \mid F \in \Sigma_{\mathsf{facts}}$ with arity k, and $t_1, \ldots, t_k \in \mathcal{T}\}$ for the set of facts instantiated with terms, partitioned into $\mathcal{F}_{\mathsf{lin}} \uplus \mathcal{F}_{\mathsf{per}}$ as expected. In addition, $\cup^{\mathsf{m}}$, $\cap^{\mathsf{m}}$, $\setminus^{\mathsf{m}}$, $\subseteq^{\mathsf{m}}$, and $\in^{\mathsf{m}}$ denote the usual operations and relations on multisets, and for a multiset m, $\mathrm{set}(m)$ denotes the set of its elements.

A *multiset rewriting* (MSR) *rule*, written $\ell \xrightarrow{a} r$, contains multisets of *facts* ℓ and r on the left and right-hand side, and is labeled with a, a multiset of actions (also facts, but disjoint from state facts) used for property specification. A MSR system $\mathcal{R}$ and an equational theory E have a semantics as a labeled transition system (LTS), whose states are multisets of ground facts from $\mathcal{F}$, the initial state is empty ([]), and the transition relation $\overset{\cdot}{\Longrightarrow}_{\mathcal{R},\mathsf{E}}$ is defined by

$$\frac{\ell \xrightarrow{a} r \in \mathcal{R} \qquad \ell' \xrightarrow{a'} r' =_{\mathsf{E}} (\ell \xrightarrow{a} r)\theta \qquad \ell' \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}} \subseteq^{\mathsf{m}} S \qquad \mathsf{set}(\ell') \cap \mathcal{F}_{\mathsf{per}} \subseteq \mathsf{set}(S)}{S \overset{a'}{\Longrightarrow}_{\mathcal{R},\mathsf{E}} S \setminus^{\mathsf{m}} (\ell' \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}}) \cup^{\mathsf{m}} r'}, \qquad (2.1)$$

where θ is a ground instance of the variables in ℓ, a, and r. Intuitively, the relation describes an update of state S to a successor state, that is possible when a given rule in $\mathcal{R}$ is applicable, i.e., an instantiation with some θ (mod E) of its left-hand side appears in S. Applying the rule consumes the linear but not the persistent facts appearing in its left-hand side, and adds the instantiations under θ of all the facts of its right-hand side to the resulting successor state.

The MSR rules used in TAMARIN feature the reserved fact symbols $\mathsf{K} \in \Sigma_{\mathsf{per}}$, and $\mathsf{Fr}, \mathsf{in}, \mathsf{out} \in \Sigma_{\mathsf{lin}}$, modeling respectively the attacker's knowledge, freshness generation, inputs, and outputs. The attacker is modeled by a set of *message deduction rules* MD_Σ, giving it the DY capabilities mentioned above. A distinguished *freshness rule*, labeled $\mathsf{Fr}(n)$, generates fresh values n, which either protocol agents or the attacker directly learn, but never both.

Finally, a protocol's observable behaviors are its *traces*, which are sequences of multisets of actions labeling a sequence of transitions. We define the sets of full traces and of filtered traces with empty labels removed.

$$\begin{aligned}\mathrm{Tr}(\mathcal{R}) &= \{\langle a_i\rangle_{1\le i\le m} \mid \\ &\quad \exists s_1,\dots,s_m.\ [] \overset{a_1}{\Longrightarrow}_{\mathcal{R},\mathsf{E}} s_1 \overset{a_2}{\Longrightarrow}_{\mathcal{R},\mathsf{E}} \dots \overset{a_m}{\Longrightarrow}_{\mathcal{R},\mathsf{E}} s_m\} \\ \mathrm{Tr}'(\mathcal{R}) &= \{\langle a_i\rangle_{1\le i\le m, a_i\ne[]} \mid \langle a_i\rangle_{1\le i\le m} \in \mathrm{Tr}(\mathcal{R})\}.\end{aligned}$$

To ensure that fresh values are unique, we exclude traces with colliding fresh values by defining

$$\begin{aligned}\mathrm{Tr_t}(\mathcal{R}) = \{&\langle a_i\rangle_{1\le i\le m} \in \mathrm{Tr}'(\mathcal{R}) \mid \\ &\forall i,j,n.\ \mathsf{Fr}(n) \in a_i \cap^{\mathsf{m}} a_j \Rightarrow i = j\}.\end{aligned}$$

We will abbreviate inclusions between each kind of trace sets using relation symbols $\preccurlyeq$, $\preccurlyeq'$, and $\preccurlyeq_{\mathrm{t}}$. For example, $\mathcal{R}_1 \preccurlyeq_{\mathrm{t}} \mathcal{R}_2$ denotes $\mathrm{Tr_t}(\mathcal{R}_1) \subseteq \mathrm{Tr_t}(\mathcal{R}_2)$, and similarly for the other two. Note that $\preccurlyeq \subseteq \preccurlyeq' \subseteq \preccurlyeq_{\mathrm{t}}$.

We focus here on TAMARIN's *trace properties* (i.e., sets of traces) such as secrecy and authentication [79]. An MSR $\mathcal{R}$ satisfies a trace property Φ, if $\mathrm{Tr_t}(\mathcal{R}) \subseteq \Phi$.

[79]: Lowe (1997), *A Hierarchy of Authentication Specification*

Example 2.2.2 (Diffie–Hellman) Continuing Example 2.2.1, we use the linear fact symbols $\mathsf{Setup}_{Alice}(\overline{init})$, $Step^1_{Alice}(\overline{init}, x)$, $Step^2_{Alice}(\overline{init}, x, g^y)$ to initialize and record the progress of agent A playing the role of Alice in the protocol. The parameters of the facts store her knowledge. It is initialized with $\overline{init} = rid, A, k_A, B, pk_B$, i.e., a thread identifier, her identity, her private key, and her partner's identity and public key. It is then extended with her share of the secret x, and the DH public key g^y she received. For Alice, the two steps of the protocol can then be modeled by the rules:

$$[\mathsf{Setup}_{Alice}(\overline{init}), \mathsf{Fr}(x)] \xrightarrow{[]} [Step^1_{Alice}(\overline{init}, x), \mathsf{out}(g^x)]$$

$$[Step^1_{Alice}(\overline{init}, x), \mathsf{in}(sign(\langle 0, B, A, g^x, Y\rangle, k_B))] \xrightarrow{[Secret(Y^x)]} [Step^2_{Alice}(\overline{init}, x, Y), \mathsf{out}(sign(\langle 1, A, B, Y, g^x\rangle, k_A))]$$

The received signature is checked by expecting $pk_B = pk(k_B)$ using pattern-matching. The action fact $Secret(Y^x)$ in the second rule is used to specify key secrecy. It records Alice's belief that the key she computes from the value Y (supposedly g^y) received from Bob remains secret. The fact $\mathsf{Setup}_{Alice}(\overline{init})$ in the first rule is produced by another rule, modeling the environment initializing Alice's knowledge:

$$[\mathsf{Fr}(rid), \mathsf{sk}(A, k_A), \mathsf{pk}(B, pk_B)] \xrightarrow{[]} [\mathsf{Setup}_{Alice}(rid, A, k_A, B, pk_B)].$$

2.2.2 Separation Logic and I/O Specifications

Separation logic enables sound and modular reasoning about heap manipulating programs by associating every allocated heap location with a *permission*. Permissions are a static concept used to verify programs, but do not affect their runtime behavior. Each permission is held by at most one function execution at each point in the program execution. To access a heap location, a function must hold the associated permission; otherwise, a verification error occurs. The *separating conjunction* $\star$ sums up the permissions in its conjuncts. Permissions to an unbounded set of locations, for instance, all locations of a linked list, can be expressed via co-recursive predicates. Moreover, *abstract* predicates are useful to specify permissions to an unknown set of locations.

Permission-based reasoning generalizes from heap locations to other kinds of program resources. Penninckx *et al.* [80] reason about a program's I/O behavior by associating each I/O operation with a permission that is required to call the operation and is then consumed. They equip the main function's precondition with an *I/O specification* that grants all permissions necessary to perform the desired I/O operations of the entire program execution. These I/O specifications can easily be encoded in standard separation logic, such that existing program verifiers supporting different programming languages can be used to verify I/O behavior.

[80]: Penninckx et al. (2015), *Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs*

Every I/O operation `io`, such as sending or receiving a value, is associated with an abstract predicate **io**, called an *I/O permission*. Intuitively, $\mathbf{io}(p_1, \bar{v}, \bar{w}, p_2)$ expresses the permission to perform `io` with outputs $\bar{v}$ and inputs $\bar{w}$. We use $\bar{x}$ to denote a vector of zero or more values. The parameters p_1 and p_2 are called *source and target places*, respectively. An abstract predicate $token(p)$ is called a *token* at place p. The I/O operation `io` moves a token from the source place p_1 to the target place p_2 by consuming $token(p_1)$ and producing $token(p_2)$. Hence, the position of the token indicates the currently allowed I/O operations.

Example 2.2.3 (Send I/O Operation) Figure 2.1 shows the signature and specification of a `send` function. The precondition requires an I/O permission **out** to send the value `v` at some source place p_1 with the corresponding token. When the send operation succeeds, the I/O

Figure 2.1: Specification of the send operation with I/O permissions. Variables starting with ? are implicitly existentially quantified. The code verifier uses && to denote the separating conjunction $\star$.

```
requires token(?p1) && out(p1,v,?p2)
ensures  ok ⟹ token(p2)
ensures !ok ⟹ token(p1) && out(p1,v,p2)
func send(v int) (ok bool)
```

permission is consumed and the token is moved to some target place p_2. In case of failure, the token remains at the source place and the I/O permission is not consumed.

The separating conjunction of I/O permissions with the same source place encodes non-deterministic choice between such permissions. Moreover, co-recursion enables repeated as well as non-terminating sequences of I/O operations.

Example 2.2.4 (I/O Specification for a Server)

$$\begin{aligned} P(p, S) &= Q(p, S) \star R(p, S) \\ Q(p_1, S) &= \exists v, p_2, p_3.\, \mathbf{in}(p_1, v, p_2) \star \mathbf{out}(p_2, v, p_3) \\ &\quad \star P(p_3, S \cup \{v\}) \\ R(p_1, S) &= \exists p_2.\, \mathbf{out}(p_1, \text{“Ping”}, p_2) \star P(p_2, S) \end{aligned}$$

The formula $\phi = token(p) \star P(p, \emptyset)$ specifies a non-terminating server that repeatedly and non-deterministically chooses between receiving and forwarding a value v or sending a “Ping” message. All received values v are recorded in the state S, which is initially empty. Input parameters, like v in **in** here, are existentially quantified to avoid imposing restrictions on the values received from the environment.

To enforce certain state updates between I/O operations, it is useful to associate permissions also with certain internal (that is, non-I/O) operations and include those *internal permissions* in an I/O specification. For instance, the above server could include an internal operation to reset the state S when it exceeds a certain size.

I/O specifications induce a transition system and hence have a trace semantics. The traces can intuitively be seen as the sequences of I/O permissions consumed by possible executions of the programs that satisfy it. We write $\mathrm{Tr}(\phi)$ for the set of traces of an I/O specification ϕ. In the example above, the sequence $\mathbf{in}(5) \cdot \mathbf{out}(5) \cdot \mathbf{out}(\text{“Ping”}) \cdot \mathbf{in}(7) \cdot \mathbf{out}(7)$ is one example of a trace of ϕ. Note that the I/O permissions' place arguments do not appear in the trace.

2.3 From Tamarin Models to I/O Specifications

In this section, we present our transformation of a Tamarin protocol model, expressed as an MSR system $\mathcal{R}$, into a set of I/O specifications ψ_i, one for each protocol role i. They serve as program specifications, against which the roles' implementations c_i are verified (Sec. 2.4). Our main result is an overall soundness guarantee stating that the traces of the complete system $C(c_1, \ldots, c_n, \mathcal{E})$, composed of the roles' verified implementations c_i and the environment $\mathcal{E}$, are contained in the traces of the protocol model $\mathcal{R}$ (Sec. 2.5):

$$C(c_1, \ldots, c_n, \mathcal{E}) \preccurlyeq_{\mathrm{t}} \mathcal{R}.$$

Hence, any trace property Φ proven for the protocol model, i.e., $\mathrm{Tr_t}(\mathcal{R}) \subseteq \Phi$, is inherited by the implementation.

The sound transformation of an MSR protocol model $\mathcal{R}$ into a set of I/O specifications is challenging:

1. Tamarin's MSR formalism is very general and expressive and offers great flexibility in modeling protocols and their properties. We want to preserve this generality as much as possible.
2. For the transformation to I/O specifications, we require a separate description of each protocol role and of the environment, with a clear interface between the two parts. This interface will be mapped to I/O permissions in the I/O specification and eventually to (e.g., I/O or cryptographic) library calls in the implementation.
3. The protocol models operate on abstract terms, whereas the implementation manipulates byte strings. We need to bridge this gap in a sound manner.

Our solution is based on a general encoding of the MSR semantics into I/O specifications. To separate the different roles' rewrite rules from each other and from the environment, we partition the fact symbols and rewrite rules accordingly. The interface between the roles and the environment is defined by identifying I/O fact symbols, for which we introduce separate I/O rules. This isolates the I/O operations from others and allows us to map them to I/O permissions and later to library functions. Moreover, we keep I/O specifications as abstract as possible by using message terms rather than byte strings. We handle the transition to byte strings in the code verification process (Sec. 2.4).

The proofs for the results stated in this section can be found in the extended version [81] of the publication on which this chapter is based.

[81]: Arquint et al. (2022), *Sound Verification of Security Protocols: From Design to Interoperable Implementations (extended version)*

2.3.1 Protocol Format

We introduce a few mild formatting assumptions on the Tamarin model. They mostly correspond to common modeling practice and serve to cleanly separate the different protocol roles and the environment. They do not restrict Tamarin's expressiveness for modeling protocols. In particular, all protocols in the Tamarin distribution would fit our assumptions after some minor modifications.

Rule Format

To model an n-role protocol, we will use a fact signature of the form

$$\Sigma_{\mathsf{facts}} = \Sigma_{\mathsf{act}} \uplus \Sigma_{\mathsf{env}} \uplus \Big(\biguplus_{1 \le i \le n} \Sigma^{i}_{\mathsf{state}} \Big),$$

where (i) Σ_{act}, Σ_{env}, and $\Sigma^{i}_{\mathsf{state}}$ are mutually disjoint sets of fact symbols, used to construct action facts (used in transition labels), environment facts, and each role i's state facts; (ii) Σ_{env} contains two disjoint subsets, Σ_{in} and Σ_{out}, of input and output fact symbols such that $\mathsf{Fr}, \mathsf{in} \in \Sigma_{\mathsf{in}}$, $\mathsf{out} \in \Sigma_{\mathsf{out}}$, and $\mathsf{K} \in \Sigma_{\mathsf{env}} \setminus (\Sigma_{\mathsf{in}} \cup \Sigma_{\mathsf{out}})$; and (iii) there is an initialization fact symbol $\mathsf{Setup}_i \in \Sigma_{\mathsf{in}}$ for each protocol role i.

We consider MSR systems $\mathcal{R}$ whose rules are as follows:

$$\mathcal{R} = \mathcal{R}_{\mathsf{env}} \uplus \Big(\biguplus_{1 \le i \le n} \mathcal{R}_i \Big).$$

We require that the rules' labels contain only facts from Σ_{act}, i.e., for all $\ell \xrightarrow{a} r \in \mathcal{R}$, $\mathrm{facts}(a) \subseteq \Sigma_{\mathsf{act}}$. Here, $\mathrm{facts}(s)$ denotes the set of fact symbols that occur in a multiset of facts s. Additionally, $\mathcal{R}_{\mathsf{env}}$ and the $\mathcal{R}_i$s are pairwise disjoint rule sets containing rules for the environment and each protocol role. Protocol rules use input and output facts to communicate with the environment. For example, the following two environment rules transfer a message to and from the attacker's knowledge:

$$[\mathsf{out}(x)] \xrightarrow{[]} [\mathsf{K}(x)] \qquad [\mathsf{K}(x)] \xrightarrow{[]} [\mathsf{in}(x)]. \tag{2.2}$$

The attacker rules MD_Σ, the freshness rule, and the rules that generate the Setup_i facts (cf. Example 2.2.2), are also in $\mathcal{R}_{\mathsf{env}}$. We assume that environment rules do not directly use the agents' internal states. Namely, for all $\ell \xrightarrow{a} r \in \mathcal{R}_{\mathsf{env}}$, $\mathrm{facts}(\ell \cup r) \subseteq \Sigma_{\mathsf{env}}$. In addition, any rule in $\mathcal{R}_{\mathsf{env}}$ producing a Setup_i fact must not produce any other facts on its right-hand side and its label must be empty. The protocol rules for role i (and only these) use role state facts from $\Sigma^i_{\mathsf{state}}$ to keep track of the role's progress and may consume facts in Σ_{in} (but must not produce them) and may produce facts in Σ_{out} (but must not consume them). More formally, we require, for all $\ell \xrightarrow{a} r \in \mathcal{R}_i$, $\mathrm{facts}(\ell) \subseteq \Sigma^i_{\mathsf{state}} \cup \Sigma_{\mathsf{in}}$ and $\mathrm{facts}(r) \subseteq \Sigma^i_{\mathsf{state}} \cup \Sigma_{\mathsf{out}}$. Finally, we require that for a protocol rule $\ell \xrightarrow{a} r \in \mathcal{R}_i$, at least one state fact appears in r, and that there is a $k_i \ge 1$ such that the tuple of the first k_i arguments of all state facts in $\ell \xrightarrow{a} r$ is the same. Intuitively, these first k_i arguments represent parameters of the run of the protocol role: their value remains fixed throughout the role's execution. They can be, for instance, the agent's identity, a thread identifier, or any value that is assumed to be known beforehand by the agent. We assume that the first one of these arguments is the thread identifier *rid*, which is of type *fresh*. For readability, we will usually group these k_i initial parameters as a tuple, denoted by $\overline{init}$.

These formatting rules impose only very mild constraints on Tamarin models. All protocol models in the Tamarin distribution could easily be adapted to conform to these constraints with only minor modifications. The main changes would be related to providing a separate setup rule for each role i and keeping the arguments of the resulting Setup_i fact as the initial arguments of all state facts as described above.

Protocol Messages

We support both the usual ways of checking received messages using pattern matching and explicit equality checks. The latter are formalized, as usual in Tamarin, as a combination of action facts labeling the given rule (e.g., $\mathsf{Eq}(x, \mathsf{hash}(z))$) and restrictions associating these facts with (a boolean combination of) equalities (e.g., $x =_{\mathsf{E}} y$ whenever $\mathsf{Eq}(x, y)$ occurs in a trace). These restrictions act as assumptions on the traces considered by Tamarin.[1] The I/O specifications resulting from our procedure require that these equality checks are implemented (Sec. 2.3.3).

Furthermore, we recommend, but do not require, the replacement of nested pairs and tuples by *formats* [82]. These are user-defined function

1: To handle these, we use a modified, but equivalent MSR semantics, where the equalities are checked at each step rather than globally on traces. This semantics allows us to translate these checks into the I/O specifications and thus enforce their correct implementation. For simplicity, we present our results under the standard semantics and refer to [81] for more details.

[82]: Mödersheim et al. (2014), *A Sound Abstraction of the Parsing Problem*

symbols, along with projections for all arguments, that behave like tuples. In the implementation, each format is mapped to a combination of tags (i.e., constant byte strings), fixed-size fields, and variable-sized fields prepended with a length field. Formats help to soundly relate term and byte string messages, if we prove that they are unambiguous and non-overlapping (see Sec. 2.4).

Example 2.3.1 (Diffie–Hellman Formatting) The rules for Alice's role from Example 2.2.2 satisfy the format conditions above. The initiator setup rule produces a fact $\mathsf{Setup}_{Alice}(\overline{init})$, whose parameters $\overline{init}$ appear as the first parameters of the state facts $Step^1_{Alice}(\overline{init}, \ldots)$ and $Step^2_{Alice}(\overline{init}, \ldots)$. Both protocol rules produce an out fact to send a message. The second rule also consumes an in fact to receive a message. To follow our recommendation to use formats, we can model the message $\langle 0, B, A, g^x, Y\rangle$ being signed as a format with five fields, rather than a tuple. Note that, at the Tamarin level, tuples containing unique tags to distinguish them behave essentially the same as formats.

2.3.2 Transformation to Component Models

We decompose an MSR system $\mathcal{R}$ that satisfies our format requirements into several component models, one for each role, and a separate environment model, which includes the attacker. In doing so, we move from a global view of the protocol, useful for security analysis, to a local view of each role, more appropriate for the implementation. In Sec. 2.3.3, we transform the component models into I/O specifications for the programs implementing them.

As a preparatory step, we refine $\mathcal{R}$ into an *interface model* which starts decoupling the roles from the environment by introducing separate rewrite rules for their interactions.

Interface Model

The protocol roles and the environment interact using input and output facts, including the built-in facts Fr, in, and out. For example, the protocol roles receive messages by consuming in facts produced by the attacker. The interface model adds an *I/O rule* for each such fact, which turns it into a buffered version. These I/O rules will later be implemented as calls to library functions.

Let Σ^-_{in} be the set Σ_{in} without the initialization facts Setup_i. We first add to the fact signature, for each non-setup input or output fact F and role i, a copy (the "buffer") F_i:

$$\Sigma^i_{\mathsf{buf}} = \{F_i \mid F \in \Sigma^-_{\mathsf{in}} \cup \Sigma_{\mathsf{out}}\} \qquad \Sigma^i_{\mathsf{role}} = \Sigma^i_{\mathsf{state}} \cup \Sigma^i_{\mathsf{buf}}$$
$$\Sigma'_{\mathsf{facts}} = \Sigma_{\mathsf{act}} \uplus \Sigma_{\mathsf{env}} \uplus (\biguplus_i \Sigma^i_{\mathsf{role}}).$$

We then replace the facts used by the protocol rules as follows. For each role i, let $\mathcal{R}'_i$ be the set of rules obtained by replacing, in all rules in $\mathcal{R}_i$, each fact $F(t_1, \ldots, t_k)$ such that $F \in \Sigma^-_{\mathsf{in}} \cup \Sigma_{\mathsf{out}}$ by $F_i(rid, t_1, \ldots, t_k)$. The latter fact has rid as an additional parameter.

We also introduce the set $\mathcal{R}_{\mathsf{io}}$ of *I/O rules*, which translate between input or output facts and their buffered versions.
The set $\mathcal{R}_{\mathsf{io}}$ contains the following rules, for each role i:

$$\begin{aligned} [F(x_1,\dots,x_k)] &\xrightarrow{[]} [F_i(rid,x_1,\dots,x_k)] && \text{for } F \in \Sigma_{\mathsf{in}}^{-} \\ [G_i(rid,x_1,\dots,x_k)] &\xrightarrow{[]} [G(x_1,\dots,x_k)] && \text{for } G \in \Sigma_{\mathsf{out}}. \end{aligned}$$

For reasons that will become clear later, we also count the role setup rules as I/O rules. Hence, we move them from $\mathcal{R}_{\mathsf{env}}$ to $\mathcal{R}_{\mathsf{io}}$, calling the remaining environment rules $\mathcal{R}_{\mathsf{env}}^{-}$.

Finally, the interface model is specified by:

$$\mathcal{R}_{\mathsf{intf}} = \mathcal{R}_{\mathsf{env}}^{-} \uplus \mathcal{R}_{\mathsf{io}} \uplus (\biguplus_i \mathcal{R}_i'). \tag{2.3}$$

Example 2.3.2 Continuing Example 2.3.1, we introduce the buffer facts in_{Alice}, out_{Alice}, and Fr_{Alice}. Recall that *rid* is included in $\overline{init}$. This yields the modified set $\mathcal{R}'_{Alice}$ for the role *Alice*:

$$\begin{aligned} &[\mathsf{Setup}_{Alice}(\overline{init}), \mathsf{Fr}_{Alice}(rid,x)] \xrightarrow{[]} \\ &\qquad [Step^1_{Alice}(\overline{init},x), \mathsf{out}_{Alice}(rid,g^x)] \\ &[Step^1_{Alice}(\overline{init},x), \mathsf{in}_{Alice}(rid, sign(\langle 0,B,A,g^x,Y\rangle,k_B))] \xrightarrow{[Secret(Y^x)]} \\ &\quad [Step^2_{Alice}(\overline{init},x,Y), \mathsf{out}_{Alice}(rid, sign(\langle 1,A,B,Y,g^x\rangle,k_A))]. \end{aligned}$$

We show that the interface model refines the original one.

Lemma 2.3.1 $\mathcal{R}_{\mathsf{intf}} \preccurlyeq' \mathcal{R}$.

Decomposition

We are now ready to decompose the interface model into the role components and the environment. In a nutshell, we assign the rules $\mathcal{R}_i'$ to the component for role i and the rules $\mathcal{R}_{\mathsf{env}}^{-}$, including the attacker rules MD_Σ, to the environment. The protocol communicates with the environment using the I/O rules. We split them into two synchronized parts, one belonging to the environment and the other to the protocol. Below, we will show that the re-composed system implements the interface model.

We explain the splitting of the I/O rules into a protocol part and an environment part using the example rule

$$[\mathsf{out}_i(rid,x)] \xrightarrow{[]} [\mathsf{out}(x)].$$

This rule models instance *rid* of role i outputting a message to the attacker. We split this rule into two parts:

$$[\mathsf{out}_i(rid,x)] \xrightarrow{[\lambda_{\mathsf{out}}(rid,x)]} [], \tag{2.4}$$

$$[] \xrightarrow{[\lambda_{\mathsf{out}}(rid,x)]} [\mathsf{out}(x)], \tag{2.5}$$

where the first rule belongs to role i and the second to the environment. We label both rules with a new action fact $\lambda_{\mathsf{out}}(rid,x)$, which uniquely identifies the original I/O rule and has as parameters all variables occurring in it. We call this fact a *synchronization label*, as we later use it for synchronizing the two parts to recover the original rule's behavior. Similarly, we split all rules in $\mathcal{R}_{\mathsf{io}}$, yielding two sets $\mathcal{R}^i_{\mathsf{io}}$ and $\mathcal{R}^e_{\mathsf{io}}$ belonging to the protocol role i and to the environment.

The components for each protocol role i and for the environment are then defined as follows.

$$\mathcal{R}^i_{\mathsf{role}} = \mathcal{R}'_i \uplus \mathcal{R}^i_{\mathsf{io}} \qquad\qquad \mathcal{R}^e_{\mathsf{env}} = \mathcal{R}^-_{\mathsf{env}} \uplus \mathcal{R}^e_{\mathsf{io}}. \tag{2.6}$$

Note that the rule sets $\mathcal{R}^i_{\mathsf{role}}$ and $\mathcal{R}^e_{\mathsf{env}}$ operate on pairwise disjoint sets of facts, namely over the signatures Σ^i_{role} and Σ_{env}, respectively. This means that they can interact with each other only by synchronizing the split I/O rules.

Example 2.3.3 (Component for Diffie–Hellman) The MSR system $\mathcal{R}^{Alice}_{\mathsf{role}}$ for Alice's role contains the two protocol rules in $\mathcal{R}'_{Alice}$ from Example 2.3.2, the output rule (2.4) described above, and similar rules for inputs and freshness generation. In addition, it contains the protocol part of the split setup rule for Alice's role from Example 2.2.2, i.e.,

$$[] \xrightarrow{[\lambda_{Alice}(rid,A,k_A,B,pk_B)]} [\mathsf{Setup}_{Alice}(rid, A, k_A, B, pk_B)].$$

The traces of the recomposition of all roles with the environment are included in the traces of the interface model. We define two kinds of parallel compositions on LTSs (induced here by the MSR systems' transition semantics). We provide their intuition first and present the formal definitions as additional information before showing the trace inclusion. The (indexed) parallel composition $|||$ interleaves the transitions of a family of component systems without communication. The (binary) parallel composition $||_\Lambda$ synchronizes transitions with labels from the set Λ, resulting in a transition labeled $[]$, and interleaves all other transitions.

We define the (indexed) interleaving parallel composition $|||$ and the (binary) synchronizing parallel composition $||_\Lambda$. These compose their argument MSR systems into an LTS.

The (indexed) interleaving parallel composition $|||_{i,rid}\mathcal{R}_i(rid)$ has as states functions f that map each pair (i, rid) to a multiset of state facts and transitions $f \xrightarrow{a} f'$ if, for some i and rid, $f(i, rid) \overset{a}{\Longrightarrow}_{\mathcal{R}_i(rid)} S'$ and $f' = f[(i, rid) \mapsto S']$, where f' agrees with f except that it maps (i, rid) to S'.

The synchronized composed system $\mathcal{R}_1 \,||_\Lambda\, \mathcal{R}_2$ has states of the form (S_1, S_2) and transitions $(S_1, S_2) \xrightarrow{a} (S'_1, S'_2)$ if either

(i) $a = []$ and there is an $a' \in_{\mathsf{E}} \Lambda$ such that $S_1 \overset{a'}{\Longrightarrow}_{\mathcal{R}_1} S'_1$ and $S_2 \overset{a'}{\Longrightarrow}_{\mathcal{R}_2} S'_2$,

(ii) $a \notin_{\mathsf{E}} \Lambda$, $S_1 \overset{a}{\Longrightarrow}_{\mathcal{R}_1} S'_1$ and $S'_2 = S_2$, or

(iii) $a \notin_{\mathsf{E}} \Lambda$, $S_2 \overset{a}{\Longrightarrow}_{\mathcal{R}_2} S'_2$ and $S'_1 = S_1$.

Here, $a' \in_{\mathsf{E}} \Lambda$ means that $a' =_{\mathsf{E}} a$ for some $a \in \Lambda$.

Lemma 2.3.2 (Decomposition) *Let $\mathcal{R}^i_{\mathsf{role}}(rid)$ be the* MSR *system $\mathcal{R}^i_{\mathsf{role}}$ for a fixed thread id rid. Then*

$$(|||_{i,rid}\mathcal{R}^i_{\mathsf{role}}(rid)) \parallel_\Lambda \mathcal{R}^e_{\mathsf{env}} \preccurlyeq \mathcal{R}_{\mathsf{intf}},$$

where $\Lambda = \bigcup_i \{a\theta \mid \exists \ell, r.\, \ell \xrightarrow{a} r \in \mathcal{R}^i_{\mathsf{io}} \wedge range(\theta) \subseteq \mathcal{M}\}$ consists of all ground instances of synchronization labels.

2.3.3 Transformation to I/O Specifications

Finally, we extract an I/O specification ψ_i from each role i's MSR system $\mathcal{R}^i_{\mathsf{role}}$, which serves as the specification for the role's implementation at the code level. ψ_i is parameterized by the thread identifier rid, and associates a token with the starting place p of the predicate P_i:

$$\psi_i(rid) = \exists p.\, token(p) \star P_i(p, rid, []).$$

The predicate $P_i(p, rid, S)$'s parameters are: a place p, a thread identifier rid, and a state S of the MSR system $\mathcal{R}^i_{\mathsf{role}}$ (i.e., a multiset of ground facts). Note that ψ_i invokes P_i with the initial state, i.e., the empty multiset [] (see Sec. 2.2.1). It is defined co-recursively as the separating conjunction over the formulas ϕ_R, one for each rewrite rule $R \in \mathcal{R}^i_{\mathsf{role}}$:

$$P_i(p, rid, S) = \bigstar_{R \in \mathcal{R}^i_{\mathsf{role}}} \phi_R(p, rid, S).$$

ϕ_R encodes an application of the rewrite rule R to the model state S. It contains an I/O or internal permission $\mathbf{R}(p, \ldots, p')$, which an implementation must hold in order to execute the program part implementing R. ϕ_R co-recursively calls $P_i(p', rid, S')$ with the target place p' of $\mathbf{R}$ and the updated state S'. We define the formulas ϕ_R separately for protocol rules in $\mathcal{R}'_i$ and for I/O rules in $\mathcal{R}^i_{\mathsf{io}}$.

Consider a protocol rule $R = \ell \xrightarrow{a} r \in \mathcal{R}'_i$ with variables $\overline{x}$. We associate the internal permission $\mathbf{R}(p, \overline{x}, \ell', a', r', p')$ to R, and define $\phi_R(p, rid, S)$ by

$$\begin{aligned}
&\phi_R(p, rid, S) = \forall \overline{x}, \ell', a', r'.\\
&\quad M(\ell', s) \wedge \ell' =_{\mathsf{E}} \ell \wedge a' =_{\mathsf{E}} a \wedge r' =_{\mathsf{E}} r \wedge \Phi_R(\overline{x})\\
&\quad \Longrightarrow \exists p'.\, \mathbf{R}(p, \overline{x}, \ell', a', r', p') \star P_i(p', rid, U(\ell', r', S))
\end{aligned}$$

where $M(\ell', S) = (\ell' \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}} \subseteq^{\mathsf{m}} S) \wedge (\mathsf{set}(\ell') \cap \mathcal{F}_{\mathsf{per}} \subseteq \mathsf{set}(S))$, $U(\ell', r', S) = S \setminus^{\mathsf{m}} (\ell' \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}}) \cup^{\mathsf{m}} r'$, and Φ_R is the conjunction of all (boolean combinations of) equality checks (mod E) that the rule R performs using a combination of action facts in a and associated restrictions (cf. Sec. 2.3.1).

This formula specifies that, for any instantiation (mod E) ℓ', a', r' of the facts in the rule, if the matching condition $M(\ell', S)$ and the equational formula Φ_R are satisfied, we have an internal permission $\mathbf{R}$ to execute the rule's implementation. This yields an updated state $U(\ell', r', S)$, on which P_i is co-recursively applied to produce the permissions for the rest of the execution. The formula Φ_R thus enforces that the implementation performs the explicit equality checks on messages specified in the rule R (see also Sec. 2.4.3).

We define similar formulas for all I/O rules. For output rules of the form $R_G = [G_i(rid, \overline{x})] \xrightarrow{[\lambda_G(rid,\overline{x})]} [] \in \mathcal{R}^i_{\mathsf{io}}$, we define the formula

$$\begin{aligned}&\phi_{R_G}(p, rid, S) = \forall \overline{x}.\ G_i(rid, \overline{x}) \in^{\mathsf{m}} S \\ &\quad \Longrightarrow \exists p'.\ \mathbf{R_G}(p, rid, \overline{x}, p') \star P_i(p', rid, S \setminus^{\mathsf{m}} [G_i(rid, \overline{x})]))\end{aligned}$$

which grants the I/O permission $\mathbf{R_G}(p, rid, \overline{x}, p')$ for any rid and terms $\overline{x}$ for which fact $G_i(rid, \overline{x})$ exists in S. P_i is called co-recursively with the target place of the permission and the updated state, where the fact $G_i(rid, \overline{x})$ is removed.

For input rules $R_F = [] \xrightarrow{[\lambda_F(rid,\overline{z})]} [F_i(rid, \overline{z})] \in \mathcal{R}^i_{\mathsf{io}}$, we define the formula

$$\begin{aligned}&\phi_{R_F}(p, rid, S) = \\ &\quad \exists p', \overline{z}.\ \mathbf{R_F}(p, rid, \overline{z}, p') \star P_i(p', rid, S \cup^{\mathsf{m}} [F_i(rid, \overline{z})]),\end{aligned}$$

which grants the I/O permission $\mathbf{R_F}(p, rid, \overline{z}, p')$ to read inputs $\overline{z}$ for any rid. Note that the input variables $\overline{z}$ are existentially quantified (cf. Sec. 2.2.2) and the fact $F_i(rid, \overline{z})$ is added to the state in the co-recursive call to P_i.

Example 2.3.4 (I/O Specification for Diffie–Hellman) Continuing Examples 2.3.2 and 2.3.3, the component system is translated into an I/O specification that features the following conjunct corresponding to the rule for the second step of Alice's role:

$$\begin{aligned}&\phi_{Alice_2}(p, rid, S) = \forall \overline{init}, x, Y, \ell', \mathscr{a}', \mathscr{r}'. \\ &\quad M(\ell', S) \wedge \\ &\quad \ell' =_{\mathsf{E}} \{\!\!\{ Step^1_{Alice}(\overline{init}, x), \mathsf{in}_{Alice}(rid, sign(\langle 0, B, A, g^x, Y\rangle, k_B)) \}\!\!\} \wedge \\ &\quad \mathscr{a}' =_{\mathsf{E}} \{\!\!\{ Secret(Y^x) \}\!\!\} \wedge \\ &\quad \mathscr{r}' =_{\mathsf{E}} \{\!\!\{ Step^2_{Alice}(\overline{init}, x, Y), \mathsf{out}_{Alice}(rid, sign(\langle 1, A, B, Y, g^x\rangle, k_A)) \}\!\!\} \wedge \\ &\quad \Longrightarrow \exists p'.\ \mathbf{Alice_2}(p, \overline{init}, x, Y, \ell', \mathscr{a}', \mathscr{r}', p') \star P_i(p', rid, U(\ell', \mathscr{r}', S))\end{aligned}$$

i.e., the permission to execute this step is granted, provided S contains instantiations of the previous state fact and the correct input fact, and that S is updated by replacing them with the new state and output facts. Recall that $\overline{init}$ abbreviates rid, A, k_A, B, pk_B.

The construction of ψ_i from $\mathcal{R}^i_{\mathsf{role}}$ can be seen as an instance of Igloo [71] and by the soundness result from that paper, we get the following trace inclusion.

[71]: Sprenger et al. (2020), *Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification*

Theorem 2.3.3 (I/O Soundness) *Sprenger et al. [71] prove*

$$\pi(\psi_i(rid)) \preccurlyeq \mathcal{R}^i_{\mathsf{role}}(rid),$$

for all MSR *systems $\mathcal{R}^i_{\mathsf{role}}(rid)$, where the fresh name rid instantiates the thread identifier in all facts, and $\pi = \pi_{int} \circ \pi_{ext}$ relabels (the* LTS *induced by) $\psi_i(rid)$. Here, π_{int} and π_{ext} are the identity functions except on the following labels:*

$$\begin{aligned}\pi_{int}(\mathbf{R}(\overline{x}, \ell', \mathscr{a}', \mathscr{r}')) &= \mathscr{a}' && \text{for } R \in \mathcal{R}'_i \\ \pi_{ext}(\mathbf{F}(rid, \overline{x})) &= [\lambda_F(rid, \overline{x})] && \text{for } F \in \mathcal{R}^i_{\mathsf{io}}.\end{aligned}$$

2.3.4 I/O Specification Generation

We implement the described transformation to fully automatically generate I/O specifications from a Tamarin protocol model. Our tool takes a Tamarin model as input and emits an I/O specification per protocol role that satisfies the mild formatting assumptions. While the tool transforms each rule in a role's MSR system independently and, thus, is generic, i.e., independent of a protocol and its properties, the emitted I/O specifications are specific to a program verifier. Therefore, the tool features two main configuration options besides the path to the input Tamarin model and the output directory. The first option selects the targeted program verifier such that the emitted I/O specifications conform to the verifier's syntax and features. Currently, the tool supports Gobra for Go and VeriFast for Java. However, additional program verifiers and programming languages can easily be supported by implementing additional pretty-printers from the internal representation of I/O specifications to the desired output format. Since the tool emits multiple modules in different files, which define internal I/O operations and terms as well as the actual I/O specifications, the second option customizes the module name prefix such that the generated import statements work as expected after embedding the emitted files in the targeted codebase. Except for embedding the emitted files, a user has to add only constructors for public constants and fresh terms in the emitted files, equip I/O operations with a specification that enforces the respective I/O permission (cf. Sec. 2.2.2), and prove that an implementation satisfies a particular role's I/O specification, as explained in the following subsection.

[83]: Arquint et al. (2022), *Sound Verification of Security Protocols: From Design to Interoperable Implementations*
[84]: Arquint et al. (2022), *Sound Verification of Security Protocols: From Design to Interoperable Implementations*

Our tool is open-source [83, 84]. The transformation is implemented in Isabelle/HOL amounting to 1723 lines of code (LOC), and used to generate Haskell code, which is integrated into a fork of Tamarin to reuse the parsing of Tamarin models. Ignoring the generated Haskell code, the tool adds 2880 LOC to the Tamarin codebase, most of which correspond to the two pretty-printers for Gobra and VeriFast.

2.4 Verified Protocol Implementations

The implementation step consists of providing the code $c_i(rid)$ implementing each role i and proving that it satisfies its I/O specification $\psi_i(rid)$. The challenge here is bridging the abstraction gap between the message terms in the I/O specifications $\psi_i(rid)$ and the byte string messages manipulated by the code. In Sec. 2.4.1, we present an extension of the code verifiers' semantics of Hoare triples to accommodate such abstractions. In Sec. 2.4.2, we explain how we concretely relate byte strings to terms. Finally, in Sec. 2.4.3, we show how we verify the roles' I/O specifications based on appropriate I/O and crypto library specifications.

2.4.1 Code Verification with Abstraction

[80]: Penninckx et al. (2015), *Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs*

In Penninckx *et al.*'s program logic [80], the statement that a program c satisfies an I/O specification ϕ is expressed as the Hoare triple

$$\{\phi\}\ c\ \{\mathsf{true}\}, \tag{2.7}$$

with the I/O specification ϕ in the precondition and the postcondition true. We assume that the program c has an LTS semantics $\mathscr{C}$ given by the programming language's operational semantics, where the labels represent the program's I/O (and internal) operations and the program's traces consist of sequences of such labels. We leave the exact semantics unspecified here, to keep our formulation generic with respect to the programming language used. The semantics of the Hoare triple (2.7) implies that the program c's traces are included in the traces of ϕ, i.e., $\mathscr{C} \preccurlyeq \phi$.

To bridge the gap between message terms and byte string messages, we extend Penninckx *et al.*'s approach by introducing an *abstraction* or relabeling function α between the implementation's transition labels and the I/O specification's transition labels. For example, α may map a concrete label $\mathbf{in}_c(l)$ to an abstract version $\mathbf{in}_a(s)$, where l is a list implementation and s is the mathematical set of l's elements. We also extend the soundness assumption on the code verifier accordingly.

Assumption 2.4.1 (Verifier Assumption)

$$\vdash_\alpha \{\phi\}\ c\ \{\mathsf{true}\} \implies \alpha(\mathscr{C}) \preccurlyeq \phi.$$

This means that a successful verification implies that the program traces, abstracted under α, are included in the I/O specification ϕ's traces. Next, we sketch a semantics for Hoare triples that implies this trace inclusion.

Extended Semantics for Hoare Triples

Following Penninckx *et al.* [80], we sketch an example of semantic assumptions on programs and Hoare triples that make our verifier assumption (Asm. 2.4.1) hold semantically. The soundness of the program logic itself, i.e., that a provable Hoare triple $\vdash_\alpha \{\phi\}\ c\ \{\psi\}$ implies its semantic validity $\models_\alpha \{\phi\}\ c\ \{\psi\}$, is orthogonal and can be established using standard techniques from the literature.

[80]: Penninckx et al. (2015), *Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs*

We assume that a programming language's semantics makes judgments of the form $s, c \Downarrow s', \tau$, meaning that the program c when started in state s terminates in state s' and produces the I/O trace τ. This semantics induces a LTS $\mathscr{C}$, whose set of traces for a given starting state s_0 is thus

$$\mathrm{Tr}(\mathscr{C}) = \{\tau \mid \exists s'.\, s_0, c \Downarrow s', \tau\}.$$

I/O specifications ϕ have both a static and a dynamic semantics, which are defined in terms of *(I/O) heaps*. Heaps are multisets of (ground) I/O permission and token predicates. The static semantics, written $h \models \phi$, intuitively means that a heap h contains (at least) the I/O permissions and tokens prescribed by ϕ. The dynamic semantics defines the set of traces allowed by an I/O specification ϕ to contain those traces that are possible in all heap models of ϕ, i.e.,

$$\mathrm{Tr}(\phi) = \{\tau \mid \forall h.\ h \models \phi \implies h \xrightarrow{\tau}\},$$

where $h \xrightarrow{\tau}$ intuitively means that it is possible to produce a trace τ by successively pushing the tokens in h through the I/O permissions in h (and thus consume these permissions).

The semantics of Hoare triples of the form $\{\phi\}\ c\ \{\mathsf{true}\}$ with respect to an abstraction function α from program-level I/O operations to abstract I/O permissions is given by

$$\begin{aligned}
&\models_\alpha \{\phi\}\ c\ \{\mathsf{true}\} \\
&\overset{\text{def}}{\iff} \forall s,\tau,s',h.\ s,c \Downarrow s',\tau \wedge h \models \phi \Longrightarrow h \xrightarrow{\alpha(\tau)} \\
&\iff \alpha(\mathrm{Tr}(\mathscr{C})) \subseteq \mathrm{Tr}(\phi) \\
&\iff \alpha(\mathscr{C}) \preccurlyeq \phi.
\end{aligned}$$

Here, $\alpha(\mathscr{C})$ denotes the LTS $\mathscr{C}$ whose transition labels are renamed under α. The final equivalence uses the equality $\alpha(\mathrm{Tr}(\mathscr{C})) = \mathrm{Tr}(\alpha(\mathscr{C}))$.

Penninckx *et al.*'s semantics is formulated for the case where α is the identity function. Both our and their semantics of Hoare triples also include non-trivial postconditions, which we omit here to simplify the presentation.

To ensure that our extension using the abstraction function α is sound, we require that the I/O operations' contracts are *consistent with* α, i.e., imply a correct mapping of transition labels under α. More precisely, suppose the specification of such an operation `op` induces a concrete transition label $\mathbf{op}_c(\overline{a})$, where $\overline{a}$ are `op`'s inputs and outputs, and the I/O permission in the precondition induces the abstract transition label $\mathbf{op}_a(\overline{b})$. Then we define α as lifting from an (overloaded) function α that maps concrete parameter types to abstract ones, i.e., $\alpha(\mathbf{op}_c(\overline{a})) = \mathbf{op}_a(\alpha(\overline{a}))$. We therefore require that $\overline{b} = \alpha(\overline{a})$ follows from `op`'s precondition (for arguments) and postcondition (for return values). Moreover, we allow α to be a partial function, in which case the specification must also imply that the concrete arguments are in its domain.

Application to Role Verification

We now apply this idea to the verification of the protocol's role implementations (using Gobra and VeriFast in our case studies). That is, we wish to establish

$$\vdash_\alpha \{\psi_i(rid)\}\ c_i(rid)\ \{\mathsf{true}\} \tag{2.8}$$

for a suitable α. An obvious possibility would be to define an abstraction function $\alpha : \mathbb{B}^\star \longrightarrow \mathcal{M}$ from byte strings to messages and then lift it to trace labels. For example, a concrete $\mathbf{in}_c(rid, b)$ would be abstracted to $\alpha(\mathbf{in}_c(rid, b)) = \mathbf{in}_a(rid, \alpha(b))$. However, this mapping assumes that each byte string corresponds to exactly one term, and consequently that *every* byte string can be uniquely parsed as a term. To minimize our assumptions, however, we do not a priori want to exclude collisions between byte strings, i.e., we allow a byte string to have several term interpretations.

In Sec. 2.4.2, we therefore relate byte strings and terms using a concretization function $\gamma : \mathcal{M} \longrightarrow \mathbb{B}^\star$. Since a byte string may be related to several terms, we cannot define a function α mapping concrete labels to abstract I/O labels. Our solution is based on adding ghost term parameters to the I/O operations in the implementation code. For example, the operation receiving a byte string b gets an additional ghost return value term m with $b = \gamma(m)$ and the corresponding transition label is $\mathbf{in}_c(rid, (b, m))$. These ghost terms aid verification (see Sec. 2.4.3), but

```
ensures seq(ciph) = enc^B(seq(key), seq(msg))
func encrypt(key, msg []byte) (ciph []byte)

ensures ok ⟹ seq(c) = enc^B(seq(k), seq(m))
func decrypt(k, c []byte) (m []byte, ok bool)

requires token(?p₁) && in(p₁,?m,?p₂)
ensures  ok ⟹ token(p₂) && seq(b) = γ(m)
ensures !ok ⟹ token(p₁) && in(p₁,m,p₂)
func receive() (b []byte,ghost m term,ok bool)
```

Figure 2.2: Simplified specifications for encryption, decryption, and receive. The function `seq` abstracts an in-memory byte array to $\mathcal{B}$. We omit Gobra's memory annotations needed to reason about heap data structures and conditions on the size of byte strings.

are not present in the executable code. We instantiate α to the function π'_{ext} that removes the byte strings from the concrete I/O operation's labels and keeps only the ghost terms used for the reasoning. For instance, $\pi'_{ext}(\mathbf{in}_c(rid, (b, m))) = \mathbf{in}_a(rid, m)$. This function is defined only for $b = \gamma(m)$, which is guaranteed by the receive operation's contract (cf. Fig. 2.2).

Our proposed method enables us to verify that preexisting real-world code satisfies I/O specifications produced from abstract Tamarin models (see Sec. 2.6).

2.4.2 Relating Terms and Byte Strings

In Tamarin's MSR semantics, messages in $\mathcal{M}$ are ground terms. We model the concrete messages and the operations on them as *byte string algebras* defined as Σ-algebras $\mathcal{B}$ with the set of byte strings $\mathbb{B}^\star$ as the carrier set. To relate terms to byte strings, we use a surjective Σ-algebra homomorphism $\gamma : \mathcal{M} \to \mathcal{B}$, which maps (fresh and public) names to byte strings and the signature's symbols to functions on byte strings:

$$\begin{array}{rcll} \gamma(n) & = & n^{\mathcal{B}} & \text{for } n \in \mathcal{N} \\ \gamma(f(t_1, \ldots, t_k)) & = & f^{\mathcal{B}}(\gamma(t_1), \ldots, \gamma(t_k)) & \text{for } f \in \Sigma^k \end{array}$$

With the requirement that γ is surjective, we avoid junk byte strings that do not represent any term (i.e., the algebra $\mathcal{B}$ is term-generated). This is without loss of generality as there are countably infinitely many public names that can be mapped to potential junk byte strings.

Note that Σ-algebra homomorphisms are required to preserve equalities. For example, a symbolic equality $\mathsf{dec}(k, \mathsf{enc}(k, m)) =_{\mathsf{E}} m$ on terms implies the equality

$$\mathsf{dec}^{\mathcal{B}}(key, \mathsf{enc}^{\mathcal{B}}(key, msg)) = msg$$

on byte strings. In the next section, we will use the byte string algebra's functions in our crypto library's specification. This enables us to reason about message parsing and construction.

2.4.3 Verifying the I/O Specification

The verification of the I/O specification generally follows the same approach as in previous work [71, 80]. Every I/O operation performed by the code requires that a corresponding I/O permission is held. The required I/O permissions must be obtained from the I/O specification. However, our introduction of abstraction makes reasoning about what is sent and, in particular, received more challenging.

[71]: Sprenger et al. (2020), *Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification*
[80]: Penninckx et al. (2015), *Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs*

```
// seq(key) = γ(k) holds
ciph, c, ok := receive(); if !ok {return}
assert seq(ciph) = γ(c)
msg, ok := decrypt(key, ciph); if !ok {return}
assert ∃u. seq(msg) = γ(u)
        && seq(ciph) = γ(enc(k, u))
PaR1(m, ...) // using the pattern requirement
assert ∃w. c =E enc(k, w) && seq(msg) = γ(w)
```

Figure 2.3: Reasoning about receiving and parsing a ciphertext.

Sending and Receiving Messages

For a sent pair of a byte string and a ghost message (in $\mathcal{M}$), we must verify that the I/O specification permits sending the message. Similarly, for a received pair of a byte string and a ghost message, we must verify that the received message matches a term in the I/O specification, describing the expected protocol message. We refer to such terms as *patterns*. In Example 2.3.4, there is a single pattern, namely $sign(\langle 0, B, A, g^x, Y \rangle, k_B)$, where the unconstrained Y is a variable and all other entities are constrained by the fact $Step^1_{Alice}(\overline{init}, x)$.

Verifying that the I/O specification permits sending a message boils down to verifying that the byte string $\gamma(m)$ for a permitted message m was constructed and then sent. This becomes straightforward by equipping the cryptographic library with suitable specifications. Consider the simplified specification of an encryption function shown in Fig. 2.2. The function `seq` abstracts an in-memory byte array into a mathematical sequence of bytes, i.e., an element of $\mathbb{B}^\star$. Due to the specification and the surjectivity of γ, the result of `encrypt(key, msg)` is equal to $\gamma(\mathsf{enc}(m_{key}, m_{msg}))$ for some messages m_{key} and m_{msg}, where $\gamma(m_{key}) =$ `seq(key)` and $\gamma(m_{msg}) =$ `seq(msg)`. To verify the construction of an entire message, we combine the information of all such calls.

Verifying that a message m returned by `receive()` (cf. Fig. 2.2) matches a pattern t is more involved. Using our cryptographic library's specifications, we can verify that $\gamma(m)$ is equal to $\gamma(t\sigma)$, where the substitution σ instantiates the variables of t with messages. Unfortunately, this does not entail that the received message m matches the pattern t. The function γ may have collisions and hence $\gamma(m)$ may equal $\gamma(t\sigma)$, while m and $t\sigma$ differ. We address this issue by requiring that instances of the I/O specification's patterns do not collide with other byte strings; we discuss below how we justify this requirement.

Definition 2.4.1 *The* pattern requirement *for a pattern $t \in \mathcal{T}$ is defined for ground messages $m \in \mathcal{M}$ by*

$$\gamma(t\sigma) = \gamma(m) \implies \exists \sigma'.\, m =_{\mathsf{E}} t\sigma'. \tag{PaR(t)}$$

This requirement states that if (ground) messages m and $t\sigma$ coincide under γ, then m must match the pattern t (mod E) with some substitution σ', which may differ from σ.

We need the pattern requirement for all patterns of the I/O specification. For code verification, we express the pattern requirement as a ghost function whose pre- and postcondition are the left-hand and right-hand side of the pattern requirement, for each pattern respectively. To apply the pattern requirement, the corresponding ghost function is called in the code. We will explain how to prove the pattern requirement for a given pattern t after illustrating the pattern requirement on an example.

```
requires token(p) && $P_{Ann}$(p,r,S) && $Step^1_{Ann}$(k)∈ᵐS
requires ∃x. γ(enc(k,x)) = γ(m)
ensures  token(p) && $P_{Ann}$(p,r,S) && ∃x′. m =_E enc(k,x′)
ghost func PaR1(m,p,r,S,k)
```

Figure 2.4: Ghost function for the pattern requirement of Example 2.4.1. There is the single pattern enc(k,x), where k is a constant and x is a variable.

Example 2.4.1 (Checking a Ciphertext) Consider a simple protocol where a role Ann expects a message matching the pattern enc(k,x), where k is a pre-shared key. We use the fact $Step^1_{Ann}(k)$ to bind k in the model state. Fig. 2.3 shows part of an implementation. The variable `key` stores the pre-shared key, expressed as `seq(key)` = γ`(k)`. After successfully receiving a byte string `ciph` with a message `c`, `seq(ciph)` = γ`(c)` holds due to `receive`'s specification (cf. Fig. 2.2). Next, the code decrypts `ciph`. If successful, `ciph` equals the byte string γ`(enc(k,u))` for some message `u` with `seq(msg)` = γ`(u)` (lines 5–6) by `decrypt`'s postcondition (cf. Fig. 2.2) and γ being a surjective homomorphism. Furthermore, we know that γ`(enc(k,u))` equals γ`(c)`, but not yet that the received message `c` matches the pattern enc(k,x) (line 8). For this, we apply the pattern requirement by calling the ghost function `PaR1` (cf. Fig. 2.4). The constant k of the pattern enc(k,x) is passed as an argument to the call, and related to the state facts of the I/O specification via the ghost function's precondition (with $Step^1_{Ann}$`(k)∈ᵐS`).

Deriving the Pattern Requirement

The pattern requirement for a given pattern t can be derived from two more basic properties. We define these properties here and prove this implication. Afterwards, we will discuss assumptions and justifications regarding these properties.

The first property is *image disjointness*, which has two parts: First, the images of (public and fresh) names under γ are pairwise disjoint and disjoint from the image of any function $f^{\mathcal{B}}$ for $f \in \Sigma$ (cf. (NID)). Second, the image of any function $f^{\mathcal{B}}$ does not collide (modulo equational theory E) with the image of any other function $g^{\mathcal{B}}$, for $f, g \in \Sigma$ (cf. (ID$_f(t)$)). To define image disjointness, we split the equational theory E into two parts, namely the associativity and commutativity (AC) part and a user-defined, convergent rewriting system R. This split allows us to normalize a term t by sequentially applying R's rules to obtain a unique normal form modulo AC, which we call R,AC-*normalized*.

Definition 2.4.2 (R,AC-Normalized) *A pattern t is* R,AC-normalized *if this pattern has converged under* R *(mod* AC*),*

$$t = t\downarrow_{\mathsf{R},\mathsf{AC}},$$

where $t\downarrow_{\mathsf{R},\mathsf{AC}}$ denotes applying the rules of the rewriting system R *until convergence (mod* AC*).*

Definition 2.4.3 (AC-Subterm) *Following Cremers et al. [85, Definition 3], we define s being a subterm of t modulo* AC *as*

$$s \sqsubseteq_{\mathsf{AC}} t \triangleq \exists s', t'.\ (s' =_{\mathsf{AC}} s) \wedge (t' =_{\mathsf{AC}} t) \wedge (s' \sqsubseteq_{synt} t'),$$

[85]: Cremers et al. (2023), *Subterm-Based Proof Techniques for Improving the Automation and Scope of Security Protocol Analysis*

where $\sqsubseteq_{synt}$ *denotes the syntactic subterm relation. Analogously, we define the strict* AC*-subterm relation* $\sqsubset_{\mathsf{AC}}$.

Definition 2.4.4 (Image Disjointness) *Image disjointness for an* R,AC*-normalized pattern t holds if (i) γ is injective on the set of names $\mathcal{N}$* (NID)*, which includes that names and results of all functions $f \in \Sigma$ are disjoint under γ as m ranges over all ground messages, and (ii) for* AC*-subterms of t, all $f \in \Sigma$, and the image of function f coinciding after concretization with some term m, there exist terms $w_1, \ldots, w_k$ such that $f(w_1, \ldots, w_k)$ is equal to m modulo the equational theory* E (ID$_f(t)$)*.*

$$\forall n \in \mathcal{N}, m \in \mathcal{M}.\ \gamma(n) = \gamma(m) \implies n = m \tag{NID}$$

$$\begin{aligned} &f(u_1, \ldots, u_k) \sqsubseteq_{\mathsf{AC}} t \wedge \gamma(f(u_1\sigma, \ldots, u_k\sigma)) = \gamma(m) \\ &\implies \exists w_1, \ldots, w_k.\ m =_{\mathsf{E}} f(w_1, \ldots, w_k). \end{aligned} \tag{ID$_f(t)$}$$

The second property is *pattern injectivity* for a pattern t. This constitutes a much weaker form of standard injectivity. It is required to hold only for AC-subterms $t' \sqsubseteq_{\mathsf{AC}} t$ and where, again, equality is guaranteed only modulo a substitution σ'.

Definition 2.4.5 (Pattern Injectivity) *Pattern injectivity holds for an* R,AC*-normalized pattern t if, for all $f \in \Sigma$ occurring in t,*

$$\begin{aligned} &f(u_1, \ldots, u_k) \sqsubseteq_{\mathsf{AC}} t \wedge f^{\mathcal{B}}(\gamma(u_1\sigma), \ldots, \gamma(u_k\sigma)) = f^{\mathcal{B}}(b_1, \ldots, b_k) \\ &\implies \exists w_1, \ldots, w_k, \sigma'.\ f(u_1, \ldots, u_k) =_{\mathsf{AC}} f(w_1, \ldots, w_k) \wedge \\ &\qquad\qquad b_1 = \gamma(w_1\sigma') \wedge \ldots \wedge b_k = \gamma(w_k\sigma'). \end{aligned} \tag{PaI$_f(t)$}$$

Proposition 2.4.1 *Given an* R,AC*-normalized, linear (where every variable occurs only once) pattern t, image disjointness and pattern injectivity for t imply the pattern requirement for t.*

Proof. We prove the proposition's statement by well-founded induction on $\sqsubset_{\mathsf{AC}}$ using the induction hypothesis

$$\begin{aligned} &\forall t' \sqsubset_{\mathsf{AC}} t.\ \mathrm{PaI}_f(t') \wedge \mathrm{ID}_f(t') \\ &\quad \implies \left(\gamma(t'\sigma) = \gamma(m) \implies \exists \sigma'.\ m =_{\mathsf{E}} t'\sigma'\right). \end{aligned} \tag{IH}$$

We assume (NID), (ID$_f(t)$), (PaI$_f(t)$), and $\gamma(t\sigma) = \gamma(m)$ and show $\exists \sigma'.\ m =_{\mathsf{E}} t\sigma'$. We proceed by case distinction on t.

- $t = x \in \mathcal{V}$: Since x is a variable, we choose $\sigma' = [x \mapsto m]$. Hence, $m =_{\mathsf{E}} x[x \mapsto m]$.
- $t = n \in \mathcal{N}$: We apply (NID) and obtain $n = m$.
- Otherwise, we have

  $$t = f(u_1, \ldots, u_k) \tag{2.9}$$

  for some $u_1, \ldots, u_k$. By instantiating $\gamma(t\sigma) = \gamma(m)$ and γ being a homomorphism, we obtain $f^{\mathcal{B}}(\gamma(u_1\sigma), \ldots, \gamma(u_k\sigma)) = \gamma(m)$. Applying (ID$_f(t)$), we get

  $$m =_{\mathsf{E}} f(v_1, \ldots, v_k) \tag{2.10}$$

  for some $v_1, \ldots, v_k$. Under γ, we have $f^{\mathcal{B}}(\gamma(u_1\sigma), \ldots, \gamma(u_k\sigma)) =$

$f^{\mathcal{B}}(\gamma(v_1), \ldots, \gamma(v_k))$. Next, we define $b_1 = \gamma(v_1), \ldots, b_k = \gamma(v_k)$ and apply $(\mathrm{PaI}_f(t))$ to obtain

$$\exists w_1, \ldots, w_k, \sigma'.\ f(u_1, \ldots, u_k) =_{\mathsf{AC}} f(w_1, \ldots, w_k) \wedge \\ b_1 = \gamma(w_1\sigma') \wedge \ldots \wedge b_k = \gamma(w_k\sigma'). \qquad (2.11)$$

We note that $w_i \sqsubset_{\mathsf{AC}} t$ holds for $1 \leq i \leq k$. Therefore, we have $\mathrm{PaI}_f(w_i)$ and $\mathrm{ID}_f(w_i)$, which allow us to apply (IH) for each w_i resulting in $\exists \sigma_i . w_i\sigma_i =_{\mathsf{E}} v_i$. Since the pattern t is linear, all variables in w_i are disjoint from the variables in all other w_j with $i \neq j$. As a consequence, the domains of all σ_i are pairwise disjoint, which allows us to combine them into a single substitution σ', where $\sigma' = \bigcup_i \sigma_i$. As f is a function symbol and $w_i\sigma' =_{\mathsf{E}} v_i$ holds, we get

$$f(w_1\sigma', \ldots, w_k\sigma') =_{\mathsf{E}} f(v_1, \ldots, v_k). \qquad (2.12)$$

Finally, we obtain

$$m \overset{(2.10)}{=_{\mathsf{E}}} f(v_1, \ldots, v_k) \overset{(2.12)}{=_{\mathsf{E}}} f(w_1\sigma', \ldots, w_k\sigma') \\ \overset{(2.11)}{=_{\mathsf{AC}}} f(u_1\sigma', \ldots, u_k\sigma') \overset{(2.9)}{=} t\sigma',$$

which proves $m =_{\mathsf{E}} t\sigma'$.

□

We split non-linear patterns into multiple linear ones. For instance, the non-linear pattern $t = \langle x, \mathsf{hash}(x) \rangle$ can be split into $t_1 = \langle x, \_ \rangle$ and $t_2 = \langle \_, \mathsf{hash}(x) \rangle$ (where _ matches any term). Conceptually, we then first match a given term $\langle u, \mathsf{hash}(u) \rangle$ against t_1, which binds x to u, and then against $\langle \_, \mathsf{hash}(u) \rangle = t_2[x \mapsto u]$. This is equivalent to matching against t. This turned out to be simpler to work with than a single linearized pattern with additional equality constraints.

Requiring that patterns are R,AC-normalized is not a restriction in practice since patterns arising in a Tamarin model are usually normalized—otherwise, such patterns can easily be rewritten to be normalized.

Assumptions and Proof Obligations

We discuss assumptions and proof obligations regarding image disjointness and pattern injectivity. In doing so, we distinguish cryptographic operations from formats.

Since we are working in a symbolic (DY) model, which assumes perfect cryptography, we maintain this assumption for cryptographic operations at the byte string level in the following form.

Assumption 2.4.2 (Cryptographic Operations) *We assume that*

(i) γ *is injective on the set of names* $\mathcal{N}$*,*
(ii) $(\mathrm{ID}_f(t))$ *holds for cryptographic* $f \in \Sigma$ *and all* $g \in \Sigma$*, and*
(iii) $(\mathrm{PaI}_f(t))$ *holds for all protocol patterns* t *and all cryptographic* $f \in \Sigma$ *occurring in* t*.*

We justify these assumptions by noting that we can expect collisions violating these assumptions to occur only with negligible probability in good cryptographic libraries. Also recall that pattern injectivity is a much weaker requirement than standard injectivity.

The situation is different for formats (cf. Sec. 2.3.1). We can expect that the formats of a well-designed protocol are unambiguously parsable (i.e., injective and hence pattern-injective) and mutually disjoint (i.e., image disjoint). We therefore require that these properties are *proved* for formats, e.g., using the techniques proposed in [82, 86, 87].

[82]: Mödersheim et al. (2014), *A Sound Abstraction of the Parsing Problem*
[86]: Ramananandro et al. (2019), *EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats*
[87]: Wallez et al. (2023), *Comparse: Provably Secure Formats for Cryptographic Protocols*

Remark 2.4.1 An obvious way to achieve image disjointness and pattern injectivity is to *tag* each construct of the byte string algebra with a different byte string. This approach is followed, e.g., in [71] but this is unrealistic for real protocols.
Alternatively, image disjointness holds if different operations result in differently sized byte strings. For operations with varying output sizes, such as stream encryption, this may require restricting the allowed argument sizes in the implementation. In some cases, this approach may allow us to *prove* image disjointness even for some cryptographic operations. Indeed, we do this for a pre-existing implementation of the WireGuard protocol, which does not use tagging. However, this approach also has its limitations; for example, AES-256 or SHA-256 have the same output size.

[71]: Sprenger et al. (2020), *Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification*

Proving Term Equalities

Obligations to prove term equalities during code verification arise from equality constraints in the I/O specification. However, while the code can check that an equality holds on the byte string level (e.g., $\gamma(x) = \gamma(\mathsf{hash}(z))$), this does not in general imply the prescribed term equality (e.g., $x =_{\mathsf{E}} \mathsf{hash}(z)$). Following [64, 65, 69, 70], we can reasonably assume that collisions violating this implication do not occur (with overwhelming probability) in actual protocol executions and thus prove the specification under the condition that byte string equality implies term equality for the two concrete byte strings at hand (e.g., $\gamma(x) = \gamma(\mathsf{hash}(z)) \Longrightarrow x =_{\mathsf{E}} \mathsf{hash}(z)$). We call this a *collision-freedom* assumption for a given equation.

[64]: Dupressoir et al. (2011), *Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols*
[65]: Dupressoir et al. (2014), *Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols*
[69]: Vanspauwen et al. (2015), *Verifying Protocol Implementations by Augmenting Existing Cryptographic Libraries with Specifications*
[70]: Vanspauwen et al. (2017), *Verifying Cryptographic Protocol Implementations that use Industrial Cryptographic APIs*

Summary

The verification of the role implementations, i.e., the Hoare triples $\vdash_{\pi'_{ext}} \{\psi_i(rid)\}\ c_i(rid)\ \{\mathsf{true}\}$, relies on the following assumptions:

1. contracts for the I/O and cryptographic libraries, where the former's operations are consistent with π'_{ext};
2. the pattern requirement for each pattern t occurring in the I/O specification; and
3. collision-freedom for all equalities Φ_R occurring in the I/O specification.

We suggest to also prove the pattern requirement for a given pattern t whenever possible, e.g., by showing image disjointness and pattern injectivity (at least) for formats and assuming them for the cryptographic operations (Asm. 2.4.2).

$$\begin{array}{rl}
\multicolumn{2}{c}{\mathcal{R}}\\
\multicolumn{2}{c}{\grave{\curlyvee}\ \ (\text{Lemma 2.3.1})}\\
\multicolumn{2}{c}{\mathcal{R}_{\mathsf{intf}}}\\
\multicolumn{2}{c}{\curlyvee\ \ (\text{Lemma 2.3.2})}\\
\multicolumn{2}{c}{\left(|||_{i,rid}\ \mathcal{R}^{i}_{\mathsf{role}}(rid)\right)\ \ \|_{\Lambda}\ \ \mathcal{R}^{e}_{\mathsf{env}}}\\
\curlyvee\ \ (\text{Thm. 2.3.3}) & \\
\pi\big(\psi_i(rid)\big) & \curlyvee\ \ (\text{Prop. 2.5.1})\\
\curlyvee\ \ (\text{Asm. 2.4.1}) & \\
\Big(|||_{i,rid}\ \pi\big(\pi'_{ext}(\mathcal{C}_i(rid))\big)\Big) & \|_{\Lambda}\ \ \pi_{ext}\big(\pi'_{ext}(\mathcal{E})\big)\\
\multicolumn{2}{c}{\curlyvee}\\
\multicolumn{2}{c}{|||_{i,rid}\ \pi_{int}\big(\mathcal{C}_i(rid)\big)\ \ \|_{\Lambda'}\ \ \mathcal{E}}
\end{array}$$

Figure 2.5: Overview of soundness proof, where π and Λ' are defined by $\pi = \pi_{int} \circ \pi_{ext}$ and $\Lambda' = (\pi_{ext} \circ \pi'_{ext})^{-1}(\Lambda)$.

2.5 Concrete Environment and Overall Soundness

We now derive an overall soundness result for our approach, which relates the abstract Tamarin model to the concrete protocol implementation.

2.5.1 Concrete Environment

To formulate such a result, we must first define a concrete environment model $\mathcal{E}$, including a concrete attacker, which can interact with the roles' implementations. These implementations communicate with the environment using I/O library functions, which include non-ghost and ghost parameters: they send and receive both byte strings (used by the program) and ghost terms (used for the reasoning), related by γ. The ghost parameters should be reflected in $\mathcal{E}$'s interface, i.e., its synchronization labels. Moreover, to fit into an overall soundness result, $\mathcal{E}$ must be trace-included in $\mathcal{R}^{e}_{\mathsf{env}}$. Hence, the concrete attacker must not be more powerful than the DY attacker, i.e., we must prevent attacks at the byte string level such as exploiting collisions.

To achieve this, we construct the concrete environment from the term-level environment $\mathcal{R}^{e}_{\mathsf{env}}$ by changing only its *interface* with the protocol. Concretely, we rename and extend every synchronization label $[\lambda_F(rid, \overline{x})]$ of (the LTS induced by) $\mathcal{R}^{e}_{\mathsf{env}}$ to the label $\mathbf{F}(rid, \gamma(\overline{x}), \overline{x})$ in $\mathcal{E}$, and we keep the labels of $\mathcal{R}^{e}_{\mathsf{env}}$'s internal actions. Note that applying the relabeling $\pi_{ext} \circ \pi'_{ext}$ to $\mathcal{E}$ recovers $\mathcal{R}^{e}_{\mathsf{env}}$'s original labels. Hence, we record the following property.

Proposition 2.5.1 $\pi_{ext}(\pi'_{ext}(\mathcal{E})) \preccurlyeq \mathcal{R}^{e}_{\mathsf{env}}$.

2.5.2 Overall Soundness Result

Our goal now is to show that any trace property proved for the Tamarin model is preserved in the concrete system

$$\left(|||_{i,rid}\ \pi_{int}(\mathcal{C}_i(rid))\right) \|_{\Lambda'} \mathcal{E},$$

which is composed of the verified programs' LTSs $\mathcal{C}_i$ and the concrete environment $\mathcal{E}$, where the programs' internal operations are mapped back to their action fact arguments and $\Lambda' = (\pi_{ext} \circ \pi'_{ext})^{-1}(\Lambda)$ synchronizes I/O permissions that also include byte strings beside terms. Note that our soundness result assumes that the role implementations are already verified and that the verifier assumption (Asm. 2.4.1) holds. The verification of the role implementations themselves and the related assumptions are discussed in Sec. 2.4.

Theorem 2.5.2 (Soundness) *Suppose Asm. 2.4.1 holds and that we have verified, for each role i, the Hoare triple $\vdash_{\pi'_{ext}} \{\psi_i(rid)\}\ c_i(rid)\ \{\mathsf{true}\}$. Then*

$$(|||_{i,rid}\ \pi_{int}(\mathcal{C}_i(rid))) \parallel_{\Lambda'} \mathcal{E} \preccurlyeq_{\mathsf{t}} \mathcal{R}.$$

Proof. We decompose the proof into a series of trace inclusions. Fig. 2.5 gives an overview of the proof.

The first trace inclusion is

$$\begin{aligned} &\big(\ |||_{i,rid}\ \pi_{int}(\mathcal{C}_i(rid))\big) \parallel_{\Lambda'} \mathcal{E} \\ \preccurlyeq\ &(|||_{i,rid}\ \pi(\pi'_{ext}(\mathcal{C}_i(rid)))) \parallel_{\Lambda} \pi_{ext}(\pi'_{ext}(\mathcal{E})), \end{aligned} \tag{2.13}$$

where the first line is obtained from the second by pushing the relabeling $\pi_{ext} \circ \pi'_{ext}$ into the parallel composition, thus changing the set of synchronizing labels from Λ to Λ'. Next, we deduce

$$\pi(\pi'_{ext}(\mathcal{C}_i(rid))) \preccurlyeq \mathcal{R}^i_{\mathsf{role}}(rid) \tag{2.14}$$

from Thm. 2.3.3 and from the combination of Asm. 2.4.1 and the assumption $\vdash_{\pi'_{ext}} \{\psi_i(rid)\}\ c_i(rid)\ \{\mathsf{true}\}$. We then leverage a general composition theorem [71, Theorem 2.3], which implies that trace inclusion is compositional for a large class of composition operators including $|||$ and $\parallel_{\Lambda}$. We apply this to the trace inclusion (2.14) and the one from Prop. 2.5.1 to derive the trace inclusion

[71]: Sprenger et al. (2020), *Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification*

$$\begin{aligned} &(|||_{i,rid}\ \pi(\pi'_{ext}(\mathcal{C}_i(rid)))) \parallel_{\Lambda} \pi_{ext}(\pi'_{ext}(\mathcal{E})) \\ \preccurlyeq\ &(|||_{i,rid}\ \mathcal{R}^i_{\mathsf{role}}(rid)) \parallel_{\Lambda} \mathcal{R}^e_{\mathsf{env}}. \end{aligned} \tag{2.15}$$

Our result now follows by combining the trace inclusions (2.13) and (2.15) with Lemmata 2.3.1 and 2.3.2 and the relation inclusions $\preccurlyeq\ \subseteq\ \preccurlyeq'\ \subseteq\ \preccurlyeq_{\mathsf{t}}$ from Sec. 2.2. □

Corollary 2.5.3 (Property Preservation) *Any trace property Φ that holds for $\mathcal{R}$ also holds for $(|||_{i,rid}\ \pi_{int}(\mathcal{C}_i(rid))) \parallel_{\Lambda'} \mathcal{E}$.*

Example 2.5.1 (Secrecy for Diffie–Hellman) We have verified the secrecy of and agreement on the exchanged key for the Tamarin protocol model from Example 2.2.2, expressed as the requirement that in all possible traces of the system, the attacker never learns a value k for which a $Secret(k)$ action occurs in the trace. We then implemented both roles by programs, and proved that they satisfy their I/O specifications (Example 2.3.4). Our soundness result and its corollary guarantee that the composed system also satisfies key secrecy and authentication.

2.6 Application and WireGuard Case Study

To provide evidence that our approach to verifying cryptographic protocol implementations is general, powerful, and scales to complex real-world protocols, we use it to verify our DH running example and the WireGuard protocol. Additionally, we apply our approach to a key component of a digital asset authentication system in Sec. 2.7. Although this component implements on a high level a passive protocol role, which does not send messages, our approach is nevertheless applicable with a minimal extension, thus, demonstrating its generality.

2.6.1 Applying Our Approach

The application of our approach involves three steps:

1. *Protocol model specification and verification in Tamarin.* Protocol models must satisfy our mild format restrictions (Sec. 2.3.1). Existing protocol models may require minor syntactic modifications. Verify the desired trace properties such as secrecy and authentication.
2. *Generation of the roles' I/O specifications.* Our tool automatically generates each protocol role's I/O specification along with definitions of types and internal operations. It accepts all protocol models satisfying our format assumptions. The tool currently supports Gobra (for Go) and VeriFast (for Java).
3. *Role implementation and verification.* Verify an existing or new role implementation against its I/O specification. This relies on user-provided (reusable) contracts for the I/O and cryptographic libraries used and on proofs of the relevant instances of the pattern requirement (e.g., using Asm. 2.4.2 and Prop. 2.4.1).

By Cor. 2.5.3 (and Asm. 2.4.1), all properties proven for the Tamarin model are inherited by the implementation.

For our DH example, we have verified a Go implementation (using Gobra) and a Java implementation (using VeriFast) against their generated specifications. These two implementations are interoperable and exchange messages via UDP. The model amounts to 119 lines that Tamarin automatically verifies in less than 4 s. Gobra verifies the 106 lines long initiator implementation in 11.0 s, requiring 1051 lines of specifications and proof annotations, which includes the specifications for library stubs. 650 lines thereof are generated by our tool. Our Java implementation is comparable in size and verifies in 0.2 s using VeriFast. We have also produced a faulty Go implementation that sends x instead of g^x as the first message. As expected, verification for this faulty implementation fails in 11.0 s because the I/O permissions do not permit sending this payload.

2.6.2 The WireGuard Key Exchange

WireGuard is an open VPN system that is widely deployed on various platforms and integrated into the Linux kernel. Its core is the WireGuard cryptographic protocol.

The WireGuard protocol mainly consists of a handshake, where two agents establish secret session keys and authenticate each other, and a transport phase, where they use these keys to set up a secure channel for message transport. We give an overview of the protocol in Fig. 2.6. The

// handshake phase

$A \to B: \langle 1, sid_I, g^{ek_I}, c_{pk_I}, c_{ts}, mac1_I, mac2_I \rangle$

$B \to A: \langle 2, sid_R, sid_I, g^{ek_R}, c_{empty}, mac1_R, mac2_R \rangle$

// transport phase

$A \to B: \langle 4, sid_R, 0, \text{aead}(k_{IR}, 0, p_0, '') \rangle$

$B \to A: \langle 4, sid_I, 0, \text{aead}(k_{RI}, 0, p'_0, '') \rangle$

$A \to B: \langle 4, sid_R, 1, \text{aead}(k_{IR}, 1, p_1, '') \rangle$

...

Figure 2.6: The WireGuard protocol.

complete protocol additionally features denial of service (DoS) protection mechanisms, which we omit.

The protocol involves two roles, the initiator (Alice) and the responder (Bob), each with long-term secret and public keys. It is assumed that Alice and Bob know each other's public keys pk_I and pk_R in advance. They may optionally use a pre-shared secret. The protocol relies on an authenticated encryption with associated data (AEAD) construction, hashing, and key derivation functions (KDFs). The exact algorithms used are irrelevant for our presentation. All protocol messages contain a tag: 1 and 2 for handshake messages, 3 for the optional DoS prevention messages (not shown), and 4 for transport messages. They also contain randomly generated unique session identifiers sid_I or sid_R (for each role).

The *handshake phase* comprises two messages. Alice and Bob generate fresh ephemeral DH keys ek_I, ek_R, and exchange the associated public keys g^{ek_I}, g^{ek_R}. They also exchange ciphertexts c_{pk_I}, c_{ts}, and c_{empty}, which encrypt Alice's public key, a timestamp, and the empty string, respectively, with keys derived from both long term and ephemeral secrets. The messages also contain message authentication codes (MACs) for the DoS protection mode, not described here. At the end of the handshake phase, both agents compute two symmetric keys k_{IR} and k_{RI}.

WireGuard Message Construction

Fig. 2.7 displays the detailed message constructions and key computations for the WireGuard protocol, where:

- *info* and *prologue* are fixed strings encoding protocol information such as the protocol version;
- aead is an AEAD algorithm, h is a hash function, kdf_1, kdf_2, kdf_3 are key derivation functions (KDFs);
- (k_I, pk_I) and (k_R, pk_R) are the initiator and responder's long-term secret and public keys;
- $(ek_I, epk_I = g^{ek_I})$ and $(ek_R, epk_R = g^{ek_R})$ are the initiator and responder's ephemeral secret and public DH keys, g being the group generator; and
- *psk* is an optional pre-shared key—if unused it is set to a string of zeros.

These two keys are then used to encrypt messages in the *transport phase*: k_{IR} for messages from initiator to responder and k_{RI} for the other direction. Alice and Bob both keep two counters n_{IR} and n_{RI}, counting the number of messages sent in each direction. When Alice sends a message, she encrypts it with k_{IR}, using the current value of n_{IR} as the AEAD nonce, and increments n_{IR}. Thus, no confusion is possible regarding the order of messages. The use of different keys and counters allows messages to be sent independently in each direction, without a need for strict alternation.

Note that the protocol mandates that Alice sends the first transport

c_{pk_I}, c_{ts}, and c_{empty} are computed as follows.

$$\begin{aligned}
c_0 &= \mathrm{h}(\mathit{info})\\
h_0 &= \mathrm{h}(\langle c_0, \mathit{prologue}\rangle)\\
h_1 &= \mathrm{h}(\langle h_0, pk_R\rangle)\\
c_1 &= \mathrm{kdf}_1(\langle c_0, epk_I\rangle)\\
h_2 &= \mathrm{h}(\langle h_1, epk_I\rangle)\\
c_2 &= \mathrm{kdf}_1(\langle c_1, g^{k_R * ek_I}\rangle)\\
k_1 &= \mathrm{kdf}_2(\langle c_1, g^{k_R * ek_I}\rangle)\\
c_{pk_I} &= \mathrm{aead}(k_1, 0, pk_I, h_2)\\
h_3 &= \mathrm{h}(\langle h_2, c_{pk_I}\rangle)\\
c_3 &= \mathrm{kdf}_1(\langle c_2, g^{k_R * k_I}\rangle)\\
k_2 &= \mathrm{kdf}_2(\langle c_2, g^{k_R * k_I}\rangle)\\
c_{ts} &= \mathrm{aead}(k_2, 0, \mathit{timestamp}, h_3)\\
h_4 &= \mathrm{h}(\langle h_3, c_{ts}\rangle)\\
c_4 &= \mathrm{kdf}_1(\langle c_3, epk_R\rangle)\\
h_5 &= \mathrm{h}(\langle h_4, epk_R\rangle)\\
c_5 &= \mathrm{kdf}_1(\langle c_4, g^{ek_R * ek_I}\rangle)\\
c_6 &= \mathrm{kdf}_1(\langle c_5, g^{ek_R * k_I}\rangle)\\
c_7 &= \mathrm{kdf}_1(\langle c_6, psk\rangle)\\
\pi &= \mathrm{kdf}_2(\langle c_6, psk\rangle)\\
k_3 &= \mathrm{kdf}_3(\langle c_6, psk\rangle)\\
h_6 &= \mathrm{h}(\langle h_5, \pi\rangle)\\
c_{empty} &= \mathrm{aead}(k_3, 0, \text{""}, h_6)
\end{aligned}$$

k_{IR} and k_{RI} are the resulting symmetric keys that are used in the subsequent transport phase: $k_{IR} = \mathrm{kdf}_1(c_7)$, $k_{RI} = \mathrm{kdf}_2(c_7)$.

Figure 2.7: The WireGuard message construction

message. The reason is that it is used by Bob to confirm she has received his key—in contrast, Alice confirms this for Bob when she receives the second handshake message.

2.6.3 Tamarin Model

We model the WireGuard protocol in Tamarin as a MSR system satisfying the assumptions from Sec. 2.3.1. Our model features rules for each of the two roles' behavior, as well as environment rules modeling the initial distribution of long-term keys. The environment may spawn any number of instances of each role, i.e., an unbounded number of sessions running in parallel, between the same or different agents.

Note that we not only model the handshake and first transport message, after which the key exchange is concluded, but also the loop that follows where agents may exchange any number of transport messages, in any order, using the computed keys. Verifying such unbounded loops is challenging for automated tools, and often leads to non-termination. For this reason, they are usually not modeled in their full generality. However, the presence of the loop in the implementation required its inclusion in the model as well, so that the implementation adheres to the model's behavior. We had to manually write three lemmata and an *oracle* (a heuristic for Tamarin's proof search) to help the tool terminate. Tamarin can then prove these lemmata and verify the model automatically.

We formulate and prove in Tamarin trace properties expressing authentication (see Tab. 2.1). More precisely, we show that, after the first transport message, the participants mutually agree on the resulting keys: if Alice believes she has exchanged k_{IR} and k_{RI} with Bob, then Bob also believes

Table 2.1: Properties verified for the WireGuard case study. By Cor. 2.5.3, the code preserves the trace properties of the protocol model.

	Verified properties
Protocol	Agreement on the keys, forward secrecy
Code	Memory safety, conformance with generated I/O spec

so, and conversely. Moreover, we prove the forward secrecy of the keys k_{IR} and k_{RI}: they remain secret from the attacker, provided neither Alice nor Bob's long-term secrets were corrupted *before the end of the key exchange*. The Tamarin file consists of 221 lines of MSR rules, and 71 lines of lemmata and properties. It is verified automatically by Tamarin in about 1.90 min.

2.6.4 Implementation and Code Verification

We separately verified the initiator and responder code of the official Go implementation of WireGuard [88] in Gobra. We first proved memory safety, which is independent of our approach, but a required initial step in tools like Gobra and VeriFast. We then verified each role implementation against its I/O specification, which we generated with our tool from the WireGuard Tamarin model. We annotated the code with specifications, namely pre- and postconditions and loop invariants, and proof annotations, namely assertions, lemma calls, and predicate unfolding commands. The code can be compiled as annotations appear inside comments. Tab. 2.1 summarizes all properties we proved.

[88]: Donenfeld (n.d.), *Go Implementation of WireGuard*

Changes to the Implementation. We modified the official Go implementation in three ways. First, we removed features not included in our Tamarin model, namely DoS protection. Second, we made changes to simplify proving memory safety. For this, we removed metrics and load balancing, which requires complex concurrency reasoning currently not supported by Gobra. Lastly, we performed changes related to our approach. Namely, we wrote stubs for cryptographic and network operations and equipped them with trusted specifications (cf. Sec. 2.4.3) and adapted the code accordingly.

Our verified implementation is interoperable with the official implementation and can relay traffic between the operating system and a VPN connection.

Verified Components. We verified both the handshake and transport phases for both roles. These components are responsible for all I/O operations that establish and interact directly with a VPN connection. The codebase contains additional I/O operations outside the verified components that, e.g., interact with the operating system and are not influenced by a VPN connection's ephemeral key material. We did not verify the setup code for network sockets and cryptographic keys. The stubs for cryptographic and network operations are trusted by assumption and thus also not verified.

I/O Specification. In addition to the I/O specifications generated by our tool, we declared the byte string algebra operations and the homomorphism γ.

Our verification of the I/O specification is standard. To verify a call to an I/O operation or an internal operation, we extract the corresponding I/O permission from the predicate $P_i(p, rid, S)$. This requires facts about the model state S. For instance, to send a byte string b, there must exist a term t with $\gamma(t) = b$ such that $\mathsf{out}_i(rid, t) \in S$. We verify such facts by relating the program state to the model state S.

Pattern Requirement. Each of the protocol's three non-linear patterns induces two instances of the pattern requirement, realized as lemma functions (cf. Fig. 2.4).

We proved the pattern requirement instances using Prop. 2.4.1 and Asm. 2.4.2 by showing that all formats are (i) image-disjoint with each other and names and (ii) pattern-injective. Point (i) holds, since all formats start with a different constant and their lengths differ from the lengths of byte strings representing names. Formats are also injective and hence satisfy (ii), as the arguments of all formats appear at fixed offsets in the byte string representation.

Statistics. Our implementation consists of 551 lines of verified Go code, excluding library stubs. Specifications and proof annotations make up 4173 lines of code, 1358 of which are generated by our tool. We required 329 lines of code to declare the term and byte string algebras and the axioms about γ. The verification runtime is about 22.9 and 23.2 s for the initiator and responder, respectively.

The case study demonstrates that our approach is applicable to pre-existing real-world security protocols with implementations of considerable size.

2.7 ADEM Case Study: Verifying Emblems to Authenticate Digital Assets

While Sec. 2.6 evaluates our approach on security protocol implementations containing key exchange mechanisms, this section applies our approach to an implementation arising in the context of Linker and Basin's Authentic Digital EMblem (ADEM) [89], a system for authenticating digital assets. This application is interesting for two main reasons. First, this implementation is different from a classical security protocol as it does not use any secrets, receives almost all necessary data as program inputs, and keeps the computed result local, i.e., does not send this result to the network. Thus, focusing on network I/O operations and proving refinement thereof is insufficient to guarantee that the implementation computes the correct result. We address this shortcoming by slightly extending our approach to treat also command-line inputs and outputs as I/O operations. Thus, we can prove that the implementation performs all checks mandated by the corresponding protocol model and computes a result that refines the model's result. Second, as we will discuss in Sec. 2.7.3, this application illustrates that gaps between implementations and abstract models arise in practice that go beyond relating concrete bytes with abstract terms. More specifically, the implementation performs certain checks concurrently while the model performs them sequentially. Therefore, the implementation may exhibit additional interleavings of these checks that are not possible in the model. To address this shortcoming, we develop a novel solution that closes the gap between concurrency patterns arising in the implementation and the sequential processing in the original abstract model, which leverages TAMARIN's semantics to require only slight adaptations of the model for incorporating concurrent processing therein.

[89]: Linker et al. (2023), *ADEM: An Authentic Digital EMblem*

2.7.1 Authentic Digital EMblem (ADEM)

ADEM is a novel authentication mechanism for digital assets, such as servers, software applications, and networks. Motivated by the International Committee of the Red Cross (ICRC), this mechanism aims at signaling protection under International Humanitarian Law (IHL) similar to the red cross, crescent, and crystal for physical objects such as hospitals, vehicles, and personnel. Similar to physical assets, digital assets belong to a party that is protected under IHL, a so-called *protected party*, and, thus, should not be attacked. Examples for such protected partys include humanitarian organizations like UNICEF and Médecins Sans Frontières.

ADEM authenticates digital assets by means of a digital emblem, which is cryptographically signed. More specifically, a protected party uses a hierarchy of signing keys to ultimately endorse a digital emblem's signing key, which authenticates the digital asset as belonging to this protected party. ADEM enables *agents*, such as military units, to check[2] not only to which protected party a digital asset belongs but also whether an authority permits this party to use an emblem. An *authority*, such as a nation state, permitting a protected party to use emblems means that this authority acknowledges that the protected party generally performs operations that may use (according to the authority's jurisdiction) emblems signaling protection under IHL and, thus, should not be attacked. Since these emblems should signal protection during armed conflicts, a *single* authority authenticating a protected party is impractical as a particular agent may not trust this authority. Therefore, ADEM envisions *multiple* authorities endorsing a protected party such that an agent can pick the most trustworthy authority therefrom.

2: To avoid confusion, we intentionally use "check" to refer to verifying a cryptographic signature and "verify" for verifying properties about a model or implementation.

ADEM is formalized in a Tamarin model for which Linker and Basin [89] prove authentication and accountability. Authentication in this context means that an emblem marks an asset as protected, unless the attacker obtains signing key material, or a component in the system misbehaves. Accountability states that misbehavior is detectable and the misbehaving component can be identified, e.g., if a protected party misuses its status to issue an emblem for an asset that actually does not enjoy protection under IHL or if an authority maliciously endorses a party as being protected. ADEM achieves accountability by making authorities and protected partys commit to their signing keys via the existing Web public key infrastructure (PKI). For this purpose, authorities and protected partys request a certificate from a certificate authority (CA) that binds their signing public key to their identity in the form of their domain name. Ultimately, Certificate Transparency (CT) enables an authority or protected party to detect if someone else requested a certificate in their name such that they can dispute this certificate, which ADEM expects to happen. As a consequence, if a signing key appearing in a CT log is misused, the bound authority or protected party is accountable. Since our approach guarantees that an implementation satisfies the same properties as proven for a Tamarin model, we refer to Linker and Basin's work [89] for the detailed definitions of these security properties and, instead, focus on the implementation and how we can prove refinement therefor.

[89]: Linker et al. (2023), *ADEM: An Authentic Digital EMblem*

2.7.2 Implementation

ADEM is implemented as a prototype in the Go programming language [90] providing several libraries and command-line tools to generate,

[90]: An Authentic Digital Emblem - GitHub Working Group (2025), *ADEM Prototypes*

distribute, and check digital emblems. While the former two functionalities are relevant for availability, the latter is the critical functionality from an authentication point of view. Hence, we focus on emblem checking as incorrectly checking an emblem and, thus, deeming an asset as protected even though it is not, results in users loosing trust in ADEM, which is an attacker's goal. Trust in ADEM is crucial due to emblems' signaling nature, which prevents attacks to protected assets only if users launching such attacks respect emblems.

The method that checks emblems[3] takes an emblem and multiple endorsements, both in binary format, and an optional set of trusted signing keys as inputs and returns, if successful, various information about the emblem, including the assets it protects and its security level. Trusted signing keys enable agents to configure which authorities they trust, and the security level depends, e.g., on whether there is an endorsement by an authority (instead of just internal endorsements by a protected party) or whether one of the involved signing keys is trusted.

3: `VerifyTokens` in [90]

An emblem and its endorsements conceptually form a tree structure, where emblems and endorsements are the nodes and "endorsed by" is the parent-child relation. The emblem is the root of this tree and is cryptographically signed by a key that is endorsed by the emblem's children in the tree. Similarly, each endorsement is again signed by a key endorsed by its children. Thus, the implementation checks that the (potentially attacker-provided) input indeed forms such a tree and traverses this tree from the leaves to the root, checking in each step that an endorsement and ultimately the emblem are valid. To avoid building this dependency tree explicitly, the implementation spawns a goroutine, i.e., a lightweight thread, for each emblem and endorsement, and uses promises to await completion of the children's checks. In particular, each goroutine deserializes the emblem or endorsement, performs the task explained next, and finally completes the corresponding promise indicating that the emblem or endorsement is checked. The goroutine's task waits first on the successful completion of the promise for their signing key; unless it is an endorsement by an authority, in which case the implementation checks that the authority's key binding exists in a certificate in a CT log and that this certificate is valid. Afterwards the task checks the emblem's or endorsement's signature. The implementation waits for all goroutines to complete before proceeding to check the contents of the emblem and endorsements, e.g., whether issuer information is consistent and constraints are met. Finally, the implementation determines the security level based on the processed emblem and endorsements.

2.7.3 Verification

Given that a security proof for ADEM exists in the form of a verified Tamarin model, it is attractive to verify the implementation that checks emblems against this model by applying our approach. However, we have to slightly extend our approach in order to relate this implementation to the model. As explained in the introductory paragraph to Sec. 2.7, the implementation's result, i.e., the security level, is not sent over the network but printed on the command line. Thus, showing that the implementation's network I/O behavior matches the model's is insufficient and, in particular, would not prove that the implementation returns a security level that is at most as trustworthy as the one determined by the model[4]. We bridge this gap by naturally extending our approach to treat network and non-network I/O operations equally, with the latter

4: This issue does not arise in the Tamarin model because the proven lemmata express that desired properties hold whenever certain actions occur in the trace (cf. Sec. 2.2.1). Therefore, lemmata in Tamarin relate via actions to the execution of transitions and do not require that these transitions perform I/O.

including interactions with the command line. More specifically, we treat the program inputs, which are typically obtained by performing some network operations beforehand, as data received from the untrusted environment in the abstract model. Similarly, we make the computed security level an explicit output operation in the model and enforce for the implementation that an I/O permission for the computed security level is returned via the postcondition, justifying printing it on the command line. This adaptation required inserting output operations with string constants corresponding to the security levels into the Tamarin model.

Proving that the implementation is memory safe and refines this slightly modified Tamarin model turned out to be challenging, which required further adaptations to the model and implementation. While we conclude on a high level that verifying such an implementation using our approach is in principle possible, we remark that there are several practical obstacles that need to be overcome, which are to be expected for an implementation and model that were designed without having program verification or applying our approach in mind. Orthogonal to our approach are language features unsupported by Gobra, which we rewrote or axiomatized, and instances of incompleteness, which we sidestepped by appropriate assumptions. We encountered two main obstacles related to our approach, both caused by a mismatch between the implementation and the model. In the following, we focus on these two obstacles and how we overcame them; we refer to Meinen's Master's Thesis [77] for the full details.

[77]: Meinen (2024), *Verification^2 of the Authentic Digital EMblem*

Concurrency. As discussed in Sec. 2.7.2, the implementation employs separate goroutines to check emblems and endorsements concurrently. These goroutines not only exploit the independence of certain endorsements but also simplify the implementation as no explicit dependency tree has to be constructed. We verify the implementation's promise mechanism, which internally uses Go channels, thanks to Gobra's support for Go channels. Additionally, the implementation uses a thread-safe data structure to manage all promises and handle their completion, which we verify using Gobra's support for mutexes and wait groups. In contrast to the implementation, the original Tamarin model processes emblems and endorsements not only sequentially but also in the opposite order, i.e., starting with the emblem and ending with endorsements by authorities. While it is possible to enable multiple goroutines to make progress in a sequential, abstract model, e.g., by sharing the corresponding I/O specification via a shared resource, all transition orderings that can occur in the implementation must be permitted by the model.

We resolve the concurrency and processing order mismatch by adapting the Tamarin model and explicitly modeling the parallel processing of emblems and endorsements. Instead of naively encoding concurrency in the model, we exploit the fact that Tamarin considers the presence of unboundedly many protocol role instances during its proof search. Accordingly, we turn the processing of an emblem or endorsement into a dedicated protocol role. This allows us to use a dedicated I/O specification for each goroutine performing emblem or endorsement checks, which heavily simplifies the concurrency reasoning on the implementation level. However, we introduce at the same time additional communication in the abstract model between protocol role instances. In the model, instances of this dedicated and newly added protocol role communicate via a secure channel with the already existing, main protocol role checking an emblem. In the implementation, this communication corresponds to spawning a goroutine with particular arguments and ultimately sending a goroutine's result via a channel to the main goroutine. We omit the inter-goroutine communication in the model that happens in the implementation via promises for simplicity; an extension is straightforward.

Internal Endorsements. The original ADEM model makes simplifying assumptions about endorsements that the implementation does not satisfy. In particular, the model assumes that an emblem is directly signed with a protected party's signing key. For realistic deployments, however, a protected party may use zero or more intermediate signing keys by signing *internal endorsements*. I.e., instead of directly signing an emblem, a protected party can use a chain of internal endorsements initially signed by its main signing key and ultimately signing an emblem. The original model omits this possibility as the authors argue on paper that an attacker cannot violate the authentication property by compromising any of these intermediate signing keys without also violating one of their assumptions, namely that internal endorsements restrict the issuance of emblems to protected assets. Since the implementation supports internal endorsements, we extend the model accordingly as we would otherwise have to prove that an emblem is directly signed by a protected party's signing key, which is violated for program inputs containing internal endorsements.

Discussion. The ADEM case study demonstrates that our approach is broadly applicable and not restricted to key exchange protocols as long as an abstract model exists. However, an abstract model has to accurately capture an implementation's behavior, which may not always be the case. Due to the abstract nature of a model, it is possible that certain operations were omitted in a model as an extension seemed straightforward at modeling time. If these omitted operations occur in an implementation, they are, however, relevant for proving refinement. Therefore, certain adaptations to the model or implementation might be necessary to enable successful verification despite our approach being powerful. Specific to ADEM is that the original TAMARIN model proves the claims of the corresponding publication [89]. While our changes result in the model more accurately reflecting the implementation, most of these changes increase at the same time the gap between the model and ADEM's publication, making it more difficult for readers to validate that the model captures the published protocol and claims.

[89]: Linker et al. (2023), *ADEM: An Authentic Digital EMblem*

2.8 Related Work

We compare our work with different kinds of approaches to formally verifying protocol implementations. We focus here on symbolic approaches providing soundness guarantees for security protocol implementations and approaches to message parsing, complementing the more general discussion in Sec. 1.1.

Sound Model Extraction and Code Generation. Bhargavan *et al.* [53] present a sound model extractor from (a first-order subset of) F# to PROVERIF models. They work with an abstract datatype of byte strings and corresponding interfaces for the cryptographic and network libraries, which they instantiate both to symbolic terms for prototyping and to actual library implementations. Thus, they assume that protocol messages are injective and disjoint. This approach is used to verify a small functional implementation of TLS 1.0 in F# ([54]) and reference implementations of TLS 1.2 and TLS 1.3 in a typed subset of JavaScript ([12]). Several works generate both models for verification and executable

[53]: Bhargavan et al. (2008), *Verified Interoperable Implementations of Security Protocols*

[54]: Bhargavan et al. (2012), *Verified Cryptographic Implementations for TLS*

[12]: Bhargavan et al. (2017), *Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate*

code from abstract protocol descriptions. While Almousa *et al.* [62] and Bugliesi *et al.* [91] prove the correctness of a partial translation from a high-level to a low-level semantics, neither paper proves the full translation's correctness. Sisto *et al.* [63] generate a ProVerif model and a refined Java implementation from an abstract Java protocol specification and prove the implementation's soundness. CryptoBap [59] extracts a ProVerif and CryptoVerif model from machine code. The authors prove on paper a theorem stating that their extraction starting from an intermediate representation preserves attack probabilities such that any upper bound proven by the protocol model verifier is an upper bound for the intermediate representation. While they use a certificate producing compiler [92] to obtain their intermediate representation from machine code together with a proof of correctness, it remains unclear whether this certificate straightforwardly composes with their on-paper proof. The same authors fully mechanize this proof in an interactive theorem prover in follow-up work [60] while moving from the computational to the symbolic model of cryptography to enable mechanization and compositionality. OwlC [52] builds on top of Owl [51] to generate a Rust implementation from a security protocol model in their domain-specific language. Alongside the implementation, OwlC generates an interaction tree (ITree) [93] specifying the network I/O, sampling of random numbers, and declassification behavior of a protocol and use Verus to verify the generated code against the generated ITree. Since Owl and Verus do not support loops and co-recursion, respectively, proving an implementation of a non-terminating server, such as Example 2.2.4, is currently not possible, even if the server's non-terminating loop is placed in a handwritten part of the codebase as suggested by the authors [52, Sec. 7.2].

[62]: Almousa et al. (2015), *Alice and Bob: Reconciling Formal Models and Implementation*

[91]: Bugliesi et al. (2016), *Security Protocol Specification and Verification with AnBx*

[63]: Sisto et al. (2018), *Formally Sound Implementations of Security Protocols with JavaSPI*

[59]: Nasrabadi et al. (2023), *CryptoBap: A Binary Analysis Platform for Cryptographic Protocols*

[92]: Lindner et al. (2019), *TrABin: Trustworthy Analyses of Binaries*

[60]: Nasrabadi et al. (2025), *Symbolic Parallel Composition for Multi-Language Protocol Verification*

[52]: Singh et al. (2025), *OwlC: Compiling Security Protocols to Verified, Secure, High-Performance Libraries*

[51]: Gancher et al. (2023), *Owl: Compositional Verification of Security Protocols via an Information-Flow Type System*

[93]: Xia et al. (2020), *Interaction Trees: Representing Recursive and Impure Programs in Coq*

These approaches usually target a single implementation language. Although adding support for additional programming or modeling languages is possible in principle, this requires a translator and a related soundness result for each new language. Furthermore, model extraction for realistic protocols lacks abstraction and conciseness: it tends to yield large models that are difficult to automatically verify.

Code Verification Only. Bhargavan *et al.* [67] modularly verify protocol code written in F# using the F7 refinement type checker [68]. They rely on protocol-specific invariants for cryptographic structures, e.g., stating which messages are public. Vanspauwen and Jacobs [69, 70] use a similar approach for protocols implemented in C and verified using VeriFast. They allow the concrete attacker to directly manipulate byte strings, which they overapproximate symbolically by a set of terms. However, it is unclear what effect these manipulations have on message parsing in the protocol roles. While the verification of global protocol properties in the earlier work [67] required additional handwritten proofs, the more recent work [39] enables their verification in a single tool, F*, by explicitly incorporating a global event trace.

[67]: Bhargavan et al. (2010), *Modular Verification of Security Protocol Code by Typing*

[68]: Bengtson et al. (2008), *Refinement Types for Secure Implementations*

[69]: Vanspauwen et al. (2015), *Verifying Protocol Implementations by Augmenting Existing Cryptographic Libraries with Specifications*

[70]: Vanspauwen et al. (2017), *Verifying Cryptographic Protocol Implementations that use Industrial Cryptographic APIs*

[39]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

While these approaches are also modular and, thus, scale to protocols like Signal, we push modularity and automation further. Our approach not only decouples proving the security properties from verifying the implementation's correctness, but it also leverages Tamarin's automated proof search.

Combined Model and Code Verification. Dupressoir *et al.* [64, 65] use the interactive prover Coq for model verification in combination with the C code verifier VCC. Their approach involves reasoning about concrete byte strings and their relation to terms. The central definitions of the

[64]: Dupressoir et al. (2011), *Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols*

[65]: Dupressoir et al. (2014), *Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols*

protocol model are duplicated in Coq and in VCC and some theorems proven in Coq are imported as axioms into VCC.

Igloo [71] is a framework for distributed system verification that soundly combines model refinement in Isabelle/HOL with code verification using I/O specifications. Their case studies include a simple authentication protocol. We follow similar steps to extract I/O specifications from Tamarin models, but do this generically and automatically for a large class of protocol models. In contrast, these steps must be repeated in Igloo for each protocol, which requires Isabelle/HOL expertise. Moreover, our way of relating terms and byte strings is more flexible and realistic than theirs, which assumes an injective function from byte strings to terms. Penninckx *et al.* [80] introduced I/O specifications for verifying programs' I/O behaviors, but did not propose a method for verifying global system properties.

[71]: Sprenger et al. (2020), *Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification*

[80]: Penninckx et al. (2015), *Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs*

Message Parsing. Mödersheim and Katsoris [82] show that an abstract symbolic model using message formats soundly abstracts a more concrete model, which includes associative message concatenation, variable and fixed length fields, and tags. Their result holds for protocols whose formats are uniquely parsable and image disjoint, and they give algorithms to check these conditions. This work has inspired our use of byte string algebras and our decomposition of the pattern requirement into image disjointness and pattern injectivity. Their focus is on abstraction soundness, whereas ours is on code verification. EverParse [86] and its successor EverParse3D [94] are frameworks to generate provably secure (i.e., injective and surjective) parsers and serializers in C for authenticated message formats. Wallez *et al.* [87] derive properties for message formats based on game-based cryptographic assumptions and propose the Comparse framework to prove these properties for particular message formats and to generate reference implementations for testing purposes. Instead of defining a concrete environment and concrete attacker (like we do), Nasrabadi *et al.* [60] lift a concrete component, e.g., implementing a protocol role, to an LTS operating on symbolic terms and define composition with a symbolic attacker (another LTS) on this level. A key ingredient to this composition is a *deduction combiner* specifying how logical predicates transfer from one LTS to another to enable deductions in the latter. Since messages are represented as variables, the attacker can, e.g., decompose a received message m only after obtaining the predicate $m \mapsto t$ from the message sender, which states that variable m maps to the message's symbolic term t [60, Example 3]. The authors discuss different deduction combiners that over- or under-approximate attacker deductions [60, Sec. II-G], where the over-approximating deduction combiner is similar to the extended symbolic model proposed by Vanspauwen and Jacobs [69].

[82]: Mödersheim et al. (2014), *A Sound Abstraction of the Parsing Problem*

[86]: Ramananandro et al. (2019), *EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats*

[94]: Swamy et al. (2022), *Hardening Attack Surfaces with Formally Proven Binary Format Parsers*

[87]: Wallez et al. (2023), *Comparse: Provably Secure Formats for Cryptographic Protocols*

[60]: Nasrabadi et al. (2025), *Symbolic Parallel Composition for Multi-Language Protocol Verification*

[69]: Vanspauwen et al. (2015), *Verifying Protocol Implementations by Augmenting Existing Cryptographic Libraries with Specifications*

Conclusions. We have proposed a novel approach to cryptographic protocol verification that soundly bridges abstract design models, specified as MSR systems, with code-level specifications. This allows us to leverage the automation and proof techniques available in Tamarin for design verification together with state-of-the-art program verifiers to obtain security guarantees for protocol implementations. Our approach is general, compatible with different code verification tools, and applicable to real-world protocols as demonstrated by our case studies. Furthermore, our approach enabled Morio and Künnemann [95] to build a runtime monitor enforcing that a program execution adheres to a Tamarin model as they adopt our decomposition of a Tamarin model into role-specific MSR systems.

[95]: Morio et al. (2024), *SpecMon: Modular Black-Box Runtime Monitoring of Security Protocols*

Invariant-Based Verification of Security Protocol Implementations 3

3.1 Introduction

The methodology presented in Chapter 2 proves refinement between an *existing* verified model and a corresponding *existing* implementation. This methodology supports realistic implementations, but requires expertise in and relies on the soundness of two tools (a protocol model verifier *and* a program verifier). Moreover, formal models may not always exist, or may not be in sync with an evolving implementation, as seen in Sec. 2.7. Additionally, the model checking approach chosen by state-of-the-art protocol model verifiers is non-modular in the sense that they explore all state transitions that are applicable to a given state. Hence, verifying security properties for protocols exhibiting looping behavior frequently necessitates auxiliary lemmata or custom oracles to ensure that the proof search terminates.

In contrast, we present in this chapter a methodology for the verification of strong security properties (e.g., injective agreement and forward secrecy) directly on the level of protocol implementations. Our methodology leverages established program verification techniques that are supported by a wide range of existing automated[1] tools (e.g., [26, 29, 32, 34, 96]), which makes it readily applicable. It is based on separation logic [24, 25], a program logic that supports the language features used to write efficient implementations, such as mutable heap data structures and concurrency. As a result, our methodology applies to realistic implementations written in mainstream programming languages such as C, Go, JavaScript, and Rust. Verification in our methodology is *modular*, that is, one can verify each method (or protocol participant) in isolation. Modularity is crucial for scalability, to reduce the re-verification effort when the code evolves, and to provide strong guarantees for libraries.

1: The proof search is automatic but relies on user-provided annotations, such as pre- and postconditions and loop invariants.

[26]: Jacobs et al. (2011), *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*
[29]: Blom et al. (2014), *The VerCors Tool for Verification of Concurrent Programs*
[32]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*
[34]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*
[96]: Santos et al. (2018), *JaVerT: JavaScript Verification Toolchain*

[24]: O'Hearn et al. (2001), *Local Reasoning about Programs that Alter Data Structures*
[25]: Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*

As is common in protocol verification, we explicitly model the global trace of a protocol execution, which allows us to express security properties in ways familiar to security experts. This trace is expressed and manipulated via *ghost code* [97], that is, program code that is used for verification purposes, but erased by the compiler before the program is executed. The ghost code required to manipulate the global protocol trace is encapsulated in the I/O and cryptographic libraries used by an implementation to ensure, e.g., that each sent message is correctly reflected on the trace.

[97]: Filliâtre et al. (2014), *The Spirit of Ghost Code*

Using ghost code allows us to cleanly separate the global trace, which is necessary to prove protocol-wide properties, from the data structures maintained locally by each participant. We treat each participant instance of a protocol (including a Dolev–Yao (DY) attacker [18]) as a concurrent thread, and the global trace as shared state among these threads. This approach allows us to reason about unboundedly many participant instances and to leverage existing verification techniques and tools for shared-data concurrency.

[18]: Dolev et al. (1983), *On the Security of Public Key Protocols*

Contributions. We make the following contributions:

- ➤ We present a modular verification methodology for protocol implementations, based on global traces and concurrent separation logic, that applies to a wide range of programming languages, protocol implementations, and verification tools.

- We show how to use separation logic's linear resources to *modularly* prove injective agreement, i.e., the absence of replay attacks. To the best of our knowledge, we present the first invariant-based verification technique for this property.
- We developed a reusable Go library that facilitates maintaining the global trace; protocol-independent properties are verified once and for all for this library and can, thus, be reused for different protocol implementations.
- We demonstrate the practicality of our approach by using the Gobra verifier [34] to verify memory safety and security of Go implementations of three protocols: Needham–Schroeder–Lowe (NSL) [98, 99], signed Diffie–Hellman (DH) [100], and WireGuard [37]. We show that our approach supports different programming languages and verifiers by additionally implementing a prototype of the reusable library for C and the VeriFast verifier [26], and using it to verify a C implementation of NSL. The implementations of our reusable verification library and the case studies are open-source [101].
- We prove soundness of our approach, in particular, that the global trace correctly reflects all relevant protocol steps and, thus, any security property proved for the trace indeed holds for the protocol implementation.

[34]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*

[98]: Needham et al. (1978), *Using Encryption for Authentication in Large Networks of Computers*

[99]: Lowe (1996), *Breaking and Fixing the Needham–Schroeder Public-Key Protocol Using FDR*

[100]: Diffie et al. (1976), *New Directions in Cryptography*

[37]: Donenfeld (2017), *WireGuard: Next Generation Kernel Network Tunnel*

[26]: Jacobs et al. (2011), *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*

[101]: Arquint et al. (2023), *A Generic Methodology for the Modular Verification of Security Protocol Implementations*

We build on and substantially extend two lines of prior work: Our use of a global trace and security labels to prove secrecy is inspired by Bhargavan *et al.* [39], but our approach achieves modularity without relying on a coding discipline (cf. Sec. 1.1.3), and thus handles existing protocol implementations soundly. Our encoding of the global trace as a concurrent data structure is inspired by Dupressoir *et al.* [64]. Their work depends on specific features of the used programming language (e.g., C's volatile fields) and verifier (VCC [66]), while we present a separation-logic-based methodology applicable across different programming languages and verifiers, as demonstrated by our case studies. The use of separation logic allows us to verify concurrent, heap-manipulating programs and prove security properties that so far were out of reach for invariant-based approaches.

[39]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

[64]: Dupressoir et al. (2011), *Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols*

[66]: Cohen et al. (2009), *VCC: A Practical System for Verifying Concurrent C*

Outline. Sec. 3.2 introduces background on trace-based verification and our attacker model. In Sec. 3.3, we explain how we encode the global trace and how we relate it formally to the local state of each participant. In Sec. 3.4, we show how to prove important security properties based on a suitable trace invariant, and how we use separation logic's linear resources to prove injective agreement. We introduce our reusable verification library in Sec. 3.5, which implements our methodology, and substantially reduces the verification effort per protocol. Sec. 3.6 describes our case studies. We explain the trust assumptions underlying our methodology and sketch its soundness proof in Sec. 3.7 and 3.8, respectively. In Sec. 3.9, we present an extension to enforce secure deletion of protocol secrets on the programming language's level. We compare the methodologies from Chapter 2 and this chapter in Sec. 3.10 before we discuss related work in Sec. 3.11.

3.2 Trace-Based Verification

A protocol's security depends on the interplay of the protocol participants in the presence of an attacker. A standard technique to verify security is to record all relevant actions of the participants and the attacker on a *global trace* and to formulate the intended security properties as properties of this trace. Verification then amounts to proving that all possible traces of a protocol satisfy the intended properties. In this section, we give a high-level overview of this approach; we provide the details in the later sections.

Attacker. We consider a DY attacker that has full control over the network and performs symbolic cryptographic operations. These operations are modeled as functions over symbolic values, so-called *terms*, and encode the perfect cryptography assumption, e.g., that decryption succeeds if and only if it uses the correct key.

An attacker can apply these functions to all terms in its knowledge, which initially consists of all publicly-known terms, including string and integer constants. An attacker obtains additional knowledge by reading messages on the network. Furthermore, an attacker may corrupt participants, which adds all terms in the state of the corrupted participant to the attacker knowledge. We model two kinds of corruption: Corrupting a *participant* leaks its long-term state, which is common to all instances of this participant, such as long-term secret keys. Corrupting a *participant session* additionally leaks short-term state, e.g., ephemeral secret keys, or exchanged nonces related to the corrupted session[2].

2: Session corruption affects the entire short-term state of a participant instance as we assume that an execution of a protocol participant's implementation corresponds to a single protocol session. We discuss in Sec. 3.9 a more fine-grained treatment of individual sessions.

Trace Entries. The global trace is a sequence of events. Each event corresponds to a high-level operation performed by a participant or the attacker. It has a name and takes event-specific arguments. E.g., event *CreateNonce*(n) records that nonce n was created. This event is protocol-independent; we also support protocol-specific events to keep track of the progress within a protocol execution and to express specific security properties. E.g., a protocol-specific event may express which nonces or keys a participant uses to communicate with a peer (cf. Sec. 3.4).

We use seven protocol-independent events: (1) A *create nonce* event records that a fresh nonce has been generated. (2) A *send message* event records that a message has been sent on the network. Both events may originate from a participant or the attacker. The remaining five protocol-independent events model the capabilities of the attacker. (3) The (unique) *root* event is the first event on every trace and contains the initial attacker knowledge. (4) An *extend attacker knowledge* event models that the attacker learns additional terms, e.g., by applying a cryptographic operation to a term already in the attacker knowledge. Corruption is represented by (5) a *participant corruption* or (6) a *session corruption* event. In both cases, we use extend-events (4) to add the newly-learned terms (from the corrupted state) to the attacker knowledge. At any point during a protocol run, the total attacker knowledge is therefore determined by the union of the root event (3), the send-events (2), and the extend-events (4). Finally, (7) a *drop message* event records that the attacker dropped a message from the network.

Trace Invariant. To reason modularly about the (unbounded) set of all possible traces, we introduce a *trace invariant*, a property that must hold for every prefix of each trace produced by a protocol. Verification then consists of two main steps: first, proving that each action of a participant or the attacker (according to the above attacker model) maintains the trace invariant and, second, showing that the trace invariant implies the intended security properties.

An important component of a trace invariant are *message invariants*, which characterize the content of a message. For instance, a message invariant might express that a message parameter is a nonce (as opposed to an arbitrary term).

3.3 Local Reasoning

In the previous section, we have summarized how we can prove security properties based on a global trace of events. In this section, we show how to verify concrete protocol implementations by relating the global trace of the protocol to the local state and operations of each protocol participant. This verification is modular and can be automated using existing verification tools.

3.3.1 Safety Verification

To support realistic, efficient, and existing protocol implementations, our verification technique needs to handle programming concepts such as mutable heap structures and concurrency. To this end, we employ separation logic [24, 25], the de facto standard for the modular verification of imperative code, as discussed in Sec. 1.1.2. Separation logic is supported by existing verifiers for many languages, including VeriFast [26] for C, Prusti [32] for Rust, and Gobra [34] for Go. All of them can be used in combination with our methodology.

[24]: O'Hearn et al. (2001), *Local Reasoning about Programs that Alter Data Structures*
[25]: Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*

[26]: Jacobs et al. (2011), *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*

[32]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*

[34]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*

In specifications, the *points-to assertion* $p \mapsto e$ expresses ownership, i.e., that the current function has an exclusive *permission* to access location p and that p has value e (we write _ if the value is irrelevant). For instance, the proof rule for heap updates (rule Write in Fig. 3.1) enforces via its precondition that the current function execution may update p only if it holds the corresponding permission; otherwise, verifying this function results in a verification failure.

Permissions are initially obtained when allocating a heap location, and are transferred between function executions upon call and return according to the callee function's specification. Permissions may also be transferred between threads, see Sec. 3.3.3.

Verifying a protocol implementation in separation logic ensures that it is memory safe (e.g., does not cause null-pointer dereferences or buffer overflows), does not abort (e.g., due to division by zero), and does not exhibit data races. Where needed for our safety or security proof, we also verify functional correctness properties. We omit the details of safety

$$\frac{}{\Gamma \vdash \left[p \mapsto \_\right]\ {*p} := e\ \left[p \mapsto e\right]}\ (\textsc{Write})$$

$$\frac{\Gamma \vdash [P_1]\ C_1\ [Q_1] \quad \Gamma \vdash [P_2]\ C_2\ [Q_2]}{\Gamma \vdash [P_1 \star P_2]\ C_1 \,||\, C_2\ [Q_1 \star Q_2]}\ (\textsc{Par}) \qquad \frac{\Gamma \vdash [P \star I_r]\ C\ [Q \star I_r]}{\Gamma, r : I_r \vdash [P]\ \textbf{with}\ (r)\ \{C\}\ [Q]}\ (\textsc{With})$$

Figure 3.1: Selected separation logic proof rules: heap writes (cf. Sec. 3.3.1) along with parallel composition and lock-protected critical sections (cf. Sec. 3.3.3). Side-conditions are omitted for simplicity.

$$\begin{aligned}
&M1. \quad A \longrightarrow B : \{\langle 1, na, A\rangle\}_{pk_B}\\
&M2. \quad B \longrightarrow A : \{\langle 2, na, nb, B\rangle\}_{pk_A}\\
&M3. \quad A \longrightarrow B : \{\langle 3, nb\rangle\}_{pk_B}
\end{aligned}$$

Figure 3.2: The NSL public key protocol, where *na* and *nb* are nonces, whose generation is omitted. $\{m\}_{pk}$ and $\langle\cdots\rangle$ denote public key encryption of plain text m under the public key *pk* and tupling, respectively. Creation and distribution of the participants' authentic keys is not part of the protocol.

proofs here because they are routine work and orthogonal to the focus of this dissertation.

3.3.2 Relating Bytes with Terms

Our global trace includes symbolic terms, such as keys, nonces, and messages. In concrete implementations, these terms are typically represented by (mutable) byte arrays. In order to relate the two, we use a *concretization function* γ, which maps a term to its byte representation. We use this function in specifications; in particular, we have annotated library functions, e.g., for cryptographic operations, to relate the term representations of their inputs and outputs. E.g., a hash function that maps the byte array *xa* (representing, e.g., a message) to the byte array *ra* (representing, e.g., a number) is specified by relating the corresponding terms: $\exists x, r.\, xa = \gamma(x) \land ra = \gamma(r) \land r = h(x)$, where h is the symbolic hash operation on terms.

Parsing a received message often requires showing that the parsed byte array b corresponds to a given term t: $b = \gamma(t)$. Proving this property generally requires that each byte array corresponds to a *unique* term. However, this requirement is typically not satisfied in realistic implementations where, e.g., a byte array of length four could store an integer or an ASCII-encoded string, which have different term representations. A possible solution [39, 71] is to enforce a unique byte-level representation for every term (for instance, by preceding it with a tag). However, this is not possible when targeting existing implementations with fixed message formats.

[39]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*
[71]: Sprenger et al. (2020), *Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification*

Therefore, we adopt a less restrictive solution here. We use the *pattern requirement* (cf. Def. 2.4.1), which allows multiple terms to have the same byte-level representation *in general*, but requires a *unique* representation for the terms corresponding to protocol messages. This requirement allows a participant to uniquely determine the term for a parsed message. It ensures that the concretization function γ is *injective* on the byte arrays received as messages. The pattern requirement is satisfied by many protocols because they include message tags to distinguish the *kinds of messages*, which in turn determines the unique relationship between a byte array and a term. At the same time, it allows clashes among the representations of other terms, such as integers and strings.

We illustrate the approach using the NSL public key protocol [99] in Fig. 3.2. After receiving message *M1*, Bob parses it as an encrypted triple. The specification of the parse operation ensures $\exists na.\, \gamma(m) = \gamma(\{1, na, A\}_{pk_B})$. Since $\{1, na, A\}_{pk_B}$ is a protocol message, we can apply the pattern requirement to derive the required information about m: $\exists na.\, m = \{1, na, A\}_{pk_B}$.

[99]: Lowe (1996), *Breaking and Fixing the Needham–Schroeder Public-Key Protocol Using FDR*

3.3.3 Global Trace Encoding

As explained in Sec. 3.2, we use a global trace of events, verify invariants over this trace, and finally prove that the invariants imply the intended security properties. For this approach to be sound, the global trace has to include all relevant events performed by the protocol participants and the attacker, which we ensure as follows.

We model each participant instance potentially participating in a protocol session, and the attacker, as a thread in a concurrent system. Each thread maintains its own (mutable) local state, which may contain short-term, session-specific data and long-term data that is shared by all instances of the participant. Multiple instances of the same protocol role are modeled as threads that execute the same code. Soundness of separation logic ensures that any verified property holds for all possible interleavings between the threads, that is, for all possible interactions between the participant instances and the attacker. Moreover, since separation logic is modular, it verifies the implementation of each participant in isolation, independent of the other threads potentially running in the system (assuming only that their implementations are also verified). Consequently, the verified properties hold for an arbitrary, unbounded number of participant instances.

Separation logic achieves thread-modular reasoning by ensuring that different threads operate on *disjoint* memory, which prevents data races and eliminates interference between threads (see below for shared state). The proof rule for parallel composition (Par in Fig. 3.1) illustrates this approach. The threads C_1 and C_2 can be verified independently. They operate on the heap locations for which they obtain permissions via their preconditions P_1 and P_2, resp. Separation logic's *separating conjunction* $*$ in the precondition of the parallel composition expresses that the permissions in P_1 and P_2 are disjoint. Note that we show the rule for a structured parallel composition statement for simplicity; our technique also supports dynamic thread creation.

Each thread needs to manipulate its own local data structures and the global trace data structure that is shared among all threads. To support mutable shared state, we can use any of the established verification techniques for concurrency reasoning. For concreteness, we use a global *lock*, which is associated with a lock invariant that needs to be established when the lock is first created. This invariant may then be assumed whenever the lock is acquired and must be proved to hold upon release. Proof rule With in Fig. 3.1 illustrates this reasoning for a critical section C that is protected by the lock r. I_r is the invariant associated with lock r, as specified by $r : I_r$ in the proof context. Conceptually, a lock owns the permissions expressed in I_r and temporarily lends these permissions to a thread on entering the critical section.

Since the global trace exists only for the purpose of verification, we model it as *ghost state* and all operations on it as *ghost operations*; both are erased during compilation. Consequently, the lock protecting this ghost data structure can also be erased. Reasoning about ghost locks is completely analogous to standard locks (and supported by separation logic program verifiers). However, since a ghost lock is erased during compilation, it does not ensure mutual exclusion. Therefore, any non-ghost operation performed between an acquire and a release must be *atomic* to ensure that erasing the ghost lock does not create thread interleavings that were not considered during verification.

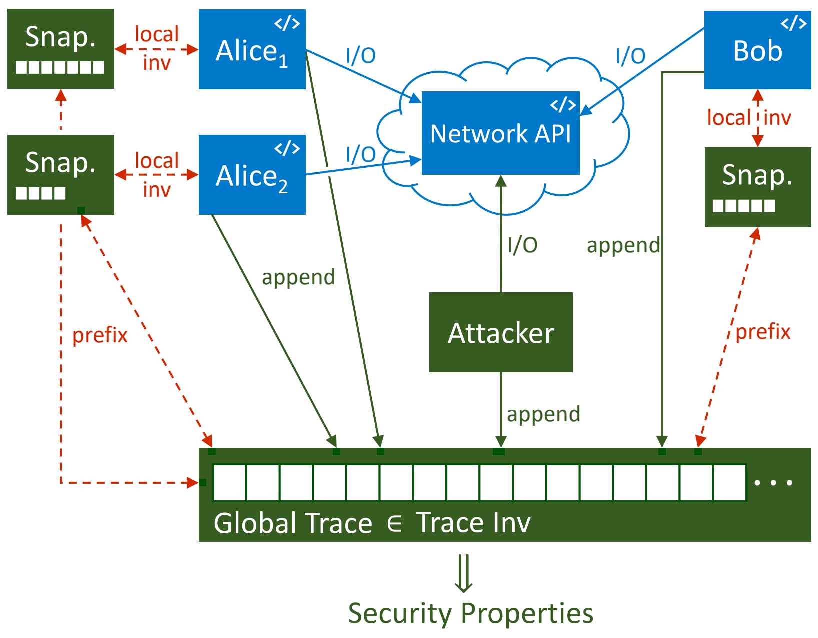

Figure 3.3: An overview of the main components of a protocol execution in our methodology. The blue boxes are components of the protocol implementation; green boxes denote ghost structures that are used for verification. Blue and green arrows denote actual and ghost method calls, resp. The red dashed arrows denote invariants relating different data structures. Participants and the attacker send and receive messages by interacting with the network. The attacker can perform additional I/O operations such as instructing the network to drop or modify messages. All protocol-relevant operations (including I/O operations) are recorded on a global trace. We verify (global) security properties by proving modularly that each protocol implementation (e.g., two and one implementations of Alice's and Bob's role, resp.) and the attacker maintain a trace invariant, and that the trace invariant implies the security properties. We enable the verification of participants by relating participant-local state with the trace via local ghost state that contains a participant's local snapshot, i.e., its last observed version of the trace.

The trace data structure provides two operations: appending an event, and reading the current state of the global trace. Fig. 3.3 illustrates how participants and the attacker interact with the global trace. The lock invariant for the global trace is the trace invariant. By formulating this invariant in separation logic, it can express ownership of heap locations and other resources, which allows us to prove security properties that are out of reach for existing invariant-based related work, as we will see in Sec. 3.4.1.

Participants must record all protocol-relevant operations on the global trace. That is, to perform an operation such as sending a message or creating a nonce, they must (1) acquire the ghost lock, (2) perform the operation, (3) append the corresponding event to the trace, and (4) release the ghost lock (and at this point prove that the trace invariant is preserved). For each relevant operation, we provide a library wrapper (see Sec. 3.5 for details) that performs these four steps[3]. Preconditions on the library functions ensure that the performed operation indeed preserves the trace invariant. Since the trace invariant (and, hence, the preconditions) contain protocol-specific properties, our library is parametric in the invariant (cf. Sec. 3.5). To ensure that *all* relevant operations are recorded on the trace, it then suffices to perform a simple syntactic check that relevant operations are performed only via the wrapper library.

3: To avoid any runtime overhead, calls to this wrapper library could be inlined (and ghost code is erased in any case).

The attacker is handled similarly. We model it as code that (1) acquires the ghost lock, (2) determines which operations the attacker could potentially perform based on its current attacker knowledge (which is recorded on the trace), (3) non-deterministically chooses any of these operations and appends the corresponding event to the trace, and (4) releases the ghost lock (and at this point proves that the trace invariant is preserved). Verifying this code ensures that all possible attacker operations preserve the trace invariant. In other words, the invariant may state only those properties that are valid under our attacker model, a property we call *attacker completeness* (sometimes referred to as *robust safety* [102] or *attacker typability* [39]).

[102]: Gordon et al. (2001), *Authenticity by Typing for Security Protocols*

[39]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

Participant and session corruption are two of the possible attacker operations in step 2 above. In both cases, step 3 adds all symbolic terms possibly present in the participant's (long-term or short-term) state to the attacker knowledge, and step 4 checks that the invariant about the attacker knowledge is maintained.

3.3.4 Local Snapshots

To prove that a protocol-relevant operation preserves the trace invariant, we frequently need to relate the arguments of the operation to earlier events on the trace. For example, when sending the first message of the NSL protocol (Fig. 3.2), Alice has to show that the message invariant holds. The message invariant specifies that *na* is a nonce, i.e., requires a prior *CreateNonce*(*na*) event on the trace.

Discharging such proof obligations requires that participants retain information about their prior operations on the global trace. Since the global trace is a shared data structure that may grow between any two accesses, participants may soundly hold on to those facts that are *stable* under extensions of the trace. For instance, if a *CreateNonce*(*na*) is present on the trace at some program point, it will also be present in all future versions of the trace.

We represent the stable information of a participant by maintaining in each participant a *local snapshot* (i.e., a local copy) of the global trace (see Fig. 3.3). Since the global trace may evolve by actions of other participants and the attacker, the local snapshot of a participant is generally a prefix of the global trace. Whenever a participant performs a protocol-relevant operation, we update its local snapshot to the current global trace. The trace invariant ensures that the local snapshots of all participants are prefixes of the global trace, and that these updates are the *only* modifications of local snapshots.

With this design, local snapshots need to be owned by the participants (to ensure their values are retained across operations of other threads), and they must *also* be owned by the ghost lock (to allow the lock invariant to relate the local snapshots to the global trace). To express this notion of shared ownership, we use fractional permissions [103], which are supported by many separation logics and distinguish between exclusive and shared ownership, which permits multiple threads in a concurrent program to simultaneously share ownership of a heap location. Conceptually, fractional permissions allow one to split an exclusive permission into several fractions each representing shared ownership; a non-zero fraction permits read access, whereas the full permission is required for write access. In specifications, we express permission to a heap location `l` with fraction `p` as `acc(l,p)`. Separating conjunction *adds* the fractions in both conjuncts and yields false if the sum for any location exceeds a full permission. For instance, `acc(l1,1) * acc(l2,1/2)` specifies full and half permissions for heap locations `l1` and `l2`, respectively. Additionally, this example implicitly specifies that the heap locations are disjoint, i.e., `l1` ≠ `l2`. Otherwise, if `l1` and `l2` were aliased, the permission amounts would add up to $3/2$ contradicting separation logic's invariant that at most a full permission exists for a heap location.

[103]: Boyland (2003), *Checking Interference with Fractional Permissions*

We split the permission to a local snapshot into two halves: One half is part of the trace invariant and lets it express properties of the local snapshot, namely that the local snapshot is a prefix of the global trace. The

```
1 na /*@, naT @*/ := CreateNonce(/*@ rvlib @*/)
2 //@ assert rvlib.Snap().NonceOccurs(naT)
```

Figure 3.4: Excerpt from a NSL implementation for Alice creating a nonce and demonstrating how to relate local state with the global trace. `//@` and `/*@ ... @*/` mark ghost code that is used for verification only. We omit the nonce's secrecy label (Sec. 3.4.2) for simplicity.

other half remains with the corresponding participant and enables the participant to retain information about the global trace. After acquiring the ghost lock, a participant temporarily obtains exclusive permission to its local snapshot by adding the half it holds with the half from the trace invariant (through the precondition $P \star I_r$ in rule WITH in Fig. 3.1) and can, thus, update the local snapshot.

By letting each participant retain a non-zero permission to its snapshot, we can rule out interference from other threads and, thus, use standard sequential reasoning to relate the content of the local snapshot to the concrete data structures maintained by the participant (via local invariants) and to prove the presence of an event on the snapshot. The example in Fig. 3.4 illustrates that. Line 1 invokes the library function `CreateNonce`. Its regular result `na` is the generated nonce; the additional ghost result `naT` is the corresponding term. `CreateNonce` takes an instance of the reusable verification library `rvlib` (cf. Sec. 3.5) as ghost argument, which allows the function to append to the global trace and update the local snapshot. Thus, its postcondition expresses the existence of the $CreateNonce(naT)$ event on the updated local snapshot. This postcondition allows the caller to prove the assertion in line 2, where `rvlib.Snap()` returns the local snapshot. The $CreateNonce(naT)$ event is the last event on this local snapshot as the local snapshot is not updated when other participants or the attacker add events to the global trace.

3.4 Proving Security Properties

In this section, we show how to define a trace invariant that lets us verify two important security properties, authentication and secrecy. Authentication means that two protocol participants are indeed communicating with each other and (depending on the particular authentication property) agree on some common values. Secrecy holds if confidential data remains unknown to the attacker. While we focus here on the proof techniques for these two standard properties, our methodology is also applicable to more complex properties such as forward secrecy, as demonstrated in Sec. 3.6.3.

3.4.1 Authentication

To prove authentication, we use protocol-specific events to record additional information beyond the exchanged messages, so that authentication properties can be expressed in a familiar way: as correspondence between these events. In this subsection, we show how to use trace invariants expressed in separation logic to prove two strong and common authentication properties: non-injective and injective agreement.

We illustrate our methodology using the NSL example from Fig. 3.2. We prove authentication using four protocol-specific events: Before sending the first message, Alice creates event $Initiate(Alice, Bob, na)$ to record

```
1  let commit = FinishA(A,B,na,nb) in
2  t.Occurs(commit) ⟹
3  let prefix, i = t.GetPrefix(commit) in
4  (prefix.Occurs(Respond(A,B,na,nb)) &&
5    !(∃A′,B′,nb′,i′. i != i′ &&
6      t.OccursAt(FinishA(A′,B′,na,nb′),i′))
7  ) || prefix.IsCorrupted({A, B})
```

Figure 3.5: Non-injective (white background) and injective (all lines) agreement from Alice's perspective with Bob on the nonces `na` and `nb`.

```
1  match ev {
2    case FinishA(A, B, na, nb):
3      UniWit(FinishA, na) &&
4      (prefix.Occurs(Respond(A, B, na, nb)) ||
5        prefix.IsCorrupted({A, B}))
6    ...
7  }
```

Figure 3.6: A simplified fragment of the trace invariant for NSL-specific events. This invariant is universally quantified over the events `ev` occurring on the trace; `prefix` is the trace prefix up to event `ev`. The highlighted line includes a separation logic resource to express that the *FinishA* event is unique w.r.t. to the nonce *na*, which allows us to prove injective agreement.

that she wants to communicate with Bob, and use the nonce *na* in the current protocol session. After receiving the first and before sending the second message, Bob in turn creates event $Respond(Alice, Bob, na, nb)$, indicating the communication partners and used nonces. Finally, the events *FinishA* and *FinishB*, with the same parameters as *Respond*, indicate successful completion of the protocol (i.e., runtime checks such as nonce comparisons succeeded) for Alice and Bob, resp. We focus on Alice's perspective in the following. We prove authentication for Bob's perspective in Sec. 3.6.3, where we also discuss authentication properties for WireGuard.

[79]: Lowe (1997), *A Hierarchy of Authentication Specification*

Non-Injective Agreement. The fact that Alice agrees with Bob on the nonces *na* and *nb*, known as *non-injective agreement* [79], is specified in Fig. 3.5 (ignore the conjunct highlighted in blue for now). This trace-based property states that if a *FinishA* event occurs on the trace (line 2) then either a *Respond* event with matching arguments occurs earlier on the trace (line 4) or one of the participants has been corrupted before an agreement was reached (line 7) as `t.Occurs(e)` yields whether event `e` occurs on trace `t`; `t.GetPrefix(e)` returns `t`'s prefix up to and including the most recent occurrence of `e`, and the index of that occurrence (i.e., the length of `prefix` minus 1). `t.OccursAt(e,i)` expresses that event `e` occurs at index `i` on trace `t`.

To prove agreement for NSL, we include the NSL-specific property from Fig. 3.6 (ignore line 3 for now) into the trace invariant. It states that for every *FinishA* event, a corresponding *Respond* event with matching arguments occurred prior on the trace, or one of the participants has been corrupted. Maintaining this invariant requires us to show the occurrence of a suitable *Respond* event (or of corruption) when Alice creates the *FinishA* event.

We discharge this proof obligation by extending the trace invariant with a message invariant for NSL's second message, which requires that the *Respond* event occurs on the trace or the message comes from the attacker. Hence, an implementation for Bob has to create a *Respond* event before sending the second message. When Alice receives the message, she may assume its message invariant (as part of the trace invariant). Since her local snapshot gets updated upon the receive-operation, the received message is recorded on the local snapshot and the message invariant becomes part of Alice's stable knowledge. So when Alice adds the *FinishA* event to the trace, she knows that either the *Respond* event occurs on the trace, or the second NSL message comes from the attacker. In the latter case, Alice can derive that corruption must have occurred because the attacker was able to construct a message containing the nonce *na*, which is accessible only to Alice and Bob (unless corrupted).

Once we established the trace invariant, it remains to show that for all traces, the invariant from Fig. 3.6 implies non-injective agreement (Fig. 3.5). This proof is a standard entailment check, which is performed automatically by program verifiers.

Injective Agreement. The stronger property *injective agreement* holds only for implementations that detect if the attacker replays messages from other protocol sessions. If successful, such a replay attack could trick participants into reusing outdated nonces (in general, key material), thereby weakening security. Proving injective agreement modularly is challenging; to the best of our knowledge, we present here the first invariant-based verification technique for injective agreement in protocol implementations (see also Sec. 3.11).

The highlighted conjunct in Fig. 3.5 strengthens non-injective to injective agreement by mandating that there is no second *FinishA* event on the trace with the same nonce na. The uniqueness of the event/nonce-pair enforces a one-to-one correspondence between *Respond* and *FinishA* events and, thus, excludes replay attacks.

To prove injective agreement, we strengthen our trace invariant to imply this property. We could in principle include a conjunct that specifies uniqueness by quantifying over the indexes into the trace. However, such an invariant would be difficult to maintain *modularly*. The necessary proof obligation for adding a *FinishA* event would require that no such event with the same first nonce already exists on the trace. However, each participant has only *partial* information about the trace stored in its local snapshot. So even if we proved the absence of an event on the local snapshot, we could not conclude its absence on the trace, such that the proof obligation cannot be discharged.

To obtain a modular verification technique for injective agreement, we leverage separation logic's permissions to encode arbitrary linear resources (non-duplicable facts). Due to the meaning of separating conjunction, $p \mapsto \_ \star p \mapsto \_$ is equivalent to false (because the permissions of the two conjuncts are not disjoint). That is, the points-to assertion $p \mapsto \_$ is a *non-duplicable (i.e., unique) resource*. We can use this fact to model the uniqueness of an event by representing the event as a separation logic permission. We use this mechanism as follows.

Conceptually, we tie event uniqueness to nonces because nonces are, by assumption of perfect cryptography, unique. When a protocol-specific event is declared, it can be specified as unique w.r.t. a specific nonce parameter. E.g., in NSL, event *FinishA* is unique w.r.t. its third parameter *na*. Subsequently, when a nonce is generated via a call to our verification library, a program annotation states for which events this nonce will be used (e.g., *FinishA*). The library call returns not only the fresh nonce (*na*), but also a linear resource for each indicated event type (technical details follow in Sec. 3.5).

This resource—called an event's *uniqueness witness*—then needs to be given up when the corresponding event is appended to the trace. That is, ownership of the resource is transferred from the participant to the ghost lock by conjoining the resource to the trace invariant. E.g., for NSL, Alice obtains the witness $\mathit{UniWit}(\mathit{FinishA}, \mathit{na})$ when creating nonce *na*. This witness is transferred to the trace invariant when she appends the event $FinishA(\_, \_, na, \_)$ to the trace, as expressed by the highlighted conjunct in Fig. 3.6, where && is interpreted as separation logic's separating conjunction $\star$. Due to the linearity of the resource, any attempt to append another *FinishA* event with *na* would fail to verify because the required witness cannot be provided a second time, which would be necessary to preserve the trace invariant.

Consequently, the invariant from Fig. 3.6 implies that the *FinishA* event with *na* is unique, which allows a standard separation logic verifier to prove the highlighted conjunct in the definition of injective agreement (Fig. 3.5).

Our discussion shows how the combination of a global trace and local snapshots allows us to prove authentication modularly, and how we can leverage the expressive power of separation logic to specify a trace invariant that lets us prove injective agreement.

3.4.2 Secrecy

Secrecy of a term *s*, e.g., a key or a nonce, states that the attacker does not learn this term except when corrupting one of the protocol participants that know the term. We can express secrecy as a property of the global trace because we can extract both the attacker knowledge and corruption events from the trace.

[39]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*
[67]: Bhargavan et al. (2010), *Modular Verification of Security Protocol Code by Typing*

Instead of directly reasoning about the concrete attacker knowledge, we follow Bhargavan *et al.* [39, 67] by over-approximating the concrete attacker knowledge to classes of terms that the attacker (possibly) knows. This over-approximation enables modular reasoning about secrecy: we impose proof obligations that prevent secrets from being leaked to the attacker by checking for every send operation that the sent message belongs to a class already known to the attacker. For instance, if a participant tried to send a (unencrypted) secret term over the network, the send operation would be rejected by the verifier. Consequently, sending a message does not change the over-approximated attacker knowledge. This knowledge is extended only when the attacker corrupts a participant or session. In this case, we add the class of terms readable by the corrupted participant or session to the knowledge.

We classify terms based on their allowed recipients by assigning them a *secrecy label*. Secrecy labels range from public (i.e., everyone including the attacker) over a set of participants to a set of particular protocol sessions. The latter is useful to classify ephemeral private keys, e.g., in our WireGuard case study, because only a participant running a particular protocol session is allowed to read these keys.

By proactively enforcing secrecy labels, we ensure that the (concrete) attacker knowledge may contain only public terms and terms whose secrecy label contains a participant or protocol session that is allowed to read the term and that has been corrupted in the past. We prove this so-called *secrecy lemma* once and for all as part of our reusable verification library (cf. Sec. 3.5).

3.5 Reusable Verification Library

We implement our methodology as a reusable verification library, which significantly reduces the verification effort per protocol: the library encap-

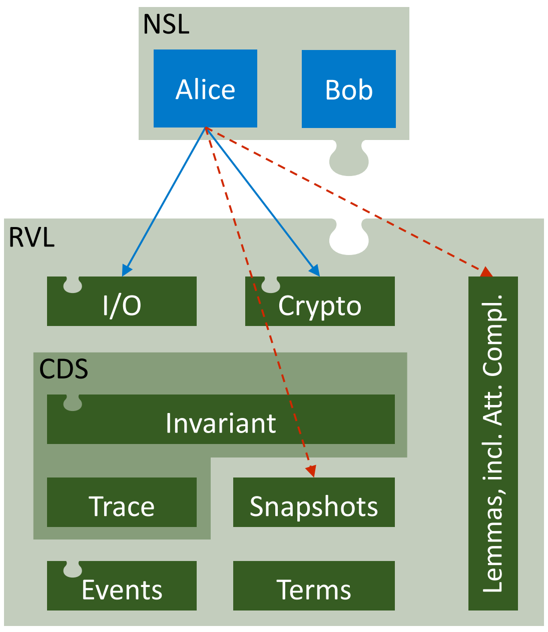

Figure 3.7: Structure of our reusable verification library (RVL). The library provides implementations for the abstractions used in our methodology: terms, events, the global trace, and local snapshots. Both the trace and all local snapshots are governed by the trace invariant. The trace is encapsulated inside a concurrent data structure (CDS) that permits shared access. The APIs for I/O and cryptographic operations apply these operations and also register the corresponding events on the trace. The RVL also provides several lemmata that have been proved for all protocols, e.g., attacker completeness. Many components of the library are parametric to accommodate protocol-specific events and invariants (and the corresponding preconditions for the I/O and crypto API). We indicate parametric components using a tab symbol near the top of the box. The parameters are supplied for a concrete protocol (here, NSL), as indicated by the tab at the bottom of the box.

sulates the global trace and provides a convenient application programming interface (API) for common network and cryptographic operations that automates trace updates. In addition, the library provides various lemmata, such as attacker completeness (Sec. 3.2), which are proved once and hold for all protocols. To enable verification of a wide range of protocols, the global trace is parametric in the events it records, and the trace invariant is parametric to account for protocol-specific properties.

To demonstrate that our methodology is widely applicable, we developed a library for the Go verifier Gobra [34], and one for the C verifier VeriFast [26]. Both library implementations are available in our open-source artifact [101]. In this section, we give an overview of the library and highlight some of its technical solutions.

[34]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*

[26]: Jacobs et al. (2011), *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*

[101]: Arquint et al. (2023), *A Generic Methodology for the Modular Verification of Security Protocol Implementations*

3.5.1 Overview

In the following, we describe the library's structure and components, explain how the library can be instantiated for different protocols, and provide data on its size and verification time.

Components. Fig. 3.7 illustrates the structure of our library (lower box) and a protocol implementation that uses it (upper box). The library provides the abstractions introduced in Sec. 3.3: terms and events abstract over concrete data structures (e.g., byte arrays) and participant operations, respectively. Events are recorded on the global trace, whose content is constrained by the trace invariant. The concurrent data structure (CDS) fully encapsulates the trace, to govern shared access and maintain the invariant. Local snapshots are prefixes of the global trace but are owned locally by the protocol participants.

The library also provides a convenient API for common network I/O and cryptographic operations: each function performs the corresponding concrete operation (e.g., sending a message or creating a nonce) and also adds the corresponding event to the trace. Suitable preconditions ensure that the operation preserves the trace invariant; they lead to proof obligations for clients using the API. Clients typically discharge these

Figure 3.8: Excerpt of the parametric trace invariant, defined via pattern matching over individual trace entries. All cases may refer to earlier events on the trace via the prefix parameter `pre`. The case for a *Send* event enforces the message invariant, which is partly defined by the library, but itself parametric. A `PEvent` represents any protocol-specific event `pe`. The corresponding case of the trace invariant comes entirely from the protocol parameter `P`.

```
pred TraceInv[P](t: Trace) {
  foreach e: Entry of t:
    let pre = ... in // trace prefix up to e
    match e {
      case Send(msg):
        MsgInv[P](msg, pre)
      case PEvent(pe):
        P::PEventInv(pe, pre)
      ...
    }
}
```

with the help of stable knowledge about the trace, which is recorded in their local snapshots.

In terms of cryptographic operations, our library currently offers asymmetric encryption, authenticated encryption with associated data (AEAD), signatures, and modular exponentiation, but can easily be extended by additional cryptographic operations. As a reference, adding the latter two features and proving the corresponding lemmata took about two person days.

Note that almost the entire library consists of ghost code that is used for verification, but will be erased by the compiler. The only non-ghost operations are the calls to the underlying I/O and cryptographic libraries. This has two important consequences. First, these calls can be inlined in the participant implementation, such that the entire library can be removed from the executable program and does not cause any runtime overhead. Second, existing protocol implementations do not have to be modified to use the library. The library provides a convenient way to systematically annotate an implementation with ghost code and proof obligations, but other forms of annotations are also possible.

Parametricity. As we discussed earlier, some events and aspects of the trace invariant (and consequently the preconditions of the I/O and cryptographic API) are protocol-specific. To capture them, we designed our library to be parametric, such that clients using the library can instantiate it for a given protocol.

Despite being parametric, our library nonetheless provides lemmata that are proven once and for all protocols, in particular, *attacker completeness* (Sec. 3.2) and the *secrecy lemma* (Sec. 3.4.2). Attacker completeness can be proved once and for all because the library is not parametric in the kinds of term abstractions it provides. The secrecy lemma directly follows from the protocol-independent parts of the trace invariant, which enforce for all protocols that implementations do not leak secrets to the attacker, i.e., messages have to be public. The library provides also several utility lemmata (e.g., that event existence is a stable trace property) that can be used when verifying a participant implementation.

Fig. 3.8 shows a small excerpt of our trace invariant. The parameter `P` provides protocol-specific events and invariants. Besides various properties of the entire trace (not shown in the figure), the trace invariant also includes event-specific invariants. We show here the invariants for *Send* events and protocol-specific events. A *Send* event requires the message invariant, which itself can be parameterized by library clients. We prove that the generic part of the message invariant is weak enough to be preserved by the attacker; it states, in particular, that the terms occurring in the message do not leak secrets. The protocol-specific part of the message invariant may constrain only encrypted data and must allow

Library	LOC	LOS	Verification time [s]
Go/Gobra	85	7688	115.0
C/VeriFast	343	3837	1.1

Table 3.1: Verified lines of code (LOC) and lines of specification (LOS) (incl. ghost code) for the library, together with the average verification times in Gobra and VeriFast.

the possibility that the encrypted data was fabricated by the attacker out of terms in the attacker knowledge. This ensures that it is maintained by all attacker actions. For a protocol-specific event, the invariant is supplied entirely by the parameter P. In the following, we explain how this parameter is represented in our library implementations.

In the Go implementation of the library, we achieve parametricity by using Go interfaces. In particular, the *generic protocol interface* declares mathematical functions (e.g., *isUnique* to indicate that an event is unique), separation logic predicates (e.g., protocol-specific event invariants), and lemmata. Clients may then supply different implementations of this interface with different definitions for these functions, predicates, and lemmata. Gobra checks via suitable proof obligations that any concrete implementation satisfies key properties specified in the interface (e.g., that protocol-specific invariants provide uniqueness witness resources for unique events). These properties can thus soundly be assumed while verifying the parametric library. Analogously, parametricity w.r.t. events is enabled by declaring an *Event* interface that protocol-specific events extend.

In VeriFast, we use its generic types (e.g., for events), abstract mathematical functions (e.g., *isUnique*), and abstract lemmata (e.g., that the event invariant is stable) to achieve parametricity and verify the library once for all protocols. When verifying implementations of a particular protocol, these abstract functions and lemmata are concretized by providing function and lemma definitions via an automated syntactic transformation. We prove that these definitions are not present while verifying the library, that is, we indeed verify the parametric version of the library, not a concrete instantiation.

Statistics. Tab. 3.1 shows the size and verification time for the two verified implementations of our library. As explained above, the library consists mostly of ghost code; only around 1 % is executable code. All methods and lemmata together are verified in ca. 2 min. The library for VeriFast is currently less complete than the one for Gobra, and lacks several useful lemmata, which explains the smaller amount of ghost code. It verifies in 1.1 s (VeriFast is usually faster than Gobra, but provides less automation). We have measured the verification times by computing the 10 % Winsorized mean of the wall-clock runtime across 10 verification runs on a 2023 Apple MacBook Pro with M3 Pro processor and macOS Sequoia 15.6. Since the library is verified once for all protocols, this effort does not have to be repeated when verifying a concrete protocol implementation.

3.5.2 Technical Solutions

In the following, we summarize the features of a verification technique and tool required to implement the main abstractions (e.g., terms, events, global traces) provided by our library.

Custom Mathematical Theories. Verification techniques frequently represent information as values of mathematical theories, such as sets, tuples, and sequences. In contrast to the corresponding data types of a programming language, these values are immutable and their operations have a direct representation in the verification logic, which simplifies reasoning.

We use mathematical theories to represent the abstractions we use in specifications and ghost code: events, the global trace, secrecy labels, and terms with equational theories. Conceptually, events form an algebraic data type (ADT), as does the global trace (a functional list). Labels and terms are also algebraic structures, but with additional properties (e.g., labels have a commutative join operator).

For secrecy labels, we axiomatize a `GetLabel` function that maps a term to its secrecy label. This axiomatization abstracts terms to secrecy labels, and models the symbolic model of cryptography. Therefore, this axiomatization has to be consistent with the equational theories of terms.

The Gobra implementation of the library represents all these structures as uninterpreted functions with appropriate axioms (analogous to how custom theories are encoded to SMT solvers). E.g., for the ADT of events, we define axioms that ADT constructors are injective in their arguments, and that different constructors produce different events. For terms, we define additional axioms to encode cryptographic equational theories, e.g., ${g^x}^y = {g^y}^x$, where g^x denotes DH exponentiation with generator g. VeriFast supports ADTs natively, which we use to represent events and the global trace. For labels and terms, we again use uninterpreted functions and axioms ("auto-lemmata") to express equational theories.

Linear Resources. Our novel support for proving injective agreement (cf. Sec. 3.4) requires reasoning about the uniqueness of certain protocol-specific events. For this purpose, we introduce (ghost) memory locations and use separation logic's (exclusive) permissions to these locations as linear resources. Separation logic predicates [104] allow us to construct linear resources with arbitrary parameters by mapping the parameter tuples injectively to a heap location. We use such predicates to represent the uniqueness witnesses from Sec. 3.4.

[104]: Parkinson et al. (2005), *Separation Logic and Abstraction*

Concurrency Reasoning. As discussed in Sec. 3.3.3, we model the global trace as a *concurrent* data structure. Our approach is compatible with any verification technique that is able to reason about shared accesses to such a data structure and to maintain an invariant over it. Moreover, to encode local snapshots (cf. Sec. 3.3.4), we require support for reasoning about properties that are stable under concurrency, which are offered by separation logic verifiers.

We model the global trace as a data structure that is protected by a ghost lock. Neither Gobra nor VeriFast support ghost locks directly, but both offer standard locks. Reasoning about ghost locks and standard locks is almost identical, with one exception: Any non-ghost operations performed between acquiring and releasing a ghost lock must be atomic (because the lock will be erased by the compiler, so it does not actually provide mutual exclusion). This property is satisfied in our library.

3.6 Case Studies

We applied our methodology to Go implementations of the NSL public key protocol, signed DH key exchange, and the WireGuard VPN protocol,

```
 1 struct Alice  {
 2   SkA byte[]
 3   PkB byte[]
 4   Na  byte[]
 5   Nb  byte[]
 6   /*@ ghost Step uint @*/
 7   ...
 8 }

10 /*@ pred LocalInvariant(a: Alice) {
11   ∃naT,nbT.
12   ... && // memory omitted
13   (a.Step == 2 ==>
14     UniWit(FinishA, naT)) &&
15   (a.Step >= 2 ==>
16     γ(naT) == a.Na &&
17     a.Snap().NonceOccurs(naT)) &&
18   (a.Step >= 3 ==>
19     γ(nbT) == a.Nb &&
20     a.Snap().Occurs(FinishI(A, B, naT, nbT)))
21 } @*/
```

Figure 3.9: The struct used for Alice's local state in the Go implementation of NSL, and an excerpt from the local invariant that relates this state to Alice's local snapshot and, thereby, to the global trace. The `Step` field is a ghost field that is used to track Alice's progress in the protocol.

and prove strong security properties. We also verified a C implementation of NSL, and obtained the same security properties as for the Go implementation. Our case studies (included in our artifact [101]) thus demonstrate the portability of our methodology across different protocols, programming languages, and verifiers, and its scalability to realistic, interoperable implementations. In this section, we first summarize each of the case studies and then discuss our experiences.

[101]: Arquint et al. (2023), *A Generic Methodology for the Modular Verification of Security Protocol Implementations*

3.6.1 Needham–Schroeder–Lowe (NSL)

We used Gobra to verify a Go implementation of the initiator and responder roles for the NSL protocol (cf. Fig. 3.2), and likewise VeriFast for a C implementation thereof. We implemented the core of the protocol as one method per participant; we also verified an alternative Go implementation of the initiator that contains one method per message to demonstrate that verification is not sensitive to the code structure. Both protocol roles store their program state locally and use an invariant to relate the local state via the term abstraction to their local snapshot and, thereby, to the global trace.

Fig. 3.9 illustrates the interplay between the local state and the local snapshot for the initiator, Alice. Alice manages her program state in a struct `Alice`. The local invariant in lines 10–21 relates Alice's local state to her local snapshot (and, thus, indirectly to the global trace). This invariant expresses ownership of the heap locations for the struct fields, which is omitted in the figure. More importantly, it specifies properties about the struct fields depending on Alice's progress within the protocol execution, which we keep track of via the `Step` field. E.g., Alice is in Step 2 after creating the nonce *naT* and sending the first message. In this case, the invariant includes the uniqueness witness (line 14), which allows Alice to create the *FinishI* event in a later protocol step. The invariant relates the concrete nonce field `Na` to its term representation `naT` using the concretization function γ (line 16). This term is used in the events on the global trace. In particular, the *CreateNonce* event for `naT` must occur on Alice's local snapshot `a.Snap()` (line 17) and, thus, on the global trace. Once Alice's protocol run has reached the final Step 3, it adds the *FinishI* event to the trace. The invariant reflects this by stating that the event

Figure 3.10: The signed DH key exchange protocol, where g^x and g^y are DH public keys and $\{\!|m|\!\}_{sk}$ denotes cryptographically signing a payload m with a secret key sk.

$$\begin{aligned}
&M1. \quad A \rightarrow B : \ g^x \\
&M2. \quad B \rightarrow A : \ \{\!|\langle 0, B, A, g^x, g^y \rangle|\!\}_{sk_B} \\
&M3. \quad A \rightarrow B : \ \{\!|\langle 1, A, B, g^y, g^x \rangle|\!\}_{sk_A}
\end{aligned}$$

is on the local snapshot (line 20). Knowledge about *FinishI*'s existence on the trace entails (via the trace invariant) properties about the *Respond* event created by Bob (recall Fig. 3.6). This knowledge, together with *FinishI*'s uniqueness witness (now stored in the trace invariant), allows us to prove injective agreement with Bob as explained in Sec. 3.4.1.

We prove for all participant implementations that they achieve (at the end of a protocol execution) injective agreement on, and secrecy for, both nonces *na* and *nb*. Additionally, we verify initialization code that creates an empty trace, generates public/private key pairs for the participants, and spawns two participant instances as goroutines to demonstrate that key distribution (although not part of the protocol) can be modeled using our methodology.

3.6.2 Signed Diffie–Hellman (DH)

In the signed DH key exchange (cf. Fig. 3.10), Alice and Bob each generate a DH secret key x and y, respectively. By transmitting the corresponding (signed) DH public keys g^x and g^y, they agree on the shared key g^{x^y} after a successful protocol run.

We prove secrecy for, and injective agreement on, the shared key. The proof is similar to the proof for NSL, which allowed us to reuse substantial parts. One noticeable difference is that proving that both participants derive the *same* shared key requires the equational theory for DH exponentiation. Our reusable verification library provides such custom theories, as discussed in Sec. 3.5.2. Another difference is that the nonces x and y are not directly part of the protocol messages (in contrast to na and nb in NSL), but are existentially quantified in the message invariants. A participant instance can determine the values of these existentially-quantified variables after receiving a protocol message, by connecting the message invariant to its own DH secret key.

3.6.3 WireGuard

As our main case study, we have picked the WireGuard VPN protocol as a real-world protocol achieving even stronger security properties than NSL. WireGuard is a modern, open-source, and cross-platform VPN that uses state-of-the-art cryptography and is part of the Linux kernel. The WireGuard protocol, which performs an authenticated key exchange, has been analyzed rigorously [105, 106]. It consists of a handshake and transport phase. During the handshake phase, the protocol participants agree on two session keys k_{IR} and k_{RI}, one per direction, that are used to symmetrically encrypt VPN packets in the transport phase.

[105]: Dowling et al. (2018), *A Cryptographic Analysis of the WireGuard Protocol*
[106]: Lipp et al. (2019), *A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol*

Implementation. We used the existing Go implementation from Chapter 2, whose memory safety proof we reused. Thanks to our reusable verification library's parametric design, instantiating our library with the concrete networking library used by the WireGuard implementation was straightforward and only required annotating cryptographic functions with suitable postconditions.

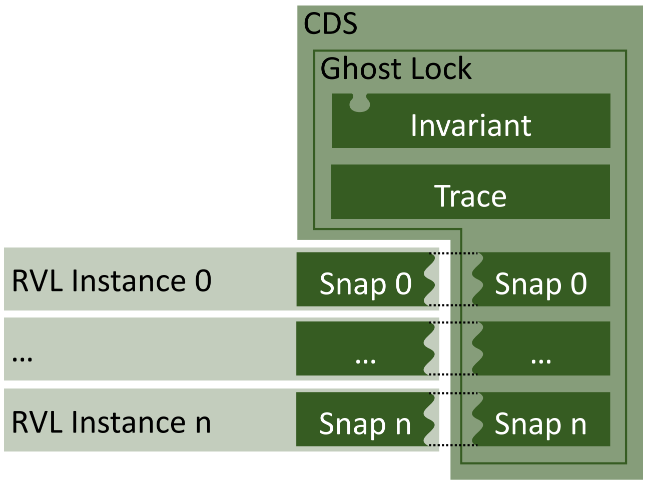

Figure 3.11: Illustration of the permission distribution. The ghost lock within the concurrent data structure (CDS) owns via its lock invariant full permissions (dark green boxes in full) to the global trace and $^1/_2$ permissions to every local snapshot (each visualized as a truncated box). Each local snapshot conceptually belongs to a protocol role and session, or more precisely to an instance of the reusable verification library (RVL). The CDS uses a mathematical map (omitted) to keep track of local snapshots by mapping from an RVL instance identifier to a local snapshot. Each RVL instance has access (omitted) to the CDS to perform operations that require acquiring and releasing the ghost lock, and owns $^1/_2$ permissions to its own local snapshot.

As explained in Sec. 2.6.4, this implementation is a subset of WireGuard's official Go implementation. It omits advanced VPN features such as protection against denial of service (DoS) attacks, session key renewal, and support for multiple concurrent VPN connections. Moreover, their implementation reduces concurrency (which we partly re-introduced, as we discuss below), and replaces a message buffer pool by single-use buffers. Our technique could handle the removed features with additional effort that is mostly orthogonal to our methodology. For instance, the implementation of DoS protection collects metrics (which does not pose a challenge for program verification) and uses a slightly different handshake (whose verification is analogous to the standard handshake; the differences are not relevant for authentication and secrecy). Supporting multiple VPN connections requires slightly more complex data structures, as do buffer pools, which can easily be handled in separation logic. Despite these simplifications, the implementation is interoperable with other WireGuard implementations and supports tunneling Internet Protocol (IP) packets via the established VPN connection to and from the operating system. Since each IP packet is encrypted using a distinct counter value, a new handshake must be performed before the counter reaches its upper limit, which is not yet implemented. Instead, the implementation stops forwarding IP packets at that point.

Our case study goes substantially beyond of Chapter 2's, which focuses on connecting Tamarin to code-level verification, and proves weak forward secrecy and non-injective agreement in the presence of long-term key corruption. We additionally consider session corruption, i.e., the possibility for an attacker to obtain ephemeral key material, and prove *strong* forward secrecy and *injective agreement with actor key compromise (AKC) security*. Furthermore, we have re-introduced (from the official WireGuard implementation) and verified the ability to send and receive transport messages in the initiator *concurrently*. This change increases Transmission Control Protocol (TCP) throughput compared to Chapter 2's implementation by a factor of 180, which illustrates how important such code optimizations are for real-world protocol implementations. The initiator verified in our work reaches 72 % of the official implementation's throughput; the additional concurrency needed to close the remaining performance gap requires standard concurrency reasoning in separation logic, which is supported by our methodology.

Verifying Concurrent Implementations

Our reusable verification library can be used in two different ways to verify a concurrent implementation. First, one can treat an instance of this library as representing a protocol run, which implies that one has to perform all trace-relevant operations by *all* threads of this protocol run with this single library instance. While possible, this approach is not very convenient for concurrent implementations, as it requires appropriate synchronization for every library access performed by the threads.

Second, we can instantiate the library once for each thread, which is the approach we took in our WireGuard case study. This second approach exploits the fact that we are using a concurrent data structure (CDS) to represent the global trace and consider all possible interleavings of trace events added by all library instances. Thus, extending the library to support multiple library instances per protocol run was straightforward as we added only a thread identifier to distinguish these library instances and their corresponding local snapshots. Prior to explaining how we use multiple library instances in an implementation, we need to introduce the internal representation and permission management of our CDS, as shown in Fig. 3.11. Our CDS internally uses a ghost heap location to store the global trace (cf. Sec. 3.3.3) and a ghost mathematical map mapping library instance identifiers to ghost heap locations storing their corresponding local snapshots (Sec. 3.3.4). To synchronize accesses to these ghost heap locations, the CDS employs a ghost lock, whose lock invariant owns full permissions to the heap location storing the global trace and to the trace invariant, and $^1/_2$ permissions to every heap location storing local snapshots. A library instance holds the remaining $^1/_2$ permissions to the heap location storing its own local snapshot and the necessary permissions to acquire the CDS's ghost lock. To enable a single protocol run to create multiple library instances, each such instance requires its own local snapshot in the CDS's ghost map, which we achieved by extending the ghost map's key from a tuple of protocol role and session identifier to a triple of protocol role, session identifier, and an optional[4] thread identifier.

4: The thread identifier is optional to ensure backward compatibility, i.e., single-threaded implementations do not have to provide such an identifier.

In our WireGuard case study, we create two library instances per protocol run for the initiator role[5]. During the handshake phase, only one of the two instances is used to generate ephemeral secret keys and derive the session keys. At the beginning of the transport phase, we launch a goroutine to concurrently take IP packets from the operating system and send them over the established VPN connection. Conversely, the main goroutine continuously receives and processes incoming VPN packets. Since we continuously prove the subsequently explained secrecy and authentication properties after processing every outgoing and incoming packet, both goroutines require not only a session key but also precise knowledge about a session key's secrecy label and events on the global trace. Since one of the two library instances was not used during the handshake phase, it does not have the necessary knowledge and, thus, cannot be immediately used in either goroutine. In particular, this instance's local snapshot is highly outdated and does not reflect the trace knowledge acquired during the handshake phase.

5: Our reusable verification library does not allow to dynamically add additional entries to the ghost map and, thus, we have to account for all library instances at initialization time. However, this restriction could easily be lifted by using a ghost map that supports concurrently inserting entries whose keys are not yet contained. This side-condition ensures that existing map entries remain stable.

We solve this problem with a simple synchronization mechanism that allows us to copy a local snapshot from one library instance

```
1 !t.AttackerKnows(s) ||
2   t.GetHs(ASess, PSess).IsCorrupted({ A,  P}) ||
3   t.IsSessionCorrupted({ASess, PSess})
```

Figure 3.12: Strong (without highlighted part) and weak forward secrecy (entire property) for a session key `s` on trace `t`.

to another. We implemented this mechanism using minimal proof obligations, meaning that the *new* local snapshot for a library instance does not need to satisfy any constraints w.r.t. to this instance's *old* local snapshot. In particular, the old local snapshot is not required to be a prefix of the new local snapshot[6]. For WireGuard, this mechanism allows us to copy the local snapshot of the library instance that is used during the handshake phase to the second library instance providing all acquired knowledge about the global trace and the session keys to this other library instance. Hence, we can use the second library instance, which is used in the goroutine sending VPN packets, as if we had already used this instance during the handshake phase and prove the security properties as expected.

6: The CDS's invariant enforces, however, that all local snapshots are prefixes of the *global* trace, which is the case for the *new* local snapshot since it is a copy of another library instance's local snapshot.

Security Properties. Since the session keys are based on ephemeral as well as long-term key material that is contributed by both protocol participants, WireGuard achieves strong security properties. In particular, we prove forward secrecy and injective agreement, both with AKC security. While WireGuard optionally incorporates a pre-shared symmetric key into the handshake to increase security, we prove all security properties in this section without considering this pre-shared key, i.e., we treat the pre-shared key as a term known to the attacker. In the following, we call the initiator *actor* and the responder *peer* when proving a property from the initiator's perspective, and vice versa for the responder's perspective.

Forward secrecy protects sessions against future corruption of the long-term secret keys. I.e., an attacker cannot compute the session keys of an already established session after learning the long-term secret keys. However, sessions that get established after corrupting the long-term secret keys are not protected because the attacker can impersonate participants by knowing their secret keys. The literature distinguishes between weak and strong forward secrecy. We were able to reuse formalizations from existing work [11, 41, 107], which are phrased as trace-based security properties and, thus, directly supported by our methodology.

[11]: Cremers et al. (2017), *A Comprehensive Symbolic Analysis of TLS 1.3*
[41]: Ho et al. (2022), *Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations*
[107]: Girol et al. (2020), *A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie–Hellman Protocols*

Weak forward secrecy for a session key *s* holds if at any point in time, one of the following three properties hold: (1) The attacker does not know *s* (line 1), (2) the actor or its peer has been corrupted before completing the handshake (line 2), or (3) the actor's or peer's session has been corrupted (line 3). In the last case, the attacker gets to read the long-term and short-term state of the corrupted participant, that is, the long-term secret key and also the session keys if the session is established. Hence, the attacker either directly obtains the session keys if the session is already established or otherwise uses the long-term secret key to impersonate the actor or its peer while establishing a session in the future. The session keys of all other sessions remain secret. Fig. 3.12 (entire figure) shows the definition of weak forward secrecy, where `A` and `P` identify the actor and peer that derive the session key `s` in their protocol sessions `ASess` and `PSess`, respectively. `t.GetHs(ASess, PSess)` returns a prefix of trace `t` up to and including the corresponding handshake's completion from the actor's perspective. The session key `s` is protected against (future) participant corruption after the handshake's completion.

Compared to weak forward secrecy, session keys satisfying *strong forward*

```
let commit = Commit(A,P,ASess,PSess,m) in
let running = Running(A,P,ASess,PSess,m) in
t.Occurs(commit) ⟹
let prefix, i = t.GetPrefix(commit) in
(prefix.Occurs(running) &&
  !(∃A',P',ASess',PSess',i'. i != i' &&
    t.OccursAt(Commit(A',P',ASess',PSess',m),i'))
) || prefix.IsCorrupted({P})
  || prefix.IsSessionCorrupted({ASess})
```

Figure 3.13: Injective agreement with AKC security on a term `m` from the actor `A` 's perspective with a peer `P`. The highlighted conjunct indicates the `Commit` event's uniqueness requirement for the given `m`.

secrecy are additionally protected against corrupting the actor, i.e., the highlighted actor is removed from line 2 in Fig. 3.12. In particular, having access to the actor's long-term secret key does not allow the attacker to obtain the established session keys. This resilience has been formalized as actor key compromise (AKC) by Basin *et al.* [108], generalizing the more widely known notion of key compromise impersonation (KCI).

[108]: Basin et al. (2014), *Actor Key Compromise: Consequences and Countermeasures*

From the initiator's perspective, WireGuard guarantees strong forward secrecy for the two session keys once the handshake has been completed. In contrast, the responder guarantees only weak forward secrecy by the end of the handshake, but achieves strong forward secrecy after receiving the first transport message. We verified strong forward secrecy at the appropriate points in the protocol for both roles.

The responder's forward secrecy guarantee is strengthened by receiving and successfully processing the first transport message because this message acts as a key confirmation. I.e., the responder checks that it derived the same session key k_{IR} as the initiator, which allows the responder to detect AKC attacks. Based on strong forward secrecy for the session keys, we further prove that the VPN payloads are treated with the same level of secrecy. This induces proof obligations that a participant sends VPN payloads to the network in a way that they can be read only by participants allowed to read the session keys (e.g., by encrypting the VPN payloads with one of the session keys).

Confirming the session keys not only enables strong forward secrecy for the session keys but also provides additional authentication guarantees: *Injective agreement with AKC security* (cf. Fig. 3.13) states that (1) an actor A agrees with a peer P on a term m with a one-to-one correspondence between the *Commit* and *Running* events unless (2) the actor's session or (3) the peer's (short-term or long-term) state has been corrupted. In particular, corrupting the actor is not sufficient to satisfy this property. In contrast, the NSL protocol only satisfies injective agreement *without* AKC security from the initiator's perspective (as presented in Sec. 3.4.1) because having access to the initiator's secret key enables the attacker to decrypt the second message, obtain the nonces na and nb, and construct a modified second message containing na and nb' with $nb \neq nb'$. Thus, there is no correspondence between *Commit* and *Running* events in the case of actor key compromise because the initiator and responder do not agree on the nonces.

3.6.4 Discussion

For each case study, Tab. 3.2 reports the size of the implementation and its specification, along with the verification time. We exclude the alternative NSL initiator implementation in Go, and the reusable verification library (recall Tab. 3.1). However, the specifications do include the (ghost) code instantiating our reusable verification library and applying the pattern

Case Studies	LOC	LOS	Verification time [s]
Go/Gobra			
NSL	207	1013	78.6
Signed DH	235	930	113.9
WireGuard	557	6339	220.8
C/VeriFast			
NSL	300	1014	4.8

Table 3.2: Verified lines of code (LOC) and lines of specification (LOS) (incl. ghost code) for our case studies, together with the average verification times in Gobra and VeriFast. We performed the measurements in the same way as in Tab. 3.1.

requirement: it amounts to 377, 373, and 1086 lines of specification (LOS) in Gobra for NSL, DH, and WireGuard, respectively, and 391 LOS in VeriFast for NSL.

Overall, the annotation overhead for Gobra ranges between 4.0 and 11.4 LOS per line of code, and is in the typical range for modular program verification. For example, Wolf *et al.* [34] report a ratio of 2.7 for a small example using concurrency in Gobra. VST [27], a separation logic-based verifier based on Coq, reports an average ratio of 13.9 for small C programs. Both works verify only memory safety and functional properties, but do not include any (arguably much more complex) security properties, whereas our numbers include safety and security. In comparison, Sec. 2.6 proves security properties for WireGuard in Gobra with a ratio of 7.6, in addition to requiring a Tamarin model of 292 lines and a Tamarin oracle implemented in Python, which ensures that Tamarin's proof search terminates. Counting the Tamarin model and oracle as specification, the overall ratio is 8.2 (and requires the use of three different languages). We will discuss the differences between our two methodologies beyond annotation overhead in Sec. 3.10.

[34]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*

[27]: Cao et al. (2018), *VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs*

The main challenge in our case studies was finding a sufficiently strong trace invariant to prove the presented security properties. For WireGuard, we had to find suitable message invariants such that the secrecy labels for the derived session keys k_{IR} and k_{RI} are sufficiently strong to prove weak and strong forward secrecy. These secrecy labels are related to the message invariants because the session keys are derived by an eightfold application of KDFs that factors in long-term and ephemeral, i.e., session-specific, DH key material that is either locally generated or received from the peer. Thus, each KDF application results in a new key with a secrecy label that depends on the secrecy labels of the input key material. To keep the annotations related to the secrecy labels in the implementation to a minimum, we have implemented a lemma for each KDF application that proves the result's secrecy label.

Moreover, the invariant for protocol-specific events has to be strong enough to prove injective agreement with AKC resilience. Our reusable verification library enables strengthening the proven authentication property from non-injective to injective by adding the uniqueness witness for each protocol-specific event. This allowed us to focus on finding a suitable invariant for non-injective agreement with AKC resilience first, and then strengthen the authentication property, which required less than 40 additional LOS.

After completing the proofs for sequential code, we re-introduced concurrency to the initiator's transport phase (recall Sec. 3.6.3). Extending the reusable verification library to allow multiple library instances per protocol run by adding the thread identifier, but without adding the synchronization mechanism as that already existed, and adapting the initiator's implementation and proof was done in an afternoon. This demonstrates that our separation-logic-based methodology enables security proofs that are robust w.r.t. nontrivial code changes.

3.7 Trust Assumptions

Our methodology allows us to prove strong security properties for implementations of security protocols. Like with all verification techniques, these proofs rely on several assumptions about the implementation and the execution environment.

We rely on the soundness of the used program verifier. Since our methodology is compatible with standard separation logic verifiers, we can mitigate this assumption by using a mature tool.

As is standard for symbolic cryptography, we assume perfect cryptographic operations (e.g., absence of hash collisions, or that ciphertexts do not leak any information). This includes that the secrecy labels' axiomatization is consistent with the equational theories, which express the properties of these cryptographic operations. We also do not verify that the implementations of the cryptographic primitives are functionally correct; while this is orthogonal to our work, our methodology could be combined with verified libraries like EverCrypt [109].

[109]: Protzenko et al. (2020), *EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider*

Furthermore, we assume that all *output* operations, i.e., sending messages, are reflected on the global trace by corresponding events, which is the case when using the I/O operations provided by our verification library. However, if an implementation uses, e.g., inline assembly or third-party libraries to send messages to the network, the global trace has to reflect these messages nonetheless. Omitting any other event does not affect soundness, only completeness.

Lastly, we assume that the protocol terms corresponding to the byte arrays in a participant's *initial* state, and those obtained from operations outside our library (e.g., read from a config file), are readable by at least that participant according to the terms' secrecy labels (recall Sec. 3.4.2). Otherwise, it would not be sufficient to model corruption of a participant by adding the class of terms readable by that participant to the attacker knowledge; the attacker could learn even more. For all terms a participant can obtain by interacting with our verification library (e.g., receiving messages, generating nonces, applying encryption), we prove in our library (via corresponding lemmata) that a participant can read these terms (and thus the terms leak as expected to the attacker in case of corruption).

3.8 Soundness

We sketch soundness of our methodology by showing that the global trace reflects all relevant protocol steps and, thus, any security property proved for the trace indeed holds for the protocol implementation. More specifically, given a distributed system of verified protocol implementations and an arbitrary attacker, the system's set of possible executions is a subset of the executions permitted by the trace invariant, which in turn is a subset of the executions that satisfy the desired security properties.

For this purpose, we define a core programming language covering all protocol-relevant operations (e.g., network I/O, cryptographic primitives), and those relevant for modeling an attacker (e.g., corruption). The language's operational semantics (Sec. 3.8.1) supports thread-local state and explicitly maintains a shared global trace. The thread-local state models the state of each participant and is manipulated via assignments in the participants' implementations. In contrast, the global trace is maintained automatically by our semantics and extended whenever a relevant protocol operation is executed. We then define a Hoare logic (Sec. 3.8.2) parameterized with a trace invariant to enable modularly verifying each participant implementation. The logic natively supports our methodology's local snapshots and the global trace. We prove that this logic is sound w.r.t. the operational semantics using a standard rule induction. Thereby, we obtain the guarantee that locally-verified participants, if composed with the attacker to a concurrent system, maintain the global trace invariant in all possible interleavings. Since the global trace reflects all relevant protocol steps, we can conclude that the aforementioned trace inclusion holds (Sec. 3.8.3). As one would expect, obtaining this global guarantee about the concurrent system from locally-verified participants turned out to be the most challenging step in the soundness proof.

Since we parameterize our Hoare logic with a trace invariant, secrecy labels and equational theories come into play only when verifying protocol role implementations and our reusable verification library against an instance of the trace invariant. Our reusable verification library instantiates the trace invariant to mandate, e.g., that a message may be sent only if its secrecy label is public, thereby ensuring that sending this message does not increase the attacker's over-approximated knowledge. Discharging this proof obligation when verifying a protocol role's implementation against our reusable verification library requires reasoning about equational theories and secrecy labels, whose axiomatization is trusted (cf. Sec. 3.5.2).

In each subsection, we relate the semantics defined for the proof sketch with the verification performed by an off-the-shelf separation-logic-based verifier (such as Gobra) against our reusable verification library. Formally connecting our dedicated Hoare logic to a standard separation logic is straightforward, based on the encoding discussed throughout the paper (using the heap to store the thread-local state, a ghost lock to synchronize access, and a lock invariant to constrain the trace and all local snapshots).

3.8.1 Language and Operational Semantics

On a high-level, we consider a distributed system consisting of multiple components: either instances of a protocol implementation, i.e., participants, or the attacker. Our programming language does not support user-defined shared variables or a heap, and each participant executes its commands in its own local state. However, security-relevant commands additionally mutate the global trace to reflect the performed operation.

Consequently, our system's configurations comprise a local configuration per component, and the global trace τ. A local configuration for a protocol participant i is characterized by its local command C_i and local state σ_i. The local configuration for the attacker is similar, but additionally contains a knowledge set k_a that stores all symbolic terms that are known to the attacker.

Definition 3.8.1 (Local Program States) *Local program states, ranged over by σ, are total functions from local variables (in the set PVars) to values (in the set PVals).*

$$PStates \triangleq PVars \rightarrow PVals$$

We define our programming language such that it directly works with symbolic terms instead of bytes, which avoids having to complicate the semantics to reflect the orthogonal issue of mapping between the bytes and terms.

Definition 3.8.2 (System Configurations) *A configuration of our distributed system has the shape*

$$\langle\langle C_1, \sigma_1\rangle, \cdots, \langle C_n, \sigma_n\rangle, \langle C_a, \sigma_a\rangle, k_a, \tau\rangle$$

where $\langle C_i, \sigma_i\rangle$ denotes the local command and local state of participant i, $\langle C_a, \sigma_a\rangle$ denotes the local command and local state of the attacker a, k_a is the attacker's knowledge set, and τ denotes the system's global trace.

Note that the local configurations of participants and the attacker have the same shape such that the same operational semantics rules, e.g., for sequential composition, are applicable to both. We achieve this common shape by keeping the attacker's knowledge set k_a separate from the attacker's local configuration $\langle C_a, \sigma_a\rangle$, even though only commands executed by the attacker possibly modify k_a.

Definition 3.8.3 (Programming Language) *We consider the following programming language, where C ranges over commands, x and $\vec{x}$ over variables and lists of variables in the set PVars, respectively, and e over expressions (modeled as total functions from PStates to PVals):*

$$\begin{aligned} C \triangleq\ & \textbf{skip} \mid C; C \mid \textbf{if}\ (e)\ \{C\}\ \textbf{else}\ \{C\} \mid \textbf{while}\ (e)\ \{C\} \mid \\ & x := e \mid send(e) \mid x := recv() \mid x := nonce() \mid \\ & x := hash(e) \mid x := pk(e) \mid x := enc(e, e) \mid x, x := dec(e, e) \mid \\ & drop(e) \mid learn(e) \mid x := choose() \mid corrupt(e) \mid \\ & \textbf{fork}\ (\vec{x})\ \{C\} \end{aligned}$$

Besides standard commands, such as sequential composition and assignment, the programming language provides several commands essential for protocol implementations: for sending and receiving a network message, for generating a nonce, hashing a term, generating a public key corresponding to a given secret key (*pk*), and encrypting and decrypting a term with a key. Additionally, the programming language provides commands only available to the attacker: dropping a message from the network, adding the value of a local variable to the attacker knowledge (*learn*), non-deterministically obtaining a term from the attacker knowledge (*choose*), and corrupting the state of specific participant (each participant has a unique id/index).

Finally, the fork command starts a new thread executing the provided command, which corresponds to spawning a new participant or the attacker. The new thread operates on its own local state, which initially maps the variables in $\vec{x}$ to the same values as the state in which the fork command is executed. This command is used to bootstrap the distributed system, as discussed in Sec. 3.8.3.

The expression language comprises symbolic terms for booleans and integers, and the usual operations thereon. We assume well-typed programs, e.g., that if-conditions are of type boolean.

Definition 3.8.4 (Operational Semantics) *Fig. 3.14 defines the small-step operational semantics for our programming language.*

The rules for standard commands such as sequential composition and conditionals, are as expected, and we will thus only discuss non-standard aspects of our programming language.

Global Trace. Recall from Sec. 3.3.3 that in our verification methodology (as implemented in Gobra), we use a concurrent ghost data structure with ghost locks to manage the global trace. In our operational semantics, we instead represent the trace as the dedicated element τ in the system's state. Regardless of the technical implementation, we must ensure three crucial properties: (1) Each operation may append only a single trace event. In our methodology, this is checked via a suitable proof obligation upon lock release; in our operational semantics, each rule adds at most one event. (2) To ensure monotonicity, the trace may only grow. Checked upon lock release in our methodology; in our operational semantics, no rule shortens the trace. (3) Every single operation must preserve the trace invariant. Checked upon lock release in our methodology; in our operational semantics, this is part of the soundness theorem (cf. Thm. 3.8.3).

Local Snapshots. Recall from Sec. 3.3.4 that each participant has a trace snapshot, which enables participants to maintain local invariants that depend on trace prefixes. To enable corresponding assertions in our program logic (Sec. 3.8.2), our operational semantics provides a local variable *snap* that is treated special in two ways: local states σ map *snap* to a sequence of trace events (not to a value in *PVals*), and program commands may not use (in particular, modify) *snap* (a straightforward syntactical constraint).

Projecting System Configurations. The Local and Attacker rules project a system configuration down to a participant- and attacker-local configuration, respectively. Except for the Corrupt and Fork rules, these projection rules simplify the definition of all other rules, as all other rules operate on either a participant-local or attacker-local configuration instead of on the full system configuration, depending on whether a command can be executed by participants and the attacker, or only by the attacker.

Network Messages. All operations modifying the network state, i.e., sending and dropping a message, are recorded on the global trace τ, and, thus, we can compute the set of receivable messages as follows:

Definition 3.8.5 (Messages on the Network)

$$msgs(\tau) \triangleq \{m \mid \forall m.\, Send(m) \in \tau \wedge Drop(m) \notin \tau\}$$

The function $msgs(\tau)$ is used in the Recv rule's side-condition to constrain the set of messages that can be received. Without loss of generality, this side-condition implies that we consider only non-blocking traces, i.e., *recv*() is invoked only if $msgs(\tau)$ is non-empty.

Nonce Freshness. The NonceGen rule's side condition captures our perfect cryptography assumption that generated nonces are always

$$\frac{\langle C_i, \langle \sigma_i, \tau \rangle\rangle \rightarrow \langle C_i', \langle \sigma_i', \tau' \rangle\rangle}{\langle \cdots, \langle C_i, \sigma_i \rangle, \cdots, k_a, \tau \rangle \rightarrow \langle \cdots, \langle C_i', \sigma_i' \rangle, \cdots, k_a, \tau' \rangle}\;(\textsc{Local}) \qquad \frac{\langle C_a, \langle \sigma_a, k_a, \tau \rangle\rangle \rightarrow \langle C_a', \langle \sigma_a', k_a', \tau' \rangle\rangle}{\langle \cdots, \langle C_a, \sigma_a \rangle, k_a, \tau \rangle \rightarrow \langle \cdots, \langle C_a', \sigma_a' \rangle, k_a', \tau' \rangle}\;(\textsc{Attacker})$$

$$\frac{}{\langle \mathbf{skip}, \langle \sigma, \tau \rangle\rangle \rightarrow \langle \epsilon, \langle \sigma, \tau \rangle\rangle}\;(\textsc{Skip}) \qquad \frac{\langle C_1, \langle \sigma, \tau \rangle\rangle \rightarrow \langle \epsilon, \langle \sigma', \tau' \rangle\rangle}{\langle C_1; C_2, \langle \sigma, \tau \rangle\rangle \rightarrow \langle C_2, \langle \sigma', \tau' \rangle\rangle}\;(\textsc{Seq}1) \qquad \frac{\langle C_1, \langle \sigma, \tau \rangle\rangle \rightarrow \langle C_1', \langle \sigma', \tau' \rangle\rangle}{\langle C_1; C_2, \langle \sigma, \tau \rangle\rangle \rightarrow \langle C_1'; C_2, \langle \sigma', \tau' \rangle\rangle}\;(\textsc{Seq}2)$$

$$\frac{}{\langle \mathbf{if}\ (e)\ \{C_1\}\ \mathbf{else}\ \{C_2\}, \langle \sigma, \tau \rangle\rangle \rightarrow \langle C_1, \langle \sigma, \tau \rangle\rangle}\;(\textsc{If}1)^{e(\sigma_i)=\mathit{True}()} \qquad \frac{}{\langle \mathbf{if}\ (e)\ \{C_1\}\ \mathbf{else}\ \{C_2\}, \langle \sigma, \tau \rangle\rangle \rightarrow \langle C_2, \langle \sigma, \tau \rangle\rangle}\;(\textsc{If}2)^{e(\sigma_i)\neq\mathit{True}()}$$

$$\frac{}{\langle \mathbf{while}\ (e)\ \{C\}, \langle \sigma, \tau \rangle\rangle \rightarrow \langle \mathbf{if}\ (e)\ \{C; \mathbf{while}\ (e)\ \{C\}\}\ \mathbf{else}\ \{\mathbf{skip}\}, \langle \sigma, \tau \rangle\rangle}\;(\textsc{While})$$

$$\frac{}{\langle x := e, \langle \sigma, \tau \rangle\rangle \rightarrow \langle \epsilon, \langle \sigma[x \mapsto e(\sigma)], \tau \rangle\rangle}\;(\textsc{Assign})$$

$$\frac{}{\langle \mathit{send}(e), \langle \sigma, \tau \rangle\rangle \rightarrow \langle \epsilon, \langle \sigma[\mathit{snap} \mapsto \tau + \mathit{Send}(e(\sigma))], \tau + \mathit{Send}(e(\sigma))\rangle\rangle}\;(\textsc{Send})$$

$$\frac{}{\langle x := \mathit{recv}(), \langle \sigma, \tau \rangle\rangle \rightarrow \langle \epsilon, \langle \sigma[x \mapsto v], \tau \rangle\rangle}\;(\textsc{Recv})^{v \in \mathit{msgs}(\tau)}$$

$$\frac{}{\langle x := \mathit{nonce}(), \langle \sigma, \tau \rangle\rangle \rightarrow \langle \epsilon, \langle \sigma[x \mapsto v, \mathit{snap} \mapsto \tau + \mathit{Nonce}(v)], \tau + \mathit{Nonce}(v)\rangle\rangle}\;(\textsc{NonceGen})^{\mathit{fresh}(v,\tau)}$$

$$\frac{}{\langle x := \mathit{hash}(e), \langle \sigma, \tau \rangle\rangle \rightarrow \langle \epsilon, \langle \sigma[x \mapsto \mathsf{hash}(e(\sigma))], \tau \rangle\rangle}\;(\textsc{Hash}) \qquad \frac{}{\langle x := \mathit{pk}(e), \langle \sigma, \tau \rangle\rangle \rightarrow \langle \epsilon, \langle \sigma[x \mapsto \mathsf{pk}(e(\sigma))], \tau \rangle\rangle}\;(\textsc{Pk})$$

$$\frac{}{\langle x := \mathit{enc}(e_1, e_2), \langle \sigma, \tau \rangle\rangle \rightarrow \langle \epsilon, \langle \sigma[x \mapsto \mathsf{enc}(e_1(\sigma), e_2(\sigma))], \tau \rangle\rangle}\;(\textsc{Enc})$$

$$\frac{}{\langle x, ok := \mathit{dec}(e_1, e_2), \langle \sigma, \tau \rangle\rangle \rightarrow \langle \epsilon, \langle \sigma[x \mapsto v, ok \mapsto \mathit{True}()], \tau \rangle\rangle}\;(\textsc{DecSucc})^{\exists v. e_2(\sigma) = \mathsf{enc}(\mathsf{pk}(e_1(\sigma)), v)}$$

$$\frac{}{\langle x, ok := \mathit{dec}(e_1, e_2), \langle \sigma, \tau \rangle\rangle \rightarrow \langle \epsilon, \langle \sigma[ok \mapsto \mathit{False}()], \tau \rangle\rangle}\;(\textsc{DecFail})^{\forall v. e_2(\sigma) \neq \mathsf{enc}(\mathsf{pk}(e_1(\sigma)), v)}$$

$$\frac{}{\langle \mathit{drop}(e), \langle \sigma, k, \tau \rangle\rangle \rightarrow \langle \epsilon, \langle \sigma[\mathit{snap} \mapsto \tau + \mathit{Drop}(e(\sigma))], k, \tau + \mathit{Drop}(e(\sigma))\rangle\rangle}\;(\textsc{Drop})$$

$$\frac{}{\langle \mathit{learn}(e), \langle \sigma, k, \tau \rangle\rangle \rightarrow \langle \epsilon, \langle \sigma, k \cup \{e(\sigma)\}, \tau \rangle\rangle}\;(\textsc{Learn}) \qquad \frac{}{\langle x := \mathit{choose}(), \langle \sigma, k, \tau \rangle\rangle \rightarrow \langle \epsilon, \langle \sigma[x \mapsto v], k, \tau \rangle\rangle}\;(\textsc{Choose})^{v \in k}$$

$$\frac{}{\langle \cdots, \langle C_i, \sigma_i \rangle, \cdots, \langle \mathit{corrupt}(i); C_a', \sigma_a \rangle, k_a, \tau \rangle \rightarrow \langle \cdots, \langle C_i, \sigma_i \rangle, \cdots, \langle C_a', \sigma_a \rangle, k_a \cup \mathit{val}(\sigma_i), \tau + \mathit{Corrupt}(i, \mathit{val}(\sigma_i))\rangle}\;(\textsc{Corrupt})$$

$$\frac{}{\langle \cdots, \langle \mathbf{fork}\ (\vec{x})\ \{C\}; C', \sigma_i \rangle, \cdots, k_a, \tau \rangle \rightarrow \langle \cdots, \langle C', \sigma_i \rangle, \cdots, \langle C, [\vec{x} \mapsto \sigma_i(\vec{x}), \mathit{snap} \mapsto \sigma_i(\mathit{snap})]\rangle, k_a, \tau \rangle}\;(\textsc{Fork})$$

Figure 3.14: Small-step semantics. Since expressions are functions from states to values, $e(\sigma)$ denotes the evaluation of expression e in state σ. $\sigma[x_1 \mapsto v_1, \cdots, x_n \mapsto v_n]$ denotes state update: a state that, for all i, $1 \leq i \leq n$, yields v_i for x_i, and the value in σ for all other variables. We extend this notion naturally to vectors of variables for the Fork rule, i.e., $[\vec{x} \mapsto \sigma_i(\vec{x}), \mathit{snap} \mapsto \sigma_i(\mathit{snap})]$ denotes a state that contains only mappings for the variables in $\vec{x}$ and *snap*, and maps them to their respective values in state σ_i. Appending to a trace is denoted by $+$, e.g., $\tau + \mathit{Nonce}(v)$. $\langle \epsilon, \langle \sigma, \tau \rangle\rangle$ denotes a terminal state.

fresh.

Definition 3.8.6 (Freshness of Nonces) *Since all previously generated nonces have been recorded on the trace τ, we can define freshness of a nonce v on the global trace τ as follows:*

$$fresh(v, \tau) \triangleq v \notin \{n \mid \forall n, l.\, Nonce(n, l) \in \tau\}$$

Corruption. The Corrupt rule expresses that the attacker knowledge is extended by all terms in the state σ_i of the corrupted participant i. The attacker can make use of these newly learned terms by executing $x := choose()$ that non-deterministically picks a term in the attacker knowledge and assigns it to the local variable x.

To avoid additional sequential composition rules operating on the system configuration (instead of a local configuration like Seq1 and Seq2), we bake sequential composition into the Corrupt and Fork rules. While this simplification requires the *corrupt*() and **fork** () {} commands to be followed by another command, this requirement is not a limitation in practice, as inserting a **skip** command fulfills this requirement without changing a program's behavior.

Forking. The Fork rule extends the system configuration by another local configuration containing the forked command and the new thread's initial state. This new state maps the variables in $\vec{x}$ and *snap* to the same value as they have in state σ_i, i.e., the state in which the fork command is executed, which enables, e.g., the sharing of public keys.

3.8.2 Program Logic

We now present a program logic that enables local reasoning about each participant, while guaranteeing that the trace invariant is maintained even when composing arbitrarily many verified participants and the attacker to a distributed system. We first present several auxiliary definitions and lemmata, and then the logic's proof rules.

Definition 3.8.7 (Trace Prefix) *We define the following predicate over two traces expressing that τ_1 is a prefix of τ_2*

$$prefix(\tau_1, \tau_2) \triangleq \; \exists p.\, \tau_1 + p = \tau_2$$

where p is a possibly empty sequence of trace events.

Lemma 3.8.1 (Prefix Reflexivity)

$$\forall \tau.\, prefix(\tau, \tau)$$

Proof. Pick p to be the empty sequence in Def. 3.8.7. □

Lemma 3.8.2 (Prefix Transitivity)

$$\forall \tau_1, \tau_2, \tau_3.\, prefix(\tau_1, \tau_2) \land prefix(\tau_2, \tau_3) \implies prefix(\tau_1, \tau_3)$$

Proof.

$$\begin{aligned}
& \mathit{prefix}(\tau_1, \tau_2) \wedge \mathit{prefix}(\tau_2, \tau_3) \\
& \overset{\text{def}}{\iff} \exists p_1, p_2.\, \tau_1 + p_1 = \tau_2 \wedge \tau_2 + p_2 = \tau_3 \\
& \implies \exists p_1, p_2.\, \tau_1 + p_1 + p_2 = \tau_3 \\
& \overset{\text{def}}{\iff} \mathit{prefix}(\tau_1, \tau_3)
\end{aligned}$$

where, in the last step, we pick p in Def. 3.8.7 to be $p_1 + p_2$. □

[110]: Vafeiadis (2011), *Concurrent Separation Logic and Operational Semantics*

Inspired by Vafeiadis [110], we express the semantics of judgments in our logic in terms of configuration safety, which we define next. Intuitively, $\mathit{safe}_n(i, C, \sigma, Q, \tau)$ expresses that it is safe to execute command C, as the ith component of the distributed system, and for n execution steps starting in a state σ; and if the command is fully executed, the predicate Q holds in the resulting final state. Furthermore, if new threads have been forked as part of executing C then it is safe to execute these forked components, too. Since we are ultimately interested in the effects on the global trace τ, configuration safety includes maintenance of the trace invariant ρ. A judgement $\models [P]\ C\ [Q]$ then expresses that it is safe to execute the command C starting from any initial state satisfying the predicate P for an arbitrary number of execution steps.

Definition 3.8.8 (Configuration Safety)

$\mathit{safe}_0(i, C, \sigma, Q, \tau)$ holds always.

$\mathit{safe}_{n+1}(i, C, \sigma, Q, \tau)$ holds if and only if

(i) $C = \epsilon \implies Q(\sigma)$ and

(ii) $$\begin{aligned}
\forall \vec{C}, \vec{C}', \vec{\sigma}, \vec{\sigma}', k_a, k_a', \tau'.\ & i \le |\vec{C}| = |\vec{\sigma}| \le |\vec{C}'| = |\vec{\sigma}'| \wedge \\
& \vec{C}_i = C \wedge \vec{C}_i' \ne C \wedge \vec{\sigma}_i = \sigma \wedge \\
& \rho(\tau) \wedge \mathit{prefix}(\mathit{snap}(\sigma), \tau) \wedge \\
& \langle \overrightarrow{\langle C, \sigma \rangle}, k_a, \tau \rangle \to \langle \overrightarrow{\langle C', \sigma' \rangle}, k_a', \tau' \rangle \\
\implies\ & \rho(\tau') \wedge \mathit{prefix}(\tau, \tau') \wedge \mathit{prefix}(\mathit{snap}(\vec{\sigma}_i'), \tau') \wedge \\
& k_a \subseteq k_a' \wedge \mathit{safe}_n(i, \vec{C}_i', \vec{\sigma}_i', Q, \tau') \wedge \\
& \left(\bigwedge_{|\vec{C}| < j \le |\vec{C}'|} \mathit{safe}_n(j, \vec{C}_j', \vec{\sigma}_j', \mathit{True}(), \tau') \wedge \mathit{prefix}(\mathit{snap}(\vec{\sigma}_j'), \tau') \right)
\end{aligned}$$

where $|\vec{V}|$ and $\vec{V}_i$ denote the length and element at index i of a vector V, resp., and $\overrightarrow{\langle C, \sigma \rangle}$ is syntactic sugar for $\langle \vec{C}_1, \vec{\sigma}_1 \rangle \cdots \langle \vec{C}_{|\vec{C}|}, \vec{\sigma}_{|\vec{\sigma}|} \rangle$.

Definition 3.8.9 (Validity)

$$\models [P]\ C\ [Q] \triangleq \forall n, i, \sigma, \tau.\, P(\sigma) \implies \mathit{safe}_n(i, C, \sigma, Q, \tau)$$

Executing zero steps is vacuously safe. Executing $n + 1$ steps is safe if (i) the command is already fully executed and the predicate Q satisfied; and otherwise if (ii) there is a transition to $\vec{C}_i'$ that maintains the trace invariant ρ, the necessary monotonicity properties (on snapshot, trace, and the attacker's knowledge set), and allows continued safe execution

$$\frac{}{\vdash [P]\ skip\ [P]}\ (\textsc{Skip}) \qquad \frac{\vdash [P]\ C_1\ [R] \quad \vdash [R]\ C_2\ [Q]}{\vdash [P]\ C_1; C_2\ [Q]}\ (\textsc{Seq}) \qquad \frac{P \models P' \quad Q' \models Q \quad \vdash [P']\ C\ [Q']}{\vdash [P]\ C\ [Q]}\ (\textsc{Cons})$$

$$\frac{\vdash [e \land P]\ C_1\ [Q] \quad \vdash [\neg e \land P]\ C_2\ [Q]}{\vdash [P]\ \textbf{if}\ (e)\ \{C_1\}\ \textbf{else}\ \{C_2\}\ [Q]}\ (\textsc{If}) \qquad \frac{\vdash [e \land P]\ C\ [P]}{\vdash [P]\ \textbf{while}\ (e)\ \{C\}\ [\neg e \land P]}\ (\textsc{While}) \qquad \frac{}{\vdash [P[e/x]]\ x := e\ [P]}\ (\textsc{Assign})$$

$$\frac{}{\vdash \big[ext(Send(e), snap) \land \forall p.\, P[snap + p + Send(e)/snap]\big]\ send(e)\ [P]}\ (\textsc{Send}) \qquad \frac{}{\vdash [\forall x.\, P]\ x := recv()\ [P]}\ (\textsc{Recv})$$

$$\frac{}{\vdash \big[ext(Nonce(x), snap) \land \forall p, x.\, P[snap + p + Nonce(x)/snap]\big]\ x := nonce()\ [P]}\ (\textsc{NonceGen})$$

$$\frac{}{\vdash [P[\mathsf{hash}(e)/x]]\ x := hash(e)\ [P]}\ (\textsc{Hash}) \qquad \frac{}{\vdash [P[\mathsf{pk}(e)/x]]\ x := pk(e)\ [P]}\ (\textsc{Pk}) \qquad \frac{}{\vdash [P[\mathsf{enc}(e_1, e_2)/x]]\ x := enc(e_1, e_2)\ [P]}\ (\textsc{Enc})$$

$$\frac{}{\vdash [\forall x.\, P[True()/ok][e_2/\mathsf{enc}(\mathsf{pk}(e_1), x)] \land P[False()/ok]]\ x, ok := dec(e_1, e_2)\ [P]}\ (\textsc{Dec})$$

$$\frac{}{\vdash \big[ext(Drop(e), snap) \land \forall p.\, P[snap + p + Drop(e)/snap]\big]\ drop(e)\ [P]}\ (\textsc{Drop})$$

$$\frac{}{\vdash [P]\ learn(e)\ [P]}\ (\textsc{Learn}) \qquad \frac{}{\vdash [\forall x.\, P]\ x := choose()\ [P]}\ (\textsc{Choose})$$

$$\frac{}{\vdash \big[(\forall v.\, ext(Corrupt(e, v), snap)) \land (\forall p, v.\, P[snap + p + Corrupt(e, v)/snap])\big]\ corrupt(e)\ [P]}\ (\textsc{Corrupt})$$

$$\frac{fv(R) \subseteq \vec{x} \quad P \models R \quad \vdash [R]\ C\ [True()] \quad \vdash [P]\ C'\ [Q]}{\vdash [P]\ \textbf{fork}\ (\vec{x})\ \{C\}; C'\ [Q]}\ (\textsc{Fork})$$

Figure 3.15: The proof rules.

of all components (i.e., of commands $\vec{C'}$) in the system, including newly forked ones (the last, iterated conjunct in the definition).

Fig. 3.15 shows the proof rules for our logic. Our assertion language is a first-order logic (for brevity not a separation logic) with the usual logical connectives and quantifiers, and access to local program variables. Pre- and postconditions can therefore refer to the local snapshot, but they *cannot* refer to the global trace. The latter corresponds to our methodology (recall Sec. 3.3.4), where pre- and postconditions also cannot directly express properties about the trace because access to it is governed by our reusable verification library's ghost lock. Instead, properties about the global trace, such as the existence of a particular trace event, must always be expressed via the local snapshot. This ensures that pre- and postconditions are stable under potential environment interference, which is needed to prove our proof rules sound.

As for the operational semantics, we again discuss only non-standard proof rules. Proof rules corresponding to commands that modify the global trace, e.g., the Send rule, enforce that the trace invariant is maintained under potential environment interference. For this purpose, we define an extensibility predicate specifying that appending a trace event n to an arbitrary extension of a trace τ maintains the trace invariant ρ.

Definition 3.8.10 (Extensibility) *A trace τ is* extensible *by a trace event n if the trace invariant ρ is maintained for any possible trace τ', given that τ is a prefix thereof:*

$$ext(n, \tau) \triangleq \forall \tau'. prefix(\tau, \tau') \wedge \rho(\tau') \implies \rho(\tau' + n)$$

Recall from Fig. 3.14 that commands modifying the global trace, e.g., the send command, also update the local snapshot to the most recent version of the trace. Analogous to the proof rule for assignments, the proof rules for trace-modifying commands, thus, require that $\forall p. P[snap + p + n/snap]$ holds in the state before executing the command, where P is the postcondition and n is a trace event (e.g., $Send(e)$). The quantified p accounts for all possible trace extensions that could have been made by the environment since the local snapshot was last updated, and thus accounts for arbitrary interleavings of participants and the attacker.

For the sake of presentation we have omitted additional assumptions that are available when discharging preconditions of snapshot-updating commands: e.g., in proof rule NonceGen, we may additionally use nonce freshness, and in proof rule Recv, we may assume that a received message was previously sent and not dropped in the meantime.

Theorem 3.8.3 (Soundness of Proof Rules)

$$\textit{If } \vdash [P]\ C\ [Q] \textit{ then } \models [P]\ C\ [Q]$$

We proof this theorem in the usual way, i.e., by structural induction on the shape of the proof tree given by the theorem's left-hand side of the implication. We proceed by case distinction on the last rule applied, and may assume the theorem (i.e., our induction hypothesis) for this rule's premises. In our proof sketch we focus on a few interesting cases—send, sequential composition, and fork—, which we present further down, as individual lemmata. Send is interesting because it illustrates a trace-updating proof rule, for which we have to show that the trace invariant is maintained. The challenge for sequential composition is to show that our definition of configuration safety allows us to prove that *each* transition in the system maintains the trace invariant. The Fork proof rule is of interest because it is the only command that extends the system configuration with additional components.

We begin by sketching the proofs for several auxiliary lemmata about configuration safety that will be useful later on.

Lemma 3.8.4 *The empty command satisfies configuration safety given that the predicate Q holds.*

$$\forall n, i, \sigma, Q, \tau. Q(\sigma) \implies safe_n(i, \epsilon, \sigma, Q, \tau)$$

Proof. We show for arbitrary n, i, σ, τ, and assuming $Q(\sigma)$, that configuration safety $safe_n(i, \epsilon, \sigma, Q, \tau)$ holds. Case (i) from the definition of *safe* holds straightforwardly. Case (ii) is satisfied because there is no transition starting in command ϵ and resulting in a different command. Hence, this case vacuously holds. □

Lemma 3.8.5 *A command C satisfying configuration safety for n execution steps is safe to execute for fewer execution steps.*

$$\forall m, n, i, C, \sigma, Q, \tau.\, m \leq n \wedge \mathit{safe}_n(i, C, \sigma, Q, \tau) \\ \implies \mathit{safe}_m(i, C, \sigma, Q, \tau)$$

Proof. Straightforward induction on m. □

Next, we present soundness lemmata for the aforementioned interesting proof rules: send, sequential composition, and fork.

Send. Soundness for the proof rule Send directly follows from the following safety lemma:

Lemma 3.8.6

$$\forall n, i, \sigma, Q, \tau.\, \mathit{ext}(\mathit{Send}(e(\sigma)), \mathit{snap}(\sigma)) \wedge \\ (\forall p.\, Q[\mathit{snap} + p + \mathit{Send}(e)/\mathit{snap}](\sigma)) \\ \implies \mathit{safe}_n(i, \mathit{send}(e), \sigma, Q, \tau)$$

Proof. We prove this lemma by induction on n using the following induction hypothesis:

$$IH(n) \triangleq \forall i, \sigma, Q, \tau.\, \mathit{ext}(\mathit{Send}(e(\sigma)), \mathit{snap}(\sigma)) \wedge \\ (\forall p.\, Q[\mathit{snap} + p + \mathit{Send}(e)/\mathit{snap}](\sigma)) \\ \implies \mathit{safe}_n(i, \mathit{send}(e), \sigma, Q, \tau)$$

In the base case ($n = 0$), $\mathit{safe}_0(i, \mathit{send}(e), \sigma, Q, \tau)$ holds by definition. For the induction step, we assume $IH(n)$ and show that $IH(n+1)$ holds. I.e., we further assume $\mathit{ext}(\mathit{Send}(e(\sigma)), \mathit{snap}(\sigma))$ and $\forall p.\, Q[\mathit{snap} + p + \mathit{Send}(e)/\mathit{snap}](\sigma)$ for arbitrary i, σ, Q, and τ. We have to prove that $\mathit{safe}_{n+1}(i, \mathit{send}(e), \sigma, Q, \tau)$ holds. Case (i) from the definition of *safe* holds trivially because $\mathit{send}(e) \neq \epsilon$. To prove case (ii), we assume the implication's left-hand side and show that the right-hand side holds. In particular, we consider a transition that executes command $\mathit{send}(e)$. According to the operational semantics, only the transition rule Local with an application of the Send rule in its premise is applicable and modifies the command in the ith component's configuration. This allows us to conclude that the considered transition must have the following shape:

$$\langle \overrightarrow{\langle C, \sigma \rangle}, k_a, \tau \rangle \rightarrow \langle \overrightarrow{\langle C', \sigma' \rangle}, k'_a, \tau' \rangle$$

where

$$|\vec{C}'| = |\vec{C}| \wedge k'_a = k_a \wedge \tau' = \tau + \mathit{Send}(e(\sigma)) \wedge \\ \vec{C}'_i = \epsilon \wedge \vec{\sigma}'_i = \vec{\sigma}_i[\mathit{snap} \mapsto \tau + \mathit{Send}(e(\sigma))] \wedge \\ (\forall j.\, i \neq j \implies \vec{C}'_j = \vec{C}_j \wedge \vec{\sigma}'_j = \vec{\sigma}_j)$$

and $\rho(\tau) \wedge \mathit{prefix}(\mathit{snap}(\vec{\sigma}_i), \tau)$ holds. We have to prove that (1) $\rho(\tau')$, (2) $\mathit{prefix}(\tau, \tau')$, (3) $\mathit{prefix}(\mathit{snap}(\vec{\sigma}'_i), \tau')$, (4) $k_a \subseteq k'_a$, and (5) $\mathit{safe}_n(i, \vec{C}'_i, \vec{\sigma}'_i, Q, \tau')$ hold. Note that no additional local configurations have been added by this command because $|\vec{C}'| = |\vec{C}|$ holds. (1) follows directly by definition of Def. 3.8.10. (2) holds by choosing $p = \mathit{Send}(e(\sigma))$ as witness in Def. 3.8.7. (3) holds by reflexivity (cf. Lemma 3.8.1). (4) holds because the attacker knowledge is unchanged. Finally, (5) follows from Lemma 3.8.4 via the following derivation to obtain $Q(\vec{\sigma_i'})$:

$$\begin{aligned}&\forall p.\, Q[snap + p + Send(e)/snap](\vec{\sigma}_i)\\ &\implies Q[\tau + Send(e)/snap](\vec{\sigma}_i)\\ &\iff Q(\vec{\sigma}_i[snap \mapsto \tau + Send(e(\vec{\sigma}_i))]) \iff Q(\vec{\sigma_i'})\end{aligned}$$

where the implication is justified by the fact that $prefix(snap(\vec{\sigma}_i), \tau)$ holds. □

Seq. In the case where the last rule applied in our proof tree is Seq, we may assume the induction hypothesis for the rule's premises, i.e., $\models [P]\ S_1\ [R]$ and $\models [R]\ S_2\ [Q]$. Soundness for this case, i.e., showing $\models [P]\ S_1;\, S_2\ [Q]$, then follows from the following safety lemma:

Lemma 3.8.7

$$\begin{aligned}&\forall n, i, S_1, S_2, \sigma_1, R, Q, \tau.\, safe_n(i, S_1, \sigma_1, R, \tau) \wedge\\ &\quad (\forall m, \sigma_2, \tau'.\, m \le n \wedge R(\sigma_2) \implies safe_m(i, S_2, \sigma_2, Q, \tau'))\\ &\implies safe_n(i, S_1;\, S_2, \sigma_1, Q, \tau)\end{aligned}$$

Proof. We perform induction on n using the following induction hypothesis:

$$\begin{aligned}IH(n) \triangleq\ &\forall i, S_1, S_2, \sigma_1, R, Q, \tau, .\\ &\quad safe_n(i, S_1, \sigma_1, R, \tau) \wedge\\ &\quad (\forall m, \sigma_2, \tau'.\, m \le n \wedge R(\sigma_2) \implies safe_m(i, S_2, \sigma_2, Q, \tau'))\\ &\implies safe_n(i, S_1;\, S_2, \sigma_1, Q, \tau)\end{aligned}$$

The base case ($n = 0$) holds by definition. In the induction step, we may assume $IH(n)$ to prove $IH(n+1)$. For arbitrary i, S_1, S_2, σ_1, R, Q, and τ we assume the left-hand side, i.e., $safe_{n+1}(i, S_1, \sigma_1, R, \tau)$ and $\forall m, \sigma_2, \tau'.\, m \le n+1 \wedge R(\sigma_2) \implies safe_m(i, S_2, \sigma_2, Q, \tau')$. It remains to prove that $safe_{n+1}(i, S_1;\, S_2, \sigma_1, Q, \tau)$ holds. The proof proceeds similarly to the proof of Lemma 3.8.6 except that in case (ii) the Local rule's premise is fulfilled by an application of either the Seq1 or Seq2 rule:

- Case Seq1: According to this transition's premise, there exists a transition $\langle S_1, \langle \sigma_i, \tau \rangle\rangle \longrightarrow \langle \epsilon, \langle \sigma_i', \tau'' \rangle\rangle$ for some τ''. Thus, we obtain by definition of $safe_{n+1}(i, S_1, \sigma_i, R, \tau)$ that $\rho(\tau'')$, $prefix(\tau, \tau'')$, $prefix(snap(\sigma_i'), \tau'')$, $k_a \subseteq k_a'$, and $safe_n(i, \epsilon, \sigma_i', R, \tau'')$ hold. We distinguish two cases, namely $n = 0$ and $n > 0$. In the first case, we obtain by definition of configuration safety $safe_0(i, S_2, \sigma_i', Q, \tau'')$. In the second case, we obtain by definition of $safe_n(i, \epsilon, \sigma_i', R, \tau'')$ that $R(\sigma_i')$ holds. Therefore, we can instantiate m, σ_2, and τ' with n, σ_i', and τ'', respectively, in the quantifier above. Thus, we obtain $safe_n(i, S_2, \sigma_i', Q, \tau'')$. This concludes the proof for both cases $n = 0$ and $n > 0$ showing that $safe_{n+1}(i, S_1;\, S_2, \sigma_i, Q, \tau)$ holds.
- Case Seq2: This transition's premise specifies that a transition $\langle S_1, \langle \sigma_i, \tau \rangle\rangle \longrightarrow \langle S_1', \langle \sigma_i', \tau'' \rangle\rangle$ for some τ'' exists. We apply the definition of $safe_{n+1}(i, S_1, \sigma_i, R, \tau)$ to obtain $\rho(\tau'')$, $prefix(\tau, \tau'')$, $prefix(snap(\sigma_i'), \tau'')$, $k_a \subseteq k_a'$, and $safe_n(i, S_1', \sigma_i', R, \tau'')$. By applying the induction hypothesis for n, we obtain $safe_n(i, S_1';\, S_2, \sigma_i', Q, \tau'')$. Thus, we showed $safe_{n+1}(i, S_1;\, S_2, \sigma_i, Q, \tau)$.

□

Fork. Soundness of the Fork proof rule follows from the following safety lemma:

Lemma 3.8.8

$$\begin{aligned}&\forall n, i, \vec{x}, S_1, S_2, \sigma_1, Q, \tau.\, \mathit{safe}_n(i, S_1, \sigma_1, Q, \tau) \land \\ &\qquad (\forall j, \sigma_2.\, [\sigma_1 \sim \sigma_2]^{\vec{x} \cup snap} \implies \mathit{safe}_n(j, S_2, \sigma_2, \mathit{True}(), \tau)) \\ &\implies \mathit{safe}_n(i, \textbf{fork } (\vec{x}) \; \{S_2\};\, S_1, \sigma_1, Q, \tau)\end{aligned}$$

where $[\sigma_1 \sim \sigma_2]^{\vec{x} \cup snap}$ *denotes that* σ_1 *maps the variables in* $\vec{x}$ *and variable snap to the same values as* σ_2 *does.*

Proof. We perform induction on n and use the following induction hypothesis:

$$\begin{aligned}IH(n) \triangleq\ &\forall i, \vec{x}, S_1, S_2, \sigma_1, Q, \tau, . \\ &\quad \mathit{safe}_n(i, S_1, \sigma_1, Q, \tau) \land \\ &\quad (\forall j, \sigma_2.\, [\sigma_1 \sim \sigma_2]^{\vec{x} \cup snap} \implies \mathit{safe}_n(j, S_2, \sigma_2, \mathit{True}(), \tau)) \\ &\quad \implies \mathit{safe}_n(i, \textbf{fork } (\vec{x}) \; \{S_2\};\, S_1, \sigma_1, Q, \tau)\end{aligned}$$

For $n = 0$, $\mathit{safe}_0(i, \textbf{fork } (\vec{x}) \; \{S_2\};\, S_1, \sigma_1, Q, \tau)$ holds by definition. In the induction step, we assume $IH(n)$ to show $IH(n+1)$. We assume the left-hand side, i.e.,

$$\mathit{safe}_{n+1}(i, S_1, \sigma_1, Q, \tau) \land \tag{3.1}$$

$$(\forall j, \sigma_2.\, [\sigma_1 \sim \sigma_2]^{\vec{x} \cup snap} \implies \mathit{safe}_{n+1}(j, S_2, \sigma_2, \mathit{True}(), \tau)) \tag{3.2}$$

and seek to show $\mathit{safe}_{n+1}(i, \textbf{fork } (\vec{x}) \; \{S_2\};\, S_1, \sigma_1, Q, \tau)$. Similar to the proof of Lemma 3.8.6, the interesting case is (ii) in which we only consider the inference rule Fork. Based on the operational semantics, we obtain

$$\begin{aligned}&(\vec{C}'_i = S_1) \land (\sigma_1 = \vec{\sigma}_i = \vec{\sigma}'_i) \land (|\vec{C}'| = |\vec{C}| + 1) \land \\ &(\vec{C}'_{|\vec{C}'|} = S_2) \land \left[\sigma_1 \sim \vec{\sigma}'_{|\vec{C}'|}\right]^{\vec{x} \cup snap} \land \\ &(\forall j.\, 1 \le j \le |\vec{C}| \implies \vec{\sigma}'_j = \vec{\sigma}_j) \land \\ &(\forall j.\, 1 \le j \le |\vec{C}| \land i \ne j \implies \vec{C}'_j = \vec{C}_j)\end{aligned}$$

Since the attacker knowledge k_a and global trace τ remain unchanged by the application of this inference rule, we have to prove that (a) $\mathit{safe}_n(i, S_1, \sigma_1, Q, \tau)$ and (b) $\mathit{safe}_n(|\vec{C}'|, S_2, \vec{\sigma}'_{|\vec{C}'|}, \mathit{True}(), \tau)$ hold. (a) follows from applying Lemma 3.8.5 to $\mathit{safe}_{n+1}(i, S_1, \sigma_1, Q, \tau)$. Since (3.2)'s left-hand side is satisfied for $\sigma_2 = \vec{\sigma}'_{|\vec{C}'|}$, we instantiate the quantifier j with $|\vec{C}'|$ to obtain $\mathit{safe}_{n+1}(|\vec{C}'|, S_2, \vec{\sigma}'_{|\vec{C}'|}, \mathit{True}(), \tau)$. We also obtain (b) by applying Lemma 3.8.5. □

3.8.3 Trace Inclusion

We can now show the desired trace inclusion (recall Sec. 3.8), which directly follows from Thm. 3.8.3.

```
func main(num_initiators, num_responders int) {
  ... // initialization code
  while (num_initiators > 0) {
    fork (initiator_args) {
      initiator(initiator_args)
    }
    num_initiators := num_initiators - 1
  }
  while (num_responders > 0) {
    fork (responder_args) {
      responder(responder_args)
    }
    num_responders := num_responders - 1
  }
  fork() { attacker() }
}
```

Figure 3.16: Sketch of a program C_{system} bootstrapping the distributed system by executing sequential initialization code to, e.g., generate public/private key pairs and forking several instances of an initiator and responder implementation and the highly non-deterministic attacker implementation. `initiator_args` and `responder_args` are abbreviations for a list of arguments that are passed to the initiator and responder implementations, respectively. E.g., the initiator's public/private key pair and the responder's public key might constitute `initiator_args`.

Theorem 3.8.9 *If we bootstrap the distributed system from a single component, with no precondition, a trace invariant that holds for the empty trace, and an initial attacker knowledge set, then the trace invariant always holds, regardless of how many transitions are performed and whether additional components (participants and the attacker) are forked.*

$$\forall C, \vec{C}', Q, \sigma, \vec{\sigma}', k_a', \tau'. \vdash [\mathit{True}()] \ C \ [Q] \land \rho(\emptyset) \land \langle\langle C, \sigma\rangle, k_a^{init}, \emptyset\rangle \to^* \langle\overrightarrow{\langle C', \sigma'\rangle}, k_a', \tau'\rangle \implies \rho(\tau')$$

where k_a^{init} is the initial attacker knowledge consisting of all public terms.

Proof. We prove this theorem by first applying soundness of our proof rules (Thm. 3.8.3) and expanding Def. 3.8.8 because the state σ trivially satisfies the precondition $\mathit{True}()$. We proceed by induction over the length of transition sequences. Since the trace invariant holds initially, is maintained by each transition, and each command in every component of the system satisfies configuration safety, we obtain $\rho(\tau')$ for every trace τ' that is possible after executing n transitions, where n is the induction variable. □

Thm. 3.8.9 implies the following trace inclusion property where ϕ is a security property implied by the trace invariant ρ, i.e., $\rho \models \phi$:

$$\forall C, Q. \vdash [\mathit{True}()] \ C \ [Q] \land \rho(\emptyset) \implies \mathit{Tr}(C) \subseteq \mathit{Tr}(\rho) \subseteq \mathit{Tr}(\phi)$$

where $\mathit{Tr}(C)$ denotes the set of all traces that result from executing arbitrary many transitions according to the small-step operational semantics. $\mathit{Tr}(\rho)$ and $\mathit{Tr}(\phi)$ are the sets of traces satisfying ρ and ϕ, respectively. I.e., $\mathit{Tr}(\rho) = \{\tau \mid \forall \tau. \rho(\tau)\}$ and $\mathit{Tr}(\phi)$ analogously.

In our verification case studies we prove Thm. 3.8.9 in three steps: In step 1, we once-and-for-all verify our reusable verification library, including a most-general attacker implementation (an iterated non-deterministic choice between all executable commands) against a partially abstract (thus sufficiently general) trace invariant. I.e., the judgement $\vdash [true] \ C_a \ [true]$ that we obtain for the attacker holds for all possible attackers and protocol-specific instantiations of this abstract trace invariant. In step 2, we implement each participant in its own program C_i; verifying these effectively yield a judgement $\vdash [P_i] \ C_i \ [Q_i]$ per participant.

In step 3, we combine these separate judgments for the protocol participants and the attacker by constructing a program C_{system} that first performs some sequential initialization code and then forks several instances of protocol participants and the attacker, as illustrated in Fig. 3.16. By taking the number of participant instances as unconstrained input parameters, we obtain a result for unboundedly many instances. Functions and non-deterministic choices are straightforward extensions to our programming language. The initialization code's purpose is to establish the participants' preconditions. E.g., in our NSL case study, we implement initialization code that generates public-private key pairs and passes the necessary keys to each protocol participant. In the case of WireGuard, the corresponding initialization code remains an assumption, which is typical for security protocol verification and corresponds to assuming that there exists a mechanism to authentically distribute public keys.

3.9 Secure Deletion

Our methodology presented so far enables proving strong security properties for protocol implementations, like forward secrecy for WireGuard. While forward secrecy is typically defined to consider compromise of long-term keys only, as opposed to also consider session key compromise, modern protocols are often designed to provide even stronger security guarantees. To protect past messages against future compromise of a session key, protocols employ a so-called *ratchet* mechanism that periodically updates the session key by applying a one-way function, e.g., a key derivation function (KDF). Therefore, if the attacker obtains a future session key, it cannot decrypt past messages as that would require inverting the one-way function. Similarly, a ratchet enables inputting additional entropy into the generation of the next session key, providing post-compromise security. Post-compromise security informally states that a session key k remains secure even if the attacker compromises a session key that was *previously* used in the same session. However, this property requires that the attacker does not obtain all entropy that got input to derive the key k starting from the compromised session key, as the attacker could otherwise perform the very same derivation to obtain k. The ratchet mechanism became well-known through the Signal protocol [111] employing the Double Ratchet Algorithm [112].

[111]: Marlinspike et al. (2016), *The X3DH Key Agreement Protocol*

[112]: Perrin et al. (2016), *The Double Ratchet Algorithm*

While employing the Double Ratchet Algorithm, or more generally a ratchet, achieves strong security guarantees for a protocol design, these security guarantees carry over to an implementation only if past key material is securely deleted. If an implementation retains, e.g., all past session keys, corrupting a participant session would allow the attacker to decrypt past messages without having to invert the employed one-way function as all session keys leak to the attacker.

In this section, we present an extension of our methodology that enforces secure deletion of past key material. Without cooperation with the compiler (as, e.g., Olmos *et al.* [113] propose for the JASMIN compiler) and operating system, secure deletion, however, remains best effort as sensitive data, such as keys, may get copied arbitrarily or delete operations therefor may get eliminated by the compiler [114]. We designed our extension to support off-the-shelf program verifiers, which made it impossible simply enforce an invariant for every heap location and stack variable stating that the stored value is either not sensitive, has been securely deleted, or is currently required to execute the protocol. More specifically, enforcing

[113]: Olmos et al. (2024), *High-Assurance Zeroization*

[114]: Yang et al. (2017), *Dead Store Elimination (Still) Considered Harmful*

Figure 3.17: Definition of the three abstract predicates used to count heap locations storing sensitive data per epoch.

```
pred EpochGuard(sessionId uint32, epoch uint32)
pred NextEpochGuard(sessionId uint32, epoch uint32)
pred IsSensitive(data []byte, sessionId uint32, epoch uint32)
```

such an invariant is not readily supported by program verifiers as, e.g., a separation-logic-based verifier like Gobra is not even aware of the complete set of allocated heap locations due to thread-local, modular reasoning. DY* [39] side-steps this quantification over the entire program state by requiring that implementations store their program state after every protocol step in serialized form on the global trace, which has several drawbacks. First, existing implementations do not satisfy this requirement; second, serializing and deserializing all program state adds runtime overhead; third and as already discussed in Sec. 1.1.3, this approach imposes a coding discipline for protocol steps as the invariant is not enforced within a protocol step, and lastly, storing *actual* program state on the global trace makes it impossible to treat the global trace as a *ghost* data structure, as the global trace's existence is required at runtime. While our extension is inspired by DY*, we overcome its drawbacks resulting in a solution that is applicable to existing implementations and off-the-shelf program verifiers and retains the global trace as a *ghost* data structure.

[39]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

Counting Sensitive Data. We split the lifetime of a security protocol's session into so-called *epochs* and extend secrecy labels to determine in which epochs sensitive data is allowed to be stored. In a nutshell, our extension hinges on keeping track of the number of heap locations per epoch storing sensitive data. This allows us to enforce, before moving to the next epoch, that an implementation has either securely deleted each heap location or proved that the stored data therein is allowed to be in memory in the next epoch, according to the stored data's secrecy label. For simplicity, the current epoch is stored locally in an instance of our verification library and, thus, we assume that only a single instance of our verification library is created per protocol session. Lifting this restriction is possible as we discuss in Chapter 5.

Permission accounting [115] is a separation-logic-based permission model that keeps track of how many so-called *shares* have been handed out from a factory resource and not yet recollected. While this permission model could be used for our purpose, most off-the-shelf program verifiers do not readily support it. In addition, allowing multiple threads to obtain a share from the same factory resource would require synchronization, e.g., via a ghost lock. We overcome the lack of support for permission accounting in off-the-shelf program verifiers and avoid synchronization when allocating a new heap location storing sensitive data and securely deleting such a heap location. For this purpose, we introduce three abstract predicates (cf. Fig. 3.17) to count the number of heap locations storing sensitive data per epoch, which rely on the support for fractional permissions offered by most program verifiers and natively support concurrent programs. We assume that epochs are sequentially numbered starting from zero and are part of a session that is identified via some session identifier. A *full* permission of `EpochGuard(s,e)` expresses that no heap location exists storing sensitive data belonging to epoch `e` in the session identified by `s`. Similarly, `NextEpochGuard(s,e+1)` keeps track of heap locations storing sensitive data belonging to the next epoch `e+1`, whose necessity will become clear later. A protocol session starts off with `EpochGuard(s,0)` $\star$ `NextEpochGuard(s,1)` as a free precondition indicating that no heap locations storing sensitive data have been allocated yet in this protocol

[115]: Bornat et al. (2005), *Permission Accounting in Separation Logic*

session. Whenever sensitive data is stored in a heap location `h`, a non-zero fraction f of `EpochGuard(s,e)` is consumed while the same amount f of `IsSensitive(h,s,e)` is produced. `IsSensitive(h,s,e)` acts as a receipt that heap location `h` stores sensitive data belonging to epoch `e`. Since all operations that depend on sensitive data occur within our verification library[7], the library is solely responsible for performing these conversions of permissions. We achieve these conversions by adding trusted specifications to cryptographic functions like nonce generation or decryption since these functions may place sensitive data that should only exist in certain epochs in newly allocated or already existing heap locations.

7: Our library uses an abstract predicate to abstract over the memory footprint of byte arrays used by the library. As this predicate is abstract, clients of our library cannot obtain permissions to and, thus, directly access the contents of these byte arrays.

The trusted specifications of our verification library maintain the following informal invariants about the predicates in Fig. 3.17. We omit invariants involving `NextEpochGuard(s,e)` and introduce them later after motivating this predicate's necessity.

$$\begin{pmatrix} \mathit{perm}(\texttt{EpochGuard(s,e1)}) > 0 \\ \wedge \\ \mathit{perm}(\texttt{EpochGuard(s,e2)}) > 0 \end{pmatrix} \implies \texttt{e1 = e2} \tag{3.3}$$

$$\mathit{perm}(\texttt{EpochGuard(s,e)}) + \sum_{\texttt{h}} \mathit{perm}(\texttt{IsSensitive(h,s,e)}) \leq 1 \tag{3.4}$$

where all free variables are for all quantified and $\mathit{perm}(p)$ returns the global permission amount of predicate instance p, implicitly quantifying over all threads in a program. Invariant (3.3) states that `EpochGuard` instances agree on a particular epoch for a session `s`. Invariant (3.4) expresses that a fraction f of a `IsSensitive(h,s,e)` resource temporarily replaces the same amount f of an `EpochGuard(s,e)` resource until heap location `h` is securely deleted. While invariant (3.4) could in theory be strengthened to an equality, it is an upper bound in practice as a client of our verification library might arbitrarily drop permission amounts, which does not result in a verification error in several program verifiers, including Gobra.

Having introduced the general mechanism behind tracking heap locations storing epoch-specific sensitive data, we now explain how this mechanism is used in practice by considering three use cases, namely creating an epoch-specific ephemeral key, securely deleting such data, and transitioning from the current to the next epoch. We omit, e.g., decrypting a ciphertext containing an epoch-specific plaintext because copying such sensitive data into a heap location is similar to creating an epoch-specific ephemeral key. We end this section with details on the implementation of secure deletion in our verification library and a case study extending WireGuard and its implementation to provide forward secrecy and post-compromise security even in the presence of ephemeral key corruption.

Epoch-Specific Key Creation. We extend our verification library's nonce creation function with an additional precondition requiring a non-zero fraction f of `EpochGuard(s,e)` if an epoch-specific nonce should be created. This precondition not only uniquely determines the current epoch (cf. invariant (3.3)) but also consumes the same fraction f of `EpochGuard(s,e)` indicating that sensitive data belonging to this epoch exists in the program state. If an epoch-specific nonce should be created, an additional postcondition returns f amounts of `IsSensitive(h,s,e)`,

```
1  //@ trusted
2  //@ requires Mem(h)
3  //@ requires 0 < f
4  //@ requires acc(IsSensitive(h, rvlib.Session(), rvlib.Epoch()), f)
5  //@ ensures  acc(EpochGuard(rvlib.Session(), rvlib.Epoch()), f)
6  func DeleteSecurely(h []byte/*@, ghost rvlib *Lib, ghost f perm @*/) {
7    // overwrite content
8    for i := range h {
9      h[i] = 0
10   }
11   // prevent compiler from optimizing away the deletion
12   runtime.KeepAlive(h)
13   return
14 }
```

Figure 3.18: Implementation and trusted specification of the secure deletion function in the reusable verification library. `Mem(h)` abstracts over the permissions for the elements of the byte slice `h`. `acc(p, f)` expresses `f` permission amounts of predicate `p` in Gobra's syntax. `rvlib.Session()` and `rvlib.Epoch()` retrieve the current session identifier and epoch, respectively, from our reusable verification library. We omit the permissions for `rvlib`, which is required to, e.g., call these two functions, in all code listings within this section.

where `h` refers to the heap location that stores the freshly created nonce and is returned by this function.

This specification highlights our native support for concurrency. In particular, permissions to the `EpochGuard(s,e)` resource can be partitioned arbitrarily among multiple threads, allowing each thread to create epoch-specific nonces by choosing an appropriately small fraction f when invoking the library's nonce creation function. As fractional permissions can be split arbitrarily often, we neither limit the number of threads that can create nonces nor the amount of creatable nonces in general.

Secure Deletion. We add a function to our verification library that securely deletes sensitive data stored in a heap location `h`. Its implementation highly depends on the programming language and compiler and must take compiler optimizations such as dead store elimination into account. Ideally, its implementation utilizes compiler intrinsics to inform the compiler of the sensitive nature of the data stored in heap location `h` and that this location should be securely deleted. Olmos *et al.* [113] and Yang *et al.* [114] survey secure deletion implementations found in various open-source projects and Fig. 3.18 shows our implementation for the Go programming language. We rely on the Go compiler intrinsic `runtime.KeepAlive` to prevent optimizations, however, at the cost of keeping the heap location `h` alive and, thus, preventing garbage collection after overwriting its content.

[113]: Olmos et al. (2024), *High-Assurance Zeroization*

[114]: Yang et al. (2017), *Dead Store Elimination (Still) Considered Harmful*

We equip our secure deletion function (cf. Fig. 3.18) with a precondition requiring full permission to the heap location `h` (line 2), which is a shorthand notation for `acc(Mem(h), 1/1)`. This permission allows the secure deletion function to overwrite the content of `h` and, unless we are in a garbage-collected programming language, deallocate `h`. In addition, we require a non-zero fraction `f` of `IsSensitive(h,s,e)` (lines 3–4) to bind the permission amount `f` that is used in the postcondition to return `f` permission amounts of `EpochGuard(s,e)` (line 5), where `s` and `e` are the current session identifier and epoch, respectively, which our verification library provides.

Clients of our verification library ideally pick the permission amount `f` when invoking the secure deletion function as large as possible, i.e., to be the same amount as was used for the allocation of the heap location `h`, as we explained for the creation of nonces. If clients choose the permission amount `f` in this way, the secure deletion function undoes the swap of resources that the allocation of `h` performed:

$$\begin{tikzcd} \texttt{acc(EpochGuard(s,e),f)} \arrow[r, shift left, "\text{alloc h}"] & \texttt{acc(IsSensitive(h,s,e),f)} \arrow[l, shift left, "\text{securely delete h}"] \end{tikzcd}$$

```
1 //@ ghost
2 //@ trusted
3 //@ requires Allowed(hT,rvlib.Snap(),rvlib.Session(),rvlib.Epoch()+1)
4 //@ requires 0 < f
5 //@ requires acc(IsSensitive(h, rvlib.Session(), rvlib.Epoch()), f)
6 //@ requires acc(NextEpochGuard(rvlib.Session(), rvlib.Epoch()+1), f)
7 //@ ensures  acc(IsSensitive(h, rvlib.Session(), rvlib.Epoch()+1), f)
8 //@ ensures  acc(EpochGuard(rvlib.Session(), rvlib.Epoch()), f)
9 //@ func CarryForward(h []byte, hT Term, rvlib *Lib, f perm)
```

Figure 3.19: Trusted ghost function in our verification library to carry a heap location `h` forward from epoch `e` to the next epoch `e+1`. `hT` is the term corresponding to `h`'s content and `Allowed` checks that this content may be present in epoch `e+1` according to `hT`'s secrecy label and taking potential corruption (as reflected in the local snapshot) into account. We omit specification that binds `h` and `hT`.

Otherwise, some fraction f' of `EpochGuard(s,e)` with $0 < f'$ is unrecoverably lost. This loss of permissions occurs because a client retains f' permission amounts of `IsSensitive(h,s,e)` and cannot call `DeleteSecurely` for heap location `h` again, since `h` is deleted during the first invocation. As we will cover next, this loss of permissions prevents any future epoch transitions.

Epoch Transition. By enforcing that we transition to the next epoch `e+1` only if we possess *full* permission to `EpochGuard(s,e)`, we ensure that all heap locations storing sensitive data belonging to epoch `e` have been securely deleted. The mechanism present so far, however, would be insufficient to support realistic security protocol implementations as we would be forced to delete all sensitive data before transitioning to the next epoch. Considering a security protocol using a ratchet mechanism, this would mean that an ephemeral key k^{e+1} of an epoch `e+1` cannot depend on the ephemeral key k^{e} of the previous epoch `e` because having to delete k^{e} before transitioning to epoch `e+1` implies that we must already compute k^{e+1} shortly before the transition, i.e., in epoch `e`. Without the possibility to carry the ephemeral key k^{e+1} forward to the next epoch `e+1`, we could transition only if we delete k^{e} *and* k^{e+1} in epoch `e`.

We enable carrying sensitive data, like k^{e+1}, forward to the next epoch `e+1` by introducing the `NextEpochGuard(s,e+1)` resource (cf. Fig. 3.17). Conceptually, we alter a heap location `h` storing sensitive data belonging to an epoch `e` to storing sensitive data belonging to the next epoch `e+1`. However, this alteration comes with a proof obligation that `h` stores data that may be present in epoch `e+1` and, thus, does not have to be securely deleted in the current epoch. As shown in Fig. 3.19, we can trade a fraction `f` of permissions to the `IsSensitive(h,s,e)` resource (line 5) for `f` permissions to `IsSensitive(h,s,e+1)` (line 7), which corresponds to marking the heap location `h` as storing sensitive data that no longer belongs to epoch `e` but now belongs to the next epoch `e+1`. To reflect that after performing this carry forward operation less sensitive data belonging to epoch `e` exists, line 8 ensures `f` permissions to the `EpochGuard(s,e)` resource, which allows transitioning to the next epoch after all sensitive data belonging to epoch `e` has either been securely deleted or carried forward to the next epoch. The precondition on line 6 consumes `f` permissions to the `NextEpochGuard(s,e+1)` resource to record that sensitive data belonging to the next epoch `e+1` will exist when this function returns. Finally, line 3 enforces the side-condition that carrying the heap location `h` forward does not violate the corresponding secrecy label.

Having introduced the `NextEpochGuard(s,e+1)` resource, our verification library maintains the following invariant involving this resource.

$$\left(\begin{array}{c} \mathit{perm}(\texttt{NextEpochGuard(s,e+1)}) \\ + \\ \sum_{\texttt{h}} \mathit{perm}(\texttt{IsSensitive(h,s,e+1)}) \end{array} \right) \leq 1 \qquad (3.5)$$

```
1 //@ ghost
2 //@ requires EpochGuard(rvlib.Session(), rvlib.Epoch())
3 //@ requires 0 ≤ f
4 //@ requires acc(NextEpochGuard(rvlib.Session(), rvlib.Epoch()+1), f)
5 //@ ensures  rvlib.Epoch() == old(rvlib.Epoch()) + 1
6 //@ ensures  acc(EpochGuard(rvlib.Session(), rvlib.Epoch()), f)
7 //@ ensures  NextEpochGuard(rvlib.Session(), rvlib.Epoch()+1)
8 //@ func Transition(f perm)
```

Figure 3.20: Specification for the library function that transitions to the next epoch. We omit the permissions that are required to access the verification library's state and bump the epoch therein.

Similar to invariant (3.4), invariant (3.5) expresses that the library swaps fractions of `NextEpochGuard(s,e+1)` for fractions of `IsSensitive(h,s, e+1)` when allocating or copying sensitive data into a heap location `h`, and vice versa when securely deleting such a heap location, while keeping their sum upper bounded by one.

Since the `NextEpochGuard(s,e+1)` resource allows us to carry sensitive data forward to the next epoch before transitioning, we obtain the specification as shown in Fig. 3.20 for the transition function in our verification library. In the following, we refer to the epoch before the transition as `e` and the next epoch as `e+1`. Line 2 requires *full* permission to the `EpochGuard(s,e)` resource, which ensures that all sensitive data belonging to epoch `e` has either been securely deleted or carried forward to the next epoch. After performing the transition, line 6 returns a fraction `f` amounts of permissions to the succeeding (cf. line 5) epoch's guard, i.e., `EpochGuard(s,e+1)`, which accounts for the carried forward sensitive data. More precisely, line 4 binds this fraction `f` to the permission amount of `NextEpochGuard(s,e+1)` that a client of our verification library passes to the transition function. The fraction `f` ideally corresponds to the upper summand in invariant (3.5)[8]. Lastly, line 7 returns full permissions to the `NextEpochGuard(s,e+2)` resource, indicating that epoch `e+2` will be the next epoch and that no sensitive data belonging to this epoch exists yet.

8: A client of our verification library might choose a smaller fraction `f` than the upper summand in (3.5), whose value we denote as s in the following. In this case ($\mathtt{f} < s$), however, this client will be unable to transition ever again as the client can recover at most `f` permissions to `EpochGuard(s,e+1)` from `f` permissions to `NextEpochGuard(s,e+2)` (line 7) by securely deleting or carrying forward heap locations. Hence, $s - \mathtt{f}$ permissions to `EpochGuard(s,e+1)` are unrecoverable, thus, making any future application of the transition function impossible.

Implementation and Case Study. To demonstrate that our extension enables proving strong security properties taking corruption of ephemeral keys into account, we forked our Go verification library and an earlier, sequential version of our WireGuard implementation (cf. Sec. 3.6.3).

We added the functions described in this section to our verification library. Furthermore, we added a function to bypass the secure deletion mechanism for a particular heap location if one can prove that this heap location stores data that is allowed to be present for the entirety of the program execution, i.e., protocol session. This function is useful to reduce the proof burden for, e.g., non-sensitive parts of a plaintext as our library conservatively assumes that plaintexts contain data that is as sensitive as the decryption key. In addition, we added a KDF implementation and corresponding term abstraction representing the application of a ratchet operation to our reusable verification library. This function takes two parameters, namely the current ephemeral key and some entropy, and returns the next ephemeral key. We assume that the underlying implementation allows recomputing an output value only if both inputs are known. We soundly over-approximate the output's secrecy label to match the second input's secrecy label[9]. We verify our extended verification library in the same once-and-for-all manner as described in Sec. 3.5. However, the library does not yet provide lemmata proving forward secrecy considering ephemeral key compromise and post-compromise security for data that has epoch-specific secrecy labels. However, we expect that these lemmata are straightforward extensions of our secrecy lemma (cf. Sec. 3.4.2).

9: The most precise secrecy label would be the intersection of both inputs' secrecy labels reflecting that an attacker requires both inputs to apply the KDF. However, to provide maximum flexibility to clients of our verification library, the KDF output's secrecy label should be customizable for a given security protocol as any secrecy label between this most precise secrecy label and public (cf. Sec. 3.4.2) would be a sound choice.

We adapted the WireGuard protocol's transport phase to continuously exchange DH public keys, compute DH shared secrets, and use these shared secrets as entropy for updating the symmetric key used for encrypting transport messages by applying a ratchet operation. Since the duration of an epoch is protocol-specific, we chose a participant's epoch to last for one transport message round trip, which minimizes the provable duration in which ephemeral keys remain in memory. Using a single ratchet is sufficient as each request is followed by a response. To support sending multiple transport messages before requiring a response, a similar design to Signal's Double Ratchet Algorithm could be employed, where the output of our KDF is not immediately used as the encryption key for transport messages but instead used as input to a second KDF that uses a message counter as additional input to derive a unique key per transport message.

To illustrate the adapted protocol, let us consider an instance of the initiator role that just completed the handshake phase and, thus, is in epoch 0 and derived the symmetric key k_{IR}^{0} for encrypting the first transport message[10]. Before sending this transport message, the initiator generates a DH secret key x^0 and includes the corresponding DH public key as additional authenticated data in the transport message's AEAD component. After receiving this transport message, the responder likewise generates a DH secret key y^0 and replies with a transport message containing the corresponding DH public key in the same way. The initiator uses this DH public key and its own DH secret key x^0 to compute the DH shared secret ss^0, which forms the entropy for performing the ratchet operation to derive the next symmetric key k_{IR}^{1}. As hinted at by k_{IR}^{1}'s superscript, we transition to epoch 1 before the initiator sends the next transport message.

10: We use superscripts to indicate the epoch in which a key is (mainly) used. Since no ratchet operation has been performed to derive k_{IR}^{0}, we have $k_{IR}^{0} = k_{IR}$, where k_{IR} follows the notation in Sec. 3.6.3.

The interplay of ephemeral keys with epochs illustrates the necessity of carrying sensitive data forward to the next epoch as ephemeral keys cannot be assigned to a single epoch. In particular, the computation of k_{IR}^{1} depends on the DH secret keys that occur in the round trip in epoch 0 while k_{IR}^{1} is used as the symmetric key for encrypting the initiator sent transport message in the next epoch 1. No matter whether we transition to epoch 1 before or after computing k_{IR}^{1}, either the shared secret ss^0 or k_{IR}^{1} must be carried forward to epoch 1.

We adapted the WireGuard implementation to generate the required DH secret keys, add their public keys to transport messages, and employ the KDF to derive the transport messages' encryption keys. Additionally, we inserted calls to securely delete the DH secret keys, DH shared secrets, and the transport messages' encryption keys when they are no longer needed, and a call to transition to the next epoch whenever a round trip is completed from the perspective of a protocol participant. We verified the adapted WireGuard initiator and responder implementations with simplifying code changes and assumptions, to demonstrate that our methodology's extension to secure deletion is powerful enough to support such a ratchet-based protocol implementation. More specifically, we unrolled the main loop sending transport messages and assume that the attacker has no access to the initial transport message keys that result from the handshake phase to simplify the verification effort. We prove particular secrecy labels for the transport message keys that we derive by applying the KDF. Even though these keys are symmetric and, thus, derived by both the initiator and responder, their proved secrecy labels differ due to partial knowledge about the other communication partner's epoch. We prove in the implementation of one protocol role that such a derived transport message key is present only in a specific protocol session and two subsequent epochs of this role and possibly any session and epoch of the other role. To achieve such secrecy labels, we indicate when generating a DH secret key in epoch e that this key may exist in epochs e and $e + 1$ by choosing an appropriate secrecy label, which then allows us to carry either the corresponding DH shared secret or the derived symmetric transport message key forward to the next epoch. This faithfully represents the circumstances that there exists a short period of time immediately after applying the KDF in which the current's and next's epochs' transport message keys are simultaneously present in memory such that corrupting this state would reveal both keys to the attacker. We refer to Hugo Queinnec's Master's Thesis [78] for further details. In Chapter 5, we will discuss how the proved secrecy labels could be further strengthened to provably restrict the sessions and epochs of the other protocol role in which these transport message keys may occur.

[78]: Queinnec (2023), *Secure Deletion of Sensitive Data in Protocol Implementations*

3.10 Refinement-Based vs. Invariant-Based Verification of Security Protocol Implementations

Having presented two orthogonal methodologies to verifying security protocol implementations—one in Chapter 2 and one in this chapter—we now compare their strengths and weaknesses, focusing on four aspects, namely their support for incremental verification, the level of automation they enable, their applicability to concurrent implementations, and how easily security guarantees can be extended from, e.g., keys to message payloads.

Incremental Verification. As security properties like secrecy or authentication typically hold only if multiple protocol roles behave correctly, a global view on the distributed system is required to prove security properties about it. Our refinement-based methodology (Chapter 2) utilizes a protocol's Tamarin model as such a global view, while our invariant-based methodology (this chapter) obtains this global view from the trace invariant specifying properties about protocol roles' behavior. In the former methodology, Tamarin provides an abstract way of stating the behavior of protocol roles, which enables fast modeling and, if necessary, iterating quickly over different versions of a protocol until a version satisfies the desired security properties. Once this abstract view is established, the refinement-based methodology allows us to verify a particular protocol role's implementation in isolation from all other roles. Although every implementation of all protocol roles must be verified to ensure that the distributed system satisfies the proven security properties, this methodology allows us to focus, e.g., on particularly critical protocol roles like the role checking an Authentic Digital EMblem (ADEM) (cf. Sec. 2.7), and to verify their implementations first, thus, providing a way to incrementally verify the implementations of all protocol roles.

In contrast, our invariant-based methodology proves security properties with respect to a trace invariant. Hence, it is crucial that this trace invariant is correct, which is established by proving all implementations of protocol roles against the same trace invariant. Without verifying at least one implementation per protocol role against the trace invariant, we cannot be sure that we can even implement a protocol role in a way that satisfies the trace invariant. Therefore, our refinement-based methodology provides more flexibility in this aspect.

Automation. As demonstrated by our evaluations, our refinement-based methodology requires less specification than the invariant-based one and a part thereof is automatically generated from the Tamarin model. In addition, the former methodology utilizes Tamarin's proof search that is specifically tailored to security protocols, allowing us to utilize advances in protocol model verifiers from the past decades. However, Tamarin's proof search is not modular in the sense that Tamarin tries to prove the absence of a valid sequence of transitions from an initial state to every state that violates a given security property. Especially for large and complex protocols that, e.g., exhibit looping behavior or make use of DH exponentiation like WireGuard, this quickly leads to a non-terminating proof search. Tamarin provides several ways to mitigate this issue, e.g., handwritten or automatically generated auxiliary lemmata [116] can not only serve as shortcuts within Tamarin's proof search[11] but also prune the search space[12]. Furthermore, Tamarin provides several heuristics to control the order in which different branches are explored during the proof search and, ultimately, a custom Python oracle provides full control over this order.

[116]: Cortier et al. (2020), *Automatic Generation of Sources Lemmas in Tamarin: Towards Automatic Proofs of Security Protocols*

11: Lemmata annotated with `reuse` are considered while proving a lemma if they syntactically precede this lemma.

12: Tamarin considers lemmata annotated with `sources` during its precomputations to determine all possible ways an attacker might obtain a particular term.

In contrast, our invariant-based methodology utilizes the trace invariant to modularize the proof. Conceptually, the trace invariant not only serves as a boundary between a protocol role's implementation and the rest of the distributed system, but also provides clear proof obligations for every protocol-relevant operation therein, such as proving that a message's secrecy label permits sending this message to the network. Although highly subjective and influenced by my expertise in Gobra[13], I personally perceive our invariant-based methodology as more goal-oriented at the cost of requiring more specification and proof annotations than our refinement-based methodology. Namely, manually controlling Tamarin's proof search to prove (or disprove) a lemma, spotting certain patterns in its proof search, and eventually writing a custom oracle is in my experience tedious as this process is time-consuming and mostly trial and error. E.g., this process took me more than one person month for the protocol described in Sec. 4.5.1. In our invariant-based methodology, it is usually clear whether the trace invariant has to be strengthened to allow a protocol role to derive certain conclusions after, e.g., receiving a message or whether additional proof annotations are required to derive a conclusion, e.g., about a key's or message's secrecy label.

13: For this reason, the remainder of this paragraph is written in the first-person singular.

Concurrency. As mentioned in Sec. 2.7.3 and Sec. 3.6.3, both methodologies support verifying concurrent implementations. Since the refinement-based methodology uses a resource in I/O separation logic to express the I/O operations that an implementation may perform, a synchronization primitive must be used to verify concurrent threads performing I/O operations. If these threads perform I/O operations that are independent of operations performed by other threads, no synchronization might be required at runtime, so ghost synchronization must be employed for verification purposes, e.g., in the form of a ghost lock. In this case, the lock invariant must provide sufficient information about the abstract state such that the threads can use the I/O specification to justify their I/O operations. Since the lock invariant conceptually owns the I/O specification and abstract state, additional ghost heap locations are required such that each thread can keep track of its contributions to the abstract state. These ghost heap locations abstractly serve the same purpose as

[117]: Owicki et al. (1976), *Verifying Properties of Parallel Programs: An Axiomatic Approach*

the auxiliary variables in Owicki and Gries's counter example [117, Fig. 1] or our local snapshots.

In contrast, our invariant-based methodology does not require ghost synchronization primitives outside the reusable verification library. As each thread has its own instance of the verification library, this methodology does not introduce dependencies between I/O operations of different threads and these I/O operations must satisfy only the trace invariant. If different threads perform I/O operations that depend on each other, e.g., sending and receiving payloads encrypted with the same key, an implementation will employ some (non-ghost) synchronization primitives or perform read-only accesses to avoid data races. In these situations, our methodology is flexible enough to use these already existing synchronization primitives to share knowledge about the trace between these threads and their respective library instances, as shown in our WireGuard case study (cf. Sec. 3.6.3). Hence, our invariant-based methodology is easier to apply to concurrent implementations.

Guarantees for Payloads. In the evaluations of both methodologies, we focused on proving security properties like secrecy and authentication for *keys* that are established by running a security protocol. In practice, these keys are subsequently used to encrypt message payloads, such as VPN packets in the case of WireGuard. Ideally, security properties established for encryption keys should extend to the message payloads that are encrypted with those keys. In both methodologies, proving security properties for message payloads and keys is analogous. In our refinement-based methodology, security properties about message payloads are proved by adapting the lemmata or labels of transitions in the TAMARIN model such that they also cover message payloads and not only keys. The security properties that TAMARIN can prove for message payloads depend not only on the security protocol establishing the encryption keys for these payloads but also on how message payloads are modeled. In our case studies, message payloads are modeled as terms that are obtained from the *attacker*, which makes the attacker in charge of choosing the payloads sent by protocol participants. Thus, secrecy for message payloads is trivially violated. Alternatively, message payloads can be modeled as *fresh* terms [118], which are a priori unknown to the attacker. However, this modeling choice imposes restrictions on the set of payloads that TAMARIN considers in its proof search. E.g., sending any public constant or a term that occurs earlier in a protocol run falls outside this set. Therefore, both modeling choices have advantages and disadvantages, which is why, e.g., Girol *et al.* [107] employ both modeling choices and select them according to the security property they aim to prove.

[118]: Basin et al. (2026), *Modeling and Analyzing Security Protocols with Tamarin - A Comprehensive Guide*

[107]: Girol et al. (2020), *A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie–Hellman Protocols*

In contrast, our invariant-based methodology uses secrecy labels to specify for a given term whether the attacker and which protocol role instances may obtain this term. Thus, it is straightforward to assign a particular secrecy label to a message payload by introducing an assumption. E.g., we specify for a VPN packet that the initiator role of WireGuard obtains from the operating system, that this packet has some arbitrary term representation and that only this particular instance of the initiator and any instance of the responder role may obtain this term, unless any of these instances got compromised. This forces the initiator to encrypt this packet using a sufficiently strong encryption key before sending the resulting ciphertext to the network. Otherwise, verification fails as the corresponding send operation does not preserve the trace invariant. Therefore, our invariant-based methodology provides a more straightforward way to specify and prove security properties for message payloads than our refinement-based methodology.

To summarize, incremental verification is supported exclusively by our refinement-based methodology and our invariant-based methodology makes verifying concurrent implementations and guarantees for message payloads easier. Regarding automation, there is no clear winner as our refinement-based methodology results in a lower specification overhead and more automation as it utilizes TAMARIN. However, in case of a non-terminating proof search, our invariant-based methodology provides a more principled approach to proving security properties at the expense of requiring more specifications.

3.11 Related Work

Complementing Sec. 1.1 and Sec. 2.8, we focus here on *modular verification of symbolic security properties*, and discuss the most closely related work first: techniques for verifying security of *realistic protocol implementations*.

Dupressoir *et al.* [64] use VCC [66] to verify memory safety, non-injective agreement, and (via an external argument in COQ) weak secrecy, of two protocols implemented in C: RPC and Otway–Rees. To our knowledge, they are the first to encode a global protocol trace ("log") as a concurrent data structure. We generalize this idea to separation logic to make it much more widely applicable, because their encoding relies on C's `volatile` fields and a VCC-specific program logic, neither of which are (widely) available in other languages and verifiers. Moreover, since their logic (unlike separation logic) does not provide linear resources, proving injective agreement would require a nontrivial extension of their work. Their set-based trace encoding prevents proving, e.g., forward secrecy (which we do); they account for principal corruption, but not session corruption (we account for both). Polikarpova *et al.* [119] extend this work by incorporating stepwise refinement to formally connect a model to an existing implementation, all encoded in VCC . This refinement decomposes the verification into smaller steps, but incurs additional overhead. Moreover, they remove the need for external arguments when proving weak secrecy. They verify the latter, and a variant of authentication, for a small stateful subset of TPM 2.0.

[64]: Dupressoir et al. (2011), *Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols*

[66]: Cohen et al. (2009), *VCC: A Practical System for Verifying Concurrent C*

[119]: Polikarpova et al. (2012), *Verifying Implementations of Security Protocols by Refinement*

Vanspauwen *et al.* [69], like us, use a separation-logic-based verifier (VERIFAST [26]), but they do not model a global trace. Consequently, properties that are commonly expressed over a trace potentially need to be assembled from individual assertions. They propose an extended symbolic model that strengthens attackers by permitting byte-wise manipulations, such as splitting and reconcatenating byte sequences, in addition to the usual symbolic manipulations. Our attacker operates on terms (standard for symbolic cryptography) but we could adapt their extension. They specify PolarSSL's API using this extended model, and then verify secrecy and non-injective agreement of an NSL-implementation (and a few less complex protocols). Unlike us, they do not consider session corruption.

[69]: Vanspauwen et al. (2015), *Verifying Protocol Implementations by Augmenting Existing Cryptographic Libraries with Specifications*

[26]: Jacobs et al. (2011), *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*

Bhargavan *et al.* [39] suggest DY*: a framework for verifying protocols implemented in F* [42], a functional language that enables type-system-based proofs, e.g., using monadic effects and refinement types. DY* introduces the idea of a parametric library for reducing the per-protocol proof effort; an idea we adopted. DY*'s type system is tailored to F*,

[39]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

[42]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

whereas our methodology supports a wide range of languages and tools. Moreover, by building on separation logic, we are able to prove stronger properties, in particular, injective agreement. Our methodology can be applied directly to existing implementations, as we demonstrate in the WireGuard case study. In contrast, DY* supports code generation, but additionally requires a handwritten (and partly protocol-specific) runtime wrapper [40]. Included in DY*'s case study is the first automated verification of Signal [111] that proves forward and post-compromise security over an unbounded number of protocol messages. Our main case study is WireGuard, for which we prove, also for an unbounded number of messages, forward secrecy and injective agreement with AKC resilience. Soundness of DY*'s global protocol trace depends on a specific coding discipline (one method per protocol step) that is not automatically enforced. If missed, the attacker is accidentally restricted, and security properties can be proven incorrectly.

[40]: Bhargavan et al. (2021), *An In-Depth Symbolic Security Analysis of the ACME Standard*

[111]: Marlinspike et al. (2016), *The X3DH Key Agreement Protocol*

An earlier line of work (e.g., [67, 68, 120]) verifies security of functional programs written in F# using the F7 type checker [68], but does not integrate equational theories, and has limited support for mutable state. Moreover, this work does not model the global protocol traces and, thus, states security properties only implicitly.

[67]: Bhargavan et al. (2010), *Modular Verification of Security Protocol Code by Typing*

[68]: Bengtson et al. (2008), *Refinement Types for Secure Implementations*

[120]: Bhargavan et al. (2010), *Typechecking Higher-Order Security Libraries*

Küsters *et al.* [121] share our goal of reusing existing program analyzers and suggest an approach that enables non-interference checkers to establish computational indistinguishability results for sequential programs. To account for closed-system assumptions (typically made by such checkers) in the presence of an attacker-controlled environment, they restrict interaction with the latter to static, exception-free methods, and primitive (i.e., value) types. How to extend their approach to trace-based properties and concurrent programs remains unclear.

[121]: Küsters et al. (2012), *A Framework for the Cryptographic Verification of Java-Like Programs*

Several security property verifiers exist that (unlike us) do not reuse existing program analyzers, e.g., Csur [122] and ASPIER [123] (for C), and JavaSec [124] (for Java). However, to reduce development costs, such domain-specific tools typically only implement semantics of a restricted language subset and, e.g., assume crucial properties such as memory safety (which may render implementations insecure, e.g., due to buffer overflows).

[122]: Goubault-Larrecq et al. (2005), *Cryptographic Protocol Analysis on Real C Code*

[123]: Chaki et al. (2009), *ASPIER: An Automated Framework for Verifying Security Protocol Implementations*

[124]: Jürjens (2006), *Security Analysis of Crypto-Based Java Programs using Automated Theorem Provers*

Prior work [41, 105–107, 125, 126] on verifying properties of WireGuard (our main case study) is concerned with verifying models of the protocol, not existing implementations.

[41]: Ho et al. (2022), *Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations*

[105]: Dowling et al. (2018), *A Cryptographic Analysis of the WireGuard Protocol*

[106]: Lipp et al. (2019), *A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol*

[107]: Girol et al. (2020), *A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie–Hellman Protocols*

[125]: Donenfeld et al. (2018), *Formal Verification of the WireGuard Protocol*

[126]: Kobeissi et al. (2019), *Noise Explorer: Fully Automated Modeling and Verification for Arbitrary Noise Protocols*

Finally, a large body of work is concerned with mechanizing the verification of computational (rather than symbolic) properties; several surveys [19, 23, 38] provide an overview. This line of work establishes stronger guarantees by making weaker, more realistic cryptographic assumptions. For instance, Owl [51] allows one to verify computational security of protocols written in a dedicated language. Like in our work, their proofs are automated and compositional. Recently, OwlC [52] added a refinement-based technique for verifying implementations that Owl generates, as discussed in detail in Sec. 2.8. Meanwhile, the first separation logics for probabilistic reasoning have been proposed [127–129], but we are not aware of automated verifiers for such logics.

[19]: Barbosa et al. (2021), *SoK: Computer-Aided Cryptography*

[23]: Blanchet (2012), *Security Protocol Verification: Symbolic and Computational Models*

[38]: Avalle et al. (2014), *Formal Verification of Security Protocol Implementations: a Survey*

[51]: Gancher et al. (2023), *Owl: Compositional Verification of Security Protocols via an Information-Flow Type System*

[52]: Singh et al. (2025), *OwlC: Compiling Security Protocols to Verified, Secure, High-Performance Libraries*

[127]: Tassarotti et al. (2019), *A Separation Logic for Concurrent Randomized Programs*

[128]: Batz et al. (2019), *Quantitative Separation Logic: A Logic for Reasoning about Probabilistic Pointer Programs*

[129]: Barthe et al. (2020), *A Probabilistic Separation Logic*

Conclusions. Our methodology for the modular verification of security protocol implementations enables proving strong security properties for realistic protocol implementations. By employing separation logic, we support efficient implementations using, e.g., heap data structures, side effects, and concurrency. Encapsulating the global trace in a concurrent ghost data structure and our use of invariants over local snapshots allow our methodology to support arbitrary code structures and data representations, which is crucial for targeting existing implementations. Separation logic also allows us to specify resources in the trace invariant to express uniqueness of protocol-specific events, which is key to modularly proving injective agreement. Our case studies, which use different programming languages and program verifiers, demonstrate that our methodology handles existing and interoperable implementations of protocols and proves strong security properties, such as forward secrecy and injective agreement.

Scaling Verification of Security Protocol Implementations to Large Codebases 4

Chapters 2 and 3 covered two orthogonal methodologies for verifying security protocol implementations. While both methodologies enable proving strong security properties, such as forward secrecy and injective agreement, they require both a proof of memory safety as a prerequisite. Memory safety guarantees crash freedom and absence of not only buffer overflows but also data races. Although memory safety is a highly desirable property for any codebase, completing a proof covering just memory safety is already a laborious task that requires expertise in program verification and involves annotating the codebase with specifications and ghost code, e.g., to update ghost data structures that simplify the proof or to assert auxiliary properties. The necessary specifications and ghost code often amount to a multiple of a codebase's actual code, making a proof of memory safety prohibitively expensive for large codebases.

We explore in this chapter how to scale the verification of security properties to large codebases by differentiating between the security-criticality of a codebase's parts and trading off between expressiveness and automation. More specifically, we exploit that large codebases mostly consist of application logic while parts implementing a security protocol are often small. We therefore propose to apply our expressive, refinement-based methodology from Chapter 2[1] to only these small and critical parts (the Core) implementing a security protocol and to prove their security *and* memory safety. To prove that all other parts of the codebase (the Application) do not violate the security properties, we apply fully-automatic static analyses. The static analyses achieve that by proving *I/O independence*, i.e., that the I/O operations within the Application are independent of the Core's security-relevant data (such as keys), and that the Application meets the Core's requirements.

1: It is also possible, in principle, to apply the methodology from Chapter 3, which is similarly expressive; we discuss this alternative in Sec. 4.6.

We have proved Diodon sound by first showing that we can safely allow the Application to perform I/O independent of the security protocol, and second that manual verification and static analyses soundly compose. We evaluate Diodon on two case studies: an implementation of the signed DH key exchange and a large (100k+ LoC) production Go codebase implementing a key exchange protocol for which we obtained secrecy and injective agreement guarantees by verifying a Core of about 1 % of the code with the auto-active program verifier Gobra in less than three person months. The evaluation demonstrates that our novel combination of an expressive program verifier and fully-automatic static analyses significantly reduces the proof effort and, thus, enables proving security properties for large, production codebases.

4.1 Introduction

Approaches to verifying existing security protocol implementations (cf. Sec. 1.1.3), including our two methodologies from Chapters 2 and 3, are sound *only* if they are applied to the *entire* implementation. Verifying only a subset of the codebase is unsound, and would fail to prevent, e.g., code seemingly unrelated to a security protocol accidentally logging key material [130, 131]. However, the required expertise and annotation

[130]: CVE (2024), *CVE-2024-47083*
[131]: CVE (2023), *CVE-2023-6746*

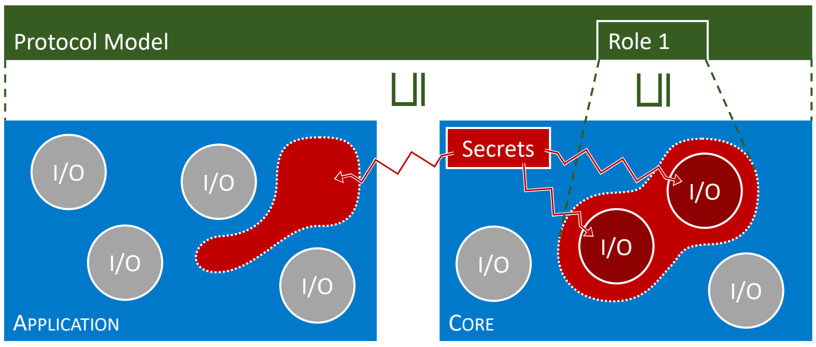

Figure 4.1: The Diodon methodology. We partition the codebase (blue) into the module implementing a protocol (Core) and the remaining codebase (Application). We prove that the Core refines (trace inclusion on the right) a particular role of the verified protocol model (green) by auto-active verification. We apply static analyses to the entire codebase to enforce that secrets (red) do not influence (red arrows) the I/O operations (gray circles) of the Application and to ensure that the Application cannot invalidate the security properties proved for the Core. Consequently, Diodon guarantees that the entire codebase refines the protocol model (trace inclusion in the middle) and, thus, enjoys all security properties proved for that model.

overhead make it infeasible to verify entire *production* codebases, which often consist of hundreds of thousands of lines of code.

In this chapter, we present Diodon^2, a proved-sound methodology that scales verification of security properties to large production codebases. Diodon works with codebases where a small, syntactically-isolated component implements a security protocol, whose security argument can be made separately from the rest of the code. Our methodology decomposes the overall codebase into this protocol implementation (the Core) and the remainder (the Application).

2: Diodon is a genus of fish known for their inflation capabilities. Erecting spines and scaling their volume by a multiple provide security, like our verification methodology.

This decomposition allows us to apply different verification techniques to the two parts. We verify the Core using our approach from Chapter 2 to show refinement w.r.t. a verified Tamarin model, which requires precise reasoning about, e.g., the payloads of I/O operations. Instead of applying the same annotation-heavy approach to the Application, we use automatic static analyses to ensure that security-relevant data of the Core (in particular, secrets such as keys) does not influence any I/O operation within the Application. If this *I/O independence* holds, the Application cannot perform any I/O operations that could interfere with the protocol and invalidate its proven security. Additionally, we use static analyses to prove that the Application satisfies the assumptions made for the proof of the Core, in particular, that the preconditions of Core functions hold when called from the Application and that the Application does not violate any invariants of Core data structures. These checks ensure that the proofs of the Core and the Application compose soundly. Consequently, the entire codebase refines the protocol model and enjoys all security properties proved for the model. Diodon significantly reduces the proof effort of verifying software that contains protocol implementations. Fig. 4.1 illustrates our methodology.

We prove I/O independence for the Application by executing an automatic taint analysis on the entire codebase to identify I/O operations that are possibly affected by secrets (also implicitly via control flow) and checking that all such operations are within the Core, which shows that the codebase's decomposition is valid and the Core is sufficiently large. It would be too restrictive to enforce that all secrets are confined within the Core. In most implementations, secrets exist outside the Core, e.g., the Application might have access to secrets either via program inputs or the Core's state (red area within the Application in Fig. 4.1). It is therefore essential to ensure (via a whole-program analysis) that the Application does not *use* these secrets to violate the security properties of the Tamarin model.

Most I/O operations within the Core correspond to a protocol step and are relevant for proving refinement w.r.t. a protocol model. In production code, however, the Core might also contain operations irrelevant to the protocol, such as logging a protocol step. To reduce the verification effort further, we also check I/O independence *within* the Core to classify each I/O operation based on whether it depends on secrets occurring in a protocol run (dark red circles in Fig. 4.1) or not (gray circles). The former need to be considered during the refinement proof, while the latter can safely be ignored. This classification simplifies the refinement proof and shortens the abstract protocol model.

We prove refinement of the Core w.r.t. a protocol model using an *auto-active* program verifier [33]. Auto-active program verifiers take as input an implementation annotated with specifications such as pre- and post-conditions and loop invariants, and attempt to verify the implementation automatically using an SMT solver.

[33]: Leino et al. (2010), *Usable Auto-Active Verification*

Auto-active verification is generally sound only if it is applied to the entire codebase because *all* callers of a function must establish its pre-condition and *all* functions must preserve data structure invariants. To ensure that our methodology is sound while avoiding this requirement for the Application, we design our methodology such that static analyses automatically discharge the proof obligations on the Application. Nevertheless, our methodology is flexible enough to permit complex interactions between the Core and Application, e.g., through concurrency and callbacks. Some assumptions remain, in particular, the absence of data races and undefined behavior; we discuss those in Sec. 4.3.4.

We prove Diodon sound, providing a blueprint for combining the distinct formalisms of auto-active verifiers and static analyses. First, we prove that a DY attacker can simulate all secret-*independent* I/O operations. Consequently, if a Tamarin model permits every secret-*dependent* I/O operation in a codebase, then this codebase refines the model. Second, we show that Diodon reasons about these secret-*dependent* I/O operations *without* verifying the entire codebase. I.e., we construct the corresponding proof for the entire codebase by starting from the proof for the Core, which we obtain from auto-active verification, and discharging the remaining proof obligations using our static analyses.

We evaluate Diodon on two Go implementations, a signed DH key exchange and a fork of the AWS Systems Manager Agent (SSM Agent) [132], a large (100k+ LoC) codebase. Part of the latter codebase implements an experimental protocol for encrypted shell sessions. We prove secrecy for and injective agreement on the session keys established by both protocols. For the SSM Agent codebase, Diodon allowed us to limit auto-active verification to only about 1% of the entire codebase, which took less than three person months. This demonstrates that Diodon enables, for the first time, the verification of strong security properties at the scale of production codebases. Our static analyses and case studies are open-source [133, 134].

[132]: Amazon Web Services, Inc. (2023), *Working with SSM Agent*

[133]: Arquint et al. (2025), *The Secrets Must Not Flow: Scaling Security Verification to Large Codebases (artifact)*
[134]: Arquint et al. (2025), *The Secrets Must Not Flow: Scaling Security Verification to Large Codebases*

Contributions. We make the following contributions:

- We present a scalable verification methodology for implementations of security protocols within large codebases, which applies to any codebase with a clear distinction between the protocol core and the rest of the code.

- We identify I/O independence, enabling concise protocol models for complex implementations.
- We show how to use static analyses to automatically discharge the Core's proof obligations, enabling Diodon to scale to large codebases.
- We prove the soundness of I/O independence w.r.t. a DY attacker, and the soundness of Diodon's combination of auto-active verification and static analyses.
- We evaluate our methodology on two case studies, an implementation of the signed DH key exchange and an AWS Systems Manager Agent fork, to demonstrate that Diodon scales to large, production codebases.

4.2 Running Example of Diodon

We demonstrate the core ideas of Diodon on a sample program in the Go programming language, which implements a simple message authentication code (MAC) protocol that sends and receives signed messages using a pre-shared key. In the remainder of this section, *we* refers to a user of Diodon. First, we manually split the codebase into Core and Application following function boundaries. We make the Core as small as possible to reduce auto-active verification efforts while making sure that the entire protocol implementation is contained therein and that we can define an invariant for the Core's API with which the Application interacts.

We model the protocol and prove security properties with the Tamarin protocol verifier [7, 8]. The goal is to prove that the entire program, i.e., the composition of the Core and Application, refines the Tamarin model and, thus, satisfies the same security properties as the protocol model. We auto-actively verify the Core using Gobra [34] and apply the automatic Argot [135] static analyses to the entire codebase.

[7]: Schmidt et al. (2012), *Automated Analysis of Diffie–Hellman Protocols and Advanced Security Properties*
[8]: Meier et al. (2013), *The TAMARIN Prover for the Symbolic Analysis of Security Protocols*
[34]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*
[135]: AWS Labs (2024), *Argot*
[25]: Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*

Core. The Core (Fig. 4.2) consists of a struct definition, two API functions, `InitChannel` and `Send`, which access this struct, and a predicate `Inv` that represents the separation logic [25] invariant used to verify the functions. Separation logic controls heap access with these permissions to reason about side effects and to prove data-race freedom, as detailed in Sec. 1.1.2. For instance, `acc(msg,1)` on line 24 passes full permission to write the contents of `msg` (if it is non-`nil`) from a caller to function `Send`, and back to the caller when the function returns. `acc(c, 1/2)` on line 33 denotes $^1/_2$ permissions, i.e., read-only access, to all fields of the struct to which `c` points (cf. fractional permissions). Separation logic *predicates* [104], like `Inv`, abstract over individual permissions to heap locations. Conceptually, we can treat a predicate instance such as `Inv(c)` as a shorthand notation for the predicate's body, which includes permissions to access the struct fields and the pre-shared key's bytes. We use `preserves` as syntactic sugar for properties that are preserved, that is, act as pre- *and* postconditions.

[104]: Parkinson et al. (2005), *Separation Logic and Abstraction*

To receive incoming packets, the Core spawns a goroutine on line 19 executing the function `continuousRecv`. We omit its implementation in the figure for space reasons. The goroutine repeatedly calls a blocking receive operation, checks the MAC's validity, and on success calls the closure that is stored in the struct field `cb` as a callback. If the callback is non-`nil`, it delivers the resulting message to the Application.

We verify the Core for any callback closure that satisfies the specification `CbSpec` (cf. line 13 & 10–11), which states that a caller must pass permission for modifying the message to the closure when invoking it and that the

```
 1  package core

 3  type Chan struct {
 4    psk []byte
 5    cb  Cb
 6  }

 8  type Cb = func(msg []byte)

10  //@ requires acc(msg, 1)
11  //@ func CbSpec(msg []byte)

13  //@ requires  cb != nil ==> cb implements CbSpec{}
14  //@ preserves psk != nil ==> acc(psk, 1)
15  //@ ensures   Inv(c)
16  func InitChannel(psk []byte, cb Cb) (c *Chan) {
17    //@ inhale AliceIOPermissions()
18    c = &Chan{append([]byte(nil), psk...), cb}
19    go continuousRecv(c)
20    return c
21  }

23  //@ preserves c != nil ==> Inv(c)
24  //@ preserves msg != nil ==> acc(msg, 1)
25  func Send(c *Chan, msg []byte) {
26    if c == nil || msg == nil { return }
27    fmt.Printf("Send %x\n", msg)
28    packet := append(msg, HMAC(msg, c.psk)...)
29    sendToNetwork(packet)
30  }

32  /*@ pred Inv(c *Chan) {
33    c != nil && acc(c, 1/2) &&
34    acc(c.psk, 1/2) && AliceIOPermissions() &&
35    (c.cb != nil ==> c.cb implements CbSpec{})
36  } @*/
```

Figure 4.2: Sample Core for a simple MAC communication. In Go, function definitions take a list of input parameters and may have a second list for outputs. We omit the `continuousRecv` goroutine's implementation that invokes the `c.cb` closure (if non-`nil`) whenever a message has been received. We simplify the representation of I/O permissions, which describe permitted protocol-relevant I/O operations, and omit proof-related statements.

closure does not have to return any permissions. On line 18, we duplicate the pre-shared key (which the Application obtains as a program input) to keep the Core's memory footprint separated from the Application. Thus, we can pass half of the permissions for accessing the struct fields to the goroutine spawned on line 19 and store the remaining permissions in the invariant `Inv`, which is then returned to the caller of `InitChannel`.

Application. The Application (Fig. 4.3) consists of a single function that creates a closure that will print any incoming message, initializes the Core with the pre-shared key `psk` and this closure, and then sends a message by invoking the `Send` function of the Core. In realistic programs, the Application might have thousands of lines of code, making auto-active verification prohibitively expensive. Diodon allows us to apply automatic static analyses instead, as detailed below.

Protocol Model. Fig. 4.4 excerpts the abstract protocol model as a multiset rewriting system in Tamarin (cf. Sec. 2.2.1) with two protocol roles, Alice and Bob, each starting off with a pre-shared key `psk`. Both roles can send and receive unboundedly many packets, each of which are the composition of a message plus the appropriate MAC. To make zero assumptions about the messages themselves, we treat them as being attacker-controlled, i.e., the sending role obtains a message from the attacker-controlled network via an `In` fact, as shown on line 3. For this protocol model, we prove that all received messages were previously sent by either Alice or Bob, unless the attacker obtains the pre-shared key, which Tamarin proves automatically.

```
1  package main

3  import . "core"

5  func main(psk []byte) {
6    cb := func(m []byte) {fmt.Printf("%x\n", m)}
7    c := InitChannel(psk, cb)
8    Send(c, []byte("hello world"))
9    fmt.Printf("Log: message sent.\n")
10   // fmt.Printf("%v\n", c)
11 }
```

Figure 4.3: Sample Application that is a client of Fig. 4.2. We omit parsing of command line arguments for presentation purposes and, thus, assume that `psk` stores the parsed pre-shared key.

```
rule Alice_Send:
    let packet = <msg, sign(msg, psk)> in
    [ Alice_1(rid, A, B, psk), In(msg) ]
  --->
    [ Alice_1(rid, A, B, psk), Out(packet) ]
rule Alice_Recv:
    let packet = <msg, sign(msg, psk)> in
    [ Alice_1(rid, A, B, psk), In(packet) ]
  --->
    [ Alice_1(rid, A, B, psk), Out(msg) ]
```

Figure 4.4: Tamarin model excerpt for the MAC protocol implemented in Fig. 4.2.

In order to prove that our program is actually a refinement of this model and, thus, inherits all proven properties, Diodon combines auto-active verification and static analyses to obtain provably-sound guarantees.

Verification. Our goal is to ensure that the composition of the Application and Core refines the abstract Tamarin model, i.e., the program's I/O behavior is contained in the model's I/O behavior. This refinement implies that any trace-based safety property proven in Tamarin also holds for the program because the program performs the same or fewer I/O operations than the protocol model. We split the refinement proof into three steps: We prove that (1) non-protocol I/O is independent of protocol secrets, (2) all remaining I/O refines a protocol role, and (3) the proof steps soundly compose.

First, we manually identify protocol-relevant calls to I/O operations within the Core. In our example, these are the `sendToNetwork` call and the corresponding network receive operation. We then perform an automatic taint analysis on the entire codebase to prove I/O independence for all other calls to I/O operations (in our example, the calls to `Printf`), i.e., we check that they do not use tainted data. Uncommenting line 10 in Fig. 4.3 would result in printing all struct fields of variable `c` including the pre-shared key `psk`, which is the only secret. I/O independence would correctly fail for this modified program, resulting in an error message indicating the flow of secret data to the print statement. In general, we treat data as a secret (i.e., tainted) if the protocol model's attacker might not know this data. Checking I/O independence ensures that we do not miss any protocol-relevant I/O operations and that the chosen Core is sufficiently large.

The Core may execute protocol-relevant operations not only by performing I/O operations, but also by communicating with the Application. For example, Alice's protocol step of taking an arbitrary message from the environment (before signing and sending it), is implemented by the Core obtaining `msg` from the Application (line 25 in Fig. 4.2). Similarly, Alice may (after receiving a packet and checking its signature) release its payload to the environment, which is implemented as passing the payload to the Application when invoking the closure `c.cb` (not shown in

Fig. 4.2). To handle such protocol-relevant operations uniformly, we treat them as *virtual* protocol-relevant I/O operations. This allows us to enforce or assume constraints on the arguments' taint status while creating the necessary proof obligations in the next step of the refinement proof. Here, the fact that releasing the payload is permitted by the protocol model (line 10 in Fig. 4.4) informs the taint analysis that the callback's argument may be considered as untainted, which allows printing it on line 6 in Fig. 4.3.

Second, we prove the Core using an auto-active program verifier, i.e., a program verifier that uses manual annotations and proof automation to prove properties about programs. While various other separation logic-based, auto-active program verifiers exist, including VeriFast [26] for C, Nagini [30] for Python, and Prusti [32] for Rust, we use the Go verifier Gobra [34] to prove the Core. This proof includes showing that the protocol model permits every protocol-relevant I/O operation, including virtual I/O. Note that step (1) ensures that these operations must all be in the Core. We use an I/O specification for each protocol role describing the permitted protocol-relevant I/O operations (cf. Sec. 2.2.2). In our example, Alice obtains the permissions to perform these operations during the initialization of the Core (line 17) and maintains them as part of the invariant (line 34), where `inhale` adds the specified permissions to the current program state by assumption. When performing a protocol-relevant I/O operation, like `sendToNetwork`, Gobra proves that the I/O specification permits this operation with the specific arguments. Otherwise, Gobra reports a verification failure.

[26]: Jacobs et al. (2011), *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*

[30]: Eilers et al. (2018), *Nagini: A Static Verifier for Python*

[32]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*

[34]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*

Third, since the Gobra proof for the Core assumes that callers respect the functions' preconditions, Diodon restricts the class of supported preconditions such that static analyses are able to prove that the Application satisfies them. For example, the precondition of `Send` requires exclusive access for the argument `msg`; Diodon enforces this condition using a combination of static pointer, escape, and pass-through analyses to ensure that no other goroutine accesses the memory pointed to by `msg`. `Send`'s other precondition requires the Core's invariant to hold, which is established by `InitChannel`. The Application could in principle violate this precondition, for example, by creating a `Chan` instance without calling `InitChannel`, or by invalidating the invariant of a `Chan` instance through field updates or concurrency. We apply this combination of static analyses to prevent all such cases (cf. Sec. 4.3.3).

Together, these three proof steps ensure that the program refines the abstract Tamarin model and inherits the security properties proved for the model.

4.3 Diodon

Our methodology, Diodon, proves security properties for implementations by refinement and scales to large codebases by significantly reducing verification effort. Diodon enables more concise protocol models than previous approaches and leverages fully automatic analyses on most of the implementation to discharge proof obligations.

We manually decompose a codebase into two syntactically isolated components, the Core implementing a security protocol, and the Application consisting of the remaining code. Typically, this decomposition is natural

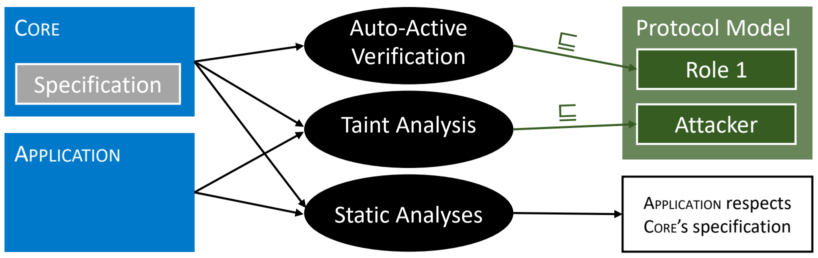

Figure 4.5: Diodon proves that the entire codebase (blue) refines a protocol model (green) by soundly composing auto-active verification with automatic static analyses. We auto-actively verify the Core based on its specification to show that the protocol-relevant I/O operations refine a protocol role (upper trace inclusion). This specification is partially generated from the protocol model, which is omitted. The static taint analysis proves that all other I/O operations within the entire codebase refine our attacker model (lower trace inclusion). Lastly, we discharge the Core's assumptions by applying automatic static analyses, proving that the Application satisfies the calling restrictions expressed in the Core's specification.

and follows module boundaries as a protocol's implementation is localized. As illustrated in Fig. 4.5, this decomposition allows us to split the proof that the entire codebase refines a protocol model into three steps and uses the best-suited tool for each step. We explain in Sec. 4.3.1 how Diodon identifies which I/O operations are protocol-relevant by performing a static taint analysis on the entire codebase. Sec. 4.3.2 covers the Core's auto-active verification using Gobra proving that protocol-relevant I/O operations refine a particular protocol role. Finally, Sec. 4.3.3 explains how we discharge the assumptions made when auto-actively verifying the Core by performing static analyses on the Application.

4.3.1 I/O Independence

One of our key insights is to distinguish between I/O operations that are relevant for a security protocol from those that are not (e.g., sending log messages to a remote server). This distinction has two main benefits. First, protocol-irrelevant operations do not have to be reflected in the abstract protocol model, which makes the model concise, more general, and easier to maintain, review, and prove secure. Second, by ensuring that protocol-irrelevant I/O operations cannot possibly invalidate the security properties proven for the protocol model, we do not have to consider them during the laborious auto-active refinement proof and instead can check simpler properties using automatic static analyses. We classify all calls to I/O operations as either protocol-relevant or protocol-irrelevant. In the Core, an I/O operation is protocol-irrelevant if and only if its specification requires no I/O permissions. In contrast, all I/O operations in the Application are implicitly considered protocol-irrelevant.

To ensure that I/O operations classified as protocol-irrelevant indeed do not interfere with the protocol or invalidate proven security properties of the protocol, we check that they do not use any secret data (such as key material); more precisely, we check *non-interference* between protocol secrets and the parameters of these I/O operations. We call this important property of an I/O operation *I/O independence*. It guarantees that an I/O operation cannot possibly invalidate the protocol's proven security properties: any I/O operation that uses only non-secret data could also have been performed by the DY attacker and, thus, was already considered by Tamarin during the protocol verification. In other words, proving that all protocol-irrelevant I/O operations satisfy I/O independence guarantees that they refine our DY attacker (cf. Sec. 4.4.1).

From a cryptographic perspective, I/O independence allows us to reduce the security of an entire codebase to the security of its Core. This reduction is valid because most of the Application can be treated as part of the attacker, while the parts of the Application that manipulate secrets (e.g., code that loads long-term keys from disk) are shown not to perform I/O, and thus can conceptually be considered part of the Core, without introducing violations of the I/O specification of its protocol role.

We prove I/O independence by performing an automatic static taint analysis on the entire codebase that includes implicit information flows from control flow. A taint analysis checks for a set of sources and sinks whether there are any flows of information from a source to a sink. The analysis starts at each source, i.e., a function which produces secret data, and explores how secret information propagates through the program by keeping track of program locations storing a *tainted* value, i.e., a value that is influenced by a source. We disallow branching on tainted data to avoid information flows via control flow.

We configure the taint analysis to consider all calls to key-generation functions within the Core and long-term secrets that are passed as program inputs, like the pre-shared key in our running example, as sources because the DY attacker does not have access to them. This set of initial sources taints all protocol secrets including session keys. E.g., if the Core implements a DH key exchange, the analysis correctly considers the generated secret key and the resulting shared key tainted because the shared key is computed from the secret key and the other party's public key. We then configure the taint analysis to treat all I/O operations in the Application as well as all protocol-irrelevant I/O operations in the Core as sinks. We use Capslock [136] to identify such I/O performing functions in the Go standard library. We consider all functions with at least one of the following capabilities as a sink: write to the file system or network, modify the state of the operating system (e.g., `os.Setenv`), perform a system call, and execute external programs (e.g., `(*os/exec.Cmd).Run`).

[136]: Google (2024), *Capslock*

We run the taint analysis on the entire codebase. If taint reaches a sink, verification fails because a secret reached a supposedly protocol-irrelevant I/O operation. Otherwise, we have correctly identified the protocol-relevant I/O operations (and thereby confirmed that we have correctly delimited the Core); it remains to reason about those I/O operations, as we discuss next.

4.3.2 Core Refinement

We auto-actively verify the entire Core, which allows us to state and prove (besides safety and functional correctness) precise constraints about protocol-relevant calls to I/O functions and their arguments. We prove that the implementation uses the payload for each I/O operation specified in the protocol model. The corresponding verification effort is feasible since, in industrial codebases like our main case study, the Core comprises only a small fraction of the codebase.

We prove that the Core refines a protocol role by building on the approach explained in Chapter 2. In particular, we equip each protocol-relevant I/O operation with a specification that requires an I/O permission for executing this operation with the provided arguments. Since we provide exactly the I/O permissions justified by the protocol role's model to the Core during its initialization, successful verification with Gobra implies that the Core executes at most the protocol-relevant I/O

```
//@ preserves c ≠ nil ⟹ inv(c)
//@ preserves a0 ≠ nil ⟹ acc(a0)
//@ preserves a1 ≠ nil ⟹ acc(a1)
//@ ensures   r ≠ nil ⟹ acc(r)
func (c *Core) ApiFn(a0, a1 *int) (r *int)
```

Figure 4.6: Example of a signature and specification of a Core API function.

operations permitted by the model and, thus, refines this protocol role. This approach differs in three significant ways from Chapter 2.

First, we do not auto-actively verify the entire codebase and, instead, verify only the Core. As we will discuss in Sec. 4.3.3, we syntactically restrict the preconditions of the Core functions so that we can apply automatic static analyses to check that each call from the Application satisfies them, which is necessary for soundness.

Second, our approach supports codebases that use multiple instances of the Core, e.g., to run multiple roles of the protocol or to run the protocol multiple times. Since Tamarin considers unboundedly many role instantiations, we can soundly create the required I/O permissions for executing a role instance whenever we create a new Core instance. These I/O permissions are bound to an instance's unique identifier such that each Core instance has its own set of I/O permissions for executing the security protocol once.

Third, to reflect that interactions in the model between the protocol and the environment may manifest as interactions between the Core and the Application in the implementation, we treat the boundary between them as a virtual network interface and enforce I/O permissions for the corresponding virtual I/O operations, as we illustrated in Sec. 4.2.

4.3.3 Analyzing the Application

We now show how to scale auto-active verification to the entire codebase. Applying auto-active verification to an entire codebase is typically not feasible within the resource constraints of industrial projects. A key insight of Diodon is that this is not necessary: we can use static analyses to automatically discharge separation logic proof obligations arising in the Application to obtain, together with the verified Core, a proof in separation logic for the *entire* codebase.

The refinement proof for the Core is valid in the context of the entire application if (1) each call to a Core function from the Application satisfies the function precondition, and (2) the Application respects permissions on memory accesses. Our soundness proof for Diodon (cf. Sec. 4.4.2) ensures that these proof obligations are sufficient and that our novel combination of static analyses can soundly discharge them. We illustrate these proof obligations and how we discharge them by considering the exemplary Core function in Fig. 4.6, taking two integer pointers as input and returning an integer pointer. This function maintains the Core invariant (if `c` is non-`nil`), needs full permissions for both inputs, and returns full permissions for the input and output parameters (if they are non-`nil`). Thus, we cannot allow, e.g., the Application to pass two aliased arguments to this function or to concurrently access heap locations pointed to by these arguments as this would violate the precondition, i.e., the permissions specified therein.

Implicit Annotations. To construct a proof for the entire codebase, we enrich the Application with a *hypothetical program instrumentation* that connects the APPLICATION to the necessary proof obligations imposed by the proof of the CORE. These *implicit annotations* track the permissions that the APPLICATION owns by using sets of heap locations and a *program invariant* specifying permissions for the heap locations in these sets. More precisely, each thread has a set lhs (short for "local heap set") for thread-local objects such as buffers, and a set ihs (short for "invariant heap set") for CORE instances. Similarly, a global set ghs ("global heap set") keeps track of objects that might be shared between threads, which becomes relevant later. These sets are mutable and, thus, their content depends on a particular program point. The sets lhs and ihs allow us to state the following local program invariant that must hold at every program point in the APPLICATION.

$$\Pi_l \triangleq \left(\bigstar_{l \in \mathsf{lhs}} \mathtt{acc}(l)\right) \star \left(\bigstar_{i \in \mathsf{ihs}} \mathtt{inv}(i)\right)$$

Here, the iterated separating conjunction $\bigstar_{e \in s} a(e)$ conjoins the assertions $a(e)$ using separating conjunction for all elements e in set s. Π_l states that a thread holds full permissions for all objects in lhs and the CORE invariant for all instances in ihs. In addition, these permissions are disjoint allowing the APPLICATION to write to heap locations in lhs without breaking the CORE invariant. When a thread obtains or gives up permissions, our implicit annotations adjust lhs and ihs to maintain the program invariant.

Fig. 4.7 shows these *implicit annotations* for calls to `ApiFn`. To highlight that each statement in the APPLICATION maintains the program invariant, we assert Π_l on lines 1 and 22. For each permission required by the callee's precondition, we remove the corresponding heap location from one of the sets to reflect that ownership is being passed to the callee. Assuming (for now) that the location was originally in the set, this removal extracts the corresponding permission from Π_l, as illustrated by the intermediate assert statements starting on lines 3, 6, and 10 for the three arguments of the call. After the call, we conversely add those heap locations to the sets for which the callee's postcondition provides permissions.

For each permission in the precondition, if the corresponding heap location was contained in one of the sets before the removal operation, then we have effectively proved that the precondition holds (syntactic restrictions ensure that the preconditions cannot contain constraints other than permission requirements, see below). In the rest of this subsection, we explain how we use static analyses to check this set containment. Then, we explain the proof obligations for memory accesses within the APPLICATION.

Guaranteeing Permissions for Parameters. For the arguments `a0` and `a1` (we will discuss the core instance `c` below), we need to prove that (1) $\{\mathtt{a0}, \mathtt{a1}\} \subseteq \mathsf{lhs}$ holds before the call to `ApiFn` and (2) `a0` and `a1` do not alias. If (2) was violated, `a1` would no longer be in lhs after removing `a0` on line 5 in Fig. 4.7, i.e., we would obtain only $\mathtt{acc}(\mathtt{a0})$ instead of $\mathtt{acc}(\mathtt{a0}) \star \mathtt{acc}(\mathtt{a1})$.

We discharge these two proof obligations by checking the conditions (C6) and (C7) in Tab. 4.1, resp., using static analyses. We check (C6) by using a thread escape analysis, which delivers judgments local(x) for a particular program point expressing that $*x$ is definitely not accessible by any other thread. We show in Sec. 4.4.2 that (C6) suffices to discharge $\{\mathtt{a0}, \mathtt{a1}\} \subseteq \mathsf{lhs}$ (if the arguments are non-`nil`) by proving a lemma that relates local(x) for a program point p with $x \in \mathsf{lhs}$. We obtain (C7) by applying a pointer analysis, which computes may-alias information, i.e., pts(x) for

```
 1 //@ assert (★_{l∈lhs} acc(l)) ⋆ (★_{i∈ihs} inv(i))
 2 //@ ihs := ihs \ {c}
 3 //@ assert (★_{l∈lhs} acc(l)) ⋆ (★_{i∈ihs} inv(i)) ⋆
 4 //@        (c ≠ nil ⟹ inv(c))
 5 //@ lhs := lhs \ {a0}
 6 //@ assert (★_{l∈lhs} acc(l)) ⋆ (★_{i∈ihs} inv(i)) ⋆
 7 //@        (c ≠ nil ⟹ inv(c)) ⋆
 8 //@        (a0 ≠ nil ⟹ acc(a0))
 9 //@ lhs := lhs \ {a1}
10 //@ assert (★_{l∈lhs} acc(l)) ⋆ (★_{i∈ihs} inv(i)) ⋆
11 //@        (c ≠ nil ⟹ inv(c)) ⋆
12 //@        (a0 ≠ nil ⟹ acc(a0)) ⋆
13 //@        (a1 ≠ nil ⟹ acc(a1))
14 r := c.ApiFn(a0, a1)
15 //@ assert (★_{l∈lhs} acc(l)) ⋆ (★_{i∈ihs} inv(i)) ⋆
16 //@        (c ≠ nil ⟹ inv(c)) ⋆
17 //@        (a0 ≠ nil ⟹ acc(a0)) ⋆
18 //@        (a1 ≠ nil ⟹ acc(a1)) ⋆
19 //@        (r ≠ nil ⟹ acc(r))
20 //@ lhs := lhs ∪ ({a0, a1, r} \ nil)
21 //@ ihs := ihs ∪ ({c} \ nil)
22 //@ assert (★_{l∈lhs} acc(l)) ⋆ (★_{i∈ihs} inv(i))
```

Figure 4.7: *Conceptually* inserted implicit annotations for a Core API call `r := c.ApiFn(a0, a1)` in the Application. The assert statements solely illustrate our deductions and, thus, can be omitted.

a pointer x, where $a \in \text{pts}(x)$ denotes that $*x$ may-alias any location allocated at site a. More precisely, we check for each pair of arguments that the sets of locations they may-alias are disjoint, which is sufficient as we restrict parameters to be shallow.

Guaranteeing the Core Invariant. Similarly to parameters, we have to prove that c ∈ ihs holds such that removing c from ihs on line 2 grants us the Core invariant `inv(c)`, if c is non-`nil`. In Sec. 4.4.2, we prove that c ∈ ihs if the following premises hold. (1) The Core instance c must have been returned as a result from a Core API function initially establishing the Core invariant, e.g., `InitChannel` in our running example. (2) All heap modifications in the Application must not modify the internal state of the Core instance, even through an alias, since this could invalidate the Core invariant.

In a single-threaded program without callbacks from the Core to the Application, the above premises are sufficient. However, in the presence of these two features, we need to ensure that the Application does not call two Core functions on the same Core instance simultaneously, which would effectively duplicate permissions and, thus, make reasoning unsound: (3) The Application must not pass the same Core reference to more than one thread, and (4) the Application must not call a Core function in a callback on the same instance that is invoking the callback.

We establish the four premises by checking the conditions (C1) to (C5) in Tab. 4.1. Conditions (C1) and (C3) can be enforced by checking that the Application calls only Core functions that establish or preserve the invariant. While the postconditions provide this information for Core instances that are passed as arguments or results, our analyses need to prevent a subtle loophole: We need to prevent Core functions from allocating a Core instance *without* establishing its invariant and letting the Application access it via global variables or shared memory. We implemented a pass-through analysis computing $\text{pass}_f(x, r)$ for a function f stating that outside of calls to f, $*x$ definitely passed through return parameter r. We use this pass-through analysis to ensure that all references to Core instances in the Application are obtained exclusively through the return parameter, such that the postcondition establishes the invariant.

	Condition	Details
C1	Core init	Core instances are created in a function ensuring the invariant in its postcondition
C2	No modification	Application does *not* write to Core instances' internal state, even through an alias
C3	Core preservation	Core instances are passed only to Core functions that preserve the invariant
C4	Core locality	Core instances are used only in the thread they are created in
C5	Core callback	Core APIs are not invoked in Application callbacks
C6	Parameter locality	Parameters to Core APIs are local
C7	Disjoint parameters	Parameters to the same Core API call do not alias one another
C8	Application access	Reads and writes in the Application occur to memory allocated in the Application or transferred from the Core

Table 4.1: Sufficient conditions checked by our static analyses, grouped into those involving Core instances, other parameters to Core functions, and memory accesses in the Application.

To establish (C2), we use a pointer analysis to ensure that all reads and writes in the Application never access a Core instance's internal state. In particular, we ensure that the Application accesses *only* heap locations that must-not-alias locations reachable from `ihs`, i.e., internal state of Core instances. Since we use a sound pointer analysis, this check conservatively over-approximates the heap locations about which the Core invariant states properties. While it is possible to access Core memory without breaking the invariant, we could not treat the Core invariant as an opaque separation logic resource when analyzing the Application, which would require a static analysis capable of reasoning about fractional permissions and arbitrary functional properties.

For (C4), we use the thread escape analysis to ensure that each Core instance does not escape its thread (we show local(`c`) for each call to Core instance `c`), guaranteeing that each thread operates on a disjoint set of Core instances `ihs`. While it is possible to safely pass Core instances between threads, this would require a significantly more sophisticated static analysis that can reason about the ordering of concurrent executions. Condition (C5) is enforced by checking that the call graph does not contain Core functions invoked transitively from Application callbacks. Allowing such calls would require proving that the same instance is not used in the inner call, which requires a more precise pointer analysis.

Our explanations generalize from the exemplary Core function in Fig. 4.6 to arbitrary Core API functions as long as they satisfy the following restrictions on pre- and postconditions. We support an arbitrary number of input and output parameters with arbitrary value and pointer types. Our restrictions mandate that Core API functions preserve the Core invariant and full permissions for each parameter of pointer type, both only under the condition that the receiver and parameters are non-`nil`. Additionally, the postcondition specifies full permissions for each return parameter if it is non-`nil` and of pointer type. These restrictions ensure that preconditions do not specify functional properties, such as require an input array to have a certain length, which we cannot check using our static analyses. As seen with our example in Fig. 4.6, we cannot rule out that the Application passes `nil` as an argument because there is no sound nilness analysis for Go to the best of our knowledge and, thus, we account for this possibility in our restrictions.

Application Memory Access. Finally, we need to ensure that the Application accesses only memory to which it has permissions. While we have already established that the Application does not write to internal state of Core instances (C2), we need to particularly consider the case where memory is transferred after its allocation between the Core and the Application. The other case, namely the Core or Application allocating memory without transferring it, is straightforward. I.e., if Core-allocated memory is never transferred to the Application then the Application cannot access it. Similarly, if Application-allocated memory is not transferred to the Core then the Application retains the corresponding permission.

Checking condition (C8) is sufficient. If Application-allocated memory is transferred to the Core, our syntactic restrictions guarantee that the Core only temporarily borrows the corresponding permissions until the Core API call returns. If the Core allocates memory and transfers it to the Application, the Core must also transfer the corresponding permissions, which we enforce via our pass-through analysis checking that this transfer happens via a return parameter as our syntactic restrictions guarantee that the postcondition specifies permissions for this return parameter. Using (C8), we prove that each memory access in the Application is to a location in either lhs or ghs (cf. Sec. 4.4.2). In the latter case, we need to reason about concurrent access. We assume that the Application is free from data races: if two accesses race, then the program is invalid according to the Go specification. If there are no races, then there is some total order in which the threads can atomically pull permission from ghs, perform the access, and then return permissions to ghs before the next thread needs to access the same location.

4.3.4 Threat Model, Assumptions, and Limitations

The Diodon methodology provides strong guarantees for large codebases, namely that a codebase satisfies the same trace-based safety properties as the abstract protocol model. Like other verification techniques, Diodon relies on assumptions about the codebase, execution environment, and the employed tools.

Diodon considers an arbitrary number of potentially concurrent protocol sessions, allowing the DY attacker to, e.g., replay messages across sessions or apply cryptographic operations thereto to construct messages of unbounded size. As is standard for symbolic cryptography, we assume cryptographic operations such as signing are perfectly secure, e.g., the attacker can create valid signatures only if it possesses the correct signing key. The attacker can obtain such keys only by observing or constructing them, never by guessing.

Our methodology allows us to prove that each implementation individually refines a particular role of an abstract protocol model. Since the security properties we prove about an abstract model are typically global, they hold only if each involved implementation refines one of the protocol roles. Next, we discuss limitations of this refinement proof, grouped by limitations of the methodology itself and limitations of our instantiation in Go.

The Diodon methodology requires a partitioning of a codebase into Core and Application, while satisfying the syntactic restrictions for the Core API specifications. This partitioning limits applicability, not soundness as the taint analysis checking I/O independence guides correct partitioning and fails otherwise. Additionally, Diodon requires the absence of undefined behavior in the codebase, which we prove for the Core. However, this remains an assumption for the Application, which could be mitigated by performing an additional static analysis establishing this property. E.g., we could use Astrée [137] for a subset of C and C++. Finally, we inherit the *pattern requirement* from Arquint *et al.* [73], which allows multiple terms to have the same byte-level representation in general, but requires a unique representation for terms corresponding to protocol messages.

[137]: AbsInt Angewandte Informatik GmbH (2025), *Astrée Static Analyzer for C and C++*

[73]: Arquint et al. (2023), *Sound Verification of Security Protocols: From Design to Interoperable Implementations*

Our instantiation of Diodon in Go uses several tools to discharge proof obligations, and we rely on the soundness of each tool: the abstract protocol model verifier, the auto-active program verifier, and the static analyses. The risk that any of these tools is unsound can be mitigated by choosing mature tools such as Tamarin and Gobra.

More specifically, the Core's auto-active verification relies on trusted specifications for libraries, such as the I/O or cryptographic libraries that, e.g., consume I/O permissions or specify the cryptographic relations between input and output parameters. Diodon could be combined with verified libraries like EverCrypt [109] to reduce this trust assumption.

[109]: Protzenko et al. (2020), *EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider*

Furthermore, our taint analysis relies on the correct specification of secrets and I/O operations (we use an existing tool [136] to identify I/O operations). E.g., not treating the pre-shared key in the running example as a secret would allow us to perform I/O operations in the Application that depend on this key.

[136]: Google (2024), *Capslock*

The employed static analyses assume that the entire codebase is free of data races and, thus, exhibits defined behavior only [138]. While we auto-actively prove race freedom for the Core, this remains an assumption for the Application. Our implicit annotations clearly indicate where in the Application we rely on this assumption. Additionally, the static analyses do not soundly handle certain hard-to-analyze features such as the `unsafe` package (e.g., allowing arbitrary pointer arithmetic), `cgo` (i.e., the ability to invoke C functions), or reflection. We rely on the codebase not using them in a way that would invalidate the analysis results. Diodon could be extended by additional static analyses to reduce these assumptions, e.g., by performing a data race analysis and checking for uses of the `unsafe` and `cgo` packages and reflection. As such, these assumptions are not an inherent limitation of Diodon itself. We report case-studies-related limitations of the static analyses in Sec. 4.5.

[138]: Go developers (2022), *The Go Memory Model*

4.4 Soundness

To prove Diodon sound, we reason separately about allowing protocol-independent I/O operations in a codebase and combining auto-active verification with static analyses.

In Sec. 4.4.1, we prove that a codebase c containing protocol-dependent *and* protocol-independent I/O operations refines a given Tamarin model if the I/O specification ϕ, corresponding to a protocol role in this Tamarin model, permits all protocol-dependent I/O operations in the codebase. For this part of the soundness proof, we assume that the *entire* codebase c satisfies the Hoare triple $\left[\phi\right]\ c\ [\mathsf{true}]$, where protocol-independent I/O operations do not consume an I/O permission and, thus, can be performed at arbitrary points within c and independently of the I/O specification ϕ. Such a Hoare triple could be obtained by auto-actively verifying the *entire* codebase c, which Diodon does not require.

In Sec. 4.4.2, we show that we constructively obtain the Hoare triple $\left[\phi\right]\ c\ [\mathsf{true}]$ for an entire codebase c using Diodon by auto-actively verifying only parts thereof, namely the Core, and executing our static analyses on c, if we assume crash freedom and absence of undefined behavior for the parts of c that are not auto-actively verified. Note however that we need crash freedom only for satisfying the definition of Hoare triples, which include crash freedom. Thus, crash freedom is not an inherent requirement for Diodon as crashes do not invalidate our security guarantees.

By combining both results, we obtain that applying Diodon to a codebase c proves that c refines a Tamarin model, despite the presence of protocol-independent I/O operations, and auto-actively verifying the Core only, as long as we discharge the side conditions using our static analyses.

4.4.1 I/O Independence

We show that we can soundly allow protocol-independent I/O operations in a codebase by treating these I/O operations as a refinement of our attacker model. For this purpose, we extend Arquint *et al.*'s soundness proof [81, App. E] to accommodate such I/O operations, and we adopt their notation for legibility. More specifically, we add these I/O operations to the concrete system and show that this concrete system refines an abstract system that is composed of only protocol roles and our attacker, which corresponds to a protocol's Tamarin model.

[81]: Arquint et al. (2022), *Sound Verification of Security Protocols: From Design to Interoperable Implementations (extended version)*

Since we permit a codebase c to perform independent I/O operations in addition to I/O operations permitted by an I/O specification ϕ, we adapt the verifier assumption [81, Asm. 1] to account for both types of I/O operations.

Assumption 4.4.1 (Verifier Assumption)

$$\vdash_\alpha \left[\phi\right]\ c\ [\mathsf{true}] \land \mathbb{T}(c, s) = \mathsf{true} \implies \alpha(\mathscr{C}) \preccurlyeq \phi \mid\mid\mid \delta.$$

I.e., we assume that successfully verifying a program c against an I/O specification ϕ and successfully executing the taint analysis $\mathbb{T}$ with some configuration s specifying sources and sinks of tainted data implies that the program's traces abstracted under a relabeling function α are included in the parallel composition of the I/O specification's traces ϕ and the traces of performing independent I/O operations δ.

We assume that the program's traces are described by the labeled transition system (LTS) semantics $\mathscr{C}$, which is provided by the operational semantics of the programming language in which c is implemented[3]. α abstracts the program's traces, e.g., referring to specific function names, to traces of operations that match the naming as used in ϕ and δ. δ represents the set of traces resulting from generating fresh nonces and using received payloads as well as public constants to construct and send payloads, as will be made explicit in Def. 4.4.2.

3: We leave the programming language intentionally unspecified to keep our soundness result general.

Note that Asm. 4.4.1 expresses besides the trace inclusion itself that the states of ϕ and δ are independent such that their parallel composition is possible. We obtain this independence by successfully executing our taint analysis. In particular, our taint analysis establishes that protocol-independent I/O operations do not operate on tainted data. We configure the taint analysis such that long-term and short-term secrets known by a protocol role but not the attacker are a source of taint. Hence, these

I/O operations and all steps necessary to compute their data are either already part of δ or the necessary computation steps can be replicated and added thereto, such that δ is independent of ϕ.

The other direction, namely that the I/O specification ϕ is independent of from δ, holds by construction of ϕ. Since we generate ϕ by a series of transformations from a protocol role's abstract model and use syntactically distinct elements to represent this protocol role's state and express the transitions that form δ, as we shall see next, δ cannot influence ϕ.

For a set of function symbols Σ operating over terms, *MD* denotes the set of transition rules that the attacker can apply. $\mathsf{K}(x)$ represents the fact that the attacker knows the term x and the fact symbols out and in represent that a protocol participant sent and might receive a particular term, respectively. Therefore, *MD* captures all operations available to the attacker, namely receiving a sent term, sending a term, adding a public constant to its knowledge, generating a fresh nonce, and applying a function $f \in \Sigma$.

Definition 4.4.1 (Attacker Message Deduction Rules) *As defined in [81, Def. 9], MD_Σ denotes the set of message deduction rules representing our* DY *attacker for Σ:*

$$\begin{aligned}
[\mathsf{out}(x)] &\xrightarrow{[]} [\mathsf{K}(x)] \\
[\mathsf{K}(x)] &\xrightarrow{[\mathsf{K}(x)]} [\mathsf{in}(x)] \\
[] &\xrightarrow{[]} [\mathsf{K}(x \in pub)] \\
[\mathsf{Fr}(x \in fresh)] &\xrightarrow{[]} [\mathsf{K}(x)] \\
[\mathsf{K}(x_1), \ldots, \mathsf{K}(x_k)] &\xrightarrow{[]} [\mathsf{K}(f(x_1, \ldots, x_k))] \quad \textit{for } f \in \Sigma \textit{ with arity } k
\end{aligned}$$

[81]: Arquint et al. (2022), *Sound Verification of Security Protocols: From Design to Interoperable Implementations (extended version)*

Similarly, we define MD_Σ^{ind} in Def. 4.4.2, which consists of the transition rules a protocol-independent component can execute. These transition rules represent sending known terms to the network and receiving terms from the network, using public constants, generating nonces, and applying functions to learn new terms. We assume that these transition rules cover all operations that a protocol-independent component might perform. In contrast to *MD*, MD_Σ^{ind} operates on syntactically different, reserved fact symbols. Avoiding these name clashes simplifies defining a simulation relation for proving Lemma 4.4.1.

While ind represents knowledge of a particular term, $\mathsf{out}_{\mathsf{ind}}$ and $\mathsf{in}_{\mathsf{ind}}$ represent a term sent to and received from the network, respectively. ind, $\mathsf{out}_{\mathsf{ind}}$, and $\mathsf{in}_{\mathsf{ind}}$ are in the same class of fact symbols as their analogous counterparts K, out, and in, respectively. I.e., K and ind are persistent fact symbols Σ_{per} capturing the property that knowledge is monotonically increasing. This means that applying a transition rule does not consume such facts and, thus, their multiplicity in the multiset comprising the state is irrelevant. In contrast, out, in, $\mathsf{out}_{\mathsf{ind}}$, and $\mathsf{in}_{\mathsf{ind}}$ are in the class of linear fact symbols Σ_{lin}, meaning that applying a transition rule that states such a fact in its premise will remove this fact from the state while such a fact occurring in the rule's conclusion adds it to the state. Additionally, $\mathsf{out}_{\mathsf{ind}}$ and $\mathsf{in}_{\mathsf{ind}}$ are in the class of output and input fact symbols Σ_{out} and Σ_{in}, respectively, as suggested by their intuitive semantics.

Definition 4.4.2 (Protocol-Independent Message Deduction Rules)

$$\begin{aligned}
[\mathsf{ind}(x)] &\xrightarrow{[]} [\mathsf{out}_{\mathsf{ind}}(x)]\\
[\mathsf{in}_{\mathsf{ind}}(x)] &\xrightarrow{[]} [\mathsf{ind}(x)]\\
[] &\xrightarrow{[]} [\mathsf{ind}(x \in pub)]\\
[\mathsf{Fr}(x \in fresh)] &\xrightarrow{[]} [\mathsf{ind}(x)]\\
[\mathsf{ind}(x_1), \ldots, \mathsf{ind}(x_k)] &\xrightarrow{[]} [\mathsf{ind}(f(x_1, \ldots, x_k))] \quad \textit{for } f \in \Sigma \textit{ with arity } k
\end{aligned}$$

where ind, $\mathsf{out}_{\mathsf{ind}}$, *and* $\mathsf{in}_{\mathsf{ind}}$ *are reserved fact symbols and* $\mathsf{ind} \in \Sigma_{\mathsf{per}}$, $\mathsf{out}_{\mathsf{ind}} \in \Sigma_{\mathsf{out}} \cap \Sigma_{\mathsf{lin}}$, *and* $\mathsf{in}_{\mathsf{ind}} \in \Sigma_{\mathsf{in}} \cap \Sigma_{\mathsf{lin}}$.

Although we present $MD_{\Sigma}^{\mathsf{ind}}$ on the same abstraction level as the attacker deduction rules MD_{Σ} to make them more legible, these deduction rules are *not* part of the MSR system $\mathcal{R}$. Instead, we transform these rules according to Sec. 2.3 and make them part of the component system as described next.

We introduce buffered versions for the $\mathsf{out}_{\mathsf{ind}}$ and $\mathsf{in}_{\mathsf{ind}}$ facts and split the rules in $MD_{\Sigma}^{\mathsf{ind}}$involving I/O into two separate rules each, which we synchronize using transition labels. This split allows us to assign half of the rules to the component executing protocol-independent operations $\mathcal{R}_{\mathsf{ind}}(rid)$ and assign the remaining rules $\mathcal{R}_{\mathsf{io}}^{+}$ to the environment forming $\mathcal{R}_{\mathsf{env}}^{e+}$. We use χ^{+} to synchronize the execution of these now separated rules.

Definition 4.4.3 ($\mathcal{R}_{\mathsf{ind}}(rid)$) *$\mathcal{R}_{\mathsf{ind}}(rid)$ consists of the following multiset transition rules.*

$$\begin{aligned}
[\mathsf{ind}(rid, x)] &\xrightarrow{[\lambda^{s}_{\mathsf{out}_{\mathsf{ind}}}(rid, x)]} []\\
[] &\xrightarrow{[\lambda^{s}_{\mathsf{in}_{\mathsf{ind}}}(rid, x)]} [\mathsf{ind}(rid, x)]\\
[] &\xrightarrow{[]} [\mathsf{ind}(rid, x \in pub)]\\
[] &\xrightarrow{[\lambda^{s}_{\mathsf{Fr}_{\mathsf{ind}}}(rid, x)]} [\mathsf{ind}(rid, x)]\\
\begin{bmatrix}\mathsf{ind}(rid, x_1),\\ \ldots,\\ \mathsf{ind}(rid, x_k)\end{bmatrix} &\xrightarrow{[]} [\mathsf{ind}(rid, f(x_1, \ldots, x_k))]\\
& \qquad \textit{for } f \in \Sigma \textit{ with arity } k
\end{aligned}$$

Definition 4.4.4 ($\mathcal{R}_{\mathsf{env}}^{e+}$) *$\mathcal{R}_{\mathsf{env}}^{e+} = \mathcal{R}_{\mathsf{env}}^{e} \uplus \mathcal{R}_{\mathsf{io}}^{+}$ where $\mathcal{R}_{\mathsf{env}}^{e}$ is defined as in Sec. 2.3.2 and $\mathcal{R}_{\mathsf{io}}^{+}$ consists of the following multiset transition rules.*

$$\begin{aligned}
[] &\xrightarrow{[\lambda^{e}_{\mathsf{out}_{\mathsf{ind}}}(rid, x)]} [\mathsf{out}_{\mathsf{ind}}(x)]\\
[\mathsf{in}_{\mathsf{ind}}(x)] &\xrightarrow{[\lambda^{e}_{\mathsf{in}_{\mathsf{ind}}}(rid, x)]} []\\
[\mathsf{Fr}(x \in fresh)] &\xrightarrow{[\lambda^{e}_{\mathsf{Fr}_{\mathsf{ind}}}(rid, x)]} []
\end{aligned}$$

Definition 4.4.5 (χ^{+}) *We define the partial synchronization function χ^{+} : $(\bigcup_{i,rid}(\mathcal{R}_{\mathsf{role}}^{i}(rid) \cup \mathcal{R}_{\mathsf{ind}}(rid))) \times \mathcal{R}_{\mathsf{env}}^{e+} \rightarrow \mathcal{E}$ that synchronizes events of the* *two systems* $|||_{i,rid} \left(\mathcal{R}^i_{\mathsf{role}}(rid) \,|||\, \mathcal{R}_{\mathsf{ind}}(rid)\right)$ *and* $\mathcal{R}^{e+}_{\mathsf{env}}$*, i.e.,*

$$\chi^+(e, e') = \begin{cases} [] & \text{if } e = F^s(rid, x) \text{ and } e' = F^e(rid, x) \\ \chi(e, e') & \text{if } e \neq F^s(rid, x) \text{ and } e' \neq F^e(rid, x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $F \in \{\lambda_{\mathsf{out}_{\mathsf{ind}}}, \lambda_{\mathsf{in}_{\mathsf{ind}}}, \lambda_{\mathsf{Fr}_{\mathsf{ind}}}\}$ *and the partial function* χ *[81, App. E.5] synchronizes labels occurring in* $\mathcal{R}^i_{\mathsf{role}}$ *and* $\mathcal{R}^e_{\mathsf{env}}$ *and* $[]$ *denotes the empty transition label.*

[81]: Arquint et al. (2022), *Sound Verification of Security Protocols: From Design to Interoperable Implementations (extended version)*

Lemma 4.4.1 (Protocol-Independent Components Refine the Attacker)

$$\left(|||_{i,rid} \left(\mathcal{R}^i_{\mathsf{role}}(rid) \,|||\, \mathcal{R}_{\mathsf{ind}}(rid)\right)\right) \parallel_{\chi^+} \mathcal{R}^{e+}_{\mathsf{env}}$$
$$\preccurlyeq \left(|||_{i,rid} \mathcal{R}^i_{\mathsf{role}}(rid)\right) \parallel_{\chi} \mathcal{R}^e_{\mathsf{env}}$$

Given $\mathcal{R}_{\mathsf{ind}}(rid)$ and $\mathcal{R}^{e+}_{\mathsf{env}}$, Lemma 4.4.1 states that we can treat the system (on the first line) consisting of possibly unboundedly many instances of components executing a protocol role and executing protocol-independent operations as a refinement of the system on the second line that does not have components executing protocol-independent operations and uses an environment without the rules in $\mathcal{R}^+_{\mathsf{io}}$.

The following proof proceeds by setting up a simulation relation that merges the states of components executing protocol-independent operations with the environment and renames certain fact symbols. Using this simulation relation, we show that each transition in the concrete system can be simulated by the abstract system. While this simulation is straightforward for transitions executed by components that are present in both, the concrete and abstract system, concrete transitions corresponding to protocol-independent operations are more insightful as we show that the abstract environment, namely our DY attacker model, can simulate those transitions.

Proof. We denote $\mathcal{E}$ and $\mathcal{E}'$ the abstract and concrete systems, respectively, and prove this lemma by establishing refinement with a stuttering simulation relation $\mathcal{R}$ between states of the abstract system $\mathcal{E}$ and states of the concrete system $\mathcal{E}'$. I.e., $\mathcal{E} = \left(|||_{i,rid} \mathcal{R}^i_{\mathsf{role}}(rid)\right) \parallel_{\chi} \mathcal{R}^e_{\mathsf{env}}$ and $\mathcal{E}' = \left(|||_{i,rid} \left(\mathcal{R}^i_{\mathsf{role}}(rid) \,|||\, \mathcal{R}_{\mathsf{ind}}(rid)\right)\right) \parallel_{\chi^+} \mathcal{R}^{e+}_{\mathsf{env}}$. Using a stuttering simulation relation in contrast to a standard simulation relation allows us to relate the abstract and concrete states even if the concrete system performs additional transitions that do not have a corresponding transition in the abstract system, i.e., the abstract system can stutter as long as the observable behaviors of the two systems remain the same. Accordingly, we use $\dot{\longrightarrow}_{\mathcal{E}}$ and $\dot{\longrightarrow}_{\mathcal{E}'}$ to denote a transition step in the abstract system $\mathcal{E}$ and concrete system $\mathcal{E}'$, respectively. Additionally, we use $\dot{\longrightarrow}_{\mathcal{R}^i_{\mathsf{role}}(rid)}$ and $\dot{\longrightarrow}_{\mathcal{R}^e_{\mathsf{env}}}$ for transitions performed by the individual components in the abstract system and, similarly, $\dot{\longrightarrow}_{\mathcal{R}^i_{\mathsf{role}}(rid)}$, $\dot{\longrightarrow}_{\mathcal{R}_{\mathsf{ind}}(rid)}$ and $\dot{\longrightarrow}_{\mathcal{R}^{e+}_{\mathsf{env}}}$ for the concrete system's components.

The abstract states are of the shape $((s_{i,rid})_{1 \le i \le n, \text{ for each } rid}, s_e)$. We use primed variables for referring to concrete states, which are of the shape $((s'_{i,rid}, s'_{ind,i,rid})_{1 \le i \le n, \text{ for each } rid}, s'_e)$, i.e., they are composed of two multisets of facts for each i, rid, and one for the environment. Each multiset $s'_{i,rid}$ corresponds to the state of instance rid executing the protocol role i, while

$s'_{ind,i,rid}$ corresponds to the state of the component executing protocol-independent operations, which is conceptually co-located with $s'_{i,rid}$ but guaranteed by our taint analysis to operate on distinct state.

We use a stuttering simulation relation $\mathcal{R}$, such that $(s, s') \in \mathcal{R}$ iff

$$s = ((s'_{i,rid})_{1 \le i \le n,\ \text{for each } rid},\ r((\cup^{\mathsf{m}}_{i,rid} s'_{ind,i,rid}) \cup^{\mathsf{m}} s'_e)),$$

where $s' = ((s'_{i,rid}, s'_{ind,i,rid})_{1 \le i \le n,\ \text{for each } rid}, s'_e)$ and r is the identity function except for the cases specified below. We lift r to operate on multiset of facts. This lifted version removes duplicate K facts because K is a persistent fact symbol.

$$\left.\begin{aligned} r(\mathsf{ind}(rid, x)) &= \mathsf{K}(x) \\ r(\mathsf{out}_{\mathsf{ind}}(x)) &= \mathsf{K}(x) \\ r(\mathsf{in}_{\mathsf{ind}}(x)) &= \mathsf{K}(x) \end{aligned}\right\} \text{for all } rid, x.$$

I.e., to derive s_e from s', we, first, combine all facts in the states of protocol-independent components $s'_{ind,i,rid}$ with s'_e by applying multiset union $\cup^{\mathsf{m}}$ and, second, rename and deduplicate these facts according to the renaming function r.

It is clear that the initial states are related, i.e., $(s, s') \in \mathcal{R}$ with $s = ((\emptyset, \dots, \emptyset), \emptyset)$ and $s' = ((\emptyset, \emptyset), \dots, (\emptyset, \emptyset), \emptyset)$. We now show that for all states $(s_1, s'_1) \in \mathcal{R}$ and for all concrete transition steps $s'_1 \xrightarrow{e}_{\mathcal{E}'} s'_2$ there exists an abstract transition $s_1 \xrightarrow{e}_{\mathcal{E}} s_2$ such that $(s_2, s'_2) \in \mathcal{R}$. We use the following naming convention to refer to individual multisets within the abstract and concrete states, respectively, for $j \in \{1, 2\}$:

$$\begin{aligned} s_j &= ((s_{j,i,rid})_{1 \le i \le n,\ \text{for each } rid}, s_{j,e}) \\ s'_j &= ((s'_{j,i,rid}, s'_{j,ind,i,rid})_{1 \le i \le n,\ \text{for each } rid}, s'_{j,e}) \end{aligned}$$

Based on the definition of the parallel and synchronizing composition, $|||$ and $||_{\chi^+}$, resp., we distinguish the following two cases for the transition step $s'_1 \xrightarrow{e}_{\mathcal{E}'} s'_2$:

- $e = \chi^+(F^s(rid, x), F^e(rid, x))$ for $F \in \{\lambda_{\mathsf{out}_{\mathsf{ind}}}, \lambda_{\mathsf{in}_{\mathsf{ind}}}, \lambda_{\mathsf{Fr}_{\mathsf{ind}}}\}$:
  Since $s'_1 \xrightarrow{e}_{\mathcal{E}'} s'_2$, we have:

$$s'_{1,ind,i,rid} \xrightarrow{F^s(rid,x)}_{\mathcal{R}_{\mathsf{ind}}(rid)} s'_{2,ind,i,rid}$$

$$s'_{1,e} \xrightarrow{F^e(rid,x)}_{\mathcal{R}^{e+}_{\mathsf{env}}} s'_{2,e}$$

  and all other component states remain unchanged, i.e.,

$$\begin{aligned} s'_{2,i,rid} &= s'_{1,i,rid} \\ s'_{2,j,rid'} &= s'_{1,j,rid'} \\ s'_{2,ind,j,rid'} &= s'_{1,ind,j,rid'} \end{aligned}$$

  for all $(j, rid') \neq (i, rid)$. We now need to distinguish the cases where $F = \lambda_{\mathsf{out}_{\mathsf{ind}}}$, $F = \lambda_{\mathsf{in}_{\mathsf{ind}}}$, and $F = \lambda_{\mathsf{Fr}_{\mathsf{ind}}}$.
  - $F = \lambda_{\mathsf{out}_{\mathsf{ind}}}$: By definition of the transition rule F^s, we have $\mathsf{ind}(rid, x) \in s'_{1,ind,i,rid}$ and $s'_{2,ind,i,rid} = s'_{1,ind,i,rid} \setminus^m \{\!|\mathsf{ind}(rid, x)|\!\}$. Similarly, by definition of F^e, we have $s'_{2,e} = s'_{1,e} \cup^m \{\!|\mathsf{out}_{\mathsf{ind}}(x)|\!\}$. By definition of χ^+, the transition label e is the empty label $[]$.

We simulate this transition in $\mathcal{E}$ by stuttering, i.e., $s_2 = s_1$. Since r renames both facts $\mathsf{ind}(rid, x)$ and $\mathsf{out_{ind}}(x)$ to $\mathsf{K}(x)$ and $(s_1, s_1') \in \mathcal{R}$, we have $\mathsf{K}(x) \in s_{1,e}$. Additionally, the multiset minus and multiset union operations cancel out after applying r such that $s_{2,e} = s_{1,e}$. Therefore, $(s_2, s_2') \in \mathcal{R}$.

- $F = \lambda_{\mathsf{in_{ind}}}$: This case is analogous to $F = \lambda_{\mathsf{out_{ind}}}$.
- $F = \lambda_{\mathsf{Fr_{ind}}}$: By definition of F^s and F^e, we have

$$\mathsf{Fr}(x \in \mathit{fresh}) \in s'_{1,e},$$

$$s'_{2,e} = s'_{1,e} \setminus^m \{\!|\mathsf{Fr}(x)|\!\}, \text{ and}$$
$$s'_{2,ind,i,rid} = s'_{1,ind,i,rid} \cup^m \{\!|\mathsf{ind}(rid, x)|\!\}.$$

Since $(s_1, s_1') \in \mathcal{R}$, we obtain $\mathsf{Fr}(x) \in s_{1,e}$ enabling us to apply the attacker's message deduction rule (from MD_Σ) $[\mathsf{Fr}(x \in \mathit{fresh})] \xrightarrow{[]} [\mathsf{K}(x)]$, which results in $s_{2,e} = s_{1,e} \setminus^m \{\!|\mathsf{Fr}(x)|\!\} \cup^m \{\!|\mathsf{K}(x)|\!\}$. Due to the renaming function r applied to $s'_{2,ind,i,rid}$, we obtain $(s_2, s_2') \in \mathcal{R}$.

▶ $e = \chi(e, e')$:
We consider the following four subcases based on the definition of χ:

- $e = \chi(\lambda^s_{F,i,rid}(\bar{m}), \lambda^e_{F,i,rid}(\bar{m}))$ for some F, i, rid, $\bar{m}$:
By definition, neither $\mathcal{R}_{\mathsf{ind}}(rid)$ nor $\mathcal{R}^+_{\mathsf{io}}$ contain any transition rule with a matching transition label. Hence, this transition step synchronizes a step in $\mathcal{R}^i_{\mathsf{role}}$ and $\mathcal{R}^e_{\mathsf{env}}$. By definition of our composition operators and since $s_1' \xrightarrow{e}_{\mathcal{E}'} s_2'$, we have

$$s'_{1,i,rid} \xrightarrow{\lambda^s_{F,i,rid}(\bar{m})}_{\mathcal{R}^{i,rid}_{\mathsf{role}}(rid)} s'_{2,i,rid}$$
$$s'_{1,e} \xrightarrow{\lambda^e_{F,i,rid}(\bar{m})}_{\mathcal{R}^e_{\mathsf{env}}} s'_{2,e}$$

and

$$s'_{2,j,rid'} = s'_{1,j,rid'}$$
$$s'_{2,ind,i,rid} = s'_{1,ind,i,rid}$$
$$s'_{2,ind,j,rid'} = s'_{1,ind,j,rid'}$$

for all $(j, rid') \neq (i, rid)$.
Since the renaming function r behaves like the identity function for facts occurring in the premise and conclusion of rules $\lambda^s_{F,i,rid}(\bar{m})$ and $\lambda^e_{F,i,rid}(\bar{m})$, the same rules can be applied in the abstract states $s_{1,i,rid}$ and $s_{1,e}$. I.e., we have

$$s_{1,i,rid} \xrightarrow{\lambda^s_{F,i,rid}(\bar{m})}_{\mathcal{R}^{i,rid}_{\mathsf{role}}(rid)} s_{2,i,rid}$$
$$s_{1,e} \xrightarrow{\lambda^e_{F,i,rid}(\bar{m})}_{\mathcal{R}^e_{\mathsf{env}}} s_{2,e}$$

and $(s_2, s_2') \in \mathcal{R}$.
- $e = \chi(e', \mathit{skip})$ for some $e' \in \mathcal{R}^i_{\mathsf{role}}(rid)$:
Then, $e' = e$ and by definition of our composition operators,

we obtain $s'_{1,i,rid} \xrightarrow{e}_{\mathcal{R}^i_{\mathsf{role}}(rid)} s'_{2,i,rid}$ and

$$\begin{aligned} s'_{2,j,rid'} &= s'_{1,j,rid'} \\ s'_{2,ind,i,rid} &= s'_{1,ind,i,rid} \\ s'_{2,ind,j,rid'} &= s'_{1,ind,j,rid'} \\ s'_{2,e} &= s'_{1,e} \end{aligned}$$

for all $(j, rid') \neq (i, rid)$. Since $(s_1, s'_1) \in \mathcal{R}$, we further have $s_{1,i,rid} = s'_{1,i,rid}$, $s_{1,i,rid} \xrightarrow{e}_{\mathcal{R}^i_{\mathsf{role}}(rid)} s_{2,i,rid}$, and, thus, $s_{2,i,rid} = s'_{2,i,rid}$. Therefore, $(s_2, s'_2) \in \mathcal{R}$.

- $e = \chi(e', skip)$ for some $e' \in \mathcal{R}_{\mathsf{ind}}(rid)$:
  $e' \neq F^s(rid, x)$ for $F \in \{\lambda_{\mathsf{out_{ind}}}, \lambda_{\mathsf{in_{ind}}}, \lambda_{\mathsf{Fr_{ind}}}\}$ by definition of χ^+. Therefore, e' must be the transition adding a public constant or applying a k-ary function f to the state of $\mathcal{R}_{\mathsf{ind}}(rid)$. We can simulate either transition in the abstract system $\mathcal{E}$ by performing the corresponding message deduction rule in MD_Σ, which updates the abstract state in the same way after merging the states of the environment and of the components performing protocol-independent operations and applying the renaming function r. Thus, $(s_2, s'_2) \in \mathcal{R}$.
- $e = \chi(skip, e')$ for some $e' \in \mathcal{R}^{e+}_{\mathsf{env}}$:
  Then, $e' = e$ and, by definition of the composition operators, we obtain $s'_{1,e} \xrightarrow{e}_{\mathcal{R}^{e+}_{\mathsf{env}}} s'_{2,e}$, $s'_{2,i,rid} = s'_{1,i,rid}$, and $s'_{2,ind,i,rid} = s'_{1,ind,i,rid}$ for all i, rid. By definition of χ^+, e cannot have the shape $F^e(rid, x)$ for some rid, x, and $F \in \{\lambda_{\mathsf{out_{ind}}}, \lambda_{\mathsf{in_{ind}}}, \lambda_{\mathsf{Fr_{ind}}}\}$, which rules out the transitions in $\mathcal{R}^+_{\mathsf{io}}$. Thus, $e \in \mathcal{R}^e_{\mathsf{env}}$. Since $(s_1, s'_1) \in \mathcal{R}$, we have $s'_{1,e} \subseteq^m s_{1,e}$. Since e's guard is stable under supermultiset, the rewrite rule e can be applied in state $s_{1,e}$, i.e., $s_{1,e} \xrightarrow{e}_{\mathcal{R}^e_{\mathsf{env}}} s_{2,e}$. As this abstract transition only modifies the submultiset $s'_{1,e}$ by adding or removing facts for which the renaming function r behaves as the identity function and leaves all other $s_{1,i,rid}$ and $s_{1,e} \setminus^m s'_{1,e}$ unchanged, we obtain $s_{2,e} = r((\cup^{\mathsf{m}}_{i,rid} s'_{1,ind,i,rid}) \cup^{\mathsf{m}} s'_{2,e})$. Thus, $s_1 \xrightarrow{e}_{\mathcal{E}} s_2$ and $(s_2, s'_2) \in \mathcal{R}$.

□

Theorem 4.4.2 (Soundness) *Suppose Asm. 4.4.1 holds and that we have verified, for each role i, the Hoare triple $\vdash_{\pi'_{ext}} [\psi_i(rid)]\ c_i(rid)\ [\mathsf{true}]$. Then*

$$(|||_{i,rid}\ \pi_{int}(\mathcal{C}_i(rid)))\ ||_{\chi'}\ \mathcal{E} \preccurlyeq_t \mathcal{R}.$$

Thm. 4.4.2 states that composing unboundedly many instances of each role's LTS $\mathcal{C}_i(rid)$ with the concrete environment $\mathcal{E}$ refines the protocol model $\mathcal{R}$. While this theorem is identical to Thm. 2.5.2, the proof differs since $\mathcal{C}$ and each $\mathcal{C}_i(rid)$ contained therein describe larger sets of traces in Asm. 4.4.1 than in Asm. 2.4.1.

Proof. We decompose the proof into a similar series of trace inclusions as Sec. 2.5.2 but add an additional trace inclusion to abstract the protocol-independent I/O operations to the environment, which contains the DY attacker (cf. Lemma 4.4.1).

The first trace inclusion is

$$\begin{aligned}&\left(|||_{i,rid}\ \pi_{\mathrm{int}}(\mathscr{C}_i(rid))\right) \parallel_{\chi'} \mathscr{E}\\ \preccurlyeq\ &\left(|||_{i,rid}\ \pi(\pi'_{\mathrm{ext}}(\mathscr{C}_i(rid)))\right) \parallel_{\chi^+} \pi_{\mathrm{ext}}(\pi'_{\mathrm{ext}}(\mathscr{E})),\end{aligned} \tag{4.1}$$

where we obtain the first line from the second by pushing the relabeling $\pi_{\mathrm{ext}} \circ \pi'_{\mathrm{ext}}$ into the parallel composition, thus changing the set of synchronization labels from χ^+ to χ'.

By combining Asm. 4.4.1, the assumption $\vdash_{\pi'_{\mathrm{ext}}} \left[\psi_i(rid)\right]\ c_i(rid)\ [\mathsf{true}]$, and Thm. 2.3.3, we obtain

$$\pi(\pi'_{\mathrm{ext}}(\mathscr{C}_i(rid))) \preccurlyeq \mathscr{R}^i_{\mathsf{role}}(rid)\ |||\ \mathscr{R}_{\mathsf{ind}}(rid), \tag{4.2}$$

where $\mathscr{R}_{\mathsf{ind}}(rid)$ is a multiset rewriting (MSR) system capturing the execution of protocol-independent I/O operations. All facts in this MSR system are instantiated with the thread id *rid*, which helps in distinguishing the facts belonging to each $\mathscr{R}_{\mathsf{ind}}$ instance. Additionally, (4.2) implicitly specifies that the MSR systems $\mathscr{R}^i_{\mathsf{role}}(rid)$ and $\mathscr{R}_{\mathsf{ind}}(rid)$ operate independently, i.e., on different multisets of facts. By performing the taint analysis, we ensure that $\mathscr{R}^i_{\mathsf{role}}(rid)$ does not influence $\mathscr{R}_{\mathsf{ind}}(rid)$. Checking the opposite, i.e., that $\mathscr{R}_{\mathsf{ind}}(rid)$ does not influence $\mathscr{R}^i_{\mathsf{role}}(rid)$ by performing a second taint analysis is not necessary. We explicitly track throughout code-level verification the multiset of facts representing the state of $\mathscr{R}^i_{\mathsf{role}}(rid)$, which is only manipulated by internal and I/O library functions corresponding to state updates permitted by $\mathscr{R}^i_{\mathsf{role}}(rid)$. Therefore, this state cannot be influenced by $\mathscr{R}_{\mathsf{ind}}(rid)$.

We can leverage a general composition theorem [71, Thm. 2.3] that implies that trace inclusion is compositional for a large class of composition operators including $|||$ and $\parallel_\Lambda$. Applying this theorem to (4.2) and Prop. 2.5.1 establishes the trace inclusion

[71]: Sprenger et al. (2020), *Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification*

$$\begin{aligned}&\left(|||_{i,rid}\ \pi(\pi'_{\mathrm{ext}}(\mathscr{C}_i(rid)))\right) \parallel_{\chi^+} \pi_{\mathrm{ext}}(\pi'_{\mathrm{ext}}(\mathscr{E}))\\ \preccurlyeq\ &\left(|||_{i,rid}\ \left(\mathscr{R}^i_{\mathsf{role}}(rid)\ |||\ \mathscr{R}_{\mathsf{ind}}(rid)\right)\right) \parallel_{\chi^+} \mathscr{R}^{e+}_{\mathsf{env}}.\end{aligned} \tag{4.3}$$

Applying Lemma 4.4.1 in connection with Lemma 2.3.1 and Lemma 2.3.2 completes the proof. □

4.4.2 Combining Auto-Active Verification and Static Analyses

In this subsection, we sketch soundness of our combination of auto-active program verification and fully automatic static analyses by constructing a proof in concurrent separation logic [25, 110] for the entire codebase. More specifically, we give an invariant that is maintained by each statement in our programming language and present proof rules that use, besides certain side conditions, only this invariant in their premises and conclusions. We use Diodon's static analyses to discharge these side conditions. Therefore, we can compose the proof rules to prove $\vdash \left[\phi\right]\ c\ [\mathsf{true}]$ for an I/O specification ϕ and an entire codebase c.

[25]: Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*
[110]: Vafeiadis (2011), *Concurrent Separation Logic and Operational Semantics*

To focus on the main proof insights, we deliberately keep the considered programming language simple (cf. Def. 4.4.6) and consider the case where an execution of the codebase c corresponds to at most one execution of a protocol role, which is represented by the I/O specification ϕ. We discuss limitations and extensions lifting these restrictions at the end of this subsection.

Prerequisites. We consider an imperative, concurrent, and heap-manipulating programming language as shown in Def. 4.4.6. For simplicity, we omit function boundaries and statements creating complex control flow. Furthermore, we assume that programs are in static single assignment (SSA) form such that we do not have to consider variable reassignments for the purpose of our proof. Besides statements to allocate, read, and write a heap location, we make each auto-actively verified API function of the Core a dedicated statement in the language even though these statements are themselves implemented as sequences of statements, which are considered by our static analyses. $c := \textbf{CoreAlloc}(\bar{e})$ corresponds to calling the Core's constructor and creating a new Core instance c. We use $\bar{r} := \textbf{CoreApi\_k}(c, \bar{e})$ to represent invoking the k-th API function[4] on a Core instance c using input arguments $\bar{e}$ and return arguments $\bar{r}$. $s_1; s_2$ denotes standard sequential composition of two statements and $\textbf{fork}\ (\bar{x})\ \{s\}$ spawns a new thread executing statement s while passing variables $\bar{x}$ to this thread. We syntactically require that the newly spawned thread accesses only its own local variables and variables $\bar{x}$.

4: We assume the existence of some total order for API functions, e.g., based on their declarations' syntactical ordering.

Definition 4.4.6 (Basic Programming Language) *We consider the following programming language, where S ranges over labeled statements, x over variables, ℓ over statement labels, and e over expressions. We have the usual side effect-free expressions. We use $\bar{y}$ as a shorthand notation to denote lists of kind y.*

$$\begin{aligned} S &\triangleq U^{\ell} \\ U &\triangleq \textbf{skip} \mid x := \textbf{new}() \mid x := *e \mid *x := e \mid \\ &\quad\; c := \textbf{CoreAlloc}(\bar{e}) \mid \bar{r} := \textbf{CoreApi\_k}(c, \bar{e}) \mid \\ &\quad\; S; S \mid \textbf{fork}\ (\bar{x})\ \{S\} \end{aligned}$$

We call $S; S$ and **fork** $(\bar{x})\ \{S\}$ compound *statements, while all other statements in our language are called* simple. *When not relevant, we omit a statement's label ℓ, which uniquely identifies the statement in the program text. We use these labels to refer to the program points before and after each labeled statement. We abstractly treat $c := \textbf{CoreAlloc}(\bar{e})$ and $\bar{r} := \textbf{CoreApi\_k}(c, \bar{e})$ as first-class statements in our language despite being implemented as sequences of statement that are considered by our static analyses and the auto-active program verifier. This is possible because we can treat these statements as opaque boxes from a proof construction point of view as we prove a Hoare triple for each such statement using the auto-active program verifier.*

We assume that all memory accesses in the unverified Application neither cause crashes nor data races such that we can reason about their effects. While we could have avoided assuming data race freedom by defining that all heap operations in our language are atomic, we try to stay faithful to most programming languages, which specify data races to cause undefined behavior, thus, making this assumption necessary.

Assumption 4.4.2 (Crash Freedom) *We assume that all heap accesses within the Application, $x := *e$ and $*x := e$ do not crash*[5]*, i.e., the Application dereferences only pointers to allocated heap locations as opposed to* `nil`*.*

5: We require Asm. 4.4.2 as we construct a Hoare triple for the entire codebase, whose definition includes that a program does not crash. We could avoid this assumption by altering the definition of a Hoare triple to guarantee the postcondition *only* if the program does not crash. This alternative definition would be suitable for programming languages like Go in which dereferencing `nil` is defined behavior and results in a crash, which does not invalidate our security guarantees.

Assumption 4.4.3 (Data Race Freedom) *We assume that all heap accesses within the* Application, $x := *e$ *and* $*x := e$, *are data race free. I.e., all accesses to the heap locations to which* e *and* x, *respectively, point are linearizable and, thus, do not cause data races.*

Note that Asm. 4.4.2 and Asm. 4.4.3 apply to heap accesses within the Application only, as we auto-actively prove safety for the Core.

By auto-actively verifying the Core, we prove a Hoare triple for each API function. This allows us to abstractly treat each API function as a statement in our language as long as there are no callbacks; we discuss callbacks as an extension at the end of this subsection. We syntactically restrict the specification of Core API functions, i.e., the assertions occurring in the auto-actively verified Hoare triples, such that we can discharge the side conditions using static analyses and, thus, construct a proof for the entire codebase. We state these restrictions immediately after introducing some notational conventions.

Definition 4.4.7 (Notation) *We introduce the following notation to simplify forthcoming definitions, explanations, and proofs.* $\mathtt{acc}_{\mathtt{nil}}(x)$ *denotes full permission for the heap location to which* x *points but only if* x *is non-*`nil`. *Analogously, we define* $\mathtt{inv}_{\mathtt{nil}}(x)$ *for the* Core *invariant. Lastly, we lift permissions for a heap location to lists thereof, internally using the iterated separating conjunction* $\bigstar_i$ *ranging over* i.

$$\begin{aligned}
\mathtt{acc}_{\mathtt{nil}}(x) &\triangleq x \neq \mathtt{nil} \implies \mathtt{acc}(x)\\
\mathtt{inv}_{\mathtt{nil}}(x) &\triangleq x \neq \mathtt{nil} \implies \mathtt{inv}(x)\\
\mathtt{acc}(\bar{x}) &\triangleq \bigstar_{0\leq i<len(\bar{x})}\mathtt{acc}(\bar{x}[i])\\
\mathtt{acc}_{\mathtt{nil}}(\bar{x}) &\triangleq \bigstar_{0\leq i<len(\bar{x})}\mathtt{acc}_{\mathtt{nil}}(\bar{x}[i])
\end{aligned}$$

where $len(\bar{x})$ *returns the length of list* $\bar{x}$ *and* $\bar{x}[i]$ *the i-th element therein.*

Assumption 4.4.4 (Syntactic Restrictions for Core Specification) *We make the following syntactical assumptions about the pre- and postconditions of* Core API *functions, which ultimately enable us to apply static analyses.*

$$\begin{aligned}
P_{CoreAlloc}(\bar{e}) &\triangleq \phi \star R\\
Q_{CoreAlloc}(c, \bar{e}) &\triangleq \mathtt{inv}(c) \star R'\\
P_{CoreApi\_k}(c, \bar{e}) &\triangleq \mathtt{inv}_{\mathtt{nil}}(c) \star S_k\\
Q_{CoreApi\_k}(c, \bar{e}, \bar{r}) &\triangleq \mathtt{inv}_{\mathtt{nil}}(c) \star S'_k
\end{aligned}$$

where R, R', S_k, *and* S'_k *are separation logic assertions that specify permissions for the arguments* $\bar{e}$ *and, if applicable,* $\bar{r}$. *Preconditions are free of functional properties and specify at most permissions for non-nil arguments, i.e.,* $\mathtt{acc}_{\mathtt{nil}}(\bar{e}) \models R$ *and* $\mathtt{acc}_{\mathtt{nil}}(\bar{e}) \models S_k$. *Each postcondition needs to specify the same or more permissions than the respective precondition, i.e.,* $R' \models R$ *and* $S'_k \models S_k$. *Additionally, postconditions need to specify full permission to every heap location that becomes accessible to the* Application *and that is created within the corresponding* Core *function or any function transitively called thereby. For simplicity, we disallow* **CoreAlloc**$(\bar{e})$ *to return such heap locations other than the* Core *instance itself and, thus, permissions to such heap locations can only occur in* S'_k *for the return arguments* $\bar{r}$. *Furthermore, we restrict the input arguments* $\bar{e}$ *and output arguments* $\bar{r}$ *to be shallow, i.e., their transitive closure of reachable heap locations is the singleton set, i.e.,* $\forall e \in \bar{e}.\, e \neq \mathtt{nil} \implies reach(e) = \{e\}$ *and analogously for* $\bar{r}$. *This restriction* *simplifies the reasoning about which heap locations are passed between the* Core *and* Application*. However, lifting this restriction is possible and would require that S_k and S'_k specify the permission for every reachable heap location.*

Program Invariant

In order to define composable proof rules for our language, we define a program invariant that is maintained by each statement. Our invariant conceptually partitions the heap among two dimensions, namely whether a heap location is accessible by multiple threads and whether a heap location is owned by the Application as opposed to the Core. As we will formalize later, we call a heap location h *Application-managed* if h is under the Application's control, which means that it is not covered by the Core invariant. Furthermore, we ensure that the Application only accesses memory that is Application-managed.

We make the heap partitioning explicit by introducing ghost variables tracking the heap locations belonging to each partition. We use a global ghost set pointed to by ghs tracking the set of heap locations that are accessible by multiple threads. The thread-local ghost variable lhs tracks Application-managed heap locations that are only accessible by the current thread. Lastly, the thread-local variable ihs tracks the Core instance if it is already allocated.

Relying on these ghost variables, we can define the *program invariants* Π_l and Π_g that specify the separation logic permissions held by a thread at each program point, as shown in Def. 4.4.8, where used is a pointer to a boolean specifying whether the I/O permissions ϕ have already been consumed to allocate a Core instance.

Definition 4.4.8 (Program Invariants)

$$\Pi_l \triangleq \left(\bigstar_{l \in \mathsf{lhs}} \mathtt{acc}(l)\right) \star \left(\bigstar_{i \in \mathsf{ihs}} \mathtt{inv}(i)\right)$$
$$\Pi_g \triangleq \mathtt{acc}(\mathsf{ghs}) \star \left(\bigstar_{g \in *\mathsf{ghs}} \mathtt{acc}(g)\right) \star$$
$$\mathtt{acc}(\mathsf{used}) \star \left(\neg(*\mathsf{used}) \implies \phi\right)$$

Π_l specifies permissions that are exclusively owned by each thread. The first conjunct specifies (full) permissions to every heap location in lhs, which, as we will see, holds every heap location that is accessible only by the current thread and is unrelated to Core instances. Additionally, Π_l specifies that the Core invariant $\mathtt{inv}(i)$ holds for each Core instance i. Note that $\mathtt{inv}(i)$ is a separation logic predicate that specifies permissions for a subset of the transitively reachable heap locations starting from i and possibly functional properties about these heap locations. While the definition of $\mathtt{inv}(i)$ matters for the Core's auto-active verification, we treat $\mathtt{inv}(i)$ for the purpose of the program invariant as an opaque separation logic resource.

Π_g specifies permissions to heap locations that are potentially shared among multiple threads. When accessing such a heap location, a thread can temporarily acquire the corresponding permission from Π_g, which is justified as long as all accesses to this location are linearizable. Since we assume absence of data races (cf. Asm. 4.4.3), there exists a linearization of heap accesses such that permission for manipulating g, i.e., $\mathtt{acc}(g)$, can temporarily be obtained from Π_g for the manipulation's duration. Furthermore, Π_g specifies the I/O permissions ϕ if they have not been

$$
\begin{aligned}
\mathbb{A}(\mathbf{skip}) &\rightsquigarrow \mathbf{skip} \\
\mathbb{A}(x := \mathbf{new}()) &\rightsquigarrow x := \mathbf{new}();\ \mathsf{lhs} := \mathsf{lhs} \cup_{\mathtt{nil}} \{x\} \\
\mathbb{A}(x := *e) &\rightsquigarrow \begin{cases} \mathsf{lhs} := \mathsf{lhs} \setminus \{e\};\ x := *e;\ \mathsf{lhs} := \mathsf{lhs} \cup_{\mathtt{nil}} \{e\} & \text{if } e \in \mathsf{lhs} \\ \mathbf{atomic}\ \{*\mathsf{ghs} := *\mathsf{ghs} \setminus \{e\};\ x := *e;\ *\mathsf{ghs} := *\mathsf{ghs} \cup_{\mathtt{nil}} \{e\}\} & \text{otherwise} \end{cases} \\
\mathbb{A}(*x := e) &\rightsquigarrow \begin{cases} \mathsf{lhs} := \mathsf{lhs} \setminus \{x\};\ *x := e;\ \mathsf{lhs} := \mathsf{lhs} \cup_{\mathtt{nil}} \{x\} & \text{if } x \in \mathsf{lhs} \\ \mathbf{atomic}\ \{ *\mathsf{ghs} := *\mathsf{ghs} \setminus \{x\};\ \mathsf{lhs} := \mathsf{lhs} \setminus (\mathit{reach}(e) \cap \mathsf{lhs}); & \\ \qquad *x := e;\ *\mathsf{ghs} := *\mathsf{ghs} \cup_{\mathtt{nil}} \{x\} \cup (\mathit{reach}(e) \cap \mathsf{lhs})\} & \text{otherwise} \end{cases} \\
\mathbb{A}(c := \mathbf{CoreAlloc}(\bar{e})) &\rightsquigarrow \mathbf{atomic}\ \{*\mathsf{used} := \mathsf{true}\};\ \mathsf{lhs} := \mathsf{lhs} \setminus \bar{e};\ c := \mathbf{CoreAlloc}(\bar{e});\ \mathsf{lhs} := \mathsf{lhs} \cup_{\mathtt{nil}} \bar{e};\ \mathsf{ihs} := \mathsf{ihs} \cup_{\mathtt{nil}} \{c\} \\
\mathbb{A}(\bar{r} := \mathbf{CoreApi\_k}(c, \bar{e})) &\rightsquigarrow \mathsf{ihs} := \mathsf{ihs} \setminus \{c\};\ \mathsf{lhs} := \mathsf{lhs} \setminus \bar{e};\ \bar{r} := \mathbf{CoreApi\_k}(c, \bar{e});\ \mathsf{lhs} := \mathsf{lhs} \cup_{\mathtt{nil}} \bar{e} \cup \bar{r};\ \mathsf{ihs} := \mathsf{ihs} \cup_{\mathtt{nil}} \{c\} \\
\mathbb{A}(s_1; s_2) &\rightsquigarrow \mathbb{A}(s_1);\ \mathbb{A}(s_2) \\
\mathbb{A}(\mathbf{fork}\ (\bar{x})\ \{s\}) &\rightsquigarrow \mathsf{lhs} := \mathsf{lhs} \setminus (\mathit{reach}(\bar{x}) \cap \mathsf{lhs});\ *\mathsf{ghs} := *\mathsf{ghs} \cup_{\mathtt{nil}} (\mathit{reach}(\bar{x}) \cap \mathsf{lhs});\ \mathbf{fork}\ (\bar{x})\ \{\mathsf{lhs} := \emptyset;\ \mathsf{ihs} := \emptyset;\ \mathbb{A}(s)\}
\end{aligned}
$$

Figure 4.8: Algorithm $\mathbb{A}$ transforms a codebase by inserting ghost statements. We define this algorithm by cases, i.e., describe how $\mathbb{A}$ transforms each statement s to a statement s', written as $\mathbb{A}(s) \rightsquigarrow s'$. $\mathit{reach}(e)$ computes the set of transitively reachable heap locations from expression e. The set union operation ignores `nil`, as variables might be `nil`, i.e., $S_1 \cup_{\mathtt{nil}} S_2 \triangleq (S_1 \cup S_2) \setminus \mathtt{nil}$. This ensures that `nil` is never contained in any ghost set.

used yet to create a Core instance, in which case the pointer used points to a heap location storing the value false. As mentioned, we focus in this proof on the case of creating at most one Core instance. However, this conjunct can easily be adapted to provide a family of I/O permissions such that the creation of arbitrarily-many Core instances becomes possible, as we will detail at the end of this subsection.

Since the presented program invariants rely on ghost variables to specify permissions, we have to ensure that these ghost variables stay in sync with a program's execution, i.e., the effects of each statement. Hence, we present algorithm $\mathbb{A}$ in Fig. 4.8 that augments a program with ghost statements updating the ghost variables according to each statement's effects. These ghost statements manipulate only ghost variables and aid verification without changing the input program's runtime behavior. Thus, these ghost variables and ghost statements can be erased before compilation.

Common to all cases of algorithm $\mathbb{A}$ is that for a statement s, first, the current heap is partitioned into a heap H on which s possibly operates and the remaining heap F that s leaves untouched by removing the separation logic resources for H via corresponding ghost set subtractions. The separation logic resources belonging to heap F remain in the ghost sets. Afterwards, statement s is executed that changes heap H to H', followed by merging the heaps H' and F via ghost set union operations.

Allocating a heap location operates on an empty heap and produces a new heap location, which is added to the set of local heap locations as there is no way any other thread might already have gained access thereto. Dereferencing a pointer e and reading the corresponding heap location requires a permission for the duration of this operation. Therefore, we first subtract and afterwards add this location from the current heap by manipulating either the ghost set of local or global heap locations depending on whether this heap location is contained in lhs. In case the heap location is in the ghost set of global heap locations, we insert an atomic block, which is justified by Asm. 4.4.3 stating that accesses to this heap location are linearizable.

Noteworthy are write operations to heap locations, especially in the case that a heap location is accessible by other threads as the written value becomes accessible by these threads. Hence, we first remove all local heap locations that are transitively reachable from the written value and

$$\frac{\omega(s_{\text{simple}})}{\Pi_g \vdash [\Pi_l]\ \mathbb{A}(s_{\text{simple}})\ [\Pi_l]}\ (\textsc{Simple})$$

$$\frac{\Pi_g \vdash [\Pi_l]\ \mathbb{A}(s_1)\ [\Pi_l] \quad \Pi_g \vdash [\Pi_l]\ \mathbb{A}(s_2)\ [\Pi_l]}{\Pi_g \vdash [\Pi_l]\ \mathbb{A}(s_1;\, s_2)\ [\Pi_l]}\ (\textsc{Seq}) \qquad \frac{\Pi_g \vdash [\Pi_l]\ \mathbb{A}(s)\ [\Pi_l]}{\Pi_g \vdash [\Pi_l]\ \mathbb{A}(\mathbf{fork}\ (\bar{x})\ \{s\})\ [\Pi_l]}\ (\textsc{Fork})$$

Figure 4.9: Proof rules. s_{simple} ranges over all *simple* statements; s, s_1, and s_2 range over all statements. ω denotes a statement's side conditions (cf. Fig. 4.10).

$$\begin{aligned}
\omega(x := *e) &\triangleq e \in \mathsf{lhs} \cup *\mathsf{ghs}\\
\omega(*x := e) &\triangleq x \in \mathsf{lhs} \cup *\mathsf{ghs}\\
\omega(c := \mathbf{CoreAlloc}(\bar{e})) &\triangleq\ *\mathsf{used} = \mathsf{false}\ \wedge\\
&\quad (\mathit{set}(\bar{e}) \setminus \mathtt{nil}) \subseteq \mathsf{lhs}\ \wedge\\
&\quad \mathit{disjoint}(\bar{e})\\
\omega(\bar{r} := \mathbf{CoreApi\_k}(c, \bar{e})) &\triangleq (\mathit{set}(\bar{e}) \setminus \mathtt{nil}) \subseteq \mathsf{lhs}\ \wedge\\
&\quad \mathit{disjoint}(\bar{e})\ \wedge\\
&\quad (c \in \mathsf{ihs} \vee c = \mathtt{nil})
\end{aligned}$$

Figure 4.10: Side conditions for our statements, which are amenable to static analyses. ω evaluates to true for all statements not listed above and $\mathit{set}(l)$ returns the set of elements in list l. We implicitly refer to variables' values, e.g., $v \in S$ denotes that the value of variable v is contained in set stored in variable S as opposed to the variables' syntactical representation.

add them afterwards to the ghost set of global heap location as these locations possibly escape the current thread via this write operation. Similarly, when forking a thread, the heap locations that are reachable from the variables $\bar{x}$ escape the current thread and, thus, the sets of local and global heap locations are updated accordingly.

For **CoreAlloc**$(\bar{e})$ and $\bar{r}$:= **CoreApi_k**$(c, \bar{e})$, the algorithm $\mathbb{A}$ adds and subtracts only $\bar{e}$ and $\bar{r}$ as opposed to all transitively reachable heap locations. This is sufficient because Asm. 4.4.4 restricts $\bar{e}$ and $\bar{r}$ to be shallow and, thus, no other heap locations are reachable. However, extending algorithm $\mathbb{A}$ to support non-shallow arguments would be straightforward by adding and removing $\mathit{reach}(\bar{e})$ and $\mathit{reach}(\bar{r})$ instead of $\bar{e}$ and $\bar{r}$ to and from lhs, respectively.

Proof Rules

Thanks to the program invariants and the ghost statements that algorithm $\mathbb{A}$ inserts into a program, we can define proof rules as shown in Fig. 4.9. In particular, all proof rules share the same pre- and postcondition, namely the local and global program invariants Π_l and Π_g, resp., which allow us to compose the proof rules to obtain a whole program proof. The proof rules' simplicity is enabled by their side conditions (cf. Fig. 4.10) that we discharge using our static analyses.

Besides containment of heap locations in particular ghost sets, the side conditions rely on disjointness of input arguments, which we formally define next. Informally, two arguments are disjoint if they point to different heap locations or one of the arguments is `nil`.

Definition 4.4.9 (Variable Value) *$val_\tau(x)$ denotes the value of variable x on trace τ. Since we assume that our programs are in SSA-form, this definition is independent of a particular program point. However, x must be declared such that $val_\tau(x)$ is defined.*

Definition 4.4.10 (Disjointness) *Two pointer variables x and y are disjoint if their pointer value is different or `nil` for all traces τ.*

$$disjoint(\{x, y\}) \triangleq \forall\tau.val_\tau(x) = \texttt{nil} \lor val_\tau(x) \neq val_\tau(y)$$

We straightforwardly lift this definition to lists of variables $\bar{z}$, where $disjoint(\bar{z})$ denotes pairwise disjointness between every element in $\bar{z}$.

Next, we sketch the proof rules' soundness proof, which relies on the side conditions ω. Afterwards, we define what properties our static analyses provide given that their execution succeeded and show that these properties imply the side conditions ω. We conclude by proving a corollary stating that we construct a Hoare triple for the entire codebase.

Theorem 4.4.3 (Soundness of Proof Rules)

$$\textit{If}\ \ \Pi_g \vdash [\Pi_l]\ \ \mathbb{A}(s)\ \ [\Pi_l]\,,\ \textit{then}\ \ \Pi_g \models [\Pi_l]\ \ \mathbb{A}(s)\ \ [\Pi_l]$$

Proof sketch. We perform structural induction over the input statement s to algorithm $\mathbb{A}$ and construct a proof tree in separation logic building up on the proof rules by Vafeiadis [110]. We use a small caps font to denote proof rules, such as Skip. All rules in this theorem's proof are from Vafeiadis [110], except Fork and Seq* that are straightforward extensions from the parallel and sequential composition rules, respectively. Side conditions arising in the proof trees are marked in blue and form ω (cf. Fig. 4.10).

[110]: Vafeiadis (2011), *Concurrent Separation Logic and Operational Semantics*

- $\mathbb{A}(\textbf{skip})$: Since the algorithm $\mathbb{A}$ does not insert any ghost commands and **skip** does not alter the program state, Π_l is trivially maintained. The Skip rule is immediately applicable and completes the proof tree.
- $\mathbb{A}(x := \textbf{new}())$: Fig. 4.20 shows the proof tree that uses Fig. 4.11 as a sub-proof for inserting a heap location into the ghost set of local heap locations.
- $\mathbb{A}(x := *e)$: The side condition ω ensures that $e \in \mathsf{lhs} \cup *\mathsf{ghs}$ holds. If $e \in \mathsf{lhs}$ then Fig. 4.21 is a valid proof tree for this read operation. Otherwise, $e \in *\mathsf{ghs}$ holds and Fig. 4.22 shows the corresponding proof tree.
- $\mathbb{A}(*x := e)$: For write operations, we construct a proof tree similar to read operations, as explained in the case above, except that we extract permissions for x instead of e from the program invariants and replace applications of the Read rule by Write. We can apply these rules because we possess full permission (as opposed to only partial permission) to the heap location (i.e., $\texttt{acc}(x)$).
- $\mathbb{A}(c := \textbf{CoreAlloc}(\bar{e}))$: Fig. 4.24 shows the proof tree extending the subproof that the auto-active program verifier implicitly constructs (in Fig. 4.23) while verifying the Hoare triple for $\textbf{CoreAlloc}(\bar{e})$.
- $\mathbb{A}(\bar{r} := \textbf{CoreApi\_k}(c, \bar{e}))$: We construct a proof tree in Fig. 4.26 using Fig. 4.25 as a subtree that is similar to the one for the Core allocation command with the main difference that the precondition requires $\texttt{inv}_{\texttt{nil}}(c)$ instead of the I/O permissions ϕ. The side condition $c \in \mathsf{ihs} \lor c = \texttt{nil}$ ensures that we can obtain $\texttt{inv}_{\texttt{nil}}(c)$ from Π_l within the proof.
- $\mathbb{A}(s_1; s_2)$: We apply the standard Seq rule from separation logic to combine the proof subtrees for $\mathbb{A}(s_1)$ and $\mathbb{A}(s_2)$ that we obtain by applying our induction hypothesis.

$$\dfrac{\dfrac{\dfrac{}{\Pi_g \vdash [\forall l \in \mathsf{lhs} \cup_{\texttt{nil}} \{x\}.\, \texttt{acc}(l)]\ \mathsf{lhs} := \mathsf{lhs} \cup_{\texttt{nil}} \{x\}\ [\forall l \in \mathsf{lhs}.\, \texttt{acc}(l)]}\ \textsc{Assign}}{\Pi_g \vdash [(\forall l \in \mathsf{lhs}.\, \texttt{acc}(l)) \star \texttt{acc}_{\texttt{nil}}(x)]\ \mathsf{lhs} := \mathsf{lhs} \cup_{\texttt{nil}} \{x\}\ [\forall l \in \mathsf{lhs}.\, \texttt{acc}(l)]}\ \textsc{Conseq}}{\Pi_g \vdash [\Pi_l \star \texttt{acc}_{\texttt{nil}}(x)]\ \mathsf{lhs} := \mathsf{lhs} \cup_{\texttt{nil}} \{x\}\ [\Pi_l]}\ \textsc{Frame}$$

Figure 4.11: Proof tree for $\mathsf{lhs} := \mathsf{lhs} \cup_{\texttt{nil}} \{x\}$, where $\texttt{acc}_{\texttt{nil}}(e) \triangleq e \neq \texttt{nil} \implies \texttt{acc}(e)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{emp} \vdash [\texttt{acc}(\mathsf{ghs}) \star {*}\mathsf{ghs} = v]\ {*}\,\mathsf{ghs} := {*}\mathsf{ghs} \cup_{\texttt{nil}} \{x\}\ [\texttt{acc}(\mathsf{ghs}) \star {*}\mathsf{ghs} = v \cup_{\texttt{nil}} \{x\}]}\ \textsc{Write}}{\mathbf{emp} \vdash [\texttt{acc}(\mathsf{ghs}) \star {*}\mathsf{ghs} = v \star R]\ {*}\,\mathsf{ghs} := {*}\mathsf{ghs} \cup_{\texttt{nil}} \{x\}\ [\texttt{acc}(\mathsf{ghs}) \star {*}\mathsf{ghs} = v \cup_{\texttt{nil}} \{x\} \star R]}\ \textsc{Frame}}{\mathbf{emp} \vdash \left[\texttt{acc}(\mathsf{ghs}) \star (\forall g \in {*}\mathsf{ghs}.\, \texttt{acc}(g)) \star \texttt{acc}_{\texttt{nil}}(x)\right]\ {*}\,\mathsf{ghs} := {*}\mathsf{ghs} \cup_{\texttt{nil}} \{x\}\ \left[\texttt{acc}(\mathsf{ghs}) \star \forall g \in {*}\mathsf{ghs}.\, \texttt{acc}(g)\right]}\ \textsc{Conseq}}{\mathbf{emp} \vdash \left[\Pi_g \star \texttt{acc}_{\texttt{nil}}(x)\right]\ {*}\,\mathsf{ghs} := {*}\mathsf{ghs} \cup_{\texttt{nil}} \{x\}\ \left[\Pi_g\right]}\ \textsc{Frame}$$

$$\text{with}\quad R \triangleq \forall g \in (v \cup_{\texttt{nil}} \{x\}).\, \texttt{acc}(g)$$

Figure 4.12: Proof tree for ${*}\mathsf{ghs} := {*}\mathsf{ghs} \cup_{\texttt{nil}} \{x\}$ given that Π_g is already local, where v is a *fresh* variable and the Write rule has been naturally extended to internally perform a heap read operation returning the value v for ${*}\mathsf{ghs}$ as specified in the precondition.

- $\mathbb{A}(\mathbf{fork}\ (\bar{x})\ \{s\})$: Fig. 4.28 shows the proof tree that applies the induction hypothesis to $\mathbb{A}(s)$. Since the algorithm $\mathbb{A}$ removes the permissions for heap locations only in $\mathit{reach}(\bar{x}) \cap \mathsf{lhs}$, the resulting side condition $((\mathit{reach}(\bar{x}) \cap \mathsf{lhs}) \subseteq \mathsf{lhs})$ is trivial since these heap locations are by definition contained in lhs.
  The main proof insight is that we ensure that the global invariant covers the permissions for all heap locations that become accessible by the spawned thread and establish the local invariant for the spawned thread by initializing the set of local heap locations and (local) Core instances to the empty set. $\mathit{reach}(\bar{x})$ forms an upper bound on the heap locations that command s might access because we syntactically require that s accesses only $\bar{x}$ and its own local variables.

□

Static Analyses

Since our proof rules rely on the side conditions ω (cf. Fig. 4.10), we introduce next our static analyses, cover the properties we assume they provide, and show that these properties imply ω. We end by proving a corollary that we can construct a whole program proof for a codebase given that we have auto-actively verified the Core and successfully executed the static analyses.

Pointer Analysis. A pointer analysis computes for each pointer x a set of heap locations L to where x *may* point, which we formalize as a judgement $\mathrm{pts}(x) = L$. Each heap location in L is identified by its allocation site, which corresponds to the label of a particular statement in the program's text. Note that this analysis over-approximates the set of heap locations that actually change when writing to x. The pointer

$$\dfrac{\begin{array}{c}\vdots\ \mathit{Fig}.\ 4.12\\ \mathbf{emp} \vdash \left[\Pi_g \star \texttt{acc}_{\texttt{nil}}(x)\right]\ {*}\,\mathsf{ghs} := {*}\mathsf{ghs} \cup_{\texttt{nil}} \{x\}\ \left[\Pi_g\right]\end{array}}{\Pi_g \vdash [\texttt{acc}_{\texttt{nil}}(x)]\ {*}\,\mathsf{ghs} := {*}\mathsf{ghs} \cup_{\texttt{nil}} \{x\}\ [\mathbf{emp}]}\ \textsc{Atom}$$

Figure 4.13: Proof tree for ${*}\mathsf{ghs} := {*}\mathsf{ghs} \cup_{\texttt{nil}} \{x\}$.

$$\dfrac{\dfrac{\dfrac{}{\Pi_g \vdash [\forall i \in \mathsf{ihs} \cup_{\mathtt{nil}} \{c\}.\, \mathtt{inv}(i)]\ \ \mathsf{ihs} := \mathsf{ihs} \cup_{\mathtt{nil}} \{c\}\ \ [\forall i \in \mathsf{ihs}.\, \mathtt{inv}(i)]}\,\textsc{Assign}}{\Pi_g \vdash [(\forall i \in \mathsf{ihs}.\, \mathtt{inv}(i)) \star \mathtt{inv}_{\mathtt{nil}}(c)]\ \ \mathsf{ihs} := \mathsf{ihs} \cup_{\mathtt{nil}} \{c\}\ \ [\forall i \in \mathsf{ihs}.\, \mathtt{inv}(i)]}\,\textsc{Conseq}}{\Pi_g \vdash [\Pi_l \star \mathtt{inv}_{\mathtt{nil}}(c)]\ \ \mathsf{ihs} := \mathsf{ihs} \cup_{\mathtt{nil}} \{c\}\ \ [\Pi_l]}\,\textsc{Frame}$$

Figure 4.14: Proof tree for $\mathsf{ihs} := \mathsf{ihs} \cup_{\mathtt{nil}} \{c\}$, where $\mathtt{inv}_{\mathtt{nil}}(c) \triangleq\ c \neq \mathtt{nil} \implies \mathtt{inv}(c)$.

$$\dfrac{\dfrac{e \in \mathsf{lhs} \qquad \dfrac{\dfrac{}{\Pi_g \vdash [\forall l \in \mathsf{lhs} \setminus \{e\}.\, \mathtt{acc}(l)]\ \ \mathsf{lhs} := \mathsf{lhs} \setminus \{e\}\ \ [\forall l \in \mathsf{lhs}.\, \mathtt{acc}(l)]}\,\textsc{Assign}}{\Pi_g \vdash [(\forall l \in \mathsf{lhs} \setminus \{e\}.\, \mathtt{acc}(l)) \star \mathtt{acc}(e)]\ \ \mathsf{lhs} := \mathsf{lhs} \setminus \{e\}\ \ [(\forall l \in \mathsf{lhs}.\, \mathtt{acc}(l)) \star \mathtt{acc}(e)]}\,\textsc{Frame}}{\Pi_g \vdash [\forall l \in \mathsf{lhs}.\, \mathtt{acc}(l)]\ \ \mathsf{lhs} := \mathsf{lhs} \setminus \{e\}\ \ [(\forall l \in \mathsf{lhs}.\, \mathtt{acc}(l)) \star \mathtt{acc}(e)]}\,\textsc{Conseq}}{\Pi_g \vdash [\Pi_l]\ \ \mathsf{lhs} := \mathsf{lhs} \setminus \{e\}\ \ [\Pi_l \star \mathtt{acc}(e)]}\,\textsc{Frame}$$

Figure 4.15: Proof tree for $\mathsf{lhs} := \mathsf{lhs} \setminus \{e\}$ if $e \in \mathsf{lhs}$.

$$\dfrac{\dfrac{e \in \mathsf{lhs} \lor e = \mathtt{nil} \qquad \dfrac{\dfrac{}{\Pi_g \vdash [\forall l \in \mathsf{lhs} \setminus \{e\}.\, \mathtt{acc}(l)]\ \ \mathsf{lhs} := \mathsf{lhs} \setminus \{e\}\ \ [\forall l \in \mathsf{lhs}.\, \mathtt{acc}(l)]}\,\textsc{Assign}}{\Pi_g \vdash [(\forall l \in \mathsf{lhs} \setminus \{e\}.\, \mathtt{acc}(l)) \star \mathtt{acc}_{\mathtt{nil}}(e)]\ \ \mathsf{lhs} := \mathsf{lhs} \setminus \{e\}\ \ [(\forall l \in \mathsf{lhs}.\, \mathtt{acc}(l)) \star \mathtt{acc}_{\mathtt{nil}}(e)]}\,\textsc{Frame}}{\Pi_g \vdash [\forall l \in \mathsf{lhs}.\, \mathtt{acc}(l)]\ \ \mathsf{lhs} := \mathsf{lhs} \setminus \{e\}\ \ [(\forall l \in \mathsf{lhs}.\, \mathtt{acc}(l)) \star \mathtt{acc}_{\mathtt{nil}}(e)]}\,\textsc{Conseq}}{\Pi_g \vdash [\Pi_l]\ \ \mathsf{lhs} := \mathsf{lhs} \setminus \{e\}\ \ [\Pi_l \star \mathtt{acc}_{\mathtt{nil}}(e)]}\,\textsc{Frame}$$

Figure 4.16: Alternative proof tree for $\mathsf{lhs} := \mathsf{lhs} \setminus \{e\}$ that permits e being $\mathtt{nil}$.

$$\dfrac{\dfrac{e \in *\mathsf{ghs} \qquad \dfrac{\dfrac{}{\mathbf{emp} \vdash [\mathtt{acc}(\mathsf{ghs}) \star *\mathsf{ghs} = v]\ \ *\mathsf{ghs} := *\mathsf{ghs} \setminus \{e\}\ \ [\mathtt{acc}(\mathsf{ghs}) \star *\mathsf{ghs} = v \setminus \{e\}]}\,\textsc{Write}}{\mathbf{emp} \vdash [\mathtt{acc}(\mathsf{ghs}) \star *\mathsf{ghs} = v \star R]\ \ *\mathsf{ghs} := *\mathsf{ghs} \setminus \{e\}\ \ [\mathtt{acc}(\mathsf{ghs}) \star *\mathsf{ghs} = v \setminus \{e\} \star R]}\,\textsc{Frame}}{\mathbf{emp} \vdash \big[\mathtt{acc}(\mathsf{ghs}) \star \forall g \in *\mathsf{ghs}.\, \mathtt{acc}(g)\big]\ \ *\mathsf{ghs} := *\mathsf{ghs} \setminus \{e\}\ \ \big[\mathtt{acc}(\mathsf{ghs}) \star (\forall g \in *\mathsf{ghs}.\, \mathtt{acc}(g)) \star \mathtt{acc}(e)\big]}\,\textsc{Conseq}}{\mathbf{emp} \vdash \big[\Pi_g\big]\ \ *\mathsf{ghs} := *\mathsf{ghs} \setminus \{e\}\ \ \big[\Pi_g \star \mathtt{acc}(e)\big]}\,\textsc{Frame}$$

$$\text{with}\quad R \triangleq\ (\forall g \in (v \setminus \{e\}).\, \mathtt{acc}(g)) \star \mathtt{acc}(e)$$

Figure 4.17: Proof tree for $*\mathsf{ghs} := *\mathsf{ghs} \setminus \{e\}$ that requires Π_g to be local.

$$\dfrac{\dfrac{c \in \mathsf{ihs} \lor c = \mathtt{nil} \qquad \dfrac{\dfrac{}{\Pi_g \vdash [\forall i \in \mathsf{ihs} \setminus \{c\}.\, \mathtt{inv}(l)]\ \ \mathsf{ihs} := \mathsf{ihs} \setminus \{c\}\ \ [\forall i \in \mathsf{ihs}.\, \mathtt{inv}(i)]}\,\textsc{Assign}}{\Pi_g \vdash [(\forall i \in \mathsf{ihs} \setminus \{c\}.\, \mathtt{inv}(l)) \star \mathtt{inv}_{\mathtt{nil}}(c)]\ \ \mathsf{ihs} := \mathsf{ihs} \setminus \{c\}\ \ [(\forall i \in \mathsf{ihs}.\, \mathtt{inv}(i)) \star \mathtt{inv}_{\mathtt{nil}}(c)]}\,\textsc{Frame}}{\Pi_g \vdash [\forall i \in \mathsf{ihs}.\, \mathtt{inv}(i)]\ \ \mathsf{ihs} := \mathsf{ihs} \setminus \{c\}\ \ [(\forall i \in \mathsf{ihs}.\, \mathtt{inv}(i)) \star \mathtt{inv}_{\mathtt{nil}}(c)]}\,\textsc{Conseq}}{\Pi_g \vdash [\Pi_l]\ \ \mathsf{ihs} := \mathsf{ihs} \setminus \{c\}\ \ [\Pi_l \star \mathtt{inv}_{\mathtt{nil}}(c)]}\,\textsc{Frame}$$

Figure 4.18: Proof tree for $\mathsf{ihs} := \mathsf{ihs} \setminus \{c\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg(*\mathsf{used}) \qquad \dfrac{\dfrac{}{\mathbf{emp} \vdash [\mathtt{acc}(\mathsf{used})]\ \ *\mathsf{used} := \mathsf{true}\ \ [\mathtt{acc}(\mathsf{used}) \star *\mathsf{used} = \mathit{true}]}\,\textsc{Write}}{\mathbf{emp} \vdash \big[\mathtt{acc}(\mathsf{used}) \star \phi\big]\ \ *\mathsf{used} := \mathsf{true}\ \ \big[\mathtt{acc}(\mathsf{used}) \star *\mathsf{used} = \mathit{true} \star \phi\big]}\,\textsc{Frame}}{\mathbf{emp} \vdash \big[\mathtt{acc}(\mathsf{used}) \star \neg(*\mathsf{used}) \implies \phi\big]\ \ *\mathsf{used} := \mathsf{true}\ \ \big[\mathtt{acc}(\mathsf{used}) \star *\mathsf{used} = \mathit{true} \star \phi\big]}\,\textsc{Conseq}}{\mathbf{emp} \vdash \big[\mathtt{acc}(\mathsf{used}) \star \neg(*\mathsf{used}) \implies \phi\big]\ \ *\mathsf{used} := \mathsf{true}\ \ \big[\mathtt{acc}(\mathsf{used}) \star (\neg(*\mathsf{used}) \implies \phi) \star \phi\big]}\,\textsc{Conseq}}{\mathbf{emp} \vdash \big[\Pi_g\big]\ \ *\mathsf{used} := \mathsf{true}\ \ \big[\Pi_g \star \phi\big]}\,\textsc{Frame}}{\Pi_g \vdash [\mathbf{emp}]\ \ \mathbf{atomic}\ \{*\mathsf{used} := \mathsf{true}\}\ \ \big[\phi\big]}\,\textsc{Atom}}{\Pi_g \vdash [\Pi_l]\ \ \mathbf{atomic}\ \{*\mathsf{used} := \mathsf{true}\}\ \ \big[\Pi_l \star \phi\big]}\,\textsc{Frame}$$

Figure 4.19: Proof tree for **atomic** $\{*\mathsf{used} := \mathsf{true}\}$.

$$\dfrac{\dfrac{\dfrac{}{\Pi_g \vdash [\mathbf{emp}]\ x := \mathbf{new}()\ [\mathtt{acc}(x)]}\textsc{Alloc}}{\Pi_g \vdash [\Pi_l]\ x := \mathbf{new}()\ [\Pi_l \star \mathtt{acc}(x)]}\textsc{Frame} \quad \dfrac{\dfrac{\vdots\ \mathit{Fig}.\,4.11}{\Pi_g \vdash [\Pi_l \star \mathtt{acc}_{\mathtt{nil}}(x)]\ \mathsf{lhs} := \mathsf{lhs} \cup_{\mathtt{nil}} \{x\}\ [\Pi_l]}}{\Pi_g \vdash [\Pi_l \star \mathtt{acc}(x)]\ \mathsf{lhs} := \mathsf{lhs} \cup_{\mathtt{nil}} \{x\}\ [\Pi_l]}\textsc{Conseq}}{\Pi_g \vdash [\Pi_l]\ x := \mathbf{new}();\ \mathsf{lhs} := \mathsf{lhs} \cup_{\mathtt{nil}} \{x\}\ [\Pi_l]}\textsc{Seq}$$

Figure 4.20: Proof tree for $\mathbb{A}(x := \mathbf{new}())$.

$$\dfrac{\dfrac{\vdots\ \mathit{Fig}.\,4.15}{\Pi_g \vdash [\Pi_l]\ s_1\ [\Pi_l \star \mathtt{acc}(e)]} \quad \dfrac{\dfrac{}{\Pi_g \vdash [\mathtt{acc}(e)]\ s_2\ [\mathtt{acc}(e)]}\textsc{Read}}{\Pi_g \vdash [\Pi_l \star \mathtt{acc}(e)]\ s_2\ [\Pi_l \star \mathtt{acc}(e)]}\textsc{Frame} \quad \dfrac{\dfrac{\vdots\ \mathit{Fig}.\,4.11}{\Pi_g \vdash [\Pi_l \star \mathtt{acc}_{\mathtt{nil}}(e)]\ s_3\ [\Pi_l]}}{\Pi_g \vdash [\Pi_l \star \mathtt{acc}(e)]\ s_3\ [\Pi_l]}\textsc{Conseq}}{\Pi_g \vdash [\Pi_l]\ s_1; s_2; s_3\ [\Pi_l]}\textsc{Seq}^*$$

with $s_1 \triangleq \mathsf{lhs} := \mathsf{lhs} \setminus \{e\}$ $\qquad s_2 \triangleq x := *e$ $\qquad s_3 \triangleq \mathsf{lhs} := \mathsf{lhs} \cup_{\mathtt{nil}} \{e\}$

Figure 4.21: Proof tree for $\mathbb{A}(x := *e)$ if $e \in \mathsf{lhs}$, where Seq* represents repeated application of the Seq rule. We discharge the side condition from Fig. 4.15 as $e \in \mathsf{lhs}$ holds by definition.

$$\dfrac{\dfrac{\dfrac{\dfrac{\boxed{e \in *\mathsf{ghs}}}{\vdots\ \mathit{Fig}.\,4.17}}{\mathbf{emp} \vdash [\Pi_g]\ s_1\ [\Pi_g \star \mathtt{acc}(e)]} \quad \dfrac{\dfrac{}{\mathbf{emp} \vdash [\mathtt{acc}(e)]\ s_2\ [\mathtt{acc}(e)]}\textsc{Read}}{\mathbf{emp} \vdash [\Pi_g \star \mathtt{acc}(e)]\ s_2\ [\Pi_g \star \mathtt{acc}(e)]}\textsc{Frame} \quad \dfrac{\dfrac{\vdots\ \mathit{Fig}.\,4.12}{\mathbf{emp} \vdash [\Pi_g \star \mathtt{acc}_{\mathtt{nil}}(e)]\ s_3\ [\Pi_g]}}{\mathbf{emp} \vdash [\Pi_g \star \mathtt{acc}(e)]\ s_3\ [\Pi_g]}\mathit{Fig}.\,4.12}{\mathbf{emp} \vdash [\Pi_g]\ s_1; s_2; s_3\ [\Pi_g]}\textsc{Seq}^*}{\dfrac{\Pi_g \vdash [\mathbf{emp}]\ \mathbf{atomic}\ \{s_1; s_2; s_3\}\ [\mathbf{emp}]}{\Pi_g \vdash [\Pi_l]\ \mathbf{atomic}\ \{s_1; s_2; s_3\}\ [\Pi_l]}\textsc{Frame}}\textsc{Atom}$$

with $s_1 \triangleq *\mathsf{ghs} := *\mathsf{ghs} \setminus \{e\}$ $\qquad s_2 \triangleq x := *e$ $\qquad s_3 \triangleq *\mathsf{ghs} := *\mathsf{ghs} \cup_{\mathtt{nil}} \{e\}$

Figure 4.22: Proof tree for $\mathbb{A}(x := *e)$ if $e \notin \mathsf{lhs}$.

$$\dfrac{\boxed{\mathit{Asm}.\,4.4.4} \quad \dfrac{\dfrac{\vdots\ \text{auto-active verification}}{\Pi_g \vdash [P_{\mathit{CoreAlloc}}(\bar{e})]\ c := \mathbf{CoreAlloc}(\bar{e})\ [Q_{\mathit{CoreAlloc}}(c, \bar{e})]}}{\Pi_g \vdash [P_{\mathit{CoreAlloc}}(\bar{e}) \star F]\ c := \mathbf{CoreAlloc}(\bar{e})\ [Q_{\mathit{CoreAlloc}}(c, \bar{e}) \star F]}\textsc{Frame}}{\dfrac{\Pi_g \vdash [\mathtt{acc}_{\mathtt{nil}}(\bar{e}) \star \phi]\ c := \mathbf{CoreAlloc}(\bar{e})\ [\mathtt{acc}_{\mathtt{nil}}(\bar{e}) \star \mathtt{inv}(c)]}{\Pi_g \vdash [\Pi_l \star \mathtt{acc}_{\mathtt{nil}}(\bar{e}) \star \phi]\ c := \mathbf{CoreAlloc}(\bar{e})\ [\Pi_l \star \mathtt{acc}_{\mathtt{nil}}(\bar{e}) \star \mathtt{inv}(c)]}\textsc{Frame}}\textsc{Conseq}$$

Figure 4.23: Proof tree for $c := \mathbf{CoreAlloc}(\bar{e})$ using the subproof that we extract from the auto-active program verifier. The side condition (Asm. 4.4.4) states that $P_{\mathit{CoreAlloc}}(\bar{e}) = \phi \star R$ and $\mathtt{acc}_{\mathtt{nil}}(\bar{e}) \models R$. We call F the permissions that are framed around, i.e., $\mathtt{acc}_{\mathtt{nil}}(\bar{e}) = R \star F$. The side condition further specifies that $Q_{\mathit{CoreAlloc}}(c, \bar{e}) = \mathtt{inv}(c) \star R'$ and $R' \models R$ hold, allowing us to derive $R' \star F \models \mathtt{acc}_{\mathtt{nil}}(\bar{e})$. Thus, we can apply the Conseq rule. We abuse the notation $\mathtt{acc}_{\mathtt{nil}}(\bar{e})$ to denote the iterated separating conjunction expressing $\mathtt{acc}_{\mathtt{nil}}(e)$ for each element e in $\bar{e}$, i.e., $\forall i.\, 0 \leq i < \mathit{len}(\bar{e}) \implies \mathtt{acc}_{\mathtt{nil}}(\bar{e}[i])$, where $\mathit{len}(\bar{e})$ and $\bar{e}[i]$ return the length and the i-th element of the list $\bar{e}$, respectively.

$$\dfrac{\dfrac{\dfrac{\boxed{\neg(*\mathsf{used})}}{\vdots\ \mathit{Fig}.\,4.19}}{\Pi_g \vdash [\Pi_l]\ s_1\ [\Pi_l \star \phi]} \quad \dfrac{\dfrac{\dfrac{\boxed{(\mathit{set}(\bar{e}) \setminus \mathtt{nil}) \subseteq \mathsf{lhs} \land \mathit{disjoint}(\bar{e})}}{\vdots\ \mathit{Fig}.\,4.16}}{\Pi_g \vdash [\Pi_l]\ s_2\ [R_2']}}{\Pi_g \vdash [\Pi_l \star \phi]\ s_2\ [R_2]}\textsc{Frame} \quad \dfrac{\dfrac{\boxed{\mathit{Asm}.\,4.4.4}}{\vdots\ \mathit{Fig}.\,4.23}}{\Pi_g \vdash [R_2]\ s_3\ [R_3]} \quad \dfrac{\dfrac{\vdots\ \mathit{Fig}.\,4.11}{\Pi_g \vdash [R_2']\ s_4\ [\Pi_l]}}{\Pi_g \vdash [R_3]\ s_4\ [R_4]}\textsc{Frame} \quad \dfrac{\dfrac{\vdots\ \mathit{Fig}.\,4.14}{\Pi_g \vdash [R_4']\ s_5\ [\Pi_l]}}{\Pi_g \vdash [R_4]\ s_5\ [\Pi_l]}\textsc{Seq}}{\Pi_g \vdash [\Pi_l]\ s_1; s_2; s_3; s_4; s_5\ [\Pi_l]}\textsc{Seq}^*$$

with

$s_1 \triangleq \mathbf{atomic}\ \{*\mathsf{used} := \mathsf{true}\}$ $\quad s_2 \triangleq \mathsf{lhs} := \mathsf{lhs} \setminus \bar{e}$ $\quad s_3 \triangleq c := \mathbf{CoreAlloc}(\bar{e})$ $\quad s_4 \triangleq \mathsf{lhs} := \mathsf{lhs} \cup_{\mathtt{nil}} \bar{e}$ $\quad s_5 \triangleq \mathsf{ihs} := \mathsf{ihs} \cup_{\mathtt{nil}} \{c\}$

$R_2' \triangleq \Pi_l \star \mathtt{acc}_{\mathtt{nil}}(\bar{e})$ $\quad R_2 \triangleq R_2' \star \phi$ $\quad R_3 \triangleq R_2' \star \mathtt{inv}(c)$ $\quad R_4 \triangleq \Pi_l \star \mathtt{inv}(c)$ $\quad R_4' \triangleq \Pi_l \star \mathtt{inv}_{\mathtt{nil}}(c)$

Figure 4.24: Proof tree for $\mathbb{A}(c := \mathbf{CoreAlloc}(\bar{e}))$. We naturally extend Fig. 4.11 and Fig. 4.15 to adding and removing *lists* of heap locations to and from the ghost set lhs, respectively. The latter requires their disjointness.

$$\dfrac{\boxed{Asm.\ 4.4.4} \quad \dfrac{\dfrac{\vdots\ \text{auto-active verification}}{\Pi_g \vdash \left[P_{CoreApi\_k}(c,\bar{e})\right]\ \bar{r} := \textbf{CoreApi\_k}(c,\bar{e})\ \left[Q_{CoreApi\_k}(c,\bar{e},\bar{r})\right]}\ \textsc{Frame}}{\Pi_g \vdash \left[P_{CoreApi\_k}(c,\bar{e}) \star F\right]\ \bar{r} := \textbf{CoreApi\_k}(c,\bar{e})\ \left[Q_{CoreApi\_k}(c,\bar{e},\bar{r}) \star F\right]}\ \textsc{Conseq}}{\dfrac{\Pi_g \vdash [\mathtt{inv_{nil}}(c) \star \mathtt{acc_{nil}}(\bar{e})]\ \bar{r} := \textbf{CoreApi\_k}(c,\bar{e})\ [\mathtt{inv_{nil}}(c) \star \mathtt{acc_{nil}}(\bar{e}) \star \mathtt{acc_{nil}}(\bar{r})]}{\Pi_g \vdash [\Pi_l \star \mathtt{inv_{nil}}(c) \star \mathtt{acc_{nil}}(\bar{e})]\ \bar{r} := \textbf{CoreApi\_k}(c,\bar{e})\ [\Pi_l \star \mathtt{inv_{nil}}(c) \star \mathtt{acc_{nil}}(\bar{e}) \star \mathtt{acc_{nil}}(\bar{r})]}\ \textsc{Frame}}$$

Figure 4.25: Proof tree for $\bar{r} := \textbf{CoreApi\_k}(c,\bar{e})$.

$$\dfrac{\dfrac{\boxed{c \in \mathsf{ihs} \lor c = \mathtt{nil}} \quad \vdots\ Fig.\ 4.18}{\Pi_g \vdash [\Pi_l]\ s_1\ [R_1]} \quad \dfrac{\boxed{\begin{matrix}(set(\bar{e}) \setminus \mathtt{nil}) \subseteq \mathsf{lhs} \land \\ disjoint(\bar{e})\end{matrix}} \quad \vdots\ Fig.\ 4.16}{\dfrac{\Pi_g \vdash [\Pi_l]\ s_2\ [R_2']}{\Pi_g \vdash [R_1]\ s_2\ [R_2]}\ \textsc{Frame}} \quad \dfrac{\boxed{Asm.\ 4.4.4} \quad \vdots\ Fig.\ 4.25}{\Pi_g \vdash [R_2]\ s_3\ [R_3]} \quad \dfrac{\vdots\ Fig.\ 4.11}{\dfrac{\Pi_g \vdash [R_3']\ s_4\ [\Pi_l]}{\Pi_g \vdash [R_3]\ s_4\ [R_1]}\ \textsc{Frame}} \quad \dfrac{\vdots\ Fig.\ 4.14}{\Pi_g \vdash [R_1]\ s_5\ [\Pi_l]}}{\Pi_g \vdash [\Pi_l]\ s_1;\ s_2;\ s_3;\ s_4;\ s_5\ [\Pi_l]}\ \textsc{Seq}^*$$

with

$$\begin{array}{lllll}
s_1 \triangleq \mathsf{ihs} := \mathsf{ihs} \setminus \{c\} & s_2 \triangleq \mathsf{lhs} := \mathsf{lhs} \setminus \bar{e} & s_3 \triangleq \bar{r} := \textbf{CoreApi\_k}(c,\bar{e}) & s_4 \triangleq \mathsf{lhs} := \mathsf{lhs} \cup_{\mathtt{nil}} \bar{e} \cup_{\mathtt{nil}} \bar{r} & s_5 \triangleq \mathsf{ihs} := \mathsf{ihs} \cup_{\mathtt{nil}} \{c\} \\
R_1 \triangleq \Pi_l \star \mathtt{inv_{nil}}(c) & R_2' \triangleq \Pi_l \star \mathtt{acc_{nil}}(\bar{e}) & R_2 \triangleq R_2' \star \mathtt{inv_{nil}}(c) & R_3' \triangleq R_2' \star \mathtt{acc_{nil}}(\bar{r}) & R_3 \triangleq R_3' \star \mathtt{inv_{nil}}(c)
\end{array}$$

Figure 4.26: Proof tree for $\mathbb{A}(\bar{r} := \textbf{CoreApi\_k}(c,\bar{e}))$.

analysis we are using is context insensitive, i.e., ignores control flow and ordering of statements. Thus, we omit the program location at which such a judgement holds as it holds for all program locations within a given codebase. If necessary, we could employ a context-sensitive pointer analysis to increase precision.

To formalize what the pointer analysis computes, let us first state several definitions before stating the pointer analysis' soundness, which we assume.

Definition 4.4.11 (Reachability) *$reach_\tau^p(x)$ returns the set of addresses for all heap locations that are transitively reachable from variable x at program point p on trace τ. Hence, $\forall x, \tau, p.\ val_\tau(x) \in reach_\tau^p(x)$ holds for all program points p after x is defined.*

Definition 4.4.12 (Allocation Site) *$as_\tau(h)$ returns the allocation site for a heap location h on trace τ, which is the label of the statement that allocated this heap location.*

Assumption 4.4.5 (Soundness of Pointer Analysis) *The pointer analysis computes for a variable x the heap locations $pts(x)$ to which x* may *point on all possible traces. These heap locations are identified by their allocation*

$$\dfrac{\dfrac{}{\Pi_g \vdash [\mathbf{emp}]\ \mathsf{lhs} := \emptyset\ [\mathsf{lhs} = \emptyset]}\ \textsc{Assign} \quad \dfrac{\dfrac{\dfrac{}{\Pi_g \vdash [\mathbf{emp}]\ \mathsf{ihs} := \emptyset\ [\mathsf{ihs} = \emptyset]}\ \textsc{Assign}}{\Pi_g \vdash [\mathsf{lhs} = \emptyset]\ \mathsf{ihs} := \emptyset\ [\mathsf{lhs} = \emptyset \star \mathsf{ihs} = \emptyset]}\ \textsc{Frame}}{\Pi_g \vdash [\mathsf{lhs} = \emptyset]\ \mathsf{ihs} := \emptyset\ [\Pi_l]}\ \textsc{Conseq} \quad \dfrac{\vdots\ \text{IH}}{\Pi_g \vdash [\Pi_l]\ \mathbb{A}(s)\ [\Pi_l]}}{\Pi_g \vdash [\mathbf{emp}]\ \mathsf{lhs} := \emptyset;\ \mathsf{ihs} := \emptyset;\ \mathbb{A}(s)\ [\Pi_l]}\ \textsc{Seq}^*$$

Figure 4.27: Proof tree for the sequence of statements that is executed as the newly spawned thread, where s represents an arbitrary input statement and IH denotes an application of the induction hypothesis. We omit trivial applications of the Conseq rule.

$$\dfrac{\dfrac{\vdots\, \mathit{Fig.}\ 4.15}{\Pi_g \vdash [\Pi_l]\ s_1\ [\Pi_l \star R]} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\vdots\, \mathit{Fig.}\ 4.13}{\Pi_g \vdash [R']\ s_2\ [\mathbf{emp}]}}{\Pi_g \vdash [R]\ s_2\ [\mathbf{emp}]}\,\textsc{Conseq} \qquad \dfrac{\dfrac{\vdots\, \mathit{Fig.}\ 4.27}{\Pi_g \vdash [\mathbf{emp}]\ \mathsf{lhs} := \emptyset;\ \mathsf{ihs} := \emptyset;\ \mathbb{A}(s)\ [\Pi_l]}}{\Pi_g \vdash [\mathbf{emp}]\ \mathbf{fork}\ (\bar{x})\ \{\mathsf{lhs} := \emptyset;\ \mathsf{ihs} := \emptyset;\ \mathbb{A}(s)\}\ [\mathbf{emp}]}\,\textsc{Fork}}{\Pi_g \vdash [R]\ s_2;\ \mathbf{fork}\ (\bar{x})\ \{\mathsf{lhs} := \emptyset;\ \mathsf{ihs} := \emptyset;\ \mathbb{A}(s)\}\ [\mathbf{emp}]}\,\textsc{Seq}}{\Pi_g \vdash [\Pi_l \star R]\ s_2;\ \mathbf{fork}\ (\bar{x})\ \{\mathsf{lhs} := \emptyset;\ \mathsf{ihs} := \emptyset;\ \mathbb{A}(s)\}\ [\Pi_l]}\,\textsc{Frame}}{\Pi_g \vdash [\Pi_l]\ s_1;\ s_2;\ \mathbf{fork}\ (\bar{x})\ \{\mathsf{lhs} := \emptyset;\ \mathsf{ihs} := \emptyset;\ \mathbb{A}(s)\}\ [\Pi_l]}\,\textsc{Seq}$$

with $s_1 \triangleq \mathsf{lhs} := \mathsf{lhs} \setminus (\mathit{reach}(\bar{x}) \cap \mathsf{lhs})$ $\qquad s_2 \triangleq *\mathsf{ghs} := *\mathsf{ghs} \cup_{\mathtt{nil}} (\mathit{reach}(\bar{x}) \cap \mathsf{lhs})$

$R \triangleq \forall l \in (\mathit{reach}(\bar{x}) \cap \mathsf{lhs}).\ \mathtt{acc}(l)$ $\qquad R' \triangleq \forall l \in (\mathit{reach}(\bar{x}) \cap \mathsf{lhs}).\ \mathtt{acc}_{\mathtt{nil}}(l)$

Figure 4.28: Proof tree for $\mathbb{A}(\mathbf{fork}\ (\bar{x})\ \{s\})$ that assumes the existence of a Fork rule. The side conditions stemming from Fig. 4.13 hold trivially as $\mathit{reach}(\bar{x}) \cap \mathsf{lhs} \subseteq \mathsf{lhs}$ and since $\mathit{reach}(\bar{x})$ returns a set of heap locations, which is by definition free of duplicates and, thus, its elements are pairwise disjoint.

site. We assume that the pointer analysis is sound, i.e., computes an over-approximation of the heap locations to which x actually points when looking at concrete traces.

$$\forall x, \tau.\ \mathit{val}_\tau(x) \neq \mathtt{nil} \implies \mathit{as}_\tau(\mathit{val}_\tau(x)) \in \mathit{pts}(x)$$

Lemma 4.4.4 (Disjointness from Pointer Analysis) *We can use the pointer analysis' may-point-to judgments to derive disjointness.*

$$\forall x, y. \mathit{pts}(x) \cap \mathit{pts}(y) = \emptyset \implies \mathit{disjoint}(\{x, y\})$$

Proof sketch. If x or y store the value `nil`, $\mathit{disjoint}(\{x, y\})$ holds. Otherwise, x and y are non-`nil`. We apply Asm. 4.4.5 to our premise and obtain $\forall \tau.\ \mathit{as}_\tau(\mathit{val}_\tau(x)) \neq \mathit{as}_\tau(\mathit{val}_\tau(y))$. Since x and y point on all possible traces to heap locations that were allocated at different allocation sites, the heap locations themselves must be different, i.e., $\mathit{val}_\tau(x) \neq \mathit{val}_\tau(y)$. ☐

Pass-Through Analysis. As hinted at by our ghost sets, we distinguish two types of heap locations, namely heap locations that make up Core instances and heap locations that the Application might access. Heap locations of the former type are tracked by collecting the respective Core instances in lhs. The latter type encompasses heap locations that are either allocated within the Application by **new**() or allocated within the Core and returned from a Core API call.

To distinguish these types of heap locations, we run a pass-through analysis that provides the judgments $\mathit{pt}^p_{\textsc{Core}}(a, \tau)$ and $\mathit{pt}^p_{\mathrm{ret}}(a, \tau)$ denoting that a heap location allocated at allocation site a passed through (pt) the return argument c of a $c :=$ **CoreAlloc**$(\bar{e})$ statement and through one of the return arguments $\bar{r}$ of a $\bar{r} :=$ **CoreApi_k**$(c, \bar{e})$ statement, respectively, between label a and program point p on trace τ. I.e., we have that $\mathit{pt}^p_{\textsc{Core}}(\mathit{as}_\tau(\mathit{val}_\tau(c)), \tau)$ and $\forall r \in \mathit{set}(\bar{r}).\ \mathit{pt}^p_{\mathrm{ret}}(\mathit{as}_\tau(\mathit{val}_\tau(r)), \tau)$ hold at the program point p on trace τ after executing the statement $c :=$ **CoreAlloc**$(\bar{e})$ and $\bar{r} :=$ **CoreApi_k**$(c, \bar{e})$, respectively.

Definition 4.4.13 (Application-Managed Heap Locations) *We call a heap location h* Application-managed *at program point p on trace τ if h is either allocated within the* Application *or has been returned from a*

$\bar{r} :=$ **CoreApi_k**$(c, \bar{e})$ *statement.*

$$am^p_\tau(h) \triangleq \textit{is-app}(as_\tau(h)) \lor pt^p_{ret}(as_\tau(h), \tau)$$

Escape Analysis. The goal of the escape analysis is to correctly place heap locations into lhs, *ghs, and ihs. In particular, we want to establish globally that an Application-managed heap location and a Core instance are in lhs and ihs, respectively, if they are *local*.

We first define what it means for a heap location to be local (cf. Def. 4.4.15). I.e., this definition takes all threads into account and states that a heap location h is local to a thread t if and only if t is the only thread that can potentially access h.

Locality of a heap location is approximated by our escape analysis. The result of the escape analysis is formalized in a judgement $local^p(x)$ for some variable x and program point p. The intuition is that a variable that is local points to heap locations (i.e., $*x$) that are accessible *only* by the current thread and, thus, can be modified or even referred to only by the current thread. The escape analysis is sound in that no heap location that is accessible by another thread will ever be reported as local (cf. Asm. 4.4.6), but potentially imprecise in that some locations that are not accessible by other threads will fail to be local.

Definition 4.4.14 (Accessibility) *We write $accessible^p_t(h)$ to denote that heap location h is* accessible *by thread t at program point p. A thread may access such a heap location either directly via variables or indirectly by dereferencing other heap locations. We define accessibility independently of variables and, thus, accessibility of h does not change when variables go out of scope. Instead, accessibility is monotonic for a thread's execution.*

Definition 4.4.15 (Locality) *A heap location h is* local *at program point p if it is accessible by a single thread t.*

$$\begin{aligned} localhl^p_t(h) \triangleq\ & accessible^p_t(h) \land \\ & (\forall t'. t' \neq t \implies \neg accessible^p_{t'}(h)) \end{aligned}$$

Lemma 4.4.5 (Uniqueness of Locality) *The thread t having access to a* local *heap location h is unique, i.e.,*

$$\forall h, t, t', p. localhl^p_t(h) \land localhl^p_{t'}(h) \implies t = t'.$$

Proof sketch. The lemma follows directly from Def. 4.4.15. □

Lemma 4.4.6 (Locality is Reverse Monotonic) *A* local *heap location h at program point p' must be local at every* earlier *program point p if h is accessible at p, i.e.,*

$$\begin{aligned} \forall h, t, p, p'. p \preceq p' \land accessible^p_t(h) \land localhl^{p'}_t(h) \implies \\ localhl^p_t(h). \end{aligned}$$

Proof sketch. We prove this lemma by contradiction for arbitrary h, t, p, and p'. $\neg localhl^p_t(h)$ implies that h is accessible by another thread t', i.e., $t' \neq t \wedge accessible^{p}_{t'}(h)$. Since accessibility is monotonic, h remains accessible by t' at p' contradicting $localhl^{p'}_{t}(h)$. □

Lemma 4.4.7 (Locality is Reverse Transitive) *If a* local *heap location h' is transitively reachable from another heap location h then h must also be local.*

$$\forall h, h', t, \tau, p. localhl^{p}_{t}(h') \wedge h' \in reach^{p}_{\tau}(h) \implies localhl^{p}_{t}(h)$$

Proof sketch. We prove this lemma by contradiction for arbitrary h, h', t, τ, and p. I.e., a thread t' exists such that h is accessible by t'. h' is accessible by t' via reachability from h, thus, contradicting $localhl^{p}_{t}(h')$. □

Assumption 4.4.6 (Soundness of Escape Analysis) *We assume that the escape analysis is sound, i.e., reports a heap location to which variable x points as being local* only *if the corresponding heap location is indeed local (or x is* `nil`*) for every possible trace τ, i.e.,*

$$\forall x, \tau, p. local^{p}(x) \implies val_{\tau}(x) = \texttt{nil} \vee \exists t.\, localhl^{p}_{t}(val_{\tau}(x)).$$

Based on these definitions and the soundness of our analyses, we prove several lemmata that relate accessible heap locations to our ghost sets and corollaries that lift these properties to variables and the judgments we obtain from our static analyses. We will later use these corollaries to show that these judgments discharge our proof rules' side conditions ω.

Lemma 4.4.8 (Inaccessability Implies Set Absence) *All heap locations stored in the ghost sets are accessible by at least one thread.*

$$\forall h, \tau, p. h \neq \texttt{nil} \wedge p \in \tau \wedge (\forall t. \neg accessible^{p}_{t}(h)) \implies \forall t.\, h \notin (\mathsf{lhs}_t \cup {*}\mathsf{ghs} \cup \mathsf{ihs}_t)^{p}$$

where e^{p} denotes evaluating expression e at program point p.

Proof sketch. We prove this lemma by induction over program traces. The base case for the empty trace holds trivially as lhs and ihs for every thread t and ${*}\mathsf{ghs}$ are initialized to the empty set. In the inductive step, we prove this lemma for an arbitrary heap location h', program point p', and trace τ. We assume the premise and apply the induction hypothesis for the immediately preceding program point p as $\forall t. \neg accessible^{p'}_{t}(h')$ implies $\forall t. \neg accessible^{p}_{t}(h')$ due to monotonicity. We show that $\forall t. h' \notin (\mathsf{lhs}_t \cup {*}\mathsf{ghs} \cup \mathsf{ihs}_t)^{p'}$ holds by analyzing the ghost operations that $\mathbb{A}$ inserts for a statement s. We assume without loss of generality that thread t_s executes $\mathbb{A}(s)$, which transitions from p to p'. We observe that every element that is added to lhs_{t_s}, ${*}\mathsf{ghs}$ or ihs_t is either the heap location to which a variable accessible by t_s points or a set of heap locations that are reachable from such a variable. Since h' by assumption is not accessible from any thread at p', $\mathbb{A}$ does not add h' to any ghost set. □

The next lemmata depend on certain requirements for a codebase, which we define next. As we will see, successfully executing the static analyses implies that a codebase meets these requirements.

$$
\begin{aligned}
r^p_\tau \triangleq\ & (\forall s, x, e, \ell.\, s^\ell = {*x} := e \land \ell \prec p \implies am^{\text{pre-}\ell}_\tau(val_\tau(x))) \land \\
& (\forall s, c, e, \bar{e}, \ell.\, s^\ell = c := \mathbf{CoreAlloc}(\bar{e}) \land \ell \prec p \land e \in set(\bar{e}) \implies \\
& \qquad val_\tau(e) = \texttt{nil} \lor am^{\text{pre-}\ell}_\tau(val_\tau(e)) \land local^{\text{post-}\ell}(e)) \land \\
& (\forall s, k, c, e, \bar{e}, r, \bar{r}, \ell.\, s^\ell = \bar{r} := \mathbf{CoreApi\_k}(c, \bar{e}) \land \ell \prec p \land e \in set(\bar{e}) \land r \in set(\bar{r}) \implies \\
& \qquad (val_\tau(e) = \texttt{nil} \lor am^{\text{pre-}\ell}_\tau(val_\tau(e)) \land local^{\text{post-}\ell}(e)) \land \\
& \qquad (val_\tau(c) = \texttt{nil} \lor \neg am^{\text{pre-}\ell}_\tau(val_\tau(c))) \land \\
& \qquad (val_\tau(r) = \texttt{nil} \lor local^{\text{post-}\ell}(r)))
\end{aligned}
$$

Figure 4.29: r^p_τ expresses requirements that all statements in a codebase before program point p on trace τ must satisfy. These requirements allow us to relate properties of heap locations to containment in the ghost sets. In particular, heap write statements must write to Application-managed heap locations only, arguments that are passed to the Core (i.e., $\bar{e}$ in $\mathbf{CoreAlloc}(\bar{e})$ and $\bar{r} := \mathbf{CoreApi\_k}(c, \bar{e})$ statements) must be Application-managed and local *after* executing the statement unless they are `nil`, the Core instance c must *not* be Application-managed, and return arguments from the Core, i.e., $\bar{r}$ in $\bar{r} := \mathbf{CoreApi\_k}(c, \bar{e})$, must be local or `nil`.

Lemma 4.4.9 (Locality Implies Set Containment for Application-Managed Locations) *An Application-managed heap location h is in thread t's* lhs *at program point p if h is local, and in* $*\mathsf{ghs}$ *if h is accessible by multiple threads. Both cases hold if a codebase meets the requirements r^p_τ (cf. Fig. 4.29).*

$$
\begin{aligned}
\forall h, t, \tau, p.\, (& h \neq \texttt{nil} \land p \in \tau \land accessible^p_t(h) \land \\
& am^p_\tau(h) \land r^p_\tau) \implies \\
& ((\forall t'.\, t' = t \lor \neg accessible^p_{t'}(h)) \iff h \in \mathsf{lhs}^p_t) \land \\
& ((\exists t'.\, t' \neq t \land accessible^p_{t'}(h)) \iff h \in {*\mathsf{ghs}^p})
\end{aligned}
$$

Proof sketch. We prove this lemma by induction over program traces. The base case for the empty trace holds trivially as there are no allocated and, thus, accessible heap locations yet. In the inductive step, we prove this lemma for an arbitrary heap location h', thread t, program point p', and trace τ by applying the induction hypothesis to the immediately preceding program point p and showing that we obtain the specified set containment for p'. I.e., we assume the premise and show that

$$
\begin{aligned}
& ((\forall t'.\, t' = t \lor \neg accessible^{p'}_{t'}(h')) \iff h' \in \mathsf{lhs}^{p'}_t) \land \\
& ((\exists t'.\, t' \neq t \land accessible^{p'}_{t'}(h')) \iff h' \in {*\mathsf{ghs}^{p'}})
\end{aligned}
\tag{4.4}
$$

holds. We case split on statement s (before applying $\mathbb{A}$) such that executing $\mathbb{A}(s)$ on thread t_s transitions from p to p'. We first note that the restrictions r are monotonic when going backwards on a trace, i.e., r^p_τ follows from $r^{p'}_\tau$.

- $s = \mathbf{skip}$: Since **skip** does not allocate any heap locations and leaves accessibility unchanged, we get $accessible^p_t(h')$ and apply the induction hypothesis. Because algorithm $\mathbb{A}$ leaves all ghost sets unmodified, (4.4) holds.
- $s = x := \mathbf{new}()$: If $h' = val_\tau(x)$, then $t = t_s$ as $accessible^{p'}_t(h')$ holds and no other thread can access h' yet. $\mathbb{A}$ adds h' to lhs_t and (4.4) holds as h' is in no other ghost set (by Lemma 4.4.8). Otherwise, h' is already allocated at p, and we apply the induction hypothesis to obtain (4.4) as $\mathbb{A}$ neither adds nor removes h' to and from any ghost set.

- $s = x := *e$: Since s neither allocates new heap locations nor changes accessibility of h', $\mathit{accessible}^p_t(h')$ holds, and we apply the induction hypothesis. If $h' = \mathit{val}_\tau(e)$, then $\mathit{accessible}^p_{t_s}(h')$ and, thus, $h' \in (\mathsf{lhs}_{t_s} \cup *\mathsf{ghs})^p$ hold. Hence, $\mathbb{A}$ ensures $\forall t'.\, \mathsf{lhs}^{p'}_{t'} = \mathsf{lhs}^p_{t'}$ and $\mathsf{lhs}^{p'} = \mathsf{lhs}^p$. Otherwise, $\mathbb{A}$ neither adds nor removes h' to and from any ghost set.
- $s = *x := e$: Since s does not allocate new heap locations, $\mathit{am}^p_\tau(h')$ holds. If $\mathit{val}_\tau(x) = h'$, then h' is accessible by t_s, and we apply the induction hypothesis. Since $\mathbb{A}$ leaves h' in the same ghost set, (4.4) holds. Otherwise, we focus on the case $\mathit{val}_\tau(x) \in *\mathsf{ghs}^p \wedge h' \in \mathit{reach}^p_\tau(e) \cap \mathsf{lhs}^p_{t_s}$ as $\mathbb{A}$ removes in this case h' from lhs_{t_s} and for all other cases guarantees that h' remains in the same ghost set. From $h' \in \mathsf{lhs}^p_{t_s}$ and our induction hypothesis, we get $t = t_s$ as h' is accessible only by a single thread. Since $\mathit{accessible}^p_{t_s}(\mathit{val}_\tau(x))$ and $\mathit{am}^p_\tau(\mathit{val}_\tau(x))$ (from $r^{p'}_\tau$) hold, we apply the induction hypothesis and obtain that another thread t' with $t' \neq t_s$ exists that can access $\mathit{val}_\tau(x)$. However, by writing e to $\mathit{val}_\tau(x)$, all from e reachable heap locations including h' become accessible from t' at p'. Since h' is accessible at p' from at least two different threads, namely t_s and t', we have to show that $h' \in *\mathsf{ghs}^{p'}$ and that h' is removed from lhs_{t_s}, which is guaranteed by $\mathbb{A}$.
- $s = c :=$ **CoreAlloc**$(\bar{e})$: If $\exists e.\, e \in \bar{e} \wedge \mathit{val}_\tau(e) = h'$, we get $\mathit{local}^{p'}(e)$ from $r^{p'}_\tau$. Thus, $t_s = t$ as only a single thread can access h'. From Asm. 4.4.6 and Lemma 4.4.6, $\mathit{localhl}^p_t(h')$ holds, and we apply the induction hypothesis to obtain $h' \in \mathsf{lhs}^p_t$. $\mathbb{A}$ guarantees that h' remains in lhs_t and that h' is not inserted into any other ghost set since $h' \neq \mathit{val}_\tau(c)$. Otherwise, $\mathit{accessible}^p_t(h')$ holds because s cannot change h''s accessibility as the arguments $\bar{e}$ are shallow (cf. Asm. 4.4.4) and, thus, s internally does not have access to h'. We apply the induction hypothesis and observe that $\mathbb{A}$ does not change set containment of h'.
- $s = \bar{r} :=$ **CoreApi_k**$(c, \bar{e})$: We reason similarly as in the case of **CoreAlloc**$(\bar{e})$ except that we consider a third case, namely $\exists r.\, r \in \bar{r} \wedge \mathit{val}_\tau(r) = h'$. In this case, $r^{p'}_\tau$ guarantees that h' is local and from $\mathit{accessible}^{p'}_t(h')$ follows that $t = t_s$. h' is a heap location newly allocated by s and $\mathbb{A}$ guarantees that h' is inserted into lhs_t. From Lemma 4.4.8, we get that h' is in no other ghost set.
- $s =$ **fork** $(\bar{x})$ $\{s'\}$: Let us call the newly spawned thread t'_s with $t'_s \neq t_s$. Since t'_s can access the variables $\bar{x}$, we have $\forall h.\, h \in \mathit{reach}^p_\tau(\bar{x}) \implies \mathit{accessible}^{p'}_{t_s}(h) \wedge \mathit{accessible}^{p'}_{t'_s}(h)$. If $h' \notin \mathit{reach}^p_\tau(\bar{x})$, then accessibility of h' does not change by executing s, and we apply the induction hypothesis and note that $\mathbb{A}$ does not modify set containment of h'. In particular, h' is not accessible by t'_s and, thus, $h' \notin \mathsf{lhs}^{p'}_{t'_s}$ holds as required by (4.4). Otherwise ($h' \in \mathit{reach}^p_\tau(\bar{x})$), we have to prove that $\forall t'.\, h' \notin \mathsf{lhs}^{p'}_{t'}$ and $h' \in *\mathsf{ghs}^{p'}$ hold. Since $\mathit{accessible}^p_{t_s}(h')$ holds, we apply the induction hypothesis and case split on whether $h' \in \mathsf{lhs}^p_{t_s}$ holds. If so, $\mathbb{A}$ moves h' from lhs_{t_s} to $*\mathsf{ghs}$, which is sufficient as $\forall t'.\, t' \neq t_s \implies h' \notin \mathsf{lhs}^p_{t'}$ holds. Otherwise, $h' \in *\mathsf{ghs}^p$ holds and $\mathbb{A}$ ensures $h' \in *\mathsf{ghs}^{p'}$.

□

Corollary 4.4.10 (Set Containment in $\mathsf{lhs} \cup {*}\mathsf{ghs}$) *A variable x is in a thread t's lhs_t or ${*}\mathsf{ghs}$ at program point p if x is a defined variable, all heap locations x may point to are* Application*-managed, and the requirements r^p_τ hold.*

$$\begin{aligned}\forall x,t,p,\tau.\, p \in \tau \wedge \mathit{defined}^p_t(x) \wedge r^p_\tau \wedge \\ (\forall h.\, \mathit{as}_\tau(h) \in \mathit{pts}(x) \implies \mathit{am}^p_\tau(h)) \implies \\ \mathit{val}_\tau(x) = \mathtt{nil} \vee \mathit{val}_\tau(x) \in (\mathsf{lhs}_t \cup \mathsf{ghs})^p\end{aligned}$$

where $\mathit{defined}^p_t(x)$ expresses that x is defined at p for thread t.

Proof sketch. Let x, t, p, and τ be arbitrary and assume the corollary's premise. If $\mathit{val}_\tau(x) = \mathtt{nil}$ holds, then the corollary holds trivially. Otherwise, x points at p to an allocated heap location, which we call h', that is, thus, accessible from thread t i.e., $h' = \mathit{val}_\tau(x) \wedge \mathit{accessible}^p_t(h')$. From Asm. 4.4.5 we obtain $\mathit{as}_\tau(h') \in \mathrm{pts}(x)$ and, thus, $\mathit{am}^p_\tau(h')$ holds. We apply Lemma 4.4.9 and observe that one of the equivalences' left-hand sides must be satisfied. Therefore, h' is either in lhs^p_t or ${*}\mathsf{ghs}^p$. □

Lemma 4.4.11 (Locality Implies Set Containment for Core Instances) *A heap location h at program point p that corresponds to a* Core *instance returned from an earlier* **CoreAlloc**$(\bar{e})$ *statement is in a thread t's* ihs *if h is local and the restrictions r^p_τ (Fig. 4.29) hold.*

$$\begin{aligned}\forall h,t,\tau,p.\, h \neq \mathtt{nil} \wedge p \in \tau \wedge \mathit{localhl}^p_t(h) \wedge \\ \mathit{pt}^p_{\text{Core}}(h,\tau) \wedge r^p_\tau \implies h \in \mathsf{ihs}^p_t\end{aligned}$$

Proof sketch. We prove this lemma by induction over program traces. The base case for the empty trace holds trivially as there are no allocated heap locations yet. In the inductive step, we prove this lemma for an arbitrary heap location h', thread t, program point p', and trace τ by applying the induction hypothesis to the immediately preceding program point p and showing that we obtain the specified set containment for p'. I.e., we assume the premise and show that $h' \in \mathsf{ihs}^{p'}_t$ holds. We case split on statement s (before applying $\mathbb{A}$) such that executing $\mathbb{A}(s)$ on thread t_s transitions from p to p'. We first note that the restrictions r are monotonic when going backwards on a trace, i.e., r^p_τ follows from $r^{p'}_\tau$.

- $s = \mathbf{skip}$: Since **skip** does not allocate Core instances and leaves accessibility unchanged, we get $\mathit{localhl}^p_t(h')$ and apply the induction hypothesis. We get $h' \in \mathsf{ihs}^{p'}_t$ as $\mathbb{A}$ leaves all ghost sets unmodified.
- $s = x := \mathbf{new}()$: $h' \neq \mathit{val}_\tau(x)$ holds because x points to a newly allocated heap location that has not been passed through the return argument of **CoreAlloc**$(\bar{e})$. Thus, $\mathit{localhl}^p_t(h')$ holds, and we apply the induction hypothesis. We observe that $\mathbb{A}$ leaves ihs_t unchanged.
- $s = x := {*}e$: Since s does not allocate Core instances, $\mathit{pt}^p_{\text{Core}}(h', t)$ holds, and we apply the induction hypothesis. The lemma holds as $\mathbb{A}$ does not modify ihs_t.
- $s = {*}x := e$: Identical reasoning as for reading a heap location.
- $s = c := \mathbf{CoreAlloc}(\bar{e})$: If $\mathit{val}_\tau(c) = h'$, then $\mathit{localhl}^{p'}_t(h')$ implies $t = t_s$. $\mathbb{A}$ guarantees that $h' \in \mathsf{ihs}^{p'}_t$. Otherwise, $\mathit{localhl}^p_t(h')$ and $\mathit{pt}^p_{\text{Core}}(h', \tau)$ hold, and we apply the induction hypothesis. $h' \in \mathsf{ihs}^{p'}_t$ holds as $\mathbb{A}$ does not remove elements from ihs.

- $s = \bar{r} := \textbf{CoreApi\_k}(c, \bar{e})$: Since s does not allocate Core instances, $localhl_t^p(h')$ and $pt_{\text{Core}}^p(h', \tau)$ hold, and we apply the induction hypothesis. Furthermore, $\mathbb{A}$ does not remove elements from $\mathsf{ihs}_{t'}$ for any thread t'.
- $s = \textbf{fork}\ (\bar{x})\ \{s'\}$: Let us call the newly spawned thread t_s' with $t_s' \neq t_s$. If $accessible_p^{t_s'}(h')$, then $t = t_s'$ as h' is local. However, h' can only be accessible to t_s' if h' is reachable from $\bar{x}$, which is accessible from thread t_s too. I.e., $accessible_p^{t_s}(h')$ holds contradicting $localhl_t^{p'}(h')$. Otherwise, $\mathbb{A}$ initializing $\mathsf{ihs}_{t_s'}$ to the empty set does not violate the lemma as t_s' cannot access h'. Furthermore, we apply the induction hypothesis as $pt_{\text{Core}}^p(h', \tau)$ holds, and we note that $\mathbb{A}$ does not remove any element from ihs_t.

□

Corollary 4.4.12 (Escape Analysis Implies Set Containment in ihs) *A variable x is in a thread t's ihs_t at program point p if x is a defined variable, local, all heap locations x may point to passed through the return parameter of some* **CoreAlloc**$(\bar{e})$*, and the requirements r_τ^p hold.*

$$\forall x, t, p, \tau.\ p \in \tau \land local^p(x) \land defined_t^p(x) \land r_\tau^p \land$$
$$(\forall h.\ as_\tau(h) \in pts(x) \implies pt_{\text{Core}}^p(h, \tau)) \implies$$
$$val_\tau(x) = \texttt{nil} \lor val_\tau(x) \in \mathsf{ihs}_t^p$$

Proof sketch. Let x, t, p, and τ be arbitrary and assume the corollary's premise. If $val_\tau(x) = \texttt{nil}$ holds, then the corollary holds trivially. Otherwise, x points at p to an allocated heap location, which we call h', which is, thus, accessible from thread t, i.e., $h' = val_\tau(x) \land accessible_t^p(h')$. $localhl_t^p(h')$ follows from Asm. 4.4.6. From Asm. 4.4.5 we obtain $as_\tau(h') \in \text{pts}(x)$ and, thus, $pt_{\text{Core}}^p(h', \tau)$. Applying Lemma 4.4.11 completes the proof. □

Having defined the properties that successfully executing our static analyses provides, we present next how we apply the static analyses in Diodon (Def. 4.4.16) and prove in Lemma 4.4.14 that this application discharges the side conditions ω (cf. Fig. 4.10).

As shown in Def. 4.4.16, we check for every heap read operation $x := *e$ that e points to Application-managed heap locations, which are identified by their allocation site a. Analogously, we check for heap writes $*x := e$ that x satisfies the same property. For every **CoreAlloc**$(\bar{e})$ and $\bar{r} :=$ **CoreApi_k**$(c, \bar{e})$, we check that the arguments $\bar{e}$ point to disjoint heap locations and that these heap locations are local and Application-managed. Additionally, we check for $\bar{r} :=$ **CoreApi_k**$(c, \bar{e})$ that c points to a local Core instance, i.e., a local heap location that has been returned by an earlier Core allocation call, and that the outputs $\bar{r}$ are local.

Definition 4.4.16 (Static Analyses for Diodon) *In Diodon, we execute the static analyses on a codebase to obtain the following judgments for every* *statement s at label ℓ therein, denoted as $j(s^\ell)$.*

$$\begin{aligned}
j(x := *e) &\triangleq \forall a, \tau.\, a \in pts(e) \implies am_\tau^{pre\text{-}\ell}(a)\\
j(*x := e) &\triangleq \forall a, \tau.\, a \in pts(x) \implies am_\tau^{pre\text{-}\ell}(a)\\
j(c := \textbf{CoreAlloc}(\bar{e})) &\triangleq disjoint_{as}(e) \land local_{am}^{\ell}(e)\\
j(\bar{r} := \textbf{CoreApi\_k}(c, \bar{e})) &\triangleq disjoint_{as}(e) \land local_{am}^{\ell}(e)\\
&\quad \land local_{\text{CORE}}^{\ell}(c) \land local_{ret}^{\ell}(r)
\end{aligned}$$

where

$$\begin{aligned}
disjoint_{as}(e) &\triangleq \forall i, j.\, 0 \le i < j < len(\bar{e}) \implies pts(e[i]) \cap pts(e[j]) = \emptyset\\
local_{am}^{\ell}(e) &\triangleq \forall e, h, \tau.\, e \in set(\bar{e}) \land as_\tau(h) \in pts(e) \implies\\
&\qquad local^{post\text{-}\ell}(e) \land am_\tau^{pre\text{-}\ell}(h)\\
local_{\text{CORE}}^{\ell}(c) &\triangleq \forall h, \tau.\, as_\tau(h) \in pts(c) \implies local^{pre\text{-}\ell}(c) \land pt_{\text{CORE}}^{pre\text{-}\ell}(h, \tau)\\
local_{ret}^{\ell}(r) &\triangleq \forall r, \tau.\, r \in set(\bar{r}) \implies local^{post\text{-}\ell}(r)
\end{aligned}$$

Lemma 4.4.13 (Discharging the Requirements) *We show that the judgments provided by our static analyses $j(s^\ell)$ (cf. Def. 4.4.16) for every statement s at label ℓ before program point p and our assumptions are sufficient to discharge the requirements r_τ^p (cf. Fig. 4.29).*

$$\forall p, \tau.\, p \in \tau \land (\forall s, \ell.\, \ell \prec p \land j(s^\ell)) \implies r_\tau^p$$

Proof sketch. We prove this lemma by induction over program traces. The base case for the empty trace holds trivially as there are no preceding statements s^ℓ. In the inductive step, we prove this lemma for an arbitrary program point p' and trace τ by applying the induction hypothesis to the immediately preceding program point p. I.e., we show that $\forall s', \ell'.\, \ell' \prec p' \land j(s'^{\ell'})$ and r_τ^p imply $r_\tau^{p'}$ by case splitting on statement s^ℓ, whose execution transitions from p to p'.

- $s^\ell = *x := e$: We have to prove that $am_\tau^p(val_\tau(x))$ holds. From $j(s^\ell)$ and Asm. 4.4.5, we get $val_\tau(x) = \texttt{nil} \lor am_\tau^p(val_\tau(x))$. $x \neq \texttt{nil}$ holds as the statement would otherwise crash (cf. Asm. 4.4.2).
- $s^\ell = c := \textbf{CoreAlloc}(\bar{e})$: We have to show for an arbitrary argument $e \in set(\bar{e})$ that $val_\tau(e) = \texttt{nil} \lor am_\tau^p(val_\tau(e)) \land local^{p'}(e)$ holds. If $val_\tau(e) \neq \texttt{nil}$, then we apply Asm. 4.4.5 to obtain $am_\tau^p(val_\tau(e))$ from $local_{\text{am}}^{\ell}(e)$.
- $s^\ell = \bar{r} := \textbf{CoreApi\_k}(c, \bar{e})$: We proceed identically as in the case of $\textbf{CoreAlloc}(\bar{e})$. Additionally, we have to show $val_\tau(c) = \texttt{nil} \lor \neg am_\tau^p(val_\tau(c))$ and $val_\tau(r) = \texttt{nil} \lor local^{p'}(r)$ for an arbitrary return argument $r \in \bar{r}$, which we get from $local_{\text{CORE}}^{\ell}(c)$ by applying Asm. 4.4.5 and $local_{\text{ret}}^{\ell}(r)$.
- Otherwise: $r_\tau^{p'}$ holds because no requirements for s^ℓ must be met.

□

Lemma 4.4.14 (Discharging the Side Conditions ω) *We show that the judgments provided by our static analyses $j(s)$ (cf. Def. 4.4.16) for every statement s in a codebase c together with our assumptions are sufficient to*

$$\dfrac{\dfrac{\dfrac{\vdots}{\mathbf{emp} \vdash [\mathbf{emp}]\ s_{\mathrm{init}}\ [R_g \star R_l]}\ \textsc{Frame}}{\mathbf{emp} \vdash [\phi]\ s_{\mathrm{init}}\ [R_g \star R_l \star \phi]}\ \textsc{Conseq}}{\mathbf{emp} \vdash [\phi]\ s_{\mathrm{init}}\ [\Pi_g \star \Pi_l]} \quad \dfrac{\dfrac{\dfrac{\dfrac{\vdots\ \mathit{Fig}.\,4.9}{\Pi_g \vdash [\Pi_l]\ \mathbb{A}(p)\ [\Pi_l]}\ \textsc{Conseq}}{\Pi_g \vdash [\Pi_l]\ \mathbb{A}(p)\ [\mathsf{true}]}\ \textsc{Share}}{\mathbf{emp} \vdash [\Pi_g \star \Pi_l]\ \mathbb{A}(p)\ [\Pi_g]}\ \textsc{Conseq}}{\mathbf{emp} \vdash [\Pi_g \star \Pi_l]\ \mathbb{A}(p)\ [\mathsf{true}]}$$

$$\dfrac{\qquad\qquad\qquad\qquad}{\mathbf{emp} \vdash [\phi]\ s_{\mathrm{init}};\ \mathbb{A}(p)\ [\mathsf{true}]}\ \textsc{Seq}$$

with $R_l \triangleq \mathsf{lhs} = \emptyset \star \mathsf{ihs} = \emptyset \quad R_g \triangleq \mathsf{acc}(\mathsf{ghs}) \star *\mathsf{ghs} = \emptyset \star \mathsf{acc}(\mathsf{used}) \star *\mathsf{used} = \mathsf{false}$

Figure 4.30: Proof tree showing the initial establishment of Π_l and Π_g for a codebase p. We assume that the ghost statement s_{init} initializes lhs and ihs to the empty set, as stated in R_l, and allocates two heap locations on the ghost heap storing $\emptyset$ and false to which ghs and used point, respectively (cf. R_g).

discharge the side conditions $\omega(s)$ (cf. Fig. 4.10).

$$\forall s \in c.\, (\forall s' \in c.\, j(s')) \implies \omega(s)$$

Proof sketch. We prove this lemma for an arbitrary statement s at label ℓ such that $s \in c$, assume $\forall s' \in c.\, j(s')$ and show that $\omega(s)$ holds by case splitting on statement s. Throughout the proof, we use program point p to refer to s's pre-state, i.e., $p \triangleq$ pre-ℓ. We obtain $\forall \tau.\, r_\tau^p$ from Lemma 4.4.13.

- $s = x := *e$: From Cor. 4.4.10, we get $val_\tau(e) = \mathtt{nil} \lor val_\tau(e) \in (\mathsf{lhs} \cup \mathsf{ghs})^p$. e points to an allocated heap location and cannot be $\mathtt{nil}$ as the statement would otherwise crash (cf. Asm. 4.4.2).
- $s = *x := e$: Analogous to heap reads but for x instead of e.
- $s = c :=$ **CoreAlloc**$(\bar{e})$: From $disjoint_{\mathrm{as}}(e)$, we obtain $disjoint(\bar{e})$ by applying Lemma 4.4.4. $local^{\ell}_{\mathrm{am}}(e)$ allows us to apply Lemma 4.4.9 providing $\forall e \in set(\bar{e}).\, val_\tau(e) = \mathtt{nil} \lor val_\tau(e) \in \mathsf{lhs}^p$. Lastly, $*\mathsf{used} = \mathsf{false}$ holds by our assumption that we have a single Core allocation statement in the codebase c. We lift this assumption at the end of this subsection.
- $s = \bar{r} :=$ **CoreApi_k**$(c, \bar{e})$: Likewise to the previous case, we obtain $disjoint(\bar{e})$ and $\forall e \in set(\bar{e}).\, val_\tau(e) = \mathtt{nil} \lor val_\tau(e) \in \mathsf{lhs}^p$. Left to show is $val_\tau(c) = \mathtt{nil} \lor val_\tau(c) \in \mathsf{ihs}^p$, which we obtain from $local^{\ell}_{\mathrm{CORE}}(c)$ by applying Cor. 4.4.12.
- Otherwise: $\omega(s) = \mathsf{true}$ and, thus, the lemma holds trivially.

□

Proof Construction

While we showed that we can compose the proof rules in Fig. 4.9 and discharge their side conditions ω, it remains to show that we initially establish the global context Π_g and the local program invariant Π_l, such that we obtain a proof for the entire codebase c. We close this gap in Cor. 4.4.15.

Corollary 4.4.15 (Proof Construction) *Successfully executing Diodon's static analyses on a codebase c and the Core's auto-active verification combined* *with our assumptions allow us to construct a separation logic proof for c.*

$$\begin{aligned}
&\textit{If } \forall s, k.\, s \in c \wedge j(s) \wedge \\
&\quad \Big(s = c := \textbf{CoreAlloc}(\bar{e}) \implies \\
&\qquad \Pi_g \vdash [P_{CoreAlloc}(\bar{e})]\ \ s\ \ [Q_{CoreAlloc}(c, \bar{e})]\Big) \wedge \\
&\quad \Big(s = \bar{r} := \textbf{CoreApi\_k}(c, \bar{e}) \implies \\
&\qquad \Pi_g \vdash \big[P_{CoreApi\_k}(c, \bar{e})\big]\ \ s\ \ \big[Q_{CoreApi\_k}(c, \bar{e}, \bar{r})\big]\Big), \\
&\textit{then } \textbf{emp} \vdash \big[\phi\big]\ \ s_{init};\ \mathbb{A}(c)\ \ [\mathsf{true}]
\end{aligned}$$

where s_{init} is ghost code creating and initializing the thread-local ghost sets lhs *and* ihs *for the main thread, as well as the global ghost set* *ghs *and the ghost flag* *used.

Proof sketch. All our proof rules (cf. Fig. 4.9) have the same shape, namely $\Pi_g \vdash [\Pi_l]\ \ \mathbb{A}(s)\ \ [\Pi_l]$ for a statement s. As shown by Lemma 4.4.14, the judgments obtained from the static analyses allow us to discharge the side conditions that are associated with each proof rule (Fig. 4.10). Therefore, left to show is that we initially establish Π_l and Π_g such that we can compose the proof rules to form a proof for an entire codebase c. The ghost statement s_{init} creates and initializes the ghost sets lhs, ihs, and *ghs as well as the ghost flag *used. Thus, we can complete the proof tree as shown in Fig. 4.30. This constitutes a proof for c as neither s_{init} nor the statements added by $\mathbb{A}$ modify c's runtime behavior. $\square$

We show that we obtain the desired proof for the entire codebase, namely that the codebase satisfies the I/O specification ϕ expressed as the Hoare triple $\textbf{emp} \vdash \big[\phi\big]\ \ s_{\text{init}};\ \mathbb{A}(c)\ \ [\mathsf{true}]$. This Hoare triple relies on s_{init} that initializes lhs, ihs, and *ghs to empty sets, as well as sets the ghost flag *used to false. s_{init} is similar in spirit to the ghost statements that algorithm $\mathbb{A}$ inserts as these statements are necessary to construct a proof for the codebase c. Cor. 4.4.15's premise states that our static analyses succeed on the codebase c, such that we obtain $j(s)$ for each statement s therein, and that we prove a Hoare triple for each Core function satisfying the syntactic restrictions.

We combine the proof for the entire codebase that we obtain from Cor. 4.4.15 with the result of Sec. 4.4.1 to obtain Diodon's overall soundness result. This result states that successfully executing our static analyses on codebase c and auto-actively verifying its Core suffices to prove that the traces of executing c together with other verified implementations and the environment are contained in the traces described by the abstract protocol model.

Theorem 4.4.16 (Overall Soundness) *Suppose Asm. 4.4.1 holds and that we have established, for each role i, I/O independence and Cor. 4.4.15's antecedent for a codebase $c_i(rid)$ and I/O specification $\psi_i(rid)$. Then*

$$(|||_{i,rid}\ \pi_{int}(\mathscr{C}_i(rid)))\ \|_{\chi'}\ \mathscr{E} \preccurlyeq_t \mathscr{R}.$$

Proof sketch. To obtain $\textbf{emp} \vdash \big[\psi_i(rid)\big]\ \ s_{\text{init}};\ \mathbb{A}(c_i(rid))\ \ [\mathsf{true}]$ for each role i, we apply Cor. 4.4.15. Since we omitted the turnstile subscript α (cf. Asm. 4.4.1) throughout Sec. 4.4.2 for brevity and **emp** on the turnstile's left-hand side is a notational difference only (Asm. 4.4.1 could be adapted accordingly), we apply Thm. 4.4.2 to obtain the desired result. □

Limitations

Our formalization defines a simple programming language to focus on the main ideas of our soundness proof and to show that successfully executing our static analyses discharges all side conditions. We believe this language covers the most critical features like heap manipulations and concurrency as these features are relevant for the results of our static analyses. In addition, we abstract each function making up the Core's API to a dedicated statement in our language, and assume that the specification of each such function satisfies our syntactic restrictions Asm. 4.4.4. However, there is a slight risk that this language misses Go features that would be a threat to soundness such as function boundaries, complex control flow, and callbacks; the former two features would be straightforward to add, and we discuss next how to add the latter.

To prove a Hoare triple for the entire codebase, we assume that the Application is free of crashes Asm. 4.4.2and data races Asm. 4.4.3. While our soundness proof does not make any statement in the case that the program crashes, our compositional proof informally guarantees that the trace inclusion holds for the program's prefix up to the program point at which a crash occurs, such that the crash freedom assumption could be dropped, which we leave to future work. However, data race freedom remains an assumption; more generally, we assume the absence of undefined behavior for programming languages other than Go and our formalized one. This assumption can be mitigated by performing additional static analyses.

Extensions

Having covered the main soundness result, we discuss two extensions to bridge the gap to realistic applications of Diodon as used in our case studies. We first lift the restriction of at most one Core instance to allow a codebase to create unboundedly many Core instances. Second, we allow the Core to invoke callbacks into the Application and discuss the side conditions that arise by this extension.

Unboundedly Many Core Instances. So far, our global program invariant Π_g contains the separating conjunct

$$\mathtt{acc}(\mathsf{used}) \star (\neg(*\mathsf{used}) \implies \phi).$$

As explained in Sec. 4.4.1, each execution of a protocol role is parameterized by a unique *rid*. I.e., ϕ and all I/O permissions that ϕ internally provides are parameterized by *rid* and, thus, are not interchangeable but specific to a particular *rid*. Hence, we can change the separating conjunct stated above to

$$\mathtt{acc}(\mathsf{used}) \star (\forall rid \notin *\mathsf{used} \implies \phi(rid))$$

providing a family of I/O permissions, where used points to a ghost set containing the *rid*s that have already been used. In addition, we adapt the entire program's precondition from ϕ to $\forall rid.\, \phi(rid)$ and change the translation $\mathbb{A}(c := \mathbf{CoreAlloc}(\bar{e}))$ to, first, pick a fresh rid' such that

$rid' \notin *\mathsf{used}$ and, second, adding rid' to $*\mathsf{used}$. Picking such a fresh rid' is always possible since rid ranges over $\mathbb{N}$.

Adding Callbacks to the Core. So far, we have treated the statements $\mathbf{CoreAlloc}(\bar{e})$ and $\bar{r} := \mathbf{CoreApi\_k}(c, \bar{e})$ as atomic statements in our language. These two statements are internally implemented as sequences of statements, which we hereafter call Core statements. As these statements constitute the Core, we auto-actively prove that a particular postcondition holds when control transfers back to the Application after fully executing these statements.

In the presence of callbacks, however, calling into the Core becomes non-atomic and control flow might transfer to the Application before reaching the post-state for which we know that the postcondition holds. We can treat callbacks as temporarily pausing the execution of these auto-actively verified Core statements to (sequentially) execute some statements belonging to the Application before eventually resuming execution of Core statements.

With respect to algorithm $\mathbb{A}$ and the ghost sets, interrupting the execution of Core statements to execute certain Application statements s_{app} means that heap locations on which the Core statements operate are missing from the ghost sets while executing s_{app} as we remove them from the ghost sets before executing Core statements and put them back only after the Core statements' postcondition holds. Missing permissions include both arguments $\bar{e}$ and the Core instance c. Therefore, we have to make sure that s_{app} neither accesses heap locations to which $\bar{e}$ points nor invokes API calls on the Core instance c as the Core invariant might not hold.

We can lift these restrictions by introducing additional proof obligations for the auto-active verification. More specifically, if we auto-actively prove that the Core statements satisfy a particular precondition for the callback, then we can update the ghost sets accordingly. E.g., such a precondition can specify permissions for heap locations passed to the callback or that the Core invariant holds.

In our SSM Agent case study (Sec. 4.5.1), we make use of these proof obligations for the callback delivering incoming messages to the Application as we specify that the Core transfers permission for the incoming message to the Application. Conceptually, this allows us to add the corresponding heap location to lhs before executing the statements constituting the callback because the auto-active proof guarantees that no statement in the Core thereafter accesses this heap location.

For our case studies, it was not necessary to transfer permissions from a callback back to the Core via a callback's postcondition. Extending Diodon to allow such permission transfers would require an analysis of the callback showing that the Application possesses these permissions while executing the callback and that the corresponding heap locations do not get accessed by the Application after the callback returns.

4.5 Case Studies

To demonstrate that Diodon scales to large codebases, we evaluate it on the AWS Systems Manager Agent (SSM Agent) [132], a 100k+ LOC production Go codebase. Furthermore, we apply Diodon to a small implementation

[132]: Amazon Web Services, Inc. (2023), *Working with SSM Agent*

Table 4.2: Execution time for running each tool on the SSM Agent codebase and approximate proof effort in person-months (PMs) for creating a protocol model, adding specifications, and adapting the Argot analyses, respectively.

	Tool	Proof Effort	Execution Time
Protocol Model	Tamarin	<2 PMs	3.30 min
Core Refinement	Gobra	<3 PMs	1.17 min
I/O Independence	Argot	<0.5 PM	0.48 min
Core Assumptions	Argot	<1.5 PMs	2.12 min

of the signed Diffie–Hellman (DH) key exchange to showcase that our methodology applies to other implementations and coding styles.

4.5.1 AWS Systems Manager Agent

[132]: Amazon Web Services, Inc. (2023), *Working with SSM Agent*

The AWS Systems Manager Agent (SSM Agent) [132] provides features for configuring, updating, and managing Amazon EC2 instances, and is widely used by AWS customers. A fork of this codebase implements a novel protocol which enables encrypted interactive shell sessions with remote host machines, similar to the Secure Shell (SSH) protocol, without needing to open inbound ports or manage SSH keys. This protocol establishes these shell sessions with a handshake protocol involving a signed elliptic-curve DH key exchange to derive sessions keys that are subsequently used in the transport phase to encrypt the shell commands and their results.

We apply Diodon by first modeling the protocol in Tamarin and proving secrecy and injective agreement. Second, we partition the codebase into the code implementing the protocol (the Core) and the remaining codebase (the Application), and prove I/O independence. Third, we auto-actively verify the Core using Gobra to prove that the Core refines the SSM Agent's role. Finally, we apply the automatic static analyses Argot [135] to discharge the assumptions within the Application on which the auto-active proof relies.

[135]: AWS Labs (2024), *Argot*

Tab. 4.2 overviews each tool's execution time, for which we use the 10 % Winsorized mean of the wall-clock runtime across 10 verification runs, measured on a 2023 Apple MacBook Pro with M3 Pro processor and macOS 15.6.

Protocol Model

We model in Tamarin the security protocol for establishing a remote shell session between an SSM Agent running on an EC2 instance and an AWS customer. The protocol offloads all signature operations to the AWS Key Management Service (KMS) [139] such that neither protocol role has to manage their own signing keys. We model the connections to KMS as secure channels. Furthermore, the SSM Agent sends the asymmetrically-encrypted session keys to a trusted third party to monitor the transmitted shell commands should this be necessary for regulatory reasons.

[139]: Amazon Web Services, Inc. (2024), *AWS Key Management Service*

Full Protocol Description

Fig. 4.31 shows the protocol for establishing interactive shell sessions between an SSM Agent (A) and a customer (B). The protocol includes two additional roles namely KMS (S) and an optional, trusted monitor (M) that is allowed to inspect the established shell sessions, e.g., for compliance reasons.

$$\begin{array}{lll}
M1. & A \Rightarrow S: & \langle SignReq, Id_{skA}, g^{x}, Id_{M}, Id_{B} \rangle \\
M2. & S \Rightarrow A: & \langle SignResp, sig_{x} \rangle \\
M3. & A \rightarrow B: & \langle SessReq, g^{x}, sig_{x}, Id_{skA}, Id_{M} \rangle \\
M4. & B \Rightarrow S: & \langle VerReq, Id_{A}, Id_{skA}, g^{x}, Id_{M}, Id_{B}, sig_{x} \rangle \\
M5. & S \Rightarrow B: & \langle VerResp \rangle \\
M6. & B \Rightarrow S: & \langle SignReq, Id_{skB}, g^{y}, Id_{A} \rangle \\
M7. & S \Rightarrow B: & \langle SignResp, sig_{y} \rangle \\
M8. & B \rightarrow A: & \langle SessResp, g^{y}, sig_{y}, Id_{skB}, h(g^{x*y}) \rangle \\
M9. & A \Rightarrow S: & \langle VerReq, Id_{B}, Id_{skB}, g^{y}, Id_{A}, sig_{y} \rangle \\
M10. & S \Rightarrow A: & \langle VerResp \rangle \\
M11. & A \Rightarrow S: & \langle SignReq, Id_{skA}, c_{ss}, Id_{B} \rangle \\
M12. & S \Rightarrow A: & \langle SignResp, sig_{ss} \rangle \\
M13. & A \rightarrow M: & \langle SSKey, c_{ss}, sig_{ss}, Id_{A}, Id_{skA}, Id_{B} \rangle \\
M14. & A \rightarrow B: & \langle HSDone, c_{ss}, \mathsf{senc}(\langle HSPay, z \rangle, kdf1(g^{x*y})) \rangle \\
M15. & A \rightarrow B: & \langle Msg, \mathsf{senc}(z, kdf1(g^{x*y})) \rangle \\
M16. & B \rightarrow A: & \langle Msg, \mathsf{senc}(z, kdf2(g^{x*y})) \rangle
\end{array}$$

$$\begin{array}{lll}
\text{where} & sig_{x} & \triangleq \mathsf{sign}(\langle g^{x}, Id_{M}, Id_{B} \rangle, sk_{A}) \\
& sig_{y} & \triangleq \mathsf{sign}(\langle g^{y}, Id_{A} \rangle, sk_{B}) \\
& c_{ss} & \triangleq \mathsf{enc}(\langle kdf1(g^{x*y}), kdf2(g^{x*y}) \rangle, pk_{M}) \\
& sig_{ss} & \triangleq \mathsf{sign}(\langle c_{ss}, Id_{B} \rangle, sk_{A})
\end{array}$$

Figure 4.31: Signed DH key exchange for deriving the symmetric keys $kdf1(g^{x*y})$ and $kdf2(g^{x*y})$ that are used during the transport phase, i.e., in messages *M15* and *M16*. We use $\rightarrow$ and $\Rightarrow$ to denote communication via the untrusted network and a secure channel, respectively.

Since A and B do not personally possess their secret keys for creating signatures, we explicitly model the presence of and the interactions with KMS that remotely creates and checks signatures. We model these interactions as happening on a secure channel, indicated by $\Rightarrow$, because each role instance of A and B establishes a TLS connection to KMS.

On a high-level, this protocol performs a signed elliptic-curve DH key exchange establishing two symmetric keys $kdf1(g^{x*y})$ and $kdf2(g^{x*y})$. These keys are used in the transport phase, i.e., *M15* and *M16*, to symmetrically encrypt (*senc*) payloads for sending in a particular direction. In Tamarin, we model the transport phase as a non-deterministic loop that allows each role A and B to send and receive an unbounded number of transport messages and interleave them arbitrarily.

More specifically, the protocol proceeds as follows. Role A first generates an elliptic-curve public-private key pair, which we model in Tamarin as generating a fresh term x and computing the corresponding public key via modular exponentiation denoted by g^{x}. Then, A sends message *M1* to instruct KMS to use a particular signing key belonging to A, identified by Id_{skA}, to sign the triple $\langle g^{x}, Id_{M}, Id_{B} \rangle$. This triple includes the monitor's and B's identity to prevent Mallory-in-the-middle (MITM) attacks. KMS checks whether the requested signing key actually belongs to A before creating and sending the signature in *M2* back to A. This allows A to send a session request (*M3*) to B, which includes g^{x}, the signature, and the signing key's and monitor's identities.

After receiving a session request, B first checks the received signature via KMS. For this purpose, B sends the signature itself and the components over which the signature is computed in a signature check request (*M4*) to KMS. If the signature is valid, KMS replies with a signature check response (*M5*). Otherwise, KMS aborts the protocol, which we model as not sending any response. Afterwards, B generates its elliptic-curve public-private key pair (g^y, y) and uses KMS to sign g^y and A's identity. B then sends a session response (*M8*) to A that contains B's public curve point, the signature, the identity of B's signing key, and a hash of the shared secret $h(g^{x*y})$. The latter allows A to detect early on if A and B computed different shared secrets, e.g., due to an attempted replay attack.

After receiving a session response, A computes the shared secret and checks that it derives the same shared secret's hash value. Additionally, A checks the received signature using KMS and derives the two symmetric session keys from the shared secret by applying two different key derivation functions (KDFs) *kdf1* and *kdf2*. To enable a trusted monitor M to audit the shell session, A computes c_{ss} by asymmetrically encrypting the two session keys using the monitor's public key pk_M. Next, role A requests a signature from KMS for c_{ss} and B's identity to bind these identities to the session keys. The handshake ends by sending the encrypted session keys to the monitor (*M13*) and confirming the session keys to B (*M14*). The latter message includes some version information, which we model as an attacker-chosen payload z.

Message *M13* enables M, a trusted third party, to monitor the transmitted shell commands should this be necessary for regulatory reasons (otherwise sending message *M13* can simply be skipped). For this purpose, role A sends the asymmetrically encrypted session keys to the monitor M such that M can obtain the session keys and, thus, decrypt and audit the transport messages. Note that the monitor does not need to be online during the handshake or transport phase; it is sufficient for the monitor to come online at a later time as an untrusted log server could store message *M13* and all messages sent during the transport phase until M becomes online and fetches these messages from the log server.

In Tamarin, we prove secrecy for the two symmetric session keys, i.e., the attacker does not learn these keys unless the SSM Agent's or customer's signing key or the monitor's secret key is corrupted. Additionally, we prove that the SSM Agent injectively agrees with the customer, and vice versa, on their identities and the session keys, unless one of the three aforementioned corruption cases occurs.

The abstract protocol model amounts to 319 LOC and is automatically verified by Tamarin 1.10.0 in 3.30 min using an auxiliary oracle consisting of 75 lines of Python code.

Proving I/O Independence

We perform a taint analysis to prove I/O independence. We configure the taint analysis to consider all generated elliptic-curve secret keys as sources of protocol secrets. We assume that only the Core uses the SSM Agent's signing keys and do not treat KMS responses as taint sources because KMS only sends us signatures and never key material. As described in Sec. 4.3.1, we use Capslock's capability information to automatically configure the taint analysis' sinks.

We annotated some branching operations, instructing the taint analysis to ignore that the branch condition is tainted. We identified two classes of such branching operations. The first class is justified by cryptography. E.g., we allow branching on the success of decrypting a transport message because leakage is minimal. The second class results from imprecisions of the taint analysis and corresponds to false positives, i.e., the analysis deems a branch condition tainted although it is not. To avoid another source of false positives, we configured the taint analysis to ignore taint escaping the current thread (which would otherwise always lead to errors). Such cases could be handled precisely by marking certain struct fields as potentially storing concurrently-accessed, tainted data, such that the analysis can track the taint.

The taint analysis succeeds for the SSM Agent codebase in 29.0 s, proving that there are no taint flows.

Core Refinement

The SSM Agent contains a Go package called `datachannel` that implements the protocol. More precisely, this package contains struct definitions that together store all necessary internal state. Additionally, this package exposes publicly accessible functions to initialize the internal state, perform a handshake, and send a payload, which internally rely on several private functions. We refer to these struct definitions and functions as the Core. For backward compatibility, the Core also implements a legacy protocol; we assume that this legacy protocol is disabled.

Implementation. Each Core instance corresponds to one run of the protocol with a particular AWS customer. During initialization of a new Core instance, the Core starts a new thread, responsible for receiving and processing incoming packets for this protocol run, similar to the running example. If an incoming packet contains a transport phase payload, this payload is delivered by a callback to the Application. Thus, the Core uses two different threads, one for sending messages and another one for receiving messages, which both operate on shared state. This shared state keeps track of the progress within the protocol and the secret data involved in the protocol, such as the elliptic-curve DH points and the resulting session keys.

Since the shared state is modified during the handshake, accesses must be synchronized to avoid data races. Hence, the Core employs Go channels, i.e., lightweight message passing, to signal a transfer of the shared state's ownership from one thread to another. During the handshake phase, exclusive ownership is transferred such that the threads have synchronized write access to the shared state. Afterwards, the shared state, which includes the established session keys, is used in a read-only way permitting both threads to concurrently read the shared state while sending and receiving transport messages.

Auto-Active Refinement Proof. We verify the Core using Gobra, which proves that the Core refines the Tamarin model's SSM Agent role. This proof encompasses safety, i.e., we prove that the Core does not crash and has sufficient permissions for every heap access, thus, guaranteeing absence of data races. In particular, this forces us to reason precisely about the accesses to shared state that the two threads within the Core perform.

Due to the intricate interplay of these threads, the resulting safety proof is substantial and requires GOBRA's expressivity. We isolate and axiomatize operations that GOBRA does not yet support such as simultaneously receiving on multiple channels and functionally reasoning about serialization and deserialization. For the purpose of the proof, we treat the CORE as a state machine consisting of 12 different states. This allows us to refer to these states in the CORE's invariant and precisely express for each state the permissions and progress w.r.t. the abstract protocol model.

Although the entire complexity of the proof is encapsulated in the CORE's invariant, function calls to the CORE must respect its state machine. To avoid exposing the state machine in these functions' preconditions and imposing additional restrictions on callers, we slightly changed the implementation to perform a dynamic check consisting of a comparison with `nil` and a single integer comparison ensuring that the state machine is in a correct state; otherwise, these CORE functions return a descriptive error. Thus, the CORE functions' specifications are similar to those of our running example, i.e., mention only the invariant and specify permissions for parameters without referring to the state machine. While most parameters are of primitive type or shallow, there are a few non-shallow input parameters, which the CORE treats as opaque. Similarly, the callback from the CORE to the APPLICATION delivers a non-shallow struct for which we ensure that the CORE passes permissions for all transitively reachable heap locations to the APPLICATION.

We prove safety and refinement of the CORE in 1.17 min for 749 lines of code requiring 3825 lines of specification and proof annotations; 1064 thereof are related to the I/O specification and generated automatically by TAMARIN.

Analyzing the APPLICATION

The auto-active proof for the CORE relies on callers satisfying the specified preconditions, which we establish using a combination of static analyses. We implemented automatic checks as described in Sec. 4.3.3 for conditions (C1)–(C4) and (C6)–(C8). Condition (C5) requires a more precise call graph than is currently available in our tool and is, thus, left as future work.

We implemented our analyses by forking and extending the existing ARGOT tool. Most of our analyses are obtained by interpreting the output of an existing analysis; e.g., the parameter alias check uses the off-the-shelf pointer analysis to show parameters do not alias one another.

For some conditions, our static analyses were not able to validate the APPLICATION due to tool limitations. For example, the escape analysis cannot reason about which fields are accessed after a struct escapes. This can cause the tool to raise alarms when a struct stores a CORE instance in a field. We found it was straightforward to rewrite the CORE and APPLICATION to eliminate these failures. For example, the struct leakage can be fixed by moving the relevant field accesses before thread creation, so that the new thread has access only to the values of those fields and not the entire struct, and by extension the CORE instance.

By running our escape analysis, we observed that CORE instances escape the thread in which they are created because the APPLICATION creates a closure that closes over an object that points to a CORE instance. This capture is incidental in that the closure does not access the captured CORE instance, which we verified by manual inspection. This capture can be eliminated by rewriting the application to reference only the state necessary in this closure, rather than the full object. This change would result in a more defensive implementation by reducing the scope of possibly concurrent accesses.

Our pass-through analysis is a prototype that succeeds on our second case study. However, for the SSM Agent, we obtain false positives due to allocations in functions called from both Core and Application, which could be addressed by adding calling context information.

Some Core functions take a pointer to a logger object as a parameter, which is internally thread-safe and shared between threads. We can safely ignore escape errors due to these parameters because the Core does not access any memory of the logger object; the pointer is just used as an opaque reference to invoke log functions that are part of the Application.

In summary, this case study demonstrates that Diodon allows one to obtain strong security guarantees for a production codebase that was not designed with formal verification in mind. The remaining limitations (manual overrides of false positives in the static analyses, checking condition (C5), extremely lightweight dynamic checks enforcing non-nilness and correct ordering of API calls, and minor code changes) are modest compared to the complexity of the overall verification challenge and we conjecture that we can lift them by employing more precise static analyses.

4.5.2 Signed Diffie–Hellman (DH) Key Exchange

We also apply our approach to a codebase employing *inverted I/O*, i.e., has a Core that only produces and consumes byte arrays corresponding to protocol messages while the Application performs all I/O operations. We adapted the Tamarin model and Go implementation of the signed DH key exchange from Sec. 2.6 and extended both by a transport phase that uses the established session key to send and receive unboundedly many payloads. Tamarin verifies the abstract model with 177 lines of code in 3.2 s while Gobra verifies the Core consisting of 178 lines of code in 14.2 s requiring 1726 LOS. Executing all static analyses including the taint analysis takes 9.7 s.

This case study clearly exhibits the concept of virtual I/O. The Core performs a virtual input operation for messages that the Application received from the network and forwarded to the Core. Similarly, we perform a virtual output operation for every message that the Core produces before returning this message to the Application. Therefore, we prove that the Tamarin model permits sending this message and in return, we sanitize the message from a taint analysis' perspective such that the Application can send the message without causing a false-positive taint flow.

Diodon separates the justification of sending a particular message from the actual I/O operation. This is important for tackling realistic codebases because identifying the actual send operation in a call stack is typically difficult as a message passes through several functions that, e.g., add additional protocol headers before a message is handed to the network interface controller.

4.5.3 Discussion

Our evaluation demonstrates that Diodon enables us to efficiently prove that an entire codebase refines a protocol model and therefore is secure. To obtain the security properties as proven in Tamarin for a deployment of this protocol, we have to prove the implementations of all other protocol roles analogously against the same model using Diodon.

As shown in Tab. 4.2, the efforts for applying Diodon to the SSM Agent is manageable. Thanks to I/O independence, the Tamarin model is concise and can focus on the relevant interactions between the protocol roles. In addition, I/O independence allows us to apply automatic static analyses at the code-level to reason about all protocol-irrelevant I/O operations. This contrasts existing approaches that would auto-actively verify the entire codebase and prove that every I/O operation is explicitly permitted by the model, which is completely impractical for this codebase.

To evaluate Diodon's effectiveness at preventing security vulnerabilities, we deliberately introduce bugs in our case studies. E.g., our taint analysis correctly fails if the Core's internal state, which includes the established session keys, is logged after the handshake. Additionally, sending the DH secret key in plaintext correctly results in Gobra failing to prove refinement w.r.t. the abstract protocol model. The tools' execution time in the presence of these bugs remains comparable to that for the secure implementations.

By applying Diodon we not only obtain security properties for the SSM Agent codebase but we also discovered and fixed bugs along the way. Tamarin allowed us to quickly locate and fix a MITM attack in an earlier and unreleased version of the protocol, which is possible if the intended recipient's identity is omitted in the signatures (sig_x and sig_y in Fig. 4.31). On the code level, we identified and fixed a potential data race in an earlier and unreleased version of the Core caused by insufficient synchronization between the two threads that send and receive handshake messages. We uncovered this data race because completing the safety proof for the Core's earlier version is not possible as an additional synchronization point is necessary to transfer separation logic permissions between these threads. This demonstrates the power of applying formal methods because detecting this data race with testing techniques would require to precisely time the reception of a handshake message such that the faulty memory access occurs and, thus, can be observed.

4.6 Alternative Approach to Verifying the Core

Diodon and its soundness proof in Sec. 4.4 expect that the Core is auto-actively verified using our approach from Chapter 2, i.e., w.r.t. to a Tamarin model. Instead, in this section, we sketch an alternative that auto-actively verifies the Core using our approach from Chapter 3.

As we have seen in Chapter 3, we have to prove that all operations appending trace entries to the global trace maintain the trace invariant. Since the Application can, e.g., generate random numbers and perform I/O operations, proof obligations that the trace invariant is maintained arise in the Core *and* the Application. By auto-actively verifying the Core using a protocol-specific instantiation of the trace invariant, we discharge proof obligations for trace-relevant operations within the *Core*. This auto-active verification uses the Core invariant (`Inv` in Fig. 4.2) but instead of containing an I/O specification, this invariant must maintain sufficient knowledge about the local snapshot to discharge these proof obligations, as described in Sec. 3.3.4.

We discharge proof obligations for trace-relevant operations within the *Application* by relying on I/O independence, *without* requiring auto-active verification. Following the same high-level idea of our soundness proof in Sec. 4.4.1, we can treat the Application's operations and the corresponding trace entries that the Application conceptually appends to the global trace as refining our DH attacker. Since we prove attacker completeness (cf. Sec. 3.3.3) once and for all protocols as part of our reusable verification library, all operations performed by the DH attacker provably maintain the trace invariant, independently of a concrete protocol-specific instantiation of the trace invariant that the Core uses to prove security properties. Considering, e.g., send operations within the Application, this means more specifically that (1) we assume that the Application's I/O independence implies that the DH attacker can construct and send the same payloads. Thanks to (2) attacker completeness, we already proved as part of our library that sending these payloads maintains the trace invariant, i.e., that these payloads have a secrecy label indicating that the attacker already knows the class of terms corresponding to these payloads (cf. Sec. 3.4.2). Combining (1) and (2), we can conclude that the Application's send operations (and analogously all other trace-relevant operations) maintain the trace invariant. In particular, this once-and-for-all conclusion does *not* require us to discharge any additional proof obligation for each trace-relevant operation within the Application, except for showing that the Application satisfies I/O independence. Hence, the Application remains amenable to static analyses.

To ensure sound composition of the Core and Application, we perform the same checks by executing static analyses as described in Sec. 4.3.3, which guarantee that the Application respects the Core's separation logic specification. A detail on the interactions between the Core and Application worth mentioning regards *virtual I/O*, i.e., exchanged data. Since we treat the Application as an instance of the DH attacker, all data that the Application can freely use, i.e., data that our taint analysis considers being untainted, must be readable (according to its secrecy label) by the attacker, which corresponds to the trace invariant for messages. Thus, we can either explicitly model these data exchanges by appending trace entries in the Core representing receiving data from the Application and sending data to the Application, which is similar to virtual I/O operations explained in Sec. 4.2, or place appropriate statements in the Core assuming and asserting message invariants for the exchanged data.

To summarize, the Diodon methodology is not specific to refinement-based verification of the Core but the insight of treating the Application as an instance of the DH attacker generalizes to other approaches for verifying the Core. To retain Diodon's ability to scale to large codebases, we require that no auto-active verification of the Application is necessary and the specifications of Core functions exposed to the Application fall into the supported class of specifications (cf. Asm. 4.4.4), which is the case for our invariant-based methodology from Chapter 3.

4.7 Related Work

Much prior work on verifying security protocols exists, as covered in the previous chapters. Hence, we focus on approaches for verifying security properties for *implementations* and their applicability to large and real-world codebases. We end by comparing Diodon to approaches proving strong isolation guarantees and performing dynamic verification.

Implementation and Model Generation. One approach to obtain verified protocol implementations generates secure-by-construction implementations from an abstract model, e.g., [39, 46, 47, 51, 52]. Besides previously mentioned drawbacks of generated implementations, OwlC [52] is the only work that considers embedding a generated implementation into a larger codebase. OwlC assumes that protocol secrets occur only in the generated implementation and relies on the Rust type system to shield these secrets from the rest of the codebase. In contrast, we do not adopt this restriction and, instead, check I/O independence for the Application. Furthermore, OwlC guarantees that generated functions can be called in arbitrary order and uses Rust types to ensure that, e.g., a session key established during a handshake is passed to a function that sends or receives a transport message. Instead, we use the Core invariant to maintain separation logic properties between Core API calls, which is more expressive.

[39]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*
[46]: Pozza et al. (2004), *Spi2Java: Automatic Cryptographic Protocol Java Code Generation from Spi Calculus*
[47]: Cadé et al. (2012), *From Computationally-Proved Protocol Specifications to Implementations*
[51]: Gancher et al. (2023), *Owl: Compositional Verification of Security Protocols via an Information-Flow Type System*
[52]: Singh et al. (2025), *OwlC: Compiling Security Protocols to Verified, Secure, High-Performance Libraries*

An alternative approach extracts an abstract model from an implementation, e.g., [12, 53, 56–58]. However, for this extraction to work, an implementation typically has to follow restrictive coding disciplines such that relevant protocol steps can be identified and extracted. To achieve isolation between a verified component and potentially malicious code, Kobeissi *et al.* [58] build on process isolation provided by operating systems and, thus, require verifying the entire critical process. We cannot adopt this approach because it requires changing the codebase heavily to split it into several processes and results in an, for our use case, unacceptable overhead, since each process includes its copy of the Go runtime and the Go standard library. Bhargavan *et al.* [12] impose substantial restrictions on the API of verified code, e.g., disallowing state preservation between API calls. Codebases do not normally satisfy these restrictions, including all our case studies. E.g., they use a session key for sending a transport message in one API call that was established during the handshake, i.e., a previous API call.

[12]: Bhargavan et al. (2017), *Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate*
[53]: Bhargavan et al. (2008), *Verified Interoperable Implementations of Security Protocols*
[56]: O'Shea (2008), *Using Elyjah to Analyse Java Implementations of Cryptographic Protocols*
[57]: Aizatulin et al. (2012), *Computational Verification of C Protocol Implementations by Symbolic Execution*
[58]: Kobeissi et al. (2017), *Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach*

Existing Implementations. Dupressoir *et al.* [64] and our invariant-based (Chapter 3) and refinement-based (Chapter 2) methodologies require verifying the entire codebase using an auto-active verifier (which Diodon does not). We build on the latter methodology and, to the best of our knowledge, are the first to relax this requirement to verifying just the Core and reason about the Application using lightweight static analyses.

[64]: Dupressoir et al. (2011), *Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols*

Verified Isolation. Outside the security protocol implementation community, several approaches verify strong isolation guarantees for kernels and hypervisors. E.g., Murray *et al.* [140] prove information flow security for the seL4 microkernel using Isabelle/HOL, while Li *et al.* [141] prove similar properties for a modified version of the Linux KVM hypervisor in Coq. Common to both approaches is that they prove isolation of the kernel and hypervisor with respect to arbitrary (untrusted) user-space programs and virtual machines, respectively. Such a proof is possible as kernels and hypervisors run in a higher privilege level than the untrusted

[140]: Murray et al. (2013), *seL4: From General Purpose to a Proof of Information Flow Enforcement*
[141]: Li et al. (2021), *A Secure and Formally Verified Linux KVM Hypervisor*

components, which is enforced by the underlying hardware. To enable verification, Li *et al.* [141] move all non-essential functionality of the original hypervisor into untrusted services, reducing the codebase from more than 2M to under 4000 LOC, which is in spirit similar to our manual decomposition of a codebase. Applying their approaches to our case studies is not possible because the Core does not run in a higher privilege level than the Application and, thus, is not sufficiently isolated from the Application, which necessitates statically analyzing the Application.

[141]: Li et al. (2021), *A Secure and Formally Verified Linux KVM Hypervisor*

Dynamic Verification. Several approaches employ dynamic checks at runtime to allow for partially verified codebases. Agten *et al.* [142] target single-threaded C code and generate runtime checks at the boundary between verified and unverified code to test that the verified code's specification holds. To detect violations of properties expressed in separation logic such as ownership (via permissions) and aliasing, this approach tracks the heap locations accessed by the verified codebase at runtime and computes cryptographic hashes thereover. It remains unclear whether these checks only at the boundary remain sufficient when targeting concurrent codebases or whether the runtime overhead increases further. To avoid tracking heap locations at runtime, Ho *et al.* [41] copy all heap data at this boundary to rule out aliasing.

[142]: Agten et al. (2015), *Sound Modular Verification of C Code Executing in an Unverified Context*

[41]: Ho et al. (2022), *Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations*

Gradual verification (e.g., [143, 144]) combines auto-active verification with dynamic checks but aims at helping the proof developer by allowing incomplete specifications. I.e., gradual verification enables incremental verification where each function's specification is extended over time to eventually obtain a fully specified and verified codebase. However, as long as a codebase is not fully specified and verified, gradual verification requires tracking heap locations at runtime, which results in noticeable runtime overhead.

[143]: Bader et al. (2018), *Gradual Program Verification*

[144]: Wise et al. (2020), *Gradual Verification of Recursive Heap Data Structures*

SCIO* [145] is an F* transpiler that injects dynamic checks not only at the boundary between verified and unverified code but also at call sites of I/O operations. While they can enforce access policies for I/O operations, it remains unclear how this approach extends to cryptographic message payloads. To be applicable in our context, we would need to dynamically check whether a message sent by our Application is indeed protocol-irrelevant and, thus, does not contain any secrets from the Core—not even in encrypted form. Like our work, SecRef* [146] considers the problem of verifying only a subset of a codebase due to the otherwise prohibitive proof effort. While they also allow pre- and postconditions at the boundary between verified and unverified code, they rely on dynamic checks to enforce these conditions for heap locations accessible by unverified code. For a verified component like our Core, this means that they check the entire invariant at runtime for each API call (which we do not), as they treat the unverified code as potentially modifying a Core instance's entire state. By targeting a single-threaded language (F*), SecRef* does not have to consider concurrent memory accesses (which we do).

[145]: Andrici et al. (2024), *Securing Verified IO Programs Against Unverified Code in F**

[146]: Andrici et al. (2025), *SecRef*: Securely Sharing Mutable References Between Verified and Unverified Code in F**

By contrast, Diodon performs only extremely lightweight dynamic checks enforcing non-nilness and correct ordering of API calls, and checks all other constraints *statically* to avoid runtime overhead while simultaneously requiring minimal code changes.

Conclusions 5

In this dissertation, we have presented two methodologies for the verification of security protocol implementations and a sound combination with static analyses to scale them to large codebases. Our methodologies and our program verifier Gobra contribute to making practically deployed implementations safe *and* secure by enabling the proof of strong security properties for implementations that are *actually* executed by users around the globe to communicate securely with online services—as opposed to models or crafted reference implementations commonly used in related work.

If an abstract protocol model exists, the refinement-based methodology from Chapter 2 exploits the automation offered by Tamarin, a state-of-the-art protocol model verifier, to produce protocol-role-specific specifications describing permitted I/O operations. This novel link allows us to independently verify an implementation against the specification of a protocol role using off-the-shelf program verifiers. One challenge we overcame is bridging the gap between symbolic terms, arising in the model, and concrete data types like byte arrays used in implementations. Successful verification establishes a trace inclusion relation between the verified implementation and the corresponding role in the protocol model, thus, implying that the implementation satisfies the same security properties as the protocol model. Furthermore, the program verifier's proof encompasses a safety proof, i.e., that an implementation is free of crashes, undefined behavior, and memory errors, which rules out a large class of implementation-level errors such as buffer overflows. The work in Chapter 2 had an impact on Morio and Künnemann's SpecMon [95], which adopts our decomposition of a Tamarin model for runtime monitoring.

[95]: Morio et al. (2024), *SpecMon: Modular Black-Box Runtime Monitoring of Security Protocols*

If an accurate, abstract protocol model does not exist or the trust assumptions and number of different formalisms should be minimized, our invariant-based methodology (Chapter 3) proves security properties directly on the level of implementations. Since we deal with multiple implementations for different protocol roles that, at runtime, execute concurrently and communicate over an untrusted network with each other, we prove each implementation against a trace invariant, which captures the proof-relevant behavior of all communication partners. More specifically, we model this distributed system as a concurrent program with threads, in which each thread represents such an implementation or the attacker. Building on established verification techniques for concurrency reasoning, we prove that every trace-relevant operation maintains the trace invariant—without requiring particularly structured code (so-called coding disciplines) as assumed by related work. As we use a trace invariant in separation logic, our methodology supports proving strong security properties including injective agreement—a first for invariant-based approaches. Our parameterized, reusable verification library exploits commonalities between security protocols to reduce the per-protocol proof effort and annotation overhead.

Finally, Chapter 4 presents a novel methodology to soundly reduce the auto-active verification effort and, thus, annotation overhead of the two aforementioned methodologies as, without Diodon, both these methodologies require an auto-active proof spanning the *entire* codebase.

Diodon's high-level idea is to focus expressive but laborious proof techniques on small, security-critical parts of a codebase, which implement a security protocol. These parts require expressive proof techniques like auto-active verification as security depends, e.g., on the precise payloads of sent messages. However, all other parts of a codebase cannot be ignored as they may violate security properties that would otherwise hold for the security-critical parts. We identify and check I/O independence, which enables this partitioning of the codebase and, furthermore, results in more concise protocol models. In addition to checking I/O independence, we apply a novel combination of fully-automatic static analyses to the entire codebase ensuring that the entire codebase satisfies the assumptions made by the security-critical parts. We carefully pick a class of supported assumptions that are amenable to static analyses as the proof effort needed to apply static analyses scales to large codebases, although at the cost of less precise (but still sound) results compared to auto-active program verification. Since static analyses and auto-active program verification use vastly different formalisms, we present a blueprint in Diodon's soudness proof that lets us combine their respective guarantees.

While we have successfully applied all our methodologies to real-world codebases using heap-manipulating programming languages and proved strong security properties including forward secrecy and injective agreement, there are several avenues for future work.

Stronger Security Properties. We have focused on proving commonly used security properties. However, with the raise of post-quantum cryptography, stronger attacker models and properties guaranteeing protection against these stronger attackers are being proposed.

[17]: Linker et al. (2025), *A Formal Analysis of Apple's iMessage PQ3 Protocol*

Recent work by Linker *et al.* [17] has shown that certain post-quantum attackers can be encoded in the symbolic model of cryptography, more precisely in Tamarin. While it is likely that our refinement-based methodology is immediately applicable, it would be interesting to explore how to extend our invariant-based methodology, possibly by making the entire attacker model or at least some of the attacker's capabilities parametric. Going even further, we could consider a probabilistic polynomial-time attacker as used in the computational model of cryptography. In this attacker model, we could treat programs probabilistically, e.g., using dedicated program verifiers like Caesar [147], or devise a possibilistic verification approach that provides probabilistic guarantees. Owl [51] demonstrates that the latter approach is feasible by devising a type checker that guarantees that a security property holds with overwhelming probability, however, without providing concrete bounds on the probabilities. The latter approach is attractive as it would allow us to reuse existing program verifiers like Gobra and to target existing implementations.

[147]: Schröer et al. (2023), *A Deductive Verification Infrastructure for Probabilistic Programs*

[51]: Gancher et al. (2023), *Owl: Compositional Verification of Security Protocols via an Information-Flow Type System*

Staying in the DY attacker model, Sec. 3.9 hinted at stronger guarantees in the context of secure deletion. While our current approach does not store a protocol session's current epoch on the trace but stores it locally in the respective verification library's instance, future work could investigate which security properties can be proven if the current epoch is stored on the global trace. In particular, storing the current epoch on the trace provides two benefits. First, we can allow multiple library instances per protocol session as the authority to transition to the next epoch could get shared between these instances. Second, as a protocol session's current epoch is stored on the trace, this epoch becomes logically accessible to, e.g., instances of other protocol roles. More specifically, a trace invariant can refer to the current epoch of, e.g., a message's sender, which allows us to prove stronger security properties, namely secrecy labels that refer to an epoch of another protocol role instance.

Privacy Properties. All our methodologies target security properties, each stating a trace property, and we prove that every possible trace of executing a security protocol's implementations satisfies a given security property; i.e., a security property is a proposition that depends on an *individual* trace. In contrast, privacy properties are typically defined as an equivalence relation between two or more traces, which is not a trace property. E.g., anonymity mandates that the security protocol does not reveal the identities of involved protocol participants, which is usually formalized as the attacker's inability to distinguish between two scenarios in which the security protocol is executed by protocol participants with different identities. While several advances have been made in the context of protocol model verifiers to verify such properties, as surveyed by Delaune and Hirschi [148], we are not aware of any work that verifies privacy properties for *implementations* of security protocols. Straightforwardly applying our refinement-based methodology to utilize Tamarin's support for observational equivalence [149] is unfortunately not possible as observational equivalence is not preserved by refinement. In particular, refinement guarantees only trace *inclusion* and not trace equivalence, which means that a protocol model may describe more traces than actually possible by executing the implementations. Thus, ongoing work extends our invariant-based methodology, where the key challenge lies in *modularly* proving the existence of particular traces satisfying the equivalence relation.

[148]: Delaune et al. (2017), *A Survey of Symbolic Methods for Establishing Equivalence-Based Properties in Cryptographic Protocols*

[149]: Basin et al. (2015), *Automated Symbolic Proofs of Observational Equivalence*

Extending Diodon. Diodon, as described in Chapter 4, is a first take at soundly combining proof systems of different expressive power, namely auto-active verification and static analyses. The employed static analyses and the supported class of specifications for Core methods that an Application may call was driven by the availability of static analyses in Go and the production codebase we used to evaluate Diodon. While Diodon is a solid first step, it provides several exciting directions for future work.

First, we can implement the remaining static analyses, increase precision of existing static analyses for instance by making more of them context-sensitive, and extend Diodon with further static analyses to enable stronger specifications for Core methods. E.g., a sound analysis for nilness would broaden the supported class of specifications as such an analysis could guarantee non-`nil` arguments in calls to Core methods, which would allow us to strengthen their preconditions and, thus, we could drop the `nil` checks in the Core.

Second, since Diodon is not inherently limited to Go, future work could apply it to codebases written in other programming languages. Rust would be particularly interesting as its strong type system would allow us to drop several static analyses and assumptions like data race freedom.

Third, the Application's relation to the attacker model could be explored more formally. Currently, we assume in the soundness proof that the Application performs only operations that are abstractable to operations represented by the so-called protocol-independent message deduction rules (cf. Def. 4.4.2) and prove that these operations refine the attacker model. Since I/O independence guarantees that the Application does not send tainted data, i.e., data that depends on protocol secrets, it would be interesting to embed I/O independence into these message deduction rules. E.g., the message deduction rules could explicitly track taint and constrain send operations to send only untainted data. This treatment could foster a deeper understanding of the boundary between the Core and Application and, thus, provide a solid foundation for the proof obligations concerning virtual I/O operations.

Finally, we see applications of Diodon outside the context of security protocol implementations. It is common for program verification efforts to axiomatize libraries, i.e., proof engineers equip library functions with *trusted* specifications that are used to verify client code. These specifications are sound if they suffice to verify the library functions' implementations. However, due to the high proof effort, library functions often remain unverified. Therefore, it would be interesting to adapt Diodon to these library functions. I.e., we could execute static analyses on the implementations of library functions to increase trust in the correctness of their specifications.

Bibliography

[1] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. "Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS". In: *S&P*. IEEE, 2014, pp. 98–113. DOI: 10.1109/SP.2014.14 (cited on page 1).

[2] CVE. *CVE-2014-0160*. 2013. URL: https://www.cve.org/CVERecord?id=CVE-2014-0160 (cited on page 1).

[3] CVE. *CVE-2014-1266*. 2014. URL: https://www.cve.org/CVERecord?id=CVE-2014-1266 (cited on page 1).

[4] CVE. *CVE-2022-22805*. 2022. URL: https://www.cve.org/CVERecord?id=CVE-2022-22805 (cited on page 1).

[5] CVE. *CVE-2022-22806*. 2022. URL: https://www.cve.org/CVERecord?id=CVE-2022-22806 (cited on page 1).

[6] CVE. *CVE-2021-40823*. 2021. URL: https://www.cve.org/CVERecord?id=CVE-2021-40823 (cited on page 2).

[7] Benedikt Schmidt, Simon Meier, Cas Cremers, and David A. Basin. "Automated Analysis of Diffie–Hellman Protocols and Advanced Security Properties". In: *CSF*. IEEE, 2012, pp. 78–94. DOI: 10.1109/CSF.2012.25 (cited on pages 2, 15, 102).

[8] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. "The TAMARIN Prover for the Symbolic Analysis of Security Protocols". In: *CAV*. Vol. 8044. LNCS. Springer, 2013, pp. 696–701. DOI: 10.1007/978-3-642-39799-8_48 (cited on pages 2, 15, 102).

[9] Bruno Blanchet. "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules". In: *CSFW*. IEEE, 2001, pp. 82–96. DOI: 10.1109/CSFW.2001.930138 (cited on page 2).

[10] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication". In: *S&P*. IEEE Computer Society, 2016, pp. 470–485. DOI: 10.1109/SP.2016.35 (cited on pages 2, 7).

[11] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. "A Comprehensive Symbolic Analysis of TLS 1.3". In: *CCS*. ACM, 2017, pp. 1773–1788. DOI: 10.1145/3133956.3134063 (cited on pages 2, 7, 69).

[12] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate". In: *S&P*. IEEE, 2017, pp. 483–502. DOI: 10.1109/SP.2017.26 (cited on pages 2, 6, 7, 45, 152).

[13] David A. Basin, Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, Ralf Sasse, and Vincent Stettler. "A Formal Analysis of 5G Authentication". In: *CCS*. ACM, 2018, pp. 1383–1396. DOI: 10.1145/3243734.3243846 (cited on pages 2, 7).

[14] David A. Basin, Ralf Sasse, and Jorge Toro-Pozo. "The EMV Standard: Break, Fix, Verify". In: *S&P*. IEEE, 2021, pp. 1766–1781. DOI: 10.1109/SP40001.2021.00037 (cited on page 2).

[15] David A. Basin, Ralf Sasse, and Jorge Toro-Pozo. "Card Brand Mixup Attack: Bypassing the PIN in non-Visa Cards by Using Them for Visa Transactions". In: *USENIX Security Symposium*. USENIX Association, 2021, pp. 179–194 (cited on page 2).

[16] David A. Basin, Xenia Hofmeier, Ralf Sasse, and Jorge Toro-Pozo. "Getting Chip Card Payments Right". In: *FM (1)*. Vol. 14933. LNCS. Springer, 2024, pp. 29–51. DOI: 10.1007/978-3-031-71162-6_2 (cited on page 2).

[17] Felix Linker, Ralf Sasse, and David A. Basin. "A Formal Analysis of Apple's iMessage PQ3 Protocol". In: *USENIX Security Symposium*. USENIX Association, 2025, pp. 5015–5034 (cited on pages 2, 156).

[18] Danny Dolev and Andrew Chi-Chih Yao. "On the Security of Public Key Protocols". In: *IEEE Trans. Inf. Theory* 29.2 (1983), pp. 198–207. DOI: 10.1109/TIT.1983.1056650 (cited on pages 2, 15, 49).

[19] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. "SoK: Computer-Aided Cryptography". In: *S&P*. IEEE, 2021, pp. 777–795. DOI: 10.1109/SP40001.2021.00008 (cited on pages 2, 96).

[20] Bruno Blanchet. “A Computationally Sound Mechanized Prover for Security Protocols”. In: *S&P*. IEEE Computer Society, 2006, pp. 140–154. doi: 10.1109/SP.2006.1 (cited on page 2).

[21] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. “Computer-Aided Security Proofs for the Working Cryptographer”. In: *CRYPTO*. Vol. 6841. LNCS. Springer, 2011, pp. 71–90. doi: 10.1007/978-3-642-22792-9_5 (cited on page 2).

[22] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Catalin Hritcu, Kenji Maillard, and Bas Spitters. “SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq”. In: *CSF*. IEEE, 2021, pp. 1–15. doi: 10.1109/CSF51468.2021.00048 (cited on page 2).

[23] Bruno Blanchet. “Security Protocol Verification: Symbolic and Computational Models”. In: *POST*. Vol. 7215. LNCS. Springer, 2012, pp. 3–29. doi: 10.1007/978-3-642-28641-4_2 (cited on pages 2, 96).

[24] Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. “Local Reasoning about Programs that Alter Data Structures”. In: *CSL*. Vol. 2142. LNCS. Springer, 2001, pp. 1–19. doi: 10.1007/3-540-44802-0_1 (cited on pages 4, 49, 52).

[25] John C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures”. In: *LICS*. IEEE, 2002, pp. 55–74. doi: 10.1109/LICS.2002.1029817 (cited on pages 4, 13, 49, 52, 102, 121).

[26] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. “VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java”. In: *NASA Formal Methods*. Vol. 6617. LNCS. Springer, 2011, pp. 41–55. doi: 10.1007/978-3-642-20398-5_4 (cited on pages 4, 14, 49, 50, 52, 61, 95, 105).

[27] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. “VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs”. In: *J. Autom. Reason.* 61.1-4 (2018), pp. 367–422. doi: 10.1007/S10817-018-9457-5 (cited on pages 4, 71).

[28] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. “RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types”. In: *PLDI*. ACM, 2021, pp. 158–174. doi: 10.1145/3453483.3454036 (cited on page 4).

[29] Stefan Blom and Marieke Huisman. “The VerCors Tool for Verification of Concurrent Programs”. In: *FM*. Vol. 8442. LNCS. Springer, 2014, pp. 127–131. doi: 10.1007/978-3-319-06410-9_9 (cited on pages 4, 49).

[30] Marco Eilers and Peter Müller. “Nagini: A Static Verifier for Python”. In: *CAV (1)*. Vol. 10981. LNCS. Springer, 2018, pp. 596–603. doi: 10.1007/978-3-319-96145-3_33 (cited on pages 4, 14, 105).

[31] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. “Verus: Verifying Rust Programs using Linear Ghost Types”. In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023), pp. 286–315. doi: 10.1145/3586037 (cited on page 4).

[32] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. “Leveraging Rust Types for Modular Specification and Verification”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 147:1–147:30. doi: 10.1145/3360573 (cited on pages 4, 49, 52, 105).

[33] K. Rustan M. Leino and Michał Moskal. “Usable Auto-Active Verification”. In: *Usable Verification Workshop*. 2010 (cited on pages 4, 101).

[34] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. “Gobra: Modular Specification and Verification of Go Programs”. In: *CAV*. Vol. 12759. LNCS. Springer, 2021, pp. 367–379. doi: 10.1007/978-3-030-81685-8_17 (cited on pages 4, 12, 14, 49, 50, 52, 61, 71, 102, 105).

[35] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. “Viper: A Verification Infrastructure for Permission-Based Reasoning”. In: *VMCAI*. Vol. 9583. LNCS. Springer, 2016, pp. 41–62. doi: 10.1007/978-3-662-49122-5_2 (cited on page 4).

[36] João C. Pereira, Tobias Klenze, Sofia Giampietro, Markus Limbeck, Dionysios Spiliopoulos, Felix Wolf, Marco Eilers, Christoph Sprenger, David A. Basin, Peter Müller, and Adrian Perrig. “Protocols to Code: Formal Verification of a Secure Next-Generation Internet Router”. In: *CCS*. ACM, 2025, pp. 1469–1483. doi: 10.1145/3719027.3765104 (cited on page 5).

[37] Jason A. Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel". In: *NDSS*. The Internet Society, 2017 (cited on pages 5, 50).

[38] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. "Formal Verification of Security Protocol Implementations: a Survey". In: *Formal Aspects Comput.* 26.1 (2014), pp. 99–123. DOI: 10.1007/S00165-012-0269-9 (cited on pages 5, 96).

[39] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. "DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code". In: *EuroS&P*. IEEE, 2021, pp. 523–542. DOI: 10.1109/EUROSP51992.2021.00042 (cited on pages 5, 7, 46, 50, 53, 55, 60, 86, 95, 152, 169).

[40] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. "An In-Depth Symbolic Security Analysis of the ACME Standard". In: *CCS*. ACM, 2021, pp. 2601–2617. DOI: 10.1145/3460120.3484588 (cited on pages 5, 96).

[41] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. "Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations". In: *S&P*. IEEE, 2022, pp. 107–124. DOI: 10.1109/SP46214.2022.9833621 (cited on pages 5, 69, 96, 153).

[42] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. "Dependent Types and Multi-Monadic Effects in F*". In: *POPL*. ACM, 2016, pp. 256–270. DOI: 10.1145/2837614.2837655 (cited on pages 5, 95).

[43] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. "Verified Low-Level Programming Embedded in F". In: *Proc. ACM Program. Lang.* 1.ICFP (2017), 17:1–17:29. DOI: 10.1145/3110261 (cited on page 5).

[44] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. "Formally Verified Cryptographic Web Applications in WebAssembly". In: *S&P*. IEEE, 2019, pp. 1256–1274. DOI: 10.1109/SP.2019.00064 (cited on page 5).

[45] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. "TreeSync: Authenticated Group Management for Messaging Layer Security". In: *USENIX Security Symposium*. USENIX Association, 2023, pp. 1217–1233 (cited on pages 6, 7).

[46] Davide Pozza, Riccardo Sisto, and Luca Durante. "Spi2Java: Automatic Cryptographic Protocol Java Code Generation from Spi Calculus". In: *AINA*. IEEE, 2004, pp. 400–405. DOI: 10.1109/AINA.2004.1283943 (cited on pages 6, 152).

[47] David Cadé and Bruno Blanchet. "From Computationally-Proved Protocol Specifications to Implementations". In: *ARES*. IEEE, 2012, pp. 65–74. DOI: 10.1109/ARES.2012.63 (cited on pages 6, 152).

[48] David Cadé and Bruno Blanchet. "Proved Generation of Implementations from Computationally Secure Protocol Specifications". In: *POST*. Vol. 7796. LNCS. Springer, 2013, pp. 63–82. DOI: 10.1007/978-3-642-36830-1_4 (cited on page 6).

[49] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. "Implementing and Proving the TLS 1.3 Record Layer". In: *S&P*. IEEE, 2017, pp. 463–482. DOI: 10.1109/SP.2017.58 (cited on page 6).

[50] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. "A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer". In: *S&P*. IEEE, 2021, pp. 1162–1178. DOI: 10.1109/SP40001.2021.00039 (cited on page 6).

[51] Joshua Gancher, Sydney Gibson, Pratap Singh, Samvid Dharanikota, and Bryan Parno. "Owl: Compositional Verification of Security Protocols via an Information-Flow Type System". In: *S&P*. IEEE, 2023, pp. 1130–1147. DOI: 10.1109/SP46215.2023.10179477 (cited on pages 6, 46, 96, 152, 156).

[52] Pratap Singh, Joshua Gancher, and Bryan Parno. "OwlC: Compiling Security Protocols to Verified, Secure, High-Performance Libraries". In: *USENIX Security Symposium*. USENIX Association, 2025, pp. 5071–5090 (cited on pages 6, 46, 96, 152).

[53] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. "Verified Interoperable Implementations of Security Protocols". In: *ACM Trans. Program. Lang. Syst.* 31.1 (2008), 5:1–5:61. DOI: 10.1145/1452044.1452049 (cited on pages 6, 45, 152).

[54] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zalinescu. "Verified Cryptographic Implementations for TLS". In: *ACM Trans. Inf. Syst. Secur.* 15.1 (2012), 3:1–3:32. DOI: 10.1145/2133375.2133378 (cited on pages 6, 45).

[55] Karthikeyan Bhargavan, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. "Formal Security and Functional Verification of Cryptographic Protocol Implementations in Rust". In: *CCS*. ACM, 2025, pp. 2729–2743. DOI: 10.1145/3719027.3765213 (cited on page 6).

[56] Nicholas O'Shea. "Using Elyjah to Analyse Java Implementations of Cryptographic Protocols". In: *FCS-ARSPA-WITS-2008*. 2008 (cited on pages 6, 152).

[57] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. "Computational Verification of C Protocol Implementations by Symbolic Execution". In: *CCS*. ACM, 2012, pp. 712–723. DOI: 10.1145/2382196.2382271 (cited on pages 6, 152).

[58] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. "Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach". In: *EuroS&P*. IEEE, 2017, pp. 435–450. DOI: 10.1109/EUROSP.2017.38 (cited on pages 6, 152).

[59] Faezeh Nasrabadi, Robert Künnemann, and Hamed Nemati. "CryptoBap: A Binary Analysis Platform for Cryptographic Protocols". In: *CCS*. ACM, 2023, pp. 1362–1376. DOI: 10.1145/3576915.3623090 (cited on pages 6, 7, 46).

[60] Faezeh Nasrabadi, Robert Künnemann, and Hamed Nemati. "Symbolic Parallel Composition for Multi-Language Protocol Verification". In: *CSF*. IEEE, 2025, pp. 378–393. DOI: 10.1109/CSF64896.2025.00030 (cited on pages 6, 7, 46, 47).

[61] Paolo Modesti. "AnBx: Automatic Generation and Verification of Security Protocols Implementations". In: *FPS*. Vol. 9482. LNCS. Springer, 2015, pp. 156–173. DOI: 10.1007/978-3-319-30303-1_10 (cited on page 7).

[62] Omar Almousa, Sebastian Mödersheim, and Luca Viganò. "Alice and Bob: Reconciling Formal Models and Implementation". In: *Programming Languages with Applications to Biology and Security*. Vol. 9465. LNCS. Springer, 2015, pp. 66–85. DOI: 10.1007/978-3-319-25527-9_7 (cited on pages 7, 46).

[63] Riccardo Sisto, Piergiuseppe Bettassa Copet, Matteo Avalle, and Alfredo Pironti. "Formally Sound Implementations of Security Protocols with JavaSPI". In: *Formal Aspects Comput.* 30.2 (2018), pp. 279–317. DOI: 10.1007/S00165-017-0449-8 (cited on pages 7, 46).

[64] François Dupressoir, Andrew D. Gordon, Jan Jürjens, and David A. Naumann. "Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols". In: *CSF*. IEEE, 2011, pp. 3–17. DOI: 10.1109/CSF.2011.8 (cited on pages 7, 34, 46, 50, 95, 152).

[65] François Dupressoir, Andrew D. Gordon, Jan Jürjens, and David A. Naumann. "Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols". In: *J. Comput. Secur.* 22.5 (2014), pp. 823–866. DOI: 10.3233/JCS-140508 (cited on pages 7, 34, 46).

[66] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. "VCC: A Practical System for Verifying Concurrent C". In: *TPHOLs*. Vol. 5674. LNCS. Springer, 2009, pp. 23–42. DOI: 10.1007/978-3-642-03359-9_2 (cited on pages 7, 50, 95).

[67] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. "Modular Verification of Security Protocol Code by Typing". In: *POPL*. ACM, 2010, pp. 445–456. DOI: 10.1145/1706299.1706350 (cited on pages 7, 46, 60, 96).

[68] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. "Refinement Types for Secure Implementations". In: *CSF*. IEEE, 2008, pp. 17–32. DOI: 10.1109/CSF.2008.27 (cited on pages 7, 46, 96).

[69] Gijs Vanspauwen and Bart Jacobs. "Verifying Protocol Implementations by Augmenting Existing Cryptographic Libraries with Specifications". In: *SEFM*. Vol. 9276. LNCS. Springer, 2015, pp. 53–68. DOI: 10.1007/978-3-319-22969-0_4 (cited on pages 7, 34, 46, 47, 95).

[70] Gijs Vanspauwen and Bart Jacobs. *Verifying Cryptographic Protocol Implementations that use Industrial Cryptographic APIs*. Tech. rep. Department of Computer Science, KU Leuven, Belgium, 2017 (cited on pages 7, 34, 46).

[71] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David A. Basin. "Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 152:1–152:31. DOI: 10.1145/3428220 (cited on pages 7, 14, 25, 29, 34, 36, 47, 53, 121).

[72] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. "TreeKEM: A Modular Machine-Checked Symbolic Security Analysis of Group Key Agreement in Messaging Layer Security". In: *S&P*. IEEE, 2025, pp. 4375–4390. DOI: 10.1109/SP61157.2025.00228 (cited on page 7).

[73] Linard Arquint, Felix A. Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N. Wiesner, David A. Basin, and Peter Müller. "Sound Verification of Security Protocols: From Design to Interoperable Implementations". In: *S&P*. IEEE, 2023, pp. 1077–1093. DOI: 10.1109/SP46215.2023.10179325 (cited on pages 11, 113).

[74] Linard Arquint, Malte Schwerhoff, Vaibhav Mehta, and Peter Müller. "A Generic Methodology for the Modular Verification of Security Protocol Implementations". In: *CCS*. ACM, 2023, pp. 1377–1391. DOI: 10.1145/3576915.3623105 (cited on pages 11, 12).

[75] Linard Arquint, Samarth Kishor, Jason Koenig, and Joey Dodds. "A Method for Extending Safety Proofs of Protocols to Industrial Applications". U.S. pat. 18901581. Amazon Web Services, Inc. 2024. pending (cited on page 11).

[76] Linard Arquint, Samarth Kishor, Jason R. Koenig, Joey Dodds, Daniel Kroening, and Peter Müller. "The Secrets Must Not Flow: Scaling Security Verification to Large Codebases". In: *S&P*. To appear. IEEE, 2026, pp. 492–511. DOI: 10.1109/SP63933.2026.00026 (cited on page 11).

[77] Lasse Meinen. "Verification^2 of the Authentic Digital EMblem". MA thesis. ETH Zurich, 2024. DOI: 10.3929/ethz-b-000678729 (cited on pages 12, 44).

[78] Hugo Queinnec. "Secure Deletion of Sensitive Data in Protocol Implementations". MA thesis. ETH Zurich, 2023. DOI: 10.3929/ethz-b-000641727 (cited on pages 12, 92).

[79] Gavin Lowe. "A Hierarchy of Authentication Specification". In: *CSFW*. IEEE, 1997, pp. 31–44. DOI: 10.1109/CSFW.1997.596782 (cited on pages 16, 58).

[80] Willem Penninckx, Bart Jacobs, and Frank Piessens. "Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs". In: *ESOP*. Vol. 9032. LNCS. Springer, 2015, pp. 158–182. DOI: 10.1007/978-3-662-46669-8_7 (cited on pages 17, 26, 27, 29, 47).

[81] Linard Arquint, Felix A. Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N. Wiesner, David A. Basin, and Peter Müller. "Sound Verification of Security Protocols: From Design to Interoperable Implementations (extended version)". In: *CoRR* abs/2212.04171 (2022). DOI: 10.48550/ARXIV.2212.04171 (cited on pages 19, 20, 114, 115, 117).

[82] Sebastian Mödersheim and Georgios Katsoris. "A Sound Abstraction of the Parsing Problem". In: *CSF*. IEEE, 2014, pp. 259–273. DOI: 10.1109/CSF.2014.26 (cited on pages 20, 34, 47).

[83] Linard Arquint, Felix A. Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N. Wiesner, David A. Basin, and Peter Müller. *Sound Verification of Security Protocols: From Design to Interoperable Implementations*. Tamarin model & verified Go implementation of the WireGuard VPN key exchange protocol. 2022. DOI: 10.5281/zenodo.7409524 (cited on page 26).

[84] Linard Arquint, Felix A. Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N. Wiesner, David A. Basin, and Peter Müller. *Sound Verification of Security Protocols: From Design to Interoperable Implementations*. GitHub Repository. 2022. URL: https://github.com/viperproject/protocol-verification-refinement (cited on page 26).

[85] Cas Cremers, Charlie Jacomme, and Philip Lukert. "Subterm-Based Proof Techniques for Improving the Automation and Scope of Security Protocol Analysis". In: *CSF*. IEEE, 2023, pp. 200–213. DOI: 10.1109/CSF57540.2023.00001 (cited on page 31).

[86] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. "EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats". In: *USENIX Security Symposium*. USENIX Association, 2019, pp. 1465–1482 (cited on pages 34, 47).

[87] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. “Comparse: Provably Secure Formats for Cryptographic Protocols”. In: *CCS*. ACM, 2023, pp. 564–578. DOI: 10.1145/3576915.3623201 (cited on pages 34, 47).

[88] Jason A. Donenfeld. *Go Implementation of WireGuard*. Git Repository. [Online; accessed 11-March-2021]. URL: https://git.zx2c4.com/wireguard-go (cited on page 40).

[89] Felix Linker and David A. Basin. “ADEM: An Authentic Digital EMblem”. In: *CCS*. ACM, 2023, pp. 2815–2829. DOI: 10.1145/3576915.3616578 (cited on pages 41, 42, 45).

[90] An Authentic Digital Emblem - GitHub Working Group. *ADEM Prototypes*. GitHub Repository. 2025. URL: https://github.com/adem-wg/adem-proto (cited on pages 42, 43).

[91] Michele Bugliesi, Stefano Calzavara, Sebastian Mödersheim, and Paolo Modesti. “Security Protocol Specification and Verification with AnBx”. In: *J. Inf. Secur. Appl.* 30 (2016), pp. 46–63. DOI: 10.1016/J.JISA.2016.05.004 (cited on page 46).

[92] Andreas Lindner, Roberto Guanciale, and Roberto Metere. “TrABin: Trustworthy Analyses of Binaries”. In: *Sci. Comput. Program.* 174 (2019), pp. 72–89. DOI: 10.1016/J.SCICO.2019.01.001 (cited on page 46).

[93] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. “Interaction Trees: Representing Recursive and Impure Programs in Coq”. In: *Proc. ACM Program. Lang.* 4.POPL (2020), 51:1–51:32. DOI: 10.1145/3371119 (cited on page 46).

[94] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. “Hardening Attack Surfaces with Formally Proven Binary Format Parsers”. In: *PLDI*. ACM, 2022, pp. 31–45. DOI: 10.1145/3519939.3523708 (cited on page 47).

[95] Kevin Morio and Robert Künnemann. “SpecMon: Modular Black-Box Runtime Monitoring of Security Protocols”. In: *CCS*. ACM, 2024, pp. 2741–2755. DOI: 10.1145/3658644.3690197 (cited on pages 47, 155).

[96] José Fragoso Santos, Petar Maksimovic, Daiva Naudziuniene, Thomas Wood, and Philippa Gardner. “JaVerT: JavaScript Verification Toolchain”. In: *Proc. ACM Program. Lang.* 2.POPL (2018), 50:1–50:33. DOI: 10.1145/3158138 (cited on page 49).

[97] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. “The Spirit of Ghost Code”. In: *CAV*. Vol. 8559. LNCS. Springer, 2014, pp. 1–16. DOI: 10.1007/978-3-319-08867-9_1 (cited on pages 49, 169).

[98] Roger M. Needham and Michael D. Schroeder. “Using Encryption for Authentication in Large Networks of Computers”. In: *Commun. ACM* 21.12 (1978), pp. 993–999. DOI: 10.1145/359657.359659 (cited on page 50).

[99] Gavin Lowe. “Breaking and Fixing the Needham–Schroeder Public-Key Protocol Using FDR”. In: *TACAS*. Vol. 1055. LNCS. Springer, 1996, pp. 147–166. DOI: 10.1007/3-540-61042-1_43 (cited on pages 50, 53).

[100] Whitfield Diffie and Martin E. Hellman. “New Directions in Cryptography”. In: *IEEE Trans. Inf. Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638 (cited on page 50).

[101] Linard Arquint, Malte Schwerhoff, Vaibhav Mehta, and Peter Müller. *A Generic Methodology for the Modular Verification of Security Protocol Implementations*. Artifact containing the reusable verification libraries and the case studies. 2023. DOI: 10.5281/zenodo.8330913 (cited on pages 50, 61, 65).

[102] Andrew D. Gordon and Alan Jeffrey. “Authenticity by Typing for Security Protocols”. In: *CSFW*. IEEE Computer Society, 2001, pp. 145–159. DOI: 10.1109/CSFW.2001.930143 (cited on pages 55, 169).

[103] John Boyland. “Checking Interference with Fractional Permissions”. In: *SAS*. Vol. 2694. LNCS. Springer, 2003, pp. 55–72. DOI: 10.1007/3-540-44898-5_4 (cited on page 56).

[104] Matthew J. Parkinson and Gavin M. Bierman. “Separation Logic and Abstraction”. In: *POPL*. ACM, 2005, pp. 247–258. DOI: 10.1145/1040305.1040326 (cited on pages 64, 102).

[105] Benjamin Dowling and Kenneth G. Paterson. “A Cryptographic Analysis of the WireGuard Protocol”. In: *ACNS*. Vol. 10892. LNCS. Springer, 2018, pp. 3–21. DOI: 10.1007/978-3-319-93387-0_1 (cited on pages 66, 96).

[106] Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. “A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol”. In: *EuroS&P*. IEEE, 2019, pp. 231–246. DOI: 10.1109/EUROSP.2019.00026 (cited on pages 66, 96).

[107] Guillaume Girol, Lucca Hirschi, Ralf Sasse, Dennis Jackson, Cas Cremers, and David A. Basin. “A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie–Hellman Protocols”. In: *USENIX Security Symposium*. USENIX Association, 2020, pp. 1857–1874 (cited on pages 69, 94, 96).

[108] David A. Basin, Cas Cremers, and Marko Horvat. “Actor Key Compromise: Consequences and Countermeasures”. In: *CSF*. IEEE, 2014, pp. 244–258. DOI: 10.1109/CSF.2014.25 (cited on page 70).

[109] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella Béguelin. “EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider”. In: *S&P*. IEEE, 2020, pp. 983–1002. DOI: 10.1109/SP40000.2020.00114 (cited on pages 72, 113).

[110] Viktor Vafeiadis. “Concurrent Separation Logic and Operational Semantics”. In: *MFPS*. Vol. 276. Electronic Notes in Theoretical Computer Science. Elsevier, 2011, pp. 335–351. DOI: 10.1016/J.ENTCS.2011.09.029 (cited on pages 78, 121, 127).

[111] Moxie Marlinspike and Trevor Perrin. *The X3DH Key Agreement Protocol*. Revision 1. Nov. 4, 2016. URL: https://signal.org/docs/specifications/x3dh/x3dh.pdf (cited on pages 85, 96).

[112] Trevor Perrin and Moxie Marlinspike. *The Double Ratchet Algorithm*. Revision 1. Nov. 20, 2016. URL: https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf (cited on page 85).

[113] Santiago Arranz Olmos, Gilles Barthe, Ruben Gonzalez, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, and Peter Schwabe. “High-Assurance Zeroization”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.1 (2024), pp. 375–397. DOI: 10.46586/TCHES.V2024.I1.375-397 (cited on pages 85, 88).

[114] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. “Dead Store Elimination (Still) Considered Harmful”. In: *USENIX Security Symposium*. USENIX Association, 2017, pp. 1025–1040 (cited on pages 85, 88).

[115] Richard Bornat, Cristiano Calcagno, Peter W. O’Hearn, and Matthew J. Parkinson. “Permission Accounting in Separation Logic”. In: *POPL*. ACM, 2005, pp. 259–270. DOI: 10.1145/1040305.1040327 (cited on page 86).

[116] Véronique Cortier, Stéphanie Delaune, and Jannik Dreier. “Automatic Generation of Sources Lemmas in Tamarin: Towards Automatic Proofs of Security Protocols”. In: *ESORICS (2)*. Vol. 12309. LNCS. Springer, 2020, pp. 3–22. DOI: 10.1007/978-3-030-59013-0_1 (cited on page 93).

[117] Susan S. Owicki and David Gries. “Verifying Properties of Parallel Programs: An Axiomatic Approach”. In: *Commun. ACM* 19.5 (1976), pp. 279–285. DOI: 10.1145/360051.360224 (cited on page 94).

[118] David A. Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. *Modeling and Analyzing Security Protocols with Tamarin - A Comprehensive Guide*. Information Security and Cryptography. Springer, 2026 (cited on page 94).

[119] Nadia Polikarpova and Michal Moskal. “Verifying Implementations of Security Protocols by Refinement”. In: *VSTTE*. Vol. 7152. LNCS. Springer, 2012, pp. 50–65. DOI: 10.1007/978-3-642-27705-4_5 (cited on page 95).

[120] Karthikeyan Bhargavan, Cédric Fournet, and Nataliya Guts. “Typechecking Higher-Order Security Libraries”. In: *APLAS*. Vol. 6461. LNCS. Springer, 2010, pp. 47–62. DOI: 10.1007/978-3-642-17164-2_5 (cited on page 96).

[121] Ralf Küsters, Tomasz Truderung, and Juergen Graf. “A Framework for the Cryptographic Verification of Java-Like Programs”. In: *CSF*. IEEE, 2012, pp. 198–212. DOI: 10.1109/CSF.2012.9 (cited on page 96).

[122] Jean Goubault-Larrecq and Fabrice Parrennes. “Cryptographic Protocol Analysis on Real C Code”. In: *VMCAI*. Vol. 3385. LNCS. Springer, 2005, pp. 363–379. DOI: 10.1007/978-3-540-30579-8_24 (cited on page 96).

[123] Sagar Chaki and Anupam Datta. "ASPIER: An Automated Framework for Verifying Security Protocol Implementations". In: *CSF*. IEEE, 2009, pp. 172–185. DOI: 10.1109/CSF.2009.20 (cited on page 96).

[124] Jan Jürjens. "Security Analysis of Crypto-Based Java Programs using Automated Theorem Provers". In: *ASE*. IEEE, 2006, pp. 167–176. DOI: 10.1109/ASE.2006.60 (cited on page 96).

[125] Jason A. Donenfeld and Kevin Milner. *Formal Verification of the WireGuard Protocol*. June 7, 2018. URL: https://www.wireguard.com/papers/wireguard-formal-verification.pdf (cited on page 96).

[126] Nadim Kobeissi, Georgio Nicolas, and Karthikeyan Bhargavan. "Noise Explorer: Fully Automated Modeling and Verification for Arbitrary Noise Protocols". In: *EuroS&P*. IEEE, 2019, pp. 356–370. DOI: 10.1109/EUROSP.2019.00034 (cited on page 96).

[127] Joseph Tassarotti and Robert Harper. "A Separation Logic for Concurrent Randomized Programs". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 64:1–64:30. DOI: 10.1145/3290377 (cited on page 96).

[128] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. "Quantitative Separation Logic: A Logic for Reasoning about Probabilistic Pointer Programs". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 34:1–34:29. DOI: 10.1145/3290347 (cited on page 96).

[129] Gilles Barthe, Justin Hsu, and Kevin Liao. "A Probabilistic Separation Logic". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 55:1–55:30. DOI: 10.1145/3371123 (cited on page 96).

[130] CVE. *CVE-2024-47083*. 2024. URL: https://www.cve.org/CVERecord?id=CVE-2024-47083 (cited on page 99).

[131] CVE. *CVE-2023-6746*. 2023. URL: https://www.cve.org/CVERecord?id=CVE-2023-6746 (cited on page 99).

[132] Amazon Web Services, Inc. *Working with SSM Agent*. 2023. URL: https://docs.aws.amazon.com/systems-manager/latest/userguide/ssm-agent.html (cited on pages 101, 143, 144).

[133] Linard Arquint, Samarth Kishor, Jason R. Koenig, Joey Dodds, Daniel Kroening, and Peter Müller. *The Secrets Must Not Flow: Scaling Security Verification to Large Codebases (artifact)*. Artifact containing the protocol models, the forked SSM Agent's codebase, a DH implementation codebase, and the static analysis tools. 2025. DOI: 10.5281/zenodo.17099763 (cited on page 101).

[134] Linard Arquint, Samarth Kishor, Jason R. Koenig, Joey Dodds, Daniel Kroening, and Peter Müller. *The Secrets Must Not Flow: Scaling Security Verification to Large Codebases*. GitHub Repository. Artifact repository containing the protocol models, the forked SSM Agent's codebase, a DH implementation codebase, and the static analysis tools. 2025. URL: https://github.com/viperproject/diodon-artifact (cited on page 101).

[135] AWS Labs. *Argot*. 2024. URL: https://github.com/awslabs/ar-go-tools (cited on pages 102, 144).

[136] Google. *Capslock*. 2024. URL: https://github.com/google/capslock (cited on pages 107, 113).

[137] AbsInt Angewandte Informatik GmbH. *Astrée Static Analyzer for C and C++*. 2025. URL: https://www.absint.com/astree (cited on page 113).

[138] Go developers. *The Go Memory Model*. 2022. URL: https://go.dev/ref/mem (cited on page 113).

[139] Amazon Web Services, Inc. *AWS Key Management Service*. 2024. URL: https://aws.amazon.com/kms/ (cited on page 144).

[140] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. "seL4: From General Purpose to a Proof of Information Flow Enforcement". In: *S&P*. IEEE, 2013, pp. 415–429. DOI: 10.1109/SP.2013.35 (cited on page 152).

[141] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. "A Secure and Formally Verified Linux KVM Hypervisor". In: *S&P*. IEEE, 2021, pp. 1782–1799. DOI: 10.1109/SP40001.2021.00049 (cited on pages 152, 153).

[142] Pieter Agten, Bart Jacobs, and Frank Piessens. "Sound Modular Verification of C Code Executing in an Unverified Context". In: *POPL*. ACM, 2015, pp. 581–594. DOI: 10.1145/2676726.2676972 (cited on page 153).

[143] Johannes Bader, Jonathan Aldrich, and Éric Tanter. "Gradual Program Verification". In: *VMCAI*. Vol. 10747. Lecture Notes in Computer Science. Springer, 2018, pp. 25–46. DOI: 10.1007/978-3-319-73721-8_2 (cited on page 153).

[144] Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. “Gradual Verification of Recursive Heap Data Structures”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 228:1–228:28. doi: 10.1145/3428296 (cited on page 153).

[145] Cezar-Constantin Andrici, Ştefan Ciobâcă, Catalin Hritcu, Guido Martínez, Exequiel Rivas, Éric Tanter, and Théo Winterhalter. “Securing Verified IO Programs Against Unverified Code in F*”. In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 2226–2259. doi: 10.1145/3632916 (cited on page 153).

[146] Cezar-Constantin Andrici, Danel Ahman, Catalin Hritcu, Ruxandra Icleanu, Guido Martínez, Exequiel Rivas, and Théo Winterhalter. “SecRef*: Securely Sharing Mutable References Between Verified and Unverified Code in F*”. In: *CoRR* abs/2503.00404 (2025). doi: 10.48550/ARXIV.2503.00404 (cited on page 153).

[147] Philipp Schröer, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. “A Deductive Verification Infrastructure for Probabilistic Programs”. In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (2023), pp. 2052–2082. doi: 10.1145/3622870 (cited on page 156).

[148] Stéphanie Delaune and Lucca Hirschi. “A Survey of Symbolic Methods for Establishing Equivalence-Based Properties in Cryptographic Protocols”. In: *J. Log. Algebraic Methods Program.* 87 (2017), pp. 127–144. doi: 10.1016/J.JLAMP.2016.10.005 (cited on page 157).

[149] David A. Basin, Jannik Dreier, and Ralf Sasse. “Automated Symbolic Proofs of Observational Equivalence”. In: *CCS*. ACM, 2015, pp. 1144–1155. doi: 10.1145/2810103.2813662 (cited on page 157).

Glossary

AC associativity and commutativity. 33, 34

ADEM The Authentic Digital EMblem (ADEM) is an authentication mechanism for digital assets. 12, 43–45, 47, 94

ADT algebraic data type. 66

AEAD authenticated encryption with associated data. 40, 64, 93

AKC actor key compromise. 69, 71, 72

API application programming interface. 63, 64, 97, 104, 112–115, 124, 125, 134, 144, 145, 151, 154, 155

attacker completeness A property of trace invariants stating that a trace invariant remains valid under the full set of operations available to the attacker, as defined by the attacker model. This property is also known as *robust safety* [102] or *attacker typability* [39]. 58, 63, 64, 153

AWS Amazon Web Services. xi, 10, 12, 103, 104, 145, 146, 149

CA certificate authority. 44

CDS concurrent data structure. 63, 70

CT Certificate Transparency. 44, 45

DH Diffie–Hellman. 12, 17, 18, 39, 40, 52, 66–68, 73, 93–95, 101, 103, 104, 109, 145–147, 149, 151–153

DoS denial of service. 40, 42, 69

DY Dolev–Yao. 2, 17, 18, 35, 37, 51, 53, 103, 104, 108, 109, 114, 117, 119, 122, 158

equational theory A set of equations that is used to axiomatize the behavior of otherwise uninterpreted functions. 11, 17, 33, 34, 66, 68, 74, 75, 98

forward secrecy A property of security protocols stating that the compromise of long-term keys does not compromise past session keys. See Sec. 3.6.3 for details. 11, 51, 59, 69, 71–73, 87, 89, 93, 97–99, 101, 158

fractional permissions A common permission model in separation logic readily supported by many program verifiers. One or a full permission grants read and write access to a heap location, which is created at allocation time of this heap location. Any fraction strictly larger than zero grants read-only access. Separating conjunction adds up fractions of permissions to the same heap location. 58, 88, 90, 104, 113

functional property An implementation-specific property of a program describing its desired behavior, e.g., that a sorting algorithm's result is a sorted permutation of the input. 3, 55, 73, 74, 109, 113, 125, 126, 150

ghost code Auxiliary program code asserting intermediate properties and manipulating auxiliary program state to facilitate reasoning about a program. Ghost code does not affect a program's runtime behavior and is erased before compilation. See Filliâtre *et al.* [97] for details. 5, 11, 30–33, 37, 51, 56–59, 62, 64–67, 70, 73, 75, 77, 81, 88, 91, 95, 96, 99, 101, 126–129, 132, 134, 136–139, 142–145

goroutine A lightweight thread in the Go programming language. 5, 45–47, 68, 70, 71, 104, 105, 107

I/O independence A property of I/O operations stating that an I/O operation that is classified as protocol-irrelevant does not depend on any secret data involved in the protocol. In DIODON, we prove this property by performing a static taint analysis. 101–103, 106, 108, 109, 115, 143, 146, 148, 152–154, 158, 159

I/O separation logic A logic for reasoning about I/O behavior of programs. 19, 28, 49, 95

ICRC International Committee of the Red Cross. 44

IHL International Humanitarian Law. 44

injective agreement An authentication property of security protocols stating that two parties *injectively* agree on their identities and certain values. In contrast to non-injective agreement, this property additionally rules out replay attacks. See Sec. 3.4.1 for details. 10, 11, 51, 52, 60–62, 66, 68, 69, 71–73, 97–99, 101, 103, 146, 148, 157, 158

IP Internet Protocol. 69, 70

ITREE interaction tree. 48

KCI key compromise impersonation. 72

KDF key derivation function. 40, 73, 87, 92–94, 148

KMS Key Management Service. 146–148

LOC line of code. 28, 145, 148, 155
LOS line of specification. 73, 151
LTS labeled transition system. 17, 25, 27, 29, 30, 37, 49, 116

MAC message authentication code. 40, 104, 105
MITM Mallory-in-the-middle. 147, 152
MLS Messaging Layer Security. 6, 8
MSR multiset rewriting. 17, 18, 20–23, 25–28, 31, 41, 42, 50, 118, 123

non-injective agreement An authentication property of security protocols stating that two parties agree on their identities and certain values. This property is typically expressed as a correspondence between events on a trace. See Sec. 3.4.1 for details. 60, 61, 69, 73, 97, 98
NSL Needham–Schroeder–Lowe. 12, 52, 56, 58, 60–62, 67, 68, 72, 73, 87, 98

permission accounting An alternative permission model to fractional permissions in separation logic. This model is based on a factory resource from which so-called *shares* (another separation logic resource) can be split off. The factory resource keeps track of how many shares have been split off and allows to recollect shares. 88
PKI public key infrastructure. 44
post-compromise security A property of security protocols stating that the compromise of long-term keys does not compromise future session keys. 87, 89, 93, 98
protected party A party that is protected under International Humanitarian Law (IHL) in the context of ADEM. Examples include humanitarian organisations like UNICEF and Médecins Sans Frontières. 44, 45, 47

safety A property of programs guaranteeing that a program does cause neither runtime exceptions nor undefined behavior. In particular, it covers the absence of memory errors, which include buffer overflows, data races, use-after-free errors, and reading from uninitialized memory. 3, 4, 7, 42, 46, 52, 54, 55, 69, 73, 97, 98, 101, 109, 125, 149, 150, 152, 157
SDK software development kit. 2
separation logic A logic for reasoning about programs with mutable state.
SMT satisfiability modulo theories. 4, 66, 103
SSA static single assignment. 124, 128
SSH Secure Shell. 146
SSM Agent Systems Manager Agent. 103, 145, 146, 148, 149, 151, 152

TCP Transmission Control Protocol. 69
TLS Transport Layer Security. 1, 2, 7, 8, 47, 48

VMI association of scientific staff of the Department of Computer Science at ETH Zurich. xii
VPN Virtual Private Network. 5, 10, 16, 39, 42, 67–72, 96