DISS. ETH NO. 32317

FORMALLY VERIFYING PRE-EXISTING REAL-WORLD PROGRAMS

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES

(Dr. sc. ETH Zurich)

presented by

JOÃO CARLOS MENDES PEREIRA

born on 19.12.1996

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner

Prof. Dr. David Basin, co-examiner

Prof. Dr. Marieke Huisman, co-examiner

Dr. Aymeric Fromherz, co-examiner

2026

Abstract

Software plays a crucial role in nearly every area of modern life, from communications and banking to national defense and transportation, and faulty software is a serious threat to the reliability and security of critical infrastructures. Program verification offers a principled solution to define and establish the correctness of computer programs. Over the past decades, several efforts have produced all kinds of verified programs, including operating systems, compilers, distributed systems, and cryptographic libraries. With few exceptions, however, these systems were written from the very beginning with verification in mind. To make verification tractable, such efforts tend to restrict the programming language in which the program is written, rely on code extraction, or constrain the structure of the program. Such compromises are unsuitable for verifying pre-existing programs, where the code has been written by software engineers who are not verification experts and who are unwilling to give up the flexibility of their programming languages.

This thesis advances the state of the art in verifying crucial properties of *pre-existing*, real-world codebases written in general-purpose programming languages, working within the paradigm of SMT-based deductive verifiers and separation logic. It addresses three key challenges that arise when the implementation predates its verification: (1) *organizing* the verification of a large, pre-existing codebase that is optimized, concurrent, and not shaped for verification; (2) *taming the verification performance* of SMT-based verifiers, which suffer from long, unpredictable verification times and proof instability that are aggravated by large codebases; and (3) *specifying and verifying programs that use complex language features* that production code relies on but state-of-the-art verifiers do not support well, with a particular focus on global variables and static initializers.

Addressing the first challenge, we present a methodology for *structuring* the verification of a large pre-existing system. We applied this methodology on VerifiedSCION, a multi-year effort to verify the data plane of the SCION Internet architecture, from its protocols down to its open-source Go implementation, yielding the first formally verified full-fledged Internet router. The methodology soundly combines protocol verification by refinement with code-level separation-logic verification, linking the two through an automatically generated specification of the router's I/O behavior. This decomposition made it possible for two independent teams to work in parallel, one on proving network-wide security properties of the protocol, and the other on proving the correctness of the pre-existing implementation, consisting of roughly 4,700 lines of concurrent Go code that was not restructured for verification. In the process, we uncovered various bugs and vulnerabilities that had escaped extensive prior review and testing.

Addressing the second challenge, we contribute a set of complementary strategies for *improving the verification performance* of SMT-based verifiers, centered on minimizing and managing the verification context and on selecting the verification algorithms that generate the proof obligations. These strategies were crucial to overcoming the verification performance problems we found in VerifiedSCION. We report on the strategies we adopted and remark that context management also improves proof stability. Finally, we provide an empirical analysis of where verification time is spent, identifying bottlenecks that may inform future research.

Addressing the third challenge, we present the first practical, language-agnostic technique for modularly verifying programs that use global state and static initialization. This feature is pervasive in languages such as Go and Java, and no satisfactory solution existed previously. Built on concurrent separation logic, our technique uses module invariants to specify properties of the global state and a notion of initialization levels to determine when an invariant may be soundly assumed, regardless of the exact timing of the initialization. We formalize it as a program logic, prove its soundness, and implement it in verifiers for both Go and Java.

Together, these contributions show that the verification of crucial properties of real-world software is feasible without rewriting it for verification, and they distill techniques and lessons that we believe generalize to other large verification efforts.

Zusammenfassung

Software spielt in nahezu jedem Bereich des modernen Lebens eine entscheidende Rolle, von der Kommunikation und dem Bankwesen bis hin zur Landesverteidigung und dem Verkehrswesen, und fehlerhafte Software stellt eine ernsthafte Bedrohung für die Zuverlässigkeit und Sicherheit kritischer Infrastrukturen dar. Programmverifikation bietet eine fundierte Lösung, um die Korrektheit von Computerprogrammen zu definieren und nachzuweisen. In den vergangenen Jahrzehnten haben verschiedene Projekte die unterschiedlichsten verifizierten Programme hervorgebracht, darunter Betriebssysteme, Compiler, verteilte Systeme und kryptografische Bibliotheken. Bis auf wenige Ausnahmen wurden diese Systeme jedoch von Beginn an unter Rücksichtnahme auf deren Verifikation entworfen. Um die Verifikation handhabbar zu machen, wird in solchen Bestrebungen oft die Programmiersprache, in der das Programm geschrieben ist, eingeschränkt, auf Codeextraktion gesetzt oder die Struktur des Programms beschränkt. Derartige Kompromisse sind allerdings nicht geeignet für die Verifikation bereits bestehender Programme, deren Code von Softwareentwicklern geschrieben wurde, die keine Verifikationsexperten sind und die nicht bereit sind, die Flexibilität ihrer Programmiersprachen aufzugeben.

Diese Arbeit treibt den Stand der Technik bei der Verifikation wichtiger Eigenschaften *bereits bestehender*, realer Codebasen voran, die in Allzweck-Programmiersprachen geschrieben sind, und bewegt sich dabei im Paradigma SMT-basierter deduktiver Verifizierer und der Separationslogik. Sie behandelt drei zentrale Herausforderungen, die entstehen, wenn die Implementierung ihrer Verifikation zeitlich vorausgeht: (1) die *Organisation* der Verifikation einer grossen, bereits bestehenden Codebase, die optimiert und nebenläufig ist und nicht für die Verifikation zugeschnitten wurde, (2) die *Bändigung der Verifikationsperformanz* SMT-basierter Verifizierer, die unter langen, unvorhersehbaren Verifikationszeiten und Beweisinstabilität leiden, welche durch grosse Codebases noch verschärft werden, und (3) die *Spezifikation und Verifikation von Programmen, die komplexe Sprachfeatures nutzen*, auf die sich in der Praxis verwendeter Code stützt, die aber von modernsten Verifizierern nur unzureichend unterstützt werden, mit besonderem Augenmerk auf globale Variablen und statische Initialisierer.

Zur Bewältigung der ersten Herausforderung stellen wir eine Methodik zur *Strukturierung* der Verifikation eines grossen, bereits bestehenden Systems vor. Wir haben diese Methodik auf VerifiedSCION angewandt, ein über mehrere Jahre laufendes Vorhaben zur Verifikation der Datenebene der SCION-Internetarchitektur, von ihren Protokollen bis hinunter zu ihrer Open-Source-Implementierung in Go, woraus der erste formal verifizierte vollwertige Internet-Router hervorging. Die Methodik kombiniert auf fundierte Weise die Protokollverifikation durch Verfeinerung mit der Verifikation auf Codeebene mittels Separationslogik und verbindet beide über eine automatisch generierte Spezifikation des Eingabe-Ausgabe-Verhaltens des Routers. Diese Aufteilung ermöglichte es zwei unabhängigen Teams, parallel zu arbeiten: das eine am Nachweis netzwerkweiter Sicherheitseigenschaften des Protokolls, das andere am Nachweis der Korrektheit der bereits bestehenden Implementierung, die aus rund 4700 Zeilen nebenläufigen Go-Codes besteht, der nicht für die Verifikation umstrukturiert wurde. Dabei deckten wir verschiedene Fehler und Schwachstellen auf, die einer umfangreichen vorangegangenen Begutachtung und Erprobung entgangen waren.

Zur Bewältigung der zweiten Herausforderung steuern wir eine Reihe komplementärer Strategien zur *Verbesserung der Verifikationsperformanz* SMT-basierter Verifizierer bei, die auf die Minimierung und Verwaltung des Verifikationskontexts sowie auf die Auswahl der Verifikationsalgorithmen abzielen, welche die Beweisverpflichtungen erzeugen. Diese Strategien waren entscheidend, um die Verifikationsperformanzprobleme zu überwinden, die wir in VerifiedSCION vorgefunden haben. Wir berichten über die von uns gewählten Strategien und merken an, dass die Kontextverwaltung auch die Beweisstabilität verbessert. Schliesslich liefern wir eine empirische Analyse darüber, wo Verifikationszeit verbraucht wird, und identifizieren Engpässe, die zukünftige Forschung anleiten.

Zur Bewältigung der dritten Herausforderung stellen wir die erste praxistaugliche, sprachunabhängige Technik zur modularen Verifikation von Programmen vor, die globalen Zustand und statische Initialisierung verwenden. Dieses Merkmal ist in Sprachen wie Go und Java allgegenwärtig, und bislang existierte keine zufriedenstellende Lösung. Aufbauend auf nebenläufiger Separationslogik nutzt unsere Technik Modulinvarianten, um Eigenschaften des globalen Zustands zu spezifizieren, sowie ein Konzept von Initialisierungsebenen, um zu bestimmen, wann eine Invariante korrekt angenommen werden darf, unabhängig vom genauen

Zeitpunkt der Initialisierung. Wir formalisieren sie als Programmlogik, beweisen ihre Korrektheit und implementieren sie in Verifizierern sowohl für Go als auch für Java.

Zusammengenommen zeigen diese Beiträge, dass die Verifikation entscheidender Eigenschaften realer Software möglich ist, ohne diese für die Verifikation neu zu schreiben, und sie destillieren Techniken und Erkenntnisse heraus, die sich unserer Auffassung nach auf andere grosse Verifikationsvorhaben übertragen lassen.

Acknowledgments

The six years of my doctoral studies were, in some ways, a long and winding road. Fortunately, I had the companionship and support of many people who, in one way or another, helped me persevere and accomplish my goals. I want to thank a few people in particular.

First, I would like to express my gratitude to my supervisor, Peter Müller. Peter, thank you very much for allowing me to work in the group. I am very grateful to have had you not only as a supervisor, but as a role model. During my time at ETH, the high standards you applied to yourself and your work consistently inspired me to raise the bar for myself and do my best in what I do.

I am thankful to my second supervisor and co-examiner, David Basin, for his insightful input and patience in the many VerifiedSCION meetings over the years. I would also like to express gratitude to Marieke Huisman and Aymeric Fromherz for agreeing to be my co-examiners and contributing to making my thesis better through the various questions and discussions before, during, and after my defense.

Much of the work presented in this thesis was the output of long-lasting collaborations with great researchers, and I would like to thank them all for their contributions to the work presented in this thesis and elsewhere.

Thank you, everyone who contributed to the VerifiedSCION project! This was a grueling experience at times, but I am proud of what we achieved *together*. In particular, thank you, Dionysios, for being my brother in arms and close friend, and for enduring the petty frustrations of the project with me. Despite the struggle, I look fondly on our marathons filled with Greek food, and I am so happy you were still around for our trip to CCS'25 – that was one of the happiest moments in my PhD. A special thank you goes to Markus Limbeck for his dedication in the final push before the first submission, and to Marco Eilers for his work in making Viper rise to the challenge. Thank you, everyone in the Gobra team, especially Felix, Linard, and Dionysios. Collaborating on this shared project over many years has been a very rewarding experience, and I am glad that we made Gobra the great vessel for our research ideas that it is today. Thank you all for your care, taste, and hard work. Thank you to Marco Eilers for getting excited about my ideas about static initialization one night at a Bierhalle and for joining me in exploring this topic, and thank you, Isaac van Bakel, for contributing the formalization that made our work much better after a casual conversation at OPLSS 2024. Our meetings around a whiteboard always put a smile on my face. Thank you, Aurea, for our many meetings exploring refinement. It was a pleasure witnessing your creativity being applied to research! And thank you, Lea and Andrea, for the many funny meetings around dependency analysis. What started off as a very exploratory idea turned into a cool project over the many HAUA we spent together, not only discussing research but, in general, just having a good time and roasting each other.

It was a pleasure to work with you all!

My PhD would have been very different (and much less fun) had I not had the swell group of colleagues and friends at the Programming Methodology Group. I would like to say a word of gratitude to the members of the group over the years. Thank you, Alexandra, for providing me with guidance through our conversations over the years and reminding me of all the opportunities that I should explore; Andrea, for being the best student I could have asked for – it was a pleasure to be around when you joined the group and to see you become one of us; Andrew, for our many heartfelt conversations in the office at the least expected times; Anqi, for your kindness and generosity, and for the many good moments we shared together, usually around food and bubble tea; Arshavir, for welcoming me into the group during COVID; Aurea, for the various moments we shared together (be they at a sushi restaurant, concert, or just a stroll through town) and for your sense of humor, always ready to crack a joke (Haudi); Christoph, for your availability to discuss any research idea when I first joined the group; Dionysios (and Danai), for your support, friendship, and for the many moments we spent together, at work, and outside of it; Federico, for your work in Viper performance, which was very useful in my research; Felix, for creating Gobra and for your patience in introducing me to the codebase and reviewing my contributions; Gaurav, for the many long and exciting conversations we had over the years (and that happen still to this day!) about formal methods, academia, one's life's work, and many other topics – some of our conversations have stayed with me for the longest time and inform how I look at my work and goals on a daily basis; Hongyi, for your pleasant company at bubble teas and your patience and availability to answer all kinds of questions about Chinese culture; Jack, for bursting into the group with your relaxed

attitude and for immediately becoming an essential part of it – your easy laughter is contagious, and it was dearly missed when you went back home after your first visit, and I am happy that this time you are staying for much longer; Jérôme, for the good moments spent at Bierhalles; Jonáš, for your dedication to bringing the group together in all kinds of activities; Lea, for your freakish nature and thick Austrian accent, and more seriously, for being a great friend, confidant, and my Lästerschwester, and for being supportive until the very end and enduring many a dry run for my defense; Linard, for the various moments we spent together over the past years all over the world, and for your dedication to the tools we developed and maintained together; Malte, for always being ready to share your knowledge about our tooling and your experiments, which was often very useful in designing my own experiments; Marco, for the many years of guidance and collaboration, as well as patience to listen to me just ramble on while lying on the couch at your office, and for the many good moments together like our 3AM stroll in Vilnius; Michael, for being my first office mate and showing me your inspiring work ethic; Nick, for co-hosting the after-lunch coffees at your office, providing the opportunity for all kinds of discussions, and for our own conversations around proof automation and the use of AI for verification; Sandra, for helping a lot with all kinds of tasks in the group, making sure that the group and our retreats run without a hitch; Thibault, for the many enjoyable moments in Seattle during our internship; Thomas, for diligently trying to bring the group together, often taking the initiative to organize all kinds of events (including our recent trip to Berlin!); Vytautas, for your warmth and humor, for exposing me to the various Improv events in Zürich, for the many dinners at Mensa, and for welcoming us in Vilnius – from my side, I did my part in spreading a larger-than-life figure of you (and your eating "skills") to the new members of the group; Xavier, for the many interesting points of view around formal verification; and finally, Yushuo, for being an inspiring presence in the group, nurturing various interests like playing at an orchestra and calligraphy, while still keeping everything under control at work.

Besides our group, I am glad to have had the company of the members of the Programming Language Foundations Lab. Thank you, Ralf, for your willingness to share your knowledge in any discussion about programming languages; Max, for your inquisitive mind, which often provided a lot of food for thought after our conversations; Isaac, for being a great friend and daily presence in my life, in our coffees, swims, and evenings out trying new restaurants. I will miss being able to walk into your office and steal your attention (and snacks) and talk about whatever, whenever, be it little worries about daily life or deeper questions about the goal of one's research, career, and life. Thank you, Johannes, for the initiative in co-organizing all kinds of events and, above all, the kindness you demonstrate in many ways, be it in defending someone you think was wronged, or helping me, without my request, by shouldering more work than you had to for FMFP when I was writing this thesis; and Alessio, for your very funny sense of humor and for giving me one opportunity to call you a Malaka before I left.

I am also grateful to Matthias (Roshardt) for the many fun conversations over dinner at Mensa, Foteini for our many conversations over coffee, and Matthias (Brun) for our many conversations about formal verification.

I am very grateful to have had the opportunity to intern with Chris Hawblitzel at Microsoft Research in 2023. This internship provided the much-needed distance from my then-current research and exposed me to many cool ideas on how to construct program verifiers. Thank you, Chris, for being an excellent mentor and researcher! During my time there, I had the opportunity to collaborate with many other inspiring researchers, including Galen Hunt, Jay Lorch, and Shuvendu Lahiri. In particular, I am thankful to Galen for his friendliness and warmth, the Friday doughnuts, and the many bits of wisdom. And of course, for the memorable flight over Washington State. My gratitude extends to Bryan Parno for hosting me at the very first Verus Retreat.

I am grateful to my family, especially my parents, brother, and grandparents, for their unconditional love and support, and for making sure that I always had a loving home whenever I went back. Having them cheering me on kept me going when I felt stuck and unmotivated. I am thankful to the many friends back home who, despite the distance, were still very present in my life. I especially thank Catarina for the countless hours we spent talking about life and the PhD over coffee or sushi, Luciano for all the emotional support over the years, and Armando for being a daily presence in my life, always sharing cool ideas and a positive attitude.

Finally, and *above all*, I am forever grateful to Helena. Your love, care, patience, and humor made my journey so much happier. I can't imagine my life without you, and I am very excited for the new phase in our lives that is starting now.

Contents

Contents **vii**

1 Introduction **1**
1.1 Challenges . . . 2
1.2 Contributions . . . 3
1.3 Publications . . . 4
1.4 Outline . . . 6

2 A primer on the Gobra verifier **9**
2.1 Basic annotations . . . 10
2.2 Termination . . . 11
2.3 Permissions and heap-manipulation . . . 12
2.3.1 Accessibility predicate . . . 12
2.3.2 Quantified permissions . . . 14
2.4 Predicates . . . 14
2.5 Ghost code . . . 16
2.6 Interfaces . . . 17
2.7 Concurrency . . . 19
2.7.1 First-class predicates . . . 20
2.7.2 Reasoning about mutual exclusion with `sync.Mutex` . . . 21

3 Formal Verification of a Next-Generation Internet Router **23**
3.1 Introduction . . . 23
3.2 An overview of SCION . . . 26
3.3 Overview of properties and verification . . . 27
3.3.1 Verification guarantees . . . 27
3.3.2 Modular verification approach . . . 28
3.3.3 Separating protocol and code verification . . . 28
3.3.4 Protocol verification . . . 29
3.3.5 Code verification . . . 30
3.4 Protocol modeling and verification . . . 31
3.4.1 Environment model . . . 31
3.4.2 Protocol models . . . 32
3.4.3 Protocol verification . . . 33
3.4.4 Results . . . 34
3.5 Linking protocol to code verification . . . 36
3.6 Code verification . . . 36
3.6.1 Verifying memory safety . . . 38
3.6.2 Verifying functional correctness . . . 39
3.6.3 Verifying protocol compliance . . . 41
3.6.4 Results . . . 44
3.6.5 Ongoing work . . . 46
3.7 Assumptions and Trusted Computing Base (TCB) . . . 49
3.8 Lessons learned . . . 49
3.8.1 Overall approach . . . 50
3.8.2 Continuous development . . . 50
3.8.3 Limits of automated code verification . . . 51
3.9 Related work . . . 52
3.10 Conclusion . . . 53

4 Taming Verification Performance **55**
4.1 Introduction . . . 55
4.2 Symbolic execution of Gobra . . . 57
4.2.1 Symbolic state and symbolic execution . . . 58
4.2.2 Verification context . . . 59
4.3 Improving verification performance . . . 60
4.3.1 Automatically minimizing the context in Gobra . . . 60
4.3.2 Manually managing the context with `outline` statements . . . 64
4.3.3 Manually managing the context through information hiding . . . 66
4.3.4 Picking and tweaking verification algorithms . . . 67
4.4 Verification performance in VerifiedSCION . . . 72
4.4.1 Experimental setup . . . 72
4.4.2 Q1: How is verification time split between different members? . . . 73
4.4.3 Q2: How much time is spent by the verifier on queries to the SMT solver? . . . 77
4.4.4 Q3: Which queries to the SMT solver are the slowest? . . . 78
4.4.5 A note on verification instability . . . 81
4.5 Related work . . . 82
4.6 Conclusion . . . 85
4.6.1 Future work . . . 86

5 Modular Reasoning about Global Variables and Their Initialization **89**
5.1 Introduction . . . 89
5.2 Key ideas . . . 93
5.2.1 Module invariants . . . 94
5.2.2 Definite initialization . . . 96
5.2.3 Duplicable invariants . . . 98
5.2.4 Cyclic initialization dependencies . . . 99
5.2.5 A complete example . . . 100
5.3 Formalization . . . 101
5.3.1 Syntax . . . 102
5.3.2 Translational semantics . . . 102
5.3.3 Program logic . . . 104
5.3.4 Soundness . . . 106
5.3.5 Freedom of initialization cycles . . . 106
5.4 Implementation . . . 110
5.4.1 Application to Java . . . 110
5.4.2 Application to Go . . . 112
5.5 Evaluation . . . 114
5.5.1 Case studies . . . 114
5.5.2 Real-world Go code base . . . 116
5.5.3 Ghost global state . . . 117
5.6 Related work . . . 120
5.7 Conclusion . . . 121

6 Conclusions and Future Work **123**

Bibliography **125**

Glossary **139**

1 Introduction

1.1 Challenges 2
1.2 Contributions 3
1.3 Publications 4
1.4 Outline 6

Software plays a crucial role in nearly every area of human activity, such as communications, banking, national defense, transportation, and commerce. Faulty software poses a serious threat to the reliability and security of critical computing infrastructures and can lead to catastrophic system failures and devastating attacks. In light of this problem, substantial progress has been made over the past decades on the theory and tools for *program verification*, the area of computer science concerned with *specifying what software components should do* and *mathematically proving that they do it*.

By the time I started my doctoral studies in September 2020, several major software verification projects had already been completed in different areas. In the area of operating systems, the seL4 project [1] and the attempt to verify parts of the Hyper-V hypervisor [2] pioneered such large verification efforts, followed later by CertiKOS [3]. There are also verified compilers, including CompCert [4] and CakeML [5]. In the area of distributed systems, IronFleet [6], Verdi [7], and Velisarios [8] respectively verified the Paxos, Raft, and PBFT distributed consensus protocols. The Everest project successfully verified the implementations of security protocols [9–11] and cryptographic libraries [12, 13].

In all of the listed efforts, with the notable exception of Hyper-V, the system under verification was written from the very beginning with verification in mind. While program verification is a hard problem, projects that combine software development and verification often adopt solutions that make compromises in favor of simplifying verification, such as (1) severely restricting the set of language features that are allowed [1] (e.g., disabling function calls through function pointers), (2) using code extraction to obtain implementations from higher-level languages, which prevents users from safely modifying the generated code [14, 15], and (3) restricting the structure of the program to make it amenable to verification [6, 16] (e.g., to syntactically structure programs as state machines).

These strategies take a clean-slate approach to software development and verification, and thus, they are unsuitable for verifying pre-existing codebases written by software engineers who are not experts in program verification and who may be unwilling to sacrifice the flexibility provided by their programming languages of choice. The compromises introduced by these approaches motivated this thesis, **whose main goal is to enable the successful and productive verification of crucial properties of pre-existing codebases written in general-purpose programming languages.**

This thesis presents my contributions towards this goal. In particular, it is concerned with three major challenges: (1) organizing verification efforts in pre-existing codebases, (2) ensuring that verification performance is acceptable, and (3) verifying programs that use global state and static initializers, a common feature for which no satisfying solution previously existed. We elaborate on these challenges in Section 1.1 and the contributions in Section 1.2.

The work developed in this thesis is contained within the paradigm of semi-automated verifiers backed by SMT solvers such as Dafny [17] and

[1]: Klein et al. (2009), *seL4: formal verification of an OS kernel*

[2]: Leinenbach et al. (2009), *Verifying the Microsoft Hyper-V Hypervisor with VCC*

[3]: Gu et al. (2016), *CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels*

[4]: Leroy (2009), *Formal verification of a realistic compiler*

[5]: Kumar et al. (2014), *CakeML: a verified implementation of ML*

[6]: Hawblitzel et al. (2015), *IronFleet: Proving Practical Distributed Systems Correct*

[7]: Wilcox et al. (2015), *Verdi: a framework for implementing and formally verifying distributed systems*

[8]: Rahli et al. (2018), *Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq*

[9]: Bhargavan et al. (2013), *Implementing TLS with Verified Cryptographic Security*

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

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

[12]: Zinzindohoué et al. (2017), *HACL*: A Verified Modern Cryptographic Library*

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

[14]: Protzenko et al. (2017), *Verified low-level programming embedded in F**

[15]: O'Connor et al. (2016), *Refinement through restraint: bringing down the cost of verification*

[6]: Hawblitzel et al. (2015), *IronFleet: Proving Practical Distributed Systems Correct*

[16]: Sun et al. (2024), *Anvil: Verifying Liveness of Cluster Management Controllers*

[17]: Leino (2010), *Dafny: An Automatic Program Verifier for Functional Correctness*

Viper [18], in which proof annotations are integrated directly into the code of the system under verification and proof obligations are discharged by an SMT solver such as Z3 [19]. Furthermore, it sits in the paradigm of *separation logic* [20] and related logics such as *implicit dynamic frames* [21], which address the *frame problem*[1] in software verification and naturally extend to concurrent programs.

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

[19]: De Moura et al. (2008), *Z3: An efficient SMT solver*

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

[21]: Smans et al. (2009), *Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic*

1: The frame problem consists of efficiently specifying, in a method's contract, an upper bound on the set of memory locations that the method may modify.

1.1 Challenges

This thesis advances the state of the art in verifying pre-existing codebases that were not originally developed in combination with their verification efforts. To achieve that, this dissertation addresses the following three main challenges:

Challenge 1: Structuring the verification of pre-existing codebases. Pre-existing codebases not developed with verification in mind pose several challenges. Such codebases tend to be large, optimized, concurrent, and to make heavy use of features like aliasing to minimize memory consumption and copying. They are developed by teams that are not verification experts, and their structure reflects engineering concerns rather than verification convenience. Verifying pre-existing codebases prevents the adoption of strategies that significantly simplify the verification task, such as the co-development of the implementation and its verification with a program structure amenable to verification. Large changes to the code are strongly discouraged by their developers, and thus, verification needs to take into account the current implementation in all its complexity. When the system implements a protocol, an additional challenge arises: code-level verification must be connected to protocol-level models, even though the code was not structured to mirror those models. Finally, the scale of verifying a large pre-existing system raises organizational challenges: decomposing a large verification project into manageable tasks, distributing work over different teams, and coping with the changes that are inevitable in any actively developed software project.

Challenge 2: Taming the verification performance of SMT-based program verifiers. SMT-based program verifiers owe much of their automation to the underlying SMT solvers, but this reliance is also the source of two major usability problems: *unpredictable and long verification times* and *proof instability*. Long verification times disrupt the iterative workflow of interactive proof development: when a single verifier run takes many minutes—or appears to have diverged—proof authors are forced to context switch, slowing progress considerably. Proof instability, whereby seemingly innocuous changes to the source program or its specification (e.g., renaming a variable) alter the verification results, forces authors to revisit and stabilize previously completed proofs. The causes of instability are hard to isolate because their effects can be non-local and are highly dependent on the internal state of the verifier. Two characteristics of real-world codebases exacerbate these issues. First, large functions with many control-flow paths lead to an explosion in the number of cases that verifiers must explore. Second, large *verification contexts*—comprising user-provided definitions as well as axioms generated to model the semantics of the language—can degrade both performance and stability of SMT solvers. These problems frustrate even experienced proof engineers,

as they are hard to isolate and debug, and they may severely slow down verification efforts.

Challenge 3: Specifying and verifying code that uses complex language features. Production code tends to use the full range of features provided by the programming language in which it is written, including features that are convenient for developers but hard to reason about formally and modularly, such as closures and higher-order functions, concurrency primitives, and global state with its associated static initializers. Global state and static initialization, in particular, are pervasive in imperative and object-oriented languages such as Go and Java, where they are used routinely. As discussed at the beginning of this chapter, clean-slate verification projects often cope with such features by simply avoiding them: they restrict the set of language features that may be used, or impose a program structure that sidesteps the constructs that are difficult to reason about. This option is not available when verifying pre-existing codebases. The code already exists and was written by developers who relied on these features deliberately—typically for good reasons, such as performance, modularity, or expressiveness—and who are unwilling to stop using them merely to accommodate a verifier. Removing or rewriting these constructs would require precisely the kind of invasive changes that, as argued in Challenge 1, must be kept to a minimum. Supporting these features is therefore unavoidable, yet some of them are still not satisfactorily supported by state-of-the-art verifiers. One such example is the support for modularly reasoning about global variables and their initialization (using *static initializers*), which we focus on in this thesis. A verifier that aspires to handle real-world code must provide sound and practical support for the features that developers actually use, rather than expecting the code to be restricted to a verification-friendly subset of the language.

1.2 Contributions

This dissertation presents my contributions towards addressing the three challenges listed in the previous section. Importantly, all of these contributions have been applied in VerifiedSCION, a large verification effort to prove the correctness of the data plane of the SCION Internet Architecture [22] — briefly introduced in Section 3.2 — from the protocols to its open-source implementation in Go. The contributions of this thesis are as follows:

[22]: Zhang et al. (2011), *SCION: Scalability, Control, and Isolation On Next-Generation Networks*

Contribution 1: A methodology for structuring the comprehensive verification of a pre-existing, real-world system. Addressing Challenge 1, Chapter 3 presents the first comprehensive formal verification of a full-fledged Internet router, establishing both system-wide protocol security properties and local code properties for the *SCION border router*, the component of SCION implementing the data plane. The central contribution is a methodology for *structuring* such an effort: we decompose the project along the protocol/code boundary using the Igloo methodology [23], which soundly combines protocol verification by refinement (carried out in Isabelle/HOL) with code verification based on separation logic (carried out in Gobra). The two lines of reasoning are linked by a program specification that is automatically generated from the most

[23]: Sprenger et al. (2020), *Igloo: soundly linking compositional refinement and separation logic for distributed system verification*

concrete protocol model and that completely describes the router's allowed I/O behavior. This decomposition allowed two teams to work *in parallel* with the best tool for each task, enabling network-wide security properties (path authorization, loop freedom, and valley freedom) to be soundly transferred to the executing code, and made it possible to verify roughly 4,700 lines of optimized, concurrent Go code written by an independent team, without optimizing it for verification. Along the way, the verification uncovered and fixed bugs and vulnerabilities in both the protocol and its implementation that had escaped extensive prior review and testing. We distill the techniques and processes used to decompose a multi-year effort into manageable, reusable tasks into lessons learned that we believe generalize to other large verification projects.

Contribution 2: Strategies for improving the verification performance of SMT-based verifiers. To address Challenge 2, Chapter 4 presents a set of complementary strategies for keeping verification performance under control, centered on *minimizing and managing the verification context* (e.g., automatically pruning the proof context and the `outline` statement) and on *selecting the verification algorithms* that shape the proof obligations Gobra emits. We report on which strategies proved most effective in practice and observe that the context-management techniques also improve proof stability. Finally, we contribute an empirical analysis of where Gobra spends its time on VerifiedSCION, identifying the current performance bottlenecks. This provides a starting point for future research on the performance and stability of SMT-based verifiers, especially those that use separation logic. While the analysis targets Gobra, the strategies (particularly those based on context management) carry over to other SMT-based verifiers facing sufficiently complex codebases and specifications.

Contribution 3: A practical, language-agnostic technique for verifying programs with global variables and static initialization. In the context of Challenge 3, we were confronted with verifying code that uses *global variables* and *static initializers* in VerifiedSCION. Chapter 5 presents the first practical, modular verification technique for programs that use global state and static initialization, a feature that is pervasive in mainstream languages such as Go and Java yet for which no satisfactory verification solution previously existed. The technique builds on concurrent separation logic and specifies global state with *module invariants*, while a notion of *initialization levels* establishes a partial order on modules and methods that determines when an invariant may soundly be assumed, irrespective of when exactly the module initializer runs. The technique makes only minimal assumptions about the initialization semantics, which makes it applicable to a wide range of languages, including C++, C#, Go, Java, and Scala. We formalize this technique as a program logic and implement it in two verifiers, one for Go and another for Java, which demonstrates that our solution can be implemented in languages with different initialization semantics, and demonstrate its flexibility on a range of challenging examples, including the VerifiedSCION codebase.

1.3 Publications

During my doctoral studies, I contributed to various publications, which I list here.

Chapter 3 is mostly based on the following conference article:

João Pereira, Tobias Klenze, Sofia Giampietro, Markus Limbeck, Dionysios Spiliopoulos, Felix Wolf, Marco Eilers, Christoph Sprenger, David Basin, Peter Müller, Adrian Perrig.
Protocols to Code: Formal Verification of a Secure Next-Generation Internet Router
In CCS 2025 [24]
Received a Distinguished Artifact Award

Chapter 3 is also based on the two chapters I co-authored [25, 26] for the following book:

[25]: Arquint et al. (2022), *Code-Level Verification*
[26]: Arquint et al. (2022), *Current Status and Plans*

Laurent Chuat, Markus Legner, David Basin, David Hausheer, Samuel Hitz, Peter Müller, Adrian Perrig.
The Complete Guide to SCION: From Design Principles to Formal Verification
By Springer (2022)

Chapter 5 is based on the following conference article:

João Pereira, Isaac van Bakel, Patricia Firlejczyk, Marco Eilers, Peter Müller.
Modular Reasoning about Global Variables and Their Initialization
In OOPSLA 2025 [27]

Chapter 4 provides novel material that we plan to submit to a relevant conference.

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

Chapter 3 is the result of a long collaboration between the Network Security Group, the Information Security Group, and the Programming Methodology Group at ETH Zurich. I led the efforts to verify the open-source implementation of the SCION border router and prove that the router implementation refines the models of the protocol developed by the Information Security Group. To that end, I was the main contributor to the verified codebase, provided a large part of the proofs, and determined how the proof effort was to be organized. Dionysios Spiliopoulos contributed immensely to the proofs, as well as the students we supervised together — especially Markus Limbeck. Together with Felix A. Wolf, I supervised two students — Luca Halm and Lino Telschow — who contributed to early attempts at verifying the codebase. Their attempts gave us valuable information on how to organize our specifications and verification efforts. The work carried out so far to verify secure information flow in the SCION router was done by Henri Inndorf under my close supervision. To make it possible to verify the codebase without major changes, the rest of the Gobra team and I implemented support for the Go features used in the codebase. Marco Eilers contributed crucial low-level optimizations to the symbolic execution engine used by Gobra, which was essential to keeping verification performance under control.

Chapter 4 describes various strategies to improve the verification performance of Gobra. Two of these strategies (automatically pruning the verification context and the `outline` statements) were initially introduced by Luca Halm during his bachelor's thesis, under the close supervision of

me and Felix A. Wolf, and were later reimplemented by Felix, whereas the support for the various configurations was contributed by the Viper team — especially, Malte Schwerhoff with his student, Fabian Bösiger, and Marco Eilers. This chapter contains my observations on which strategies were the most effective in managing the verification performance in the VerifiedSCION project. It provides an evaluation and analysis that were designed and carried out by me, with contributions from Florian Marek as a result of his Summer Fellowship at ETH Zurich in 2025.

The work discussed in Chapter 5 resulted directly from the need to support global variables and their initialization in Gobra to verify the SCION router. I designed and implemented various solutions to support this feature in Gobra, and realized that a language-agnostic approach for the implementation of these features in different programming languages could be devised, with only a few reasonable assumptions on the programming language. To that end, I and Marco Eilers supervised the bachelor's thesis of Patricia Firlejczyk, who came up with a language-agnostic solution for single-threaded programs. I generalized this solution to multi-threaded programs, after realizing that any program could be modeled as a multi-threaded program where each static initializer is modeled as a separate thread and every access to global variables may potentially synchronize the thread performing the access and the initializer thread. I implemented the approach described in this chapter in Gobra, and Marco implemented it in VerCors. After asking "How hard can it be?" in a casual chat at OPLSS 2024, Isaac van Bakel provided a formalization of our ideas in the Iris separation logic [28], including a formalization of the absence of initialization cycles, and proved the soundness of the approach. In the process, he identified various unsoundnesses in the different versions of the program logics. In the end, it was harder than he expected.

[28]: Jung et al. (2018), *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*

Publications beyond this dissertation

Besides the work that directly contributed to this dissertation, I contributed to the following publications, which were led by other researchers:

Felix Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Pereira, Peter Müller.
Gobra: Modular Specification and Verification of Go Programs
In CAV 2021 [29]

Aurea Bílá, João Pereira, Peter Müller.
A Refinement Methodology for Distributed Programs in Rust
In OOPSLA 2025 [30]
Received a Distinguished Artifact Award

1.4 Outline

The remainder of this thesis is structured as follows. Chapter 2 provides the necessary background on the Gobra verifier and introduces the specification and verification concepts used throughout the thesis. Chapter 3 presents the verification of the SCION border router. It describes the architecture of the verification effort, including the decomposition into

protocol-level and code-level verification, and reports on the results and lessons learned. Chapter 4 focuses on the verification performance challenges encountered during the VerifiedSCION project. It presents strategies for managing verification times and proof context, and provides an empirical analysis of the performance bottlenecks in Gobra. Chapter 5 addresses the challenge of verifying programs that use global variables and static initialization. It presents a language-agnostic verification technique, its formalization, and its implementation in two verifiers. The thesis concludes with a discussion of the results, their implications, and directions for future work.

A primer on the Gobra verifier | 2

2.1 Basic annotations 10
2.2 Termination 11
2.3 Permissions and heap-manipulation 12
2.4 Predicates 14
2.5 Ghost code 16
2.6 Interfaces 17
2.7 Concurrency 19

In this chapter, we present the Gobra [29] verifier, which was used throughout the thesis. This chapter covers important aspects of the tool, especially its approach to verifying the memory safety and correctness of heap-manipulating Go programs, which is relevant background for the chapters that follow.

Gobra is an automated, modular verifier for heap-manipulating, concurrent Go programs. It supports a large subset of Go, including Go's interfaces and many primitive data structures. Gobra verifies memory safety, crash safety, data-race freedom, and partial correctness based on user-provided specifications. Gobra uses *implicit dynamic frames* [21] — a variant of separation logic [31, 32] — to specify and prove properties about heap-allocated data structures, which uses the notion of *permissions* (Section 2.3) to identify all memory locations that may be safely accessed by a function.

Gobra takes as input a Go program extended with annotations such as pre and postconditions and loop invariants, as well as new top-level definitions used for specification and verification purposes such as predicates (Section 2.4) and implementation proofs (Section 2.6). Gobra accepts two kinds of inputs (and their combination): Go programs with specifications and proof annotations provided as comments that start with the string `//@` or are delimited by the markers `/*@` and `@*/`, and Gobra programs, which have the same syntax as the annotated Go programs with the exception that proof annotations do not occur in comments. In this thesis, we use both input forms — we opt for the latter form by default, unless we want to emphasize the distinction between the executable Go code and the user annotations.

Verification in Gobra proceeds by first translating the annotated program into the *Viper intermediate verification language* [18], which can then be verified using one of the two verifiers (a.k.a., *backends*) for Viper: one that *symbolically executes* the program [33] and discharges the program's soundness conditions on each path, and another that generates *verification conditions* [34] based on the calculus of *weakest-preconditions* [35]. Both backends discharge their proof obligations via an SMT solver, specifically the Z3 solver [19]. In case verification fails, Gobra reports at the level of the Go program which assertions it could not verify.

The remainder of this chapter is organized as follows: In Section 2.1, we introduce the basic annotations used to specify functions — pre- and postconditions, loop invariants, and pure functions. We then explain how Gobra reasons about termination in Section 2.2. Section 2.3 presents the features of Gobra to reason about heap-allocated mutable state: fractional permissions, accessibility predicates, and quantified permissions. Building on permissions, Section 2.4 introduces *predicates*, which name parameterized assertions and enable reasoning about recursive heap structures such as linked lists and trees. Section 2.5 shows how auxiliary state and code — including the *ghost types* (sequences, sets, and multisets) Gobra provides for specification — can be introduced for verification without affecting the program's runtime behavior. Section 2.6 explains how to specify and reason about Gobra's interfaces. Finally, Section 2.7 covers the verification of *concurrent* programs, introducing

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

[33]: Schwerhoff (2016), *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*

[34]: Heule et al. (2013), *Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions*

[35]: Dijkstra (1975), *Guarded commands, nondeterminacy and formal derivation of programs*

[19]: De Moura et al. (2008), *Z3: An efficient SMT solver*

first-class predicates and showing how they are used in the verification of concurrent programs.

2.1 Basic annotations

We start with a simple function `sum` computing the sum of the integers from 0 to `n`. Listing 2.1 shows a Go implementation and Listing 2.2 the corresponding annotated version in Gobra.

Listing 2.1: Go implementation of the `sum` function.

```
1 package primer

3 func sum(n int) (sum int) {
4     sum = 0
5     for i := 0; i <= n; i++ {
6         sum += i
7     }
8     return sum
9 }
```

In addition to the Go code, the annotated version has a *specification* in the form of a pre and postcondition, and an invariant for the loop. The precondition `0 <= n` restricts the argument `n` to non-negative values. A caller of `sum` has to guarantee this assertion. The postcondition `sum == n * (n+1)/2` expresses an assertion that holds after a call returns, which has to be guaranteed by the function body. Pre and postconditions can be split into multiple `requires` and `ensures` clauses. The invariant `0 <= i && i <= n + 1` and `sum == i * (i-1)/2` is used to verify the loop. An invariant has to hold before a loop and must hold at the end of the loop body. Note that the initial statement `i := 0` is considered to happen before the loop. After the loop, we know that the invariant and the negated loop condition holds.

Listing 2.2: `sum` function specified and verified in Gobra.

```
1  package primer

3  // precondition
4  requires 0 <= n
5  // postcondition
6  ensures  sum == n * (n+1)/2
7  func sum(n int) (sum int) {
8      sum = 0

10     // loop invariants
11     invariant 0 <= i && i <= n + 1
12     invariant sum == i * (i-1)/2
13     for i := 0; i <= n; i++ {
14         sum += i
15     }
16     return sum
17 }
```

As assertions, Gobra supports Go's side-effect-free, deterministic boolean expressions (e.g. `x > y + z`), implications (`==>`), conditionals (`cond ? e1 : e2`), separating conjunctions (`&&`), universal quantifiers (e.g. `forall x int :: x >= 5 ==> x >= 0`), and existential quantifiers (e.g. `exists x int :: x > 42`). More specific assertions, e.g. for the access of heap locations and interfaces are introduced in subsequent sections.

When a function does not have side-effects, such as modifying heap-allocated structures, and is deterministic, it may be annotated with a `pure` clause. Different from non-pure functions, calls to pure functions can occur in specifications. For that reason, they are a powerful specification mechanism. We illustrate pure functions in specifications in Listing 2.3,

where the function `isEven` is pure and returns whether the argument is even. The `halfRoundedUp` function uses `isEven` in its specification.

```
1  package primer

3  decreases
4  pure func isEven(n int) (res bool) {
5      return n % 2 == 0
6  }

8  ensures isEven(n) ==> res == n / 2
9  ensures !isEven(n) ==> res == n / 2 + 1
10 func halfRoundedUp(n int) (res int) {
11     if isEven(n) {
12         res = n / 2
13     } else {
14         res = n / 2 + 1
15     }
16     return res
17 }
```

Listing 2.3: Specified implementation of `halfRoundedUp`. The pure function `isEven` used in the specification of `halfRoundedUp`.

Currently, pure functions have strict syntax constraints: their body must contain a single return statement that returns a single deterministic side-effect free expression. Furthermore, `pure` functions must be proven to terminate. Thus, we add a `decreases` clause instructing Gobra to prove termination as explained in the next section.

Gobra verifies programs modularly, one function at a time. When verifying a call to a (non-pure) function, Gobra reasons solely about the callee's specification — its pre- and postcondition — and never inspects its body. This keeps verification compositional and lets it scale to large code bases, since each function is verified once and in isolation. Pure functions are the exception: Gobra also takes their body into account, treating a call to a pure function as equal to the side-effect-free expression it returns. This exception enables equational reasoning, as a call to a pure function can be replaced by the expression it returns wherever it appears.

2.2 Termination

By default, Gobra proves partial correctness, i.e., if a function terminates then its postcondition holds. To prove *total correctness*, i.e., termination, a so-called *termination measure* must be provided in the specifications of functions and in loops. A termination measure is an expression that evaluates to a value of a well-founded order (e.g., natural numbers, lists). For simplicity, we only use natural numbers in this chapter. Gobra proves that a termination measure decreases in each iteration or recursive call, which implies that a function or loop terminates. Such a termination measure can be provided using the `decreases` keyword as shown in the following for the `sum` example.

```
1  package primer

3  requires 0 <= n // precondition
4  ensures  sum == n * (n+1)/2 // postcondition
5  decreases
6  func sum(n int) (sum int) {
7      sum = 0

9      invariant 0 <= i && i <= n + 1
10     invariant sum == i * (i-1)/2
11     decreases n - i
12     for i := 0; i <= n; i++ {
13         sum += i
```

Listing 2.4: Proving termination of `sum` with `decreases` clauses.

```
14      }
15      return sum
16  }
```

The first `decreases` clause right above the function signature specifies that the `sum` function terminates. Because the `sum` function does not recurse, any constant termination measure would be sufficient. Omitting the termination measure in these cases is a syntactic shorthand for `decreases 0`. Since the body of `sum` contains a loop, we have to prove that the loop terminates in order to prove that the function terminates. The annotation `decreases n - i` specifies a termination measure for the loop. This expression is greater than or equal to zero and decreases by one in each loop iteration; thus, the loop terminates.

Besides simple arithmetic expressions, Gobra, like Viper, also supports more advanced termination measures such as predicate instances, which we do not discuss here.

Note that while proving total correctness for (impure) functions is optional, Gobra requires proving termination for pure functions and ghost functions.

2.3 Permissions and heap-manipulation

[21]: Smans et al. (2009), *Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic*

Gobra uses *implicit dynamic frames* [21], a variant of *separation logic*, to specify and verify programs that manipulate data in the heap. Each memory address is associated with an *access permission*. Access permissions are held by function executions and transferred between functions upon call and return. A function may access a location or make a call only if the appropriate permission is held.

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

Permissions are a powerful reasoning principle that Gobra uses for several purposes. Since there is only one permission for each location, it is not possible for two threads to access a location simultaneously, which rules out data races and thus allows us to reason about code without having to consider interference by other threads. Gobra's *fractional permissions* [36] allow concurrent read accesses while still ensuring exclusive writes. Moreover, allocating an array creates one permission per array slot. Any out-of-bounds access is detected by Gobra since the permission for the (non-existent) slot is not available. Finally, permissions allow modular reasoning about side effects. As long as a function holds on to the permission for a location, no other function can possibly modify this location. This allows Gobra to preserve properties of locations across calls.

2.3.1 Accessibility predicate

The permission to access a pointer `x` is denoted by the *accessibility predicate* `acc(x)`. Having permission to a pointer implies that the pointer is not `nil`, Go's version of null. Access predicates are not duplicable resources, meaning that an accessibility predicate `P` does not entail `P && P`, where `&&` is a separating conjunction.

The following code snippet declares a type `pair`, consisting of two integer fields, and a method `incr` that increments both fields of a `pair` pointer by an integer `n`.

```
1  package primer

3  type pair struct {
4      left, right int
5  }

7  // incr requires permission to modify both fields
8  requires acc(&x.left) && acc(&x.right)
9  ensures  acc(&x.left) && acc(&x.right)
10 ensures  x.left == old(x.left) + n
11 ensures  x.right == old(x.right) + n
12 func (x *pair) incr(n int) {
13     x.left += n
14     x.right += n
15 }
```

Listing 2.5: The `pair` type and an `incr` method requiring permissions to its fields.

The function[1] `incr` modifies heap locations by assigning to `x.left` and `x.right`. For these assignments, `incr` requires permissions to the pointers of the two fields `&x.left` and `&x.right`. It gets these permissions from its precondition `acc(&x.left) && acc(&x.right)`[2]. When calling `incr` a caller has to transfer these permissions to the method. The postcondition `acc(&x.left) && acc(&x.right)` specifies that the permissions are returned to the caller when the call returns. In general, postconditions are used to express which permissions are transferred back from the callee to the caller. For pure functions, the permissions transferred back to the caller are the same as those transferred from the caller (through the precondition), and thus, they are not repeated in the postconditions.

1: Strictly speaking, `incr` is a *method* in Go. Methods are special functions in Go declared with a receiver parameter (`x` in the case of `incr`). Method calls are written using the dot syntax, e.g., `x.incr(10)`.

2: As a short-hand, one can write `acc(x)` instead of `acc(&x.left) && acc(&x.right)`.

When modifying the heap it is useful to refer to the old state of the heap. An *old expression*, `old(e)` evaluates the expression `e` in the state of the precondition. As such, the postcondition `x.left == old(x.left) + n` specifies that after `incr` returns the value of `x.left` is equal to the value of `x.left` when the function was called plus `n`.

Below, we show two clients of `incr`. The function `client1` allocates a `pair` pointer. When memory is allocated, permissions to the allocated heap locations are acquired. After `p := &pair{1,2}`, permissions to both field pointers are held. The call to `incr` on the subsequent line then uses these permissions. The assertion at the end checks that `p.left == 43` holds after the call.

```
1  func client1() {
2      p := &pair{1,2}
3      p.incr(42)
4      assert p.left == 43
5  }
```

Listing 2.6: Client `client1` allocating a `pair` and calling `incr`.

Similar to `client1`, `client2` allocates a `pair` pointer, as well. However, the allocation happens implicitly. In Gobra, when a local variable is referenced, it has to be annotated with `@`. The annotation expresses that pointers in the program may alias with this variable and thus, accesses to it must be governed by permissions.

```
1  func client2() {
2      // variable declaration marked with a @ so that Gobra
3      // allows its reference to be taken
4      x@ := pair{1,2}
5      (&x).incr(42)
6      assert x.left == 43
7  }
```

Listing 2.7: Client `client2` using an `@`-annotated variable.

Gobra uses *fractional permissions* [36] to distinguish between permissions that only allow reading from a memory location, and write permissions.

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

A permission amount is a rational number between 0 and 1. Modifying the value of a heap location requires full permission, denoted as `acc(x, write)`. Reading from a heap location requires any non-zero permission amount. Read permissions are usually denoted as fractions, e.g., `acc(x, 1/2)`. We can also write `acc(x, _)` for an existentially quantified positive permission amount.

2.3.2 Quantified permissions

Gobra provides two key mechanisms for specifying permission to a (potentially unbounded) number of heap locations: recursive predicates (Section 2.4) and *quantified permissions* [37].

[37]: Müller et al. (2016), *Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution*

Quantified permissions enable point-wise specifications, suitable for modeling heap structures that can be traversed in multiple directions, random-access data structures such as arrays, and unordered data structures such as graphs. The basic idea is to allow resource assertions to occur within the body of a `forall` quantifier. In our example, function `addToSlice` receives a slice `s` and adds the integer `n` to each element of the slice.

Listing 2.8: Using quantified permissions to specify `addToSlice`.

```
1  package primer
2
3  requires forall k int :: 0 <= k && k < len(s) ==> acc(&s[k])
4  ensures  forall k int :: 0 <= k && k < len(s) ==> acc(&s[k])
5  ensures  forall k int :: 0 <= k && k < len(s) ==> s[k] == old(s[k]) + n
6  func addToSlice(s []int, n int) {
7      invariant 0 <= i && i <= len(s)
8      invariant forall k int :: 0 <= k && k < len(s) ==>
9          acc(&s[k])
10     invariant forall k int :: i <= k && k < len(s) ==>
11         s[k] == old(s[k])
12     invariant forall k int :: 0 <= k && k < i ==>
13         s[k] == old(s[k]) + n
14     for i := 0; i < len(s); i += 1 {
15         s[i] = s[i] + n
16     }
17 }
```

To be able to change the contents of the slice, the function must have access to all elements of `s`. This is provided by the precondition `forall k int :: 0 <= k && k < len(s) ==> acc(&s[k])` asserting access to all locations `&s[k]`, where `k` is a valid index of the slice.

The function below is a client of the `addToSlice` function. The `make` call allocates the permissions required for the call.

Listing 2.9: A client of the `addToSlice` function.

```
1  func addToSliceClient() {
2      s := make([]int, 10)
3      assert forall i int :: 0 <= i && i < 10 ==> s[i] == 0
4      addToSlice(s, 10)
5      assert forall i int :: 0 <= i && i < 10 ==> s[i] == 10
6  }
```

2.4 Predicates

Predicates give a name to a parameterized assertion. A predicate can have any number of parameters, and its body can be any self-framing Gobra assertion using only these parameters as variable names. Predicate definitions can be recursive, allowing them to denote permission to and properties of recursive heap structures such as linked lists and trees.

The following snippet defines a type `node`, representing a node of a linked list. The predicate `list`, parameterized by `ptr`, abstracts the permissions of a linked list. The predicate body contains permissions to the `value` and `next` fields of the head and, recursively, contains permissions to the rest of the list if the tail is not empty, that is, if `ptr.next != nil`.

```
1  package primer

3  type node struct {
4      value int
5      next *node
6  }

8  pred list(ptr *node) {
9      acc(&ptr.value) &&
10     acc(&ptr.next) &&
11     (ptr.next != nil ==> list(ptr.next))
12 }
```

Listing 2.10: The `node` type and the recursive `list` predicate.

Predicate instances such as `list(ptr)` are resources that may be owned fully or partially by function executions. The full ownsership of a predicate instance `acc(list(ptr))` can be expressed succinctly as `list(ptr)`. Predicate instances can occur in function specifications. In Gobra, predicates are *iso-recursive* [38], meaning that predicate instances are not equivalent to their body. As such, the following program does not verify when uncommenting the second assert statement:

[38]: Summers et al. (2013), *A Formal Semantics for Isorecursive and Equirecursive State Abstractions*

```
1  requires list(ptr)
2  ensures  list(ptr)
3  func testPredFail(ptr * node) {
4      // the following assertion succeeds
5      assert list(ptr)
6      // the following assertion fails
7      // assert acc(&ptr.value) && acc(&ptr.next)
8  }
```

Listing 2.11: Iso-recursive predicates: `testPredFail` does not verify with the commented assertion uncommented.

To make the snippet verify, Gobra requires an additional `unfold` annotation, telling Gobra to replace a predicate instance with its body. Conversely, the `fold` annotation exchanges the predicate body with the predicate instance. Using `unfold`, the adapted version of the previous snippet verifies:

```
1  requires list(ptr)
2  ensures  list(ptr)
3  func testPred(ptr * node) {
4      assert list(ptr)
5      unfold list(ptr) // exchanges list(ptr) with its body
6      assert acc(&ptr.value) && acc(&ptr.next) // succeeds
7      fold list(ptr)   // folding back the predicate for the
        postcondition
8  }
```

Listing 2.12: Using `fold` and `unfold` to manipulate predicate instances.

In cases where statements are not allowed (e.g., specifications and bodies of pure functions), we use `unfolding` expressions, which temporarily unfold and fold a predicate instance. For instance, in the body of the `head` function below, the only way to acquire access to `ptr.value` is by `unfolding list(ptr)`.

```
1  requires list(ptr)
2  decreases
3  pure func head(ptr *node) int {
4      return unfolding list(ptr) in ptr.value
5  }
```

Listing 2.13: Using an `unfolding` expression in the pure function `head`.

The recursive `contains` function below checks whether a value occurs in a list. Since the function recurses on the tail of the list, it must be proven

to terminate. We use the `list` predicate instance itself as the termination measure: at each recursive call the permission to the (sub-)list is strictly smaller, which justifies the recursion.

Listing 2.14: The `contains` function using the `list` predicate as its termination measure.

```
requires list(ptr)
decreases list(ptr)
pure func (ptr *node) contains(value int) bool {
    return unfolding list(ptr) in
        ptr.value == value ||
        (ptr.next != nil && ptr.next.contains(value))
}
```

Note that the concrete permission amounts a pure function requires in its precondition are inconsequential. Since a pure function performs no heap modifications, any positive permission to the relevant locations suffices and, because a pure call implicitly returns all the permissions it received, the chosen amount has no effect on the caller.

Gobra predicates can also be abstract, i.e., have no body, to represent opaque permissions. Abstract predicates cannot be unfolded.

2.5 Ghost code

Often, introducing additional state, such as additional variables or arguments is useful for specification and verification. Consider the following `contains` function. The function returns whether or not an element `x` is contained in a slice `s`. One way to express that an element is contained in a slice is to specify that there is an index within the slice at which the element is stored. In the example, we use the ghost out-parameter `idx`, annotated with the keyword `ghost`, to store this index, allowing us to write an appropriate postcondition.

Listing 2.15: Using a ghost out-parameter in `contains`.

```
package primer

requires forall k int :: 0 <= k && k < len(s) ==> acc(&s[k], 1/2)
ensures  forall k int :: 0 <= k && k < len(s) ==> acc(&s[k], 1/2)
ensures  isContained ==> 0 <= idx && idx < len(s) && s[idx] == x
func contains(s []int, x int) (isContained bool, ghost idx int) {

    invariant 0 <= i && i <= len(s)
    invariant forall k int :: 0 <= k && k < len(s) ==> acc(&s[k], 1/4)
    for i := 0; i < len(s); i += 1 {
        if s[i] == x {
            return true, i
        }
    }

    return false, 0
}
```

Apart from the ghost in and out-parameters, Gobra offers additional ghost constructs: ghost variables, ghost types, ghost statements, and ghost methods and functions. A ghost variable is defined via a ghost declaration. For instance, the statement `ghost var x int = 42` declares a ghost variable `x` and assigns it 42.

Ghost types such as sequences (`seq[T]`), sets (`set[T]`), multisets (`mset[T]`), and permission amounts (`perm`) are mathematical types useful for specification purposes, supporting common operations such as sequence concatenation (`seq1 ++ seq2`), set union (`set1 union set2`), membership (`x in set1`), multiplicity (`x # set1`), sequence length (`len(seq1)`), and set cardinality (`len(set1)`).

Ghost if-then-else and loops are used to perform case splits and inductions, respectively. To make an actual statement a ghost statement, we add the `ghost` keyword before the statement. Pure ghost functions can be used to abstract heap state to a mathematical representation. The `toSeq` function below takes as parameter a slice and returns a mathematical sequence containing the elements of the slice. A ghost function or method is annotated with the keyword `ghost`. For ghost functions and methods, all in- and out-parameters are implicitly ghost.

Gobra checks that no ghost code influences actual code. Note that the opposite does not have to hold. For instance, the value of an actual variable can be assigned to a ghost variable.

```
1  ghost
2  requires forall j int :: 0 <= j && j < len(s) ==> acc(&s[j])
3  ensures  len(res) == len(s)
4  ensures  forall j int :: {s[j]} {res[j]} 0 <= j && j < len(s) ==>
5      s[j] == res[j]
6  decreases len(s)
7  pure func toSeq(s []int) (res seq[int]) {
8      return (len(s) == 0 ? seq[int]{} :
9          toSeq(s[:len(s)-1]) ++ seq[int]{s[len(s) - 1]})
10 }
```

Listing 2.16: The ghost pure function `toSeq` abstracting a slice to a mathematical sequence.

2.6 Interfaces

In Go, interfaces consist of sets of method signatures. In Gobra, all these method signatures additionally have a specification. Furthermore, an interface can define predicates to abstract over the heap structure of the underlying dynamic value. Consider the following declaration for an interface `stream`. The interface has two methods, the pure method `hasNext`, returning whether more elements are available, and the method `next`, returning the next element of the stream. The predicate `mem` abstracts over the permission to change the state of the `stream`.

```
1  package primer

3  type stream interface {
4    hasNext() bool

6    next() interface{}
7  }
```

Listing 2.17: Go declaration of the `stream` interface.

```
1  package primer

3  type stream interface {
4      pred mem()

6      decreases
7      requires acc(mem(), 1/2)
8      pure hasNext() bool

10     requires mem() && hasNext()
11     ensures  mem()
12     next() interface{}
13 }
```

Listing 2.18: Gobra specification of the `stream` interface.

The following snippet illustrates an implementation of the `stream` interface. The function `typeOf` returns the dynamic type of an interface. The type assertion `y.(int)` requires that the interface `y` has the dynamic type `int` and returns the dynamic value of the interface. Note that without a preceding `typeOf(y) == type[int]` the type assertion fails.

Listing 2.19: A `counter` type implementing the methods of the `stream` interface.

```
// implementation
type counter struct{ f, max int }

requires acc(x, 1/2)
decreases
pure func (x *counter) hasNext() bool {
    return x.f < x.max
}

requires acc(&x.f) && acc(&x.max, 1/2)
requires x.hasNext()
ensures  acc(&x.f) && acc(&x.max, 1/2) && x.f == old(x.f)+1
ensures  typeOf(y) == type[int] && y.(int) == old(x.f)
func (x *counter) next() (y interface{}) {
    y = x.f
    x.f += 1
    return
}
```

[39]: Liskov et al. (1994), *A Behavioral Notion of Subtyping*

Before a value of type `*counter` can be assigned to the `stream` interface, one has to show that `*counter` is a *behavioral subtype* of `stream` [39]. This means that, for all methods, the precondition of the interface method implies the precondition of the implementation method and the postcondition of the implementation method implies the postcondition of the interface method. In Gobra, such proofs are done via *implementation proofs*. The snippet below shows such an implementation proof for `*counter` and `stream`. The definition of the predicate `mem` is part of the implementation proof and defines the permissions that `*counter` requires to satisfy the interface specification. The implementation proof clause is denoted with the keyword `implements`. Inside the implementation proof clause, for each method of the interface, there is a proof method. The pre and postcondition of the proof method are omitted as they are determined by the corresponding method of the interface. Implementation proofs are heavily syntactically restricted. The body must have a single call to the method of the implementation with the right receiver and the in- and out-parameters in the correct order. Additionally, the body is only allowed to contain assert statements and fold and unfold annotations to manipulate predicate instances. In the example, we require unfolds and folds to shift from the `mem()` predicate instance of the interface methods to the direct pointer permissions of the implementation methods and vice versa.

Listing 2.20: Implementation proof that `*counter` is a behavioral subtype of `stream`.

```
// implementation proof
pred (x *counter) mem() { acc(x) }

(*counter) implements stream {
    pure (x *counter) hasNext() bool {
        return unfolding acc(x.mem(), 1/2) in x.hasNext()
    }

    (x *counter) next() (res interface{}) {
        unfold x.mem()
        res = x.next()
        fold x.mem()
    }
}
```

If an implementation proof or part of it is omitted, Gobra tries to infer the missing parts. As a special case, the empty interface `interface{}` is trivially implemented by all types. As such, in `*counter.next`, the statement `y = x.f` assigning an integer to the empty interface does not require a manual implementation proof.

The snippet below shows a small client that instantiates a `*counter`, assigns it to the `stream` interface, and calls the method `next`. Note that Gobra can show that the precondition of `y.next()`, namely `y.hasNext()` holds.

```
1 // client code
2 func client() {
3     x := &counter{0, 50}
4     var y stream = x
5     fold y.mem()
6     var z interface{} = y.next()
7 }
```

Listing 2.21: A client using a `*counter` through the `stream` interface.

Comparability

In Go, comparing two interfaces can cause a runtime panic. This happens if both dynamic values of an interface are incomparable. For that purpose, Gobra provides the function `isComparable`. The function takes as input an interface or a type and returns whether it is comparable according to Go. The snippet below shows a variation of a list where the values are interfaces. As an invariant, written in the `list` predicate, all elements of the list are comparable, meaning that a comparison with another interface is guaranteed to not cause a runtime panic. When the value of an implementation is put into an interface, Gobra generates the information of whether the resulting interface is comparable.

```
1  package primer

3  type node struct {
4      value interface{}
5      next *node
6  }

8  pred list(ptr *node) {
9      acc(&ptr.value)         &&
10     isComparable(ptr.value) &&
11     acc(&ptr.next)          &&
12     (ptr.next != nil ==> list(ptr.next))
13 }

15 requires  list(ptr)
16 decreases list(ptr)
17 pure func contains(ptr *node, value interface{}) bool {
18     return unfolding list(ptr) in
19         ptr.value == value ||
20         (ptr.next != nil && contains(ptr.next, value))
21 }
```

Listing 2.22: A list of comparable interface values specified using `isComparable`.

2.7 Concurrency

Gobra supports verification of concurrent Go programs that incorporate multi-threaded programming via goroutines, channel message passing, and shared memory. Gobra ensures that every verified program is free of race conditions. The snippet below shows code with a race condition that cannot be verified when uncommenting the second Go statement. The function `concurrentInc` would spawn two goroutines that concurrently modify the value of `x`. Only one of the spawned goroutines can own the permission to `x` at any point in time. When the first goroutine is started `go inc(&x)`, the permission `acc(x)` is transferred to the goroutine. Different from a normal call, the permissions in the postcondition of the

goroutine are not returned after `go inc(&x)`. As a consequence, starting the second goroutine fails, as `acc(x)` is not held anymore by the execution of `concurrentInc`.

Listing 2.23: A program with a race condition that fails to verify.

```
1  package primer

3  requires acc(x)
4  ensures acc(x)
5  func inc(x *int) {
6      *x = *x + 1
7  }

9  func concurrentInc() {
10     x@ := 1
11     go inc(&x)
12     // the following statement leads to a verification error
13     // due to a permission error
14     // go inc(&x)
15 }
```

The race condition in the previous example can be avoided by using concurrency primitives to synchronize the accesses to `x`. In Gobra, concurrency primitives make extensive use of Gobra's first-class predicates. Because of that, we first introduce predicate expressions in the next section. Afterwards, we show a race-condition free version of `concurrentInc` that uses the `Mutex` type defined in the `sync` package. While we do not discuss it here, Gobra can also be used to verify programs with channels.

2.7.1 First-class predicates

Gobra has support for first-class predicates, i.e., expressions with a predicate type. First-class predicates are of type `pred(x1 T1, ..., xn Tn)`. The types `T1, ..., Tn` define that the predicate has an arity of `n` with the corresponding parameter types.

To instantiate a first-class predicate, Gobra provides *predicate constructors*. A predicate constructor `P{d1, ..., dn}` partially applies a declared predicate `P` with the constructor arguments `d1, ..., dn`. A constructor argument is either an expression or a wildcard `_`, symbolizing that this argument of `P` remains unapplied. In particular, the type of `P{d1, ..., dn}` is `pred(u1,...,uk)`, where `u1,...,uk` are the types of the unapplied arguments. As an example, consider the declared predicate `pred sameValue(i1 int8, i2 uint32){ ... }`. The predicate constructor `sameValue{int8(1), _}` has type `pred(uint32)`, since the first argument is applied and the second is not. Conversely, `sameValue{_, uint32(1)}` has type `pred(int8)`. Finally, `sameValue{ int8(1), uint32(1)}` and `sameValue{ _, _}` have types `pred()` and `pred(int8, uint32)`, respectively.

The equality operator for predicate constructors is defined as a point-wise comparison, that is, `P1{d1, ..., dn}` is equal to `P2{`*d1, ... , dn*`}` if and only if `P1` and `P2` are the same declared predicate and if `di == di` for all `i` ranging from 1 to `n`.

The body of the predicate `P{d1, ..., dn}` is the body of `P` with the arguments applied accordingly. Like with other predicates, `P{d1, ..., dn}` can be instantiated and its instances may occur in assertions and in `fold` and `unfold` statements. The fold statement `fold P{d1, ..., dk}(e1, ..., en)` exchanges the first-class predicate instance with its body. The unfold statement does the reverse.

2.7.2 Reasoning about mutual exclusion with `sync.Mutex`

In the following example, we show the function `safeInc`, a data-race-free version of the previously seen function `inc` in section Concurrency. The snippet uses a lock `pmutex` to synchronize the write accesses to the pointer `x`. `safeInc`'s pre and postcondition assert that `pmutex` was initialized (`acc(pmutex.LockP(), _)`) with invariant `mutexInvariant` (`pmutex.LockInv() == mutexInvariant{x}`), protecting the access to variable `x`. In the body of `safeInc`, first `pmutex.Lock()` is called to acquire the invariant `mutexInvariant{x}`. Then the first-class predicate instance is unfolded to acquire `acc(x)`, required to modify the value of `x`. At the end, the invariant is folded back again, after which `pmutex` can be released via a call to method `Unlock`.

```
1  package primer

3  import "sync"

5  pred mutexInvariant(x *int) {
6      acc(x)
7  }

9  requires acc(pmutex.LockP(), _) &&
10     pmutex.LockInv() == mutexInvariant{x}
11 ensures  acc(pmutex.LockP(), _) &&
12     pmutex.LockInv() == mutexInvariant{x}
13 func safeInc(pmutex *sync.Mutex, x *int) {
14     pmutex.Lock()
15     unfold mutexInvariant{x}()
16     *x = *x + 1
17     fold mutexInvariant{x}()
18     pmutex.Unlock()
19 }
```

Listing 2.24: A data-race-free `safeInc` function using a `sync.Mutex`.

The snippet below shows a client. First, `mutex` is allocated and the invariant is established for the first time by folding `mutexInvariant{&x}()`. To bind an invariant to a mutex, the `SetInv` method is called. `SetInv` expects as argument a first-class predicate with arity zero. After the invariant is bound to `&mutex`, the mutex is initialized and the permission `(&mutex).LockP()` is acquired. This permission is necessary to call `Lock` and `Unlock`. At this point, the preconditions of the goroutines are satisfied and the goroutines can be started. Note that the precondition of `safeInc` does not require write permission to the lock and thus, after starting the first goroutine, there are sufficient permissions remaining to start the second goroutine.

```
1  func client() {
2      x@ := 0
3      mutex@ := sync.Mutex{}
4      fold mutexInvariant{&x}()
5      (&mutex).SetInv(mutexInvariant{&x})
6      go safeInc(&mutex, &x)
7      go safeInc(&mutex, &x)
8  }
```

Listing 2.25: A client of `safeInc` that initializes a mutex and binds its invariant.

Finally, while we do not show it here, Gobra supports Go's built-in channels. Similarly to mutexes, channels are associated with a channel invariant that expresses the properties of the messages sent over the channel. Channel invariants may also include permissions, which is a useful way to transfer permission to a location from the sender to the receiver.

Formal Verification of a Next-Generation Internet Router

3

In this chapter, we present our work on verifying the border router of the SCION protocol, which is responsible for forwarding packets between the Internet's Autonomous Systems (ASs). We describe how we successfully and modularly verified the implementation of a pre-existing protocol in the context of the VerifiedSCION project, and we distill crucial lessons that we believe may be of interest to other projects with similar aims.

3.1 Introduction 23
3.2 An overview of SCION . 26
3.3 Overview of properties and verification 27
3.4 Protocol modeling and verification 31
3.5 Linking protocol to code verification 36
3.6 Code verification 36
3.7 Assumptions and TCB . . 49
3.8 Lessons learned 49
3.9 Related work 52
3.10 Conclusion 53

3.1 Introduction

Given that Internet routers are a central part of our critical networking infrastructure and deployed on a large scale, their security and reliability are of paramount importance. In this chapter, we present the first comprehensive formal verification of Internet routers and their protocols.

Today's Internet routers run the aging Border Gateway Protocol (BGP), which does not meet modern security and reliability requirements. Proposed mechanisms to add security [40–42] have proven to be insufficient, inefficient, or they introduce new problems [43–45]. Therefore, our verification effort focuses on the SCION Internet architecture [46], a clean-slate redesign of an inter-networking infrastructure with a focus on strong security and reliability properties. SCION border routers are responsible for packet forwarding, which is simple and efficient, since each packet's path is embedded in the packet header, alongside cryptographic authenticators that authorize the path. SCION has real-world deployment: SCION services and connectivity are offered by over 20 Internet service providers reaching networks on 5 continents [47], and are used, for instance, as the networking layer for Swiss interbank clearing [48].

The security and correctness of Internet routing depend on both *system-wide, global, security properties of the protocol* (e.g., that packets can travel only along previously authorized paths, a property called *path authorization*) and *local properties of the code* (e.g., that the protocol is implemented correctly). Verifying them involved a host of challenges, in particular:

C1: How to model a packet forwarding protocol and verify its security for arbitrary network topologies under a strong attacker model?

C2: How to verify the pre-existing router implementation, a substantial code base that uses complex language features?

C3: How to verify the security and correctness of an existing, deployed router *comprehensively*, from protocol to code?

C4: How to organize the verification effort efficiently, given the size of the project, both of the protocol and codebase?

Challenge C1 concerns the protocol: We must account for the complex adversarial environment in which SCION routers operate. This involves an arbitrary network topology, an arbitrary set of authorized paths, and active and possibly colluding attackers. Moreover, the protocol is designed to achieve global properties like path authorization and loop freedom. Efficiency mandates that each router performs only local checks

[40]: Kent et al. (2000), *Secure Border Gateway Protocol (S-BGP)*
[41]: Bush (2014), *Origin Validation Operation Based on the Resource Public Key Infrastructure (RPKI)*
[42]: Lepinski et al. (2017), *BGPsec Protocol Specification*

[43]: Cooper et al. (2013), *On the risk of misbehaving RPKI authorities*
[44]: Li et al. (2014), *Even rockets cannot make pigs fly sustainably: Can BGP be secured with BGPsec?*
[45]: Rothenberger et al. (2017), *Internet Kill Switches Demystified*

[46]: Chuat et al. (2022), *The Complete Guide to SCION: From Design Principles to Formal Verification*

[47]: Wirz et al. (2025), *Scaling SCIERA: A Journey Through the Deployment of a Next-Generation Network*

[48]: SIX (n.d.), *Secure Swiss Finance Network*

related to its own position on a packet's embedded path. Showing that these checks imply the global properties requires stating and proving suitable invariants over recursive data structures that hold despite the attackers' presence. Doing this in full generality requires an expressive specification language like higher-order logic.

Challenge C2 concerns the implementation: The router code was developed independently of our verification effort and without verification in mind, with an emphasis on performance. Our goal is to verify this implementation, which has roughly 4,700 lines of optimized, concurrent Go code that heavily uses aliasing to minimize memory consumption and copying. While verifying, we want to perform as few changes to the original code as possible[1], so that our correctness proofs clearly transfer to the original implementation, and so that we can easily merge changes from the upstream SCION repository into our verified codebase. The requirement that we verify a pre-existing implementation developed by an independent team prevents the adoption of strategies that significantly simplify the verification task, such as the co-development of the implementation and its verification, with a program structure amenable to verification [6, 16]. Despite significant advances in program verification techniques, verification of complex properties about an entire pre-existing code base written in an expressive language like Go has so far been out of reach. It requires tools that support all the used language features and idioms, express all needed properties, and can do so at scale.

Challenge C3 is concerned with formally integrating these two lines of reasoning to provide sound guarantees from protocol to code. Finally, challenge C4 is concerned with finding techniques and processes to organize a multi-year project into manageable tasks, distributing work over several team members, and dealing with the changes that are inevitable in any software project.

Verification for networking data planes has so far focused on network configurations [49–52] or on network functions [53–57]. The former only consider network *models* and their configurations and is mostly restricted to either local properties of single nodes or network-wide properties of fixed, concrete networks. The latter only considers local properties of *implementations*, but no global, network-wide properties. Moreover, all works in this category are restricted to non-adversarial settings.

Approach To tackle the above challenges, we decompose our overall project into smaller tasks using different forms of modularity. Concretely, we employ the Igloo methodology [23], which soundly combines protocol verification by refinement with code verifiers based on separation logic [31]. To verify the SCION router, we model the protocol and its adversarial networking environment as a labeled transition system in Isabelle/HOL [58]. We start from an abstract protocol with a weak adversary and refine it into more concrete ones with a strong, symbolic (Dolev-Yao [59]) adversary. We build on existing basic protocol models [60, 61] and extend them with bi-directional and multi-segmented paths. We prove network-wide properties: path authorization, valley freedom, and loop freedom. The latter two were not considered in previous work.

We extract a router model from the most concrete protocol model and extend Igloo's tooling to automatically translate it into a program specification that completely describes the router's allowed I/O behavior.

We use Gobra (see Chapter 2) to verify the router's Go code against this specification. This ensures that the code correctly follows the protocol

1: Ideally, the implementation would not suffer any changes while being verified. This is not possible, however, as we often have to perform small tweaks like storing intermediate computation results in new variables so that we can refer to these results in our proofs.

[6]: Hawblitzel et al. (2015), *IronFleet: Proving Practical Distributed Systems Correct*
[16]: Sun et al. (2024), *Anvil: Verifying Liveness of Cluster Management Controllers*

[49]: Khurshid et al. (2013), *VeriFlow: Verifying Network-Wide Invariants in Real Time*
[50]: Kozen (2014), *NetKAT – A Formal System for the Verification of Networks*
[51]: Weitz et al. (2016), *Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver*
[52]: Beckett et al. (2017), *A General Approach to Network Configuration Verification*

[53]: Pirelli et al. (2022), *Automated Verification of Network Function Binaries*
[54]: Zhang et al. (2020), *Automated Verification of Customizable Middlebox Properties with Gravel*
[55]: Zaostrovnykh et al. (2019), *Verifying software network functions with no verification expertise*
[56]: Liu et al. (2018), *p4v: practical verification for programmable data planes*
[57]: Zaostrovnykh et al. (2017), *A Formally Verified NAT*

[23]: Sprenger et al. (2020), *Igloo: soundly linking compositional refinement and separation logic for distributed system verification*

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

[58]: Nipkow et al. (2002), *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*

[59]: Dolev et al. (1983), *On the security of public key protocols*

[60]: Klenze et al. (2021), *Formal Verification of Secure Forwarding Protocols*
[61]: Klenze et al. (2022), *IsaNet: A framework for verifying secure data plane protocols*

and establishes additional code properties such as crash freedom, and freedom from data races. Since verification is performed statically, the executable code and its performance are entirely unaffected. Igloo's soundness result implies that the protocol properties are preserved for the code executing in its adversarial network environment. Hence, our protocol verification results are seamlessly and soundly extended to the router implementation.

While we have presented the above tasks as sequential steps, we worked on protocol and code verification *in parallel* with two teams using the best tool for the respective task, interfaced only by the generated program specification. This decomposition substantially sped up and simplified our verification. We believe that our approach is useful for other extensive verification projects.

Contributions We summarize our main contributions.

- We present the first comprehensive verification of a full-fledged Internet router. We establish system-wide protocol security properties and local code properties. We discovered and fixed vulnerabilities and bugs in the protocol and its implementation that had escaped extensive prior reviews and testing.
- We model the SCION forwarding protocol in detail, covering crucial features of the implementation, and verify two new network-wide security properties.
- We demonstrate the feasibility of verifying advanced, performance-optimized production code developed by a different team. This required combining,extending, and scaling up various verification techniques to cover the language features and software designs used by this program, and substantial performance improvements to Gobra to handle the complexity of production code.
- We summarize our experience in verifying a complex system independently of its development and distill lessons learned that can be useful for future verification projects.

Our Isabelle/HOL and Gobra formalization and proofs were provided as an artifact to CCS 2025 and are available online [62].

[62]: Pereira et al. (2025), *Artifact for the paper "Protocols to Code: Formal Verification of a Secure Next-Generation Internet Router"*

Collaboration To reiterate, this chapter presents the results of a long collaboration between the Network Security Group, the Information Security Group, and the Programming Methodology Group at ETH Zurich. The new model of the forwarding protocol mentioned in this chapter was developed by our colleagues at the Information Security Group (Tobias Klenze, Sofia Giampietro, Christoph Sprenger, and David Basin) and are described in Sofia's Doctoral Thesis [63]. They found the protocols flaws and proposed the protocol improvements. I led the efforts from the Programming Methodology Group to verify the open-source implementation of the border router and prove that the router implementation refines the models of the protocol developed by the Information Security Group. I was the main contributor of the proofs, and determined how the proof effort was to be organized. Dionysios Spiliopoulos provided important contributions to the proofs, as well as the students we supervised together — especially Markus Limbeck. Together with Felix A. Wolf, I supervised two students — Luca Halm and Lino Telschow — who contributed to early attempts at verifying the codebase.

[63]: Giampietro (2026), *Advances in the Symbolic Verification of Cryptographic Protocols*

3.2 An overview of SCION

SCION is a secure inter-domain network architecture, providing connectivity between ASs. The control plane, performing path discovery and dissemination, executes on a service infrastructure, whereas the data plane, providing packet forwarding, runs on border routers. Although our verification effort focuses on the verification of the border router, we provide an overview of the entire architecture.

SCION network ASs in the SCION network are organized hierarchically, where each AS has bidirectional links to neighboring ASs: *CustProv* links point to provider ASs and *ProvCust* links point to customer ASs. ASs at the top of the hierarchy, i.e., without providers, are *core ASs*, which connect to each other via *Core* links. Figure 3.1 shows an example. Every AS assigns to each of its inter-AS links an *interface*, which, in combination with the AS identifier, is globally unique. SCION routers are *border routers* as they connect the internal AS network to other ASs via inter-AS links.

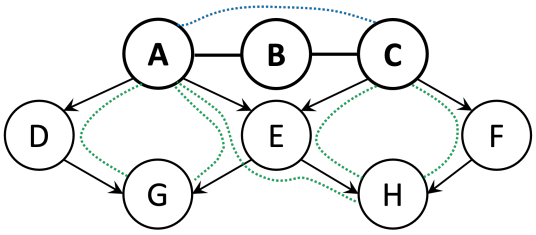


Figure 3.1: Network topology example. ASs are linked hierarchically (arrows), except among core ASs (bold). Dotted lines represent authorized path segments.

Segment construction The control plane discovers and simultaneously authorizes forwarding paths between ASs. For scalability, paths are established along several *authorized segments*. *Down-segments* are constructed along *ProvCust* links starting at a core AS, and *core segments* along *Core* links. *Up-segments* are obtained by reversing down-segments. Core segments can also be reversed for the traversal in the opposite direction. All segments are authorized by the on-path ASs using nested message authentication codes (MACs), which we present in Subsection 3.4.2. Authorizing only certain paths allows network operators to enforce their own policies according to their economic and compliance requirements.

Segments consist of a sequence of *hop fields*, each carrying the forwarding information of one AS. Each hop field contains the interfaces of the incoming and outgoing links (*prev* and *next*), and a cryptographic *authenticator* containing the nested MACs, which the respective router uses to verify that the segment was authorized by the control plane. Crucially, not only the adjacent inter-AS links, but the entire segment is authorized. This rules out subtle *path splicing* attacks [61], in which hop fields from different segments would be combined to craft a new, unauthorized segment, for example crafting path C-E-G without E's consent in Figure 3.1.

[61]: Klenze et al. (2022), *IsaNet: A framework for verifying secure data plane protocols*

Segment combination To send a packet, the source end host combines one or more authorized segments to form an AS-level path from its own AS to the destination host's AS, and embeds this path in the packet

header. Such a path can consist of up to three segments: an up-segment from the source AS to a core AS, a core-segment to another core AS, and a down-segment to the destination AS. Typically, there are multiple paths that the source can choose from.

The combination of segments follows rules that protect the economic interests of ASs. One central goal is to avoid *valleys*, where an AS forwards packets from a provider to a provider (e.g., from A to C via E in Figure 3.1). Since ASs are paid by their customers, and must pay their providers, such packets only create costs but no revenue for the valley AS.

Forwarding In the data plane, routers forward each packet according to their hop field's forwarding information after validating the authenticator. Inter-domain forwarding tables are not required since each packet contains its own forwarding state. Routing and forwarding *within* each AS are performed by the AS's intra-domain routing system and are not part of SCION. We verify SCION's forwarding protocol, which we further explain in Section 3.4.

3.3 Overview of properties and verification

In this section, we summarize the protocol and code properties we establish and the resulting overall security guarantee and present our verification approach (cf. Figure 3.2). Sections 3.4 and 3.6 provide more details on the protocol and code verification, respectively. In Section 3.7, we state the assumptions on which our verification rests.

3.3.1 Verification guarantees

We proved the central security properties of SCION's data plane protocol. Prior work [60, 61] considered only the first property.

[60]: Klenze et al. (2021), *Formal Verification of Secure Forwarding Protocols*
[61]: Klenze et al. (2022), *IsaNet: A framework for verifying secure data plane protocols*

- *Path authorization*: each segment that a packet traverses is contained in an authorized segment.
- *Valley freedom*: packets that already traversed a *ProvCust* link must not traverse a *CustProv* link.
- *Loop freedom*: packets do not follow the same link twice.

These properties protect the honest ASes' interests and rule out malicious end hosts sending packets along unauthorized, uneconomical, or impractical paths. We verify them against a strong attacker model consisting of malicious end hosts with access to the secret MAC keys of a set of *compromised ASes*.

On the code level, we verified the following properties:

- *Memory Safety:* the implementation is well-behaved, that is, it neither crashes nor causes data races.
- *Functional correctness:* the functions compute the intended results, for instance, they perform the required MAC check.
- *Protocol compliance:* the router validates and forwards SCION packets as prescribed by the protocol model.

Thus, our verification ensures that there are no bugs in the code that compromise the security of the overall system, for example, because the code lacks some check prescribed by the protocol.

As explained below, we prove the above-mentioned protocol and code properties separately and soundly link them using the Igloo methodology [23], leading to the following end-to-end guarantee:

[23]: Sprenger et al. (2020), *Igloo: soundly linking compositional refinement and separation logic for distributed system verification*

Theorem 3.3.1 *Any SCION network, whose routers execute the verified router code, satisfies the network-wide data plane security properties under the assumptions from Section 3.7.*

Liveness properties (e.g., sent packets are eventually received) would require additional bandwidth reservation [64] and DDoS prevention mechanisms, and are hence out of scope of this work.

[64]: Giuliari et al. (2021), *Colibri: a cooperative lightweight inter-domain bandwidth-reservation infrastructure*

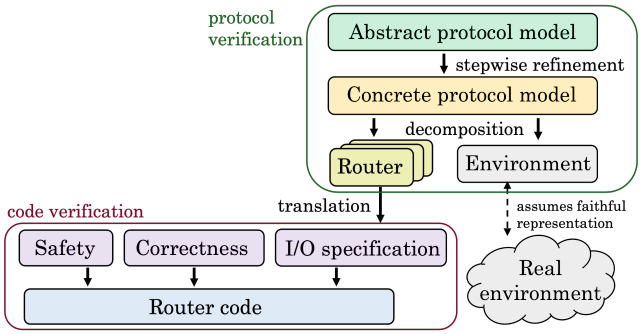


Figure 3.2: Overall verification approach.

3.3.2 Modular verification approach

A key enabling factor in our project is the decomposition of the overall verification effort into manageable chunks, addressing challenge C4. We use five decomposition strategies:

1. *Separation of protocol and implementation:* We verify the SCION data plane protocol and its implementation separately and link the two parts soundly.
2. *Protocol refinement:* We use refinement to develop the protocol model step by step and focus on one protocol aspect at a time.
3. *Protocol parametrization:* We parametrize our protocol models on the cryptographic mechanisms used to secure segments, allowing us to quickly adapt to protocol changes.
4. *Layered code properties:* We annotate and verify the implementation in three layers, each focusing on a different aspect of the intended behavior.
5. *Program modularity*: We use modular program verification, which verifies each function of the program independently.

3.3.3 Separating protocol and code verification

The first form of modularity mentioned above offers several crucial benefits. First, it allows us to leverage the strongest and most appropriate techniques and tools available for each purpose: we use stepwise refinement in Isabelle/HOL for the protocol (Subsection 3.3.4 and Section 3.4) and automated deductive verification in Gobra for the implementation (Subsection 3.3.5 and Section 3.6).

Second, it enables protocol and code verification to proceed largely *in parallel*. Furthermore, since the code properties are *layered* and the safety and many functional correctness properties are independent of the protocol verification (cf. Figure 3.2), verifying those properties allowed us to quickly identify bugs in the pre-existing implementation (and to deliver value immediately).

Finally, there is a clear and explicit interface between protocol and code verification in the form of an *I/O specification*, which we automatically generate from the router model and completely describes the router's intended I/O behavior. By verifying the implementation against this I/O specification, we establish the implementation's protocol compliance. Importantly, code verification does not require any reasoning about the attacker. A soundness result [23] guarantees that the router model and the resulting I/O specification have the same trace properties and, using compositional refinement, also guarantee a trace inclusion between the protocol models and the implementation. Consequently, all properties proved for the protocol model also hold for the implementation, as stated in Theorem 3.3.1, thereby addressing challenge C3.

[23]: Sprenger et al. (2020), *Igloo: soundly linking compositional refinement and separation logic for distributed system verification*

3.3.4 Protocol verification

Security protocol models are commonly verified using state-of-the-art automated protocol verifiers such as Tamarin [65, 66] and ProVerif [67, 68]. These tools are based on the (Dolev-Yao) attacker model, which identifies the attacker with the network. Their network model is unstructured and has no representation of a network topology. While modeling a topology is possible, one would have to limit the topology's size and/or the maximal length of authorized segments to avoid non-termination issues. Moreover, verification time would scale poorly as these limits increase.

[65]: Schmidt et al. (2012), *Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties*
[66]: Meier et al. (2013), *The TAMARIN Prover for the Symbolic Analysis of Security Protocols*
[67]: Blanchet (2001), *An Efficient Cryptographic Protocol Verifier Based on Prolog Rules*
[68]: Blanchet (2013), *Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif*

Since we aim for much stronger results, which hold for *arbitrary* network topologies and *arbitrary-length* authorized segments, these tools are not suitable for verifying SCION's data plane protocol. We therefore use the general-purpose theorem prover Isabelle/HOL [58], whose higher-order logic specification language provides the necessary expressiveness to construct a fully general protocol model that combines a Dolev-Yao attacker with an arbitrary network topology. We develop the SCION data plane protocol in several steps using *refinement* and *parametrization* to structure our models and proofs [69, 70]. These modeling and verification techniques allow us to tackle challenge C1.

[58]: Nipkow et al. (2002), *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*

[69]: Abadi et al. (1991), *The Existence of Refinement Mappings*
[70]: Lynch et al. (1995), *Forward and Backward Simulations: I. Untimed Systems*

We formalize the protocol models as labeled transition systems, where each label, called an *event*, is associated with a transition relation. A sequence of such events is called a *trace*. We formulate a series of such models at different abstraction levels and relate them via refinement proofs (cf. Figure 3.2):

Abstract model: contains all the essential protocol functionality, but only a restricted attacker, and no cryptography.

Concrete model: introduces cryptography to protect the segments, as well as a strong attacker (see Subsection 3.4.1).

Decomposed model: splits the concrete model into an environment model and a router model, and introduces separate I/O events, corresponding to the implementation's I/O library calls.

We express and prove the desired security properties as *invariants* over the models' state, which we also equivalently express as trace properties. Since refinement guarantees trace inclusion, it preserves trace properties and hence invariants. We can therefore prove each security property at its most suitable abstraction level.

3.3.5 Code verification

In contrast to most existing verification projects, the SCION router was developed independently from our verification effort and optimized for speed rather than verification (cf. challenge C2), which substantially complicates verification. In particular, the implementation uses most of Go's language features, including some that are difficult to reason about. These include threads, global state, closures, and interfaces, which require advanced reasoning about concurrency, higher-order specifications, and structural subtyping. Moreover, the router implementation minimizes the usage of heap-allocated memory by reusing data-structures, rather than allocating new instances to reduce the overhead caused by garbage collection. This further complicates our safety proofs.

Choosing the right verification tools and techniques is key to addressing these challenges. To enable modular reasoning in the presence of concurrency, we use concurrent separation logic (CSL) [71]. This choice is almost forced on us, as other widespread logics either do not handle concurrency at all or require explicit, non-modular reasoning about thread interleavings that does not scale. To better handle the size and complexity of the code, we perform automated verification based on SMT solvers, which significantly reduces the effort of writing proofs, at the cost of limiting specifications to first-order logic (which, e.g., makes specifying closures challenging).

[71]: O'Hearn (2007), *Resources, concurrency, and local reasoning*

Thus, to prove that the router implementation is well-behaved, functionally correct, and satisfies the router model's I/O specification, we use Gobra [72], an SMT-based automated deductive verifier for Go. Gobra is based on a variant of separation logic and allows for *function-modular* reasoning about concurrent programs, that is, it verifies each function independently against a user-written specification. Each function specification consists of a precondition, which expresses constraints on the parameters and states in which a function may be called, and a postcondition, which describes the function's results and its side effects. As usual for deductive verification tools, programmers must supply loop invariants. Before our project, Gobra supported most, but not all, of the language features used in the router implementation; thus, we had to extend it to support the missing features. Crucially, Gobra already supported some advanced separation logic specification constructs not available in many automated verifiers, which significantly simplifies reasoning about some code patterns used by the implementation (see Section 3.6).

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

Gobra is designed to be sound, i.e., not miss errors. However, it may produce spurious errors (false positives), mostly when the SMT solver fails to prove a valid condition. In these cases, programmers can introduce additional annotations to guide the proof search so that verification succeeds, like asserting intermediate lemmas in the program text, or by using some strategies described in Chapter 4.

3.4 Protocol modeling and verification

We describe our modeling and verification of the SCION data plane protocol. We build upon an existing model [60, 61], which covers only the first two levels in Figure 3.2 and only considers up-segments, ignoring other segment types and their combination. Furthermore, that model addresses path authorization, but specifies neither valley freedom nor loop freedom. We hence (i) extend this prior model to cover more features of the actual SCION protocol and all desired security properties from Subsection 3.3.1, (ii) add the third level in Figure 3.2, and (iii) extract the I/O specification for the code verification, tripling the size of the prior work. Moreover, for the new protocol attacks that we identified (see Subsection 3.4.4), all but one rely on the additional protocol features.

[60]: Klenze et al. (2021), *Formal Verification of Secure Forwarding Protocols*
[61]: Klenze et al. (2022), *IsaNet: A framework for verifying secure data plane protocols*

3.4.1 Environment model

Network and control plane model The state of our transition systems consists of packets in the sets $int(A)$, the internal network of AS A, and $ext(A, i, B, j)$, the inter-AS link between interface i of AS A and interface j of AS B. We do not model intra-AS networking, as it is not managed by SCION.

Representing the segments produced by the control plane, we parametrize our model over a set of authorized up-, down- and core-segments. Prior work covered only up-segments.

Attacker model Recall that the verified properties protect honest ASes against malicious senders. We assume that

- a subset of ASes may be compromised, that is, their secret keys are revealed to the attacker, but all routers follow the protocol;
- end-hosts in any AS may be compromised, regardless of whether their AS is compromised or not.

Instead of explicitly modeling compromised end hosts and routers deviating from the protocol, our model accounts for them by allowing the attacker to inject packets anywhere (*int* and *ext*) in the network, not only at compromised ASes.

In our abstract model, which does not (yet) use cryptography, the attacker can freely combine path segments to form packets, *but the segments must be authorized*. In the concrete model, we lift this limitation and introduce a full-fledged Dolev-Yao attacker [59], who manipulates symbolically represented messages and can eavesdrop on and inject new packets globally. The attacker's knowledge K consists of eavesdropped messages, all authorized segments, and the secret keys of a set of compromised ASes. The attacker's message forging capabilities are modeled as a closure operator $\mathsf{DY}(K)$, which closes the set K under the messages the attacker can derive (e.g., by constructing MACs with known keys) and hence inject. As is common in symbolic models, we assume perfect cryptography. Hence, only protocol flaws relating to the *use* of cryptographic primitives are considered, not flaws in the primitives themselves.

[59]: Dolev et al. (1983), *On the security of public key protocols*

3.4.2 Protocol models

The abstract and concrete models describe how packets are created, processed, and forwarded.

Packet structure

A packet consists of up to three segments. Each segment consists of a direction flag (*dir*), a *segment identifier* (*segID*), and a sequence of hop fields (cf. Section 3.2). Counters keep track of the current segment and hop field. Figure 3.3 shows an example two-segment packet for the path D-A-E-H in Figure 3.1.

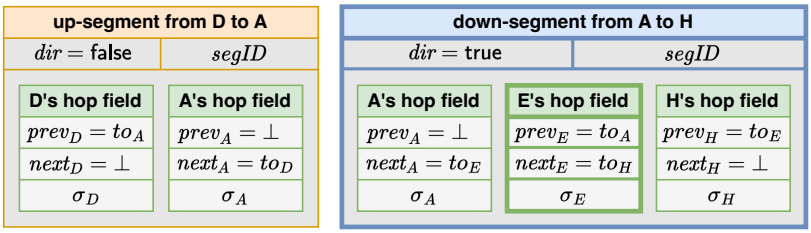


Figure 3.3: A two-segment packet at hop E of the path D-A-E-H in Figure 3.1 (payload not shown). For up-segments, where *dir* = false, the meaning of *prev* and *next* is reversed. The first hop field's *prev* and the last hop field's *next* are empty (⊥). The current segment and hop field have bold frames.

Packets in the abstract and concrete model differ in their use of the segment identifier *segID* and their hop field's authenticator σ_X. In the abstract model, where the attacker cannot modify segments, the AS's name suffices as an authenticator, and segment identifiers are not used. In the concrete model, these fields are instantiated with cryptographic values.

Concrete cryptographic authenticators During segment construction, each AS X on a segment creates a hop field with its local forwarding information (the interfaces $prev_X$ and $next_X$) and authorizes the segment's use by embedding a cryptographic authenticator σ_X in the hop field. This authenticator is a MAC constructed using a local MAC key K_X that is shared between the border routers within AS X:

$$\sigma_X = \mathsf{MAC}_{K_X}(prev_X,\ next_X,\ segID). \tag{3.1}$$

Here, *segID* is the (mutable) segment identifier, which the first AS on a segment initializes with a random value RND. Each subsequent AS X, after computing its σ_X, updates

$$segID := \sigma_X \oplus segID, \tag{3.2}$$

where $\oplus$ is exclusive-or (XOR). Each hop field's MAC thus protects the hop's *prev* and *next* fields and also, via nested MACs, all preceding hop fields. During forwarding, AS X reconstructs σ_X using Equation 3.1, checks that it matches the hop field's authenticator, and forwards the packet with *segID* updated as in Equation 3.2.

In an earlier version of SCION, the MAC directly included the MAC of the previous hop field. This simpler protocol was easier to verify, but it made segment truncation more difficult. In a truncated up-segment, the final router would lack the parent MAC of the last hop field required for authenticator verification, unless it was explicitly included in the segment. The XOR-based solution eliminates the need to include extra

MACs for verification, as it folds the accumulated MACs into the *segID* field.

Events

Our models each have a *router event*, which forwards packets, and an *attacker event*, which injects attacker-fabricated packets into the network.

The router event for a router in AS A receives a packet *pkt* either from an external interface i or from the internal network, checks its validity, updates it, and sends it either to the next hop on the path over interface j or the internal network to be processed by a different router. The router uses two validity checks: $\mathit{ifs_valid}(\mathit{pkt}, A, i, j)$, which ensures the validity of interfaces, in particular that the packet is received from the interface specified in the hop field, and $\mathit{crypto_valid}(\mathit{pkt}, A)$, which checks the validity of the segment identifier and authenticator (only in the concrete model). When a packet moves to a new segment, the router also enforces the segment combination rules (cf. Section 3.2). In the updated packet, denoted by $\mathit{update}(\mathit{pkt})$, the counters for the current hop field and, if needed, segment are incremented and, in the concrete model, the segment identifier is updated.

Overall, the checks and updates by the router event depend on: (i) whether a packet is leaving or entering an AS, (ii) whether forwarding to the next AS happens on the same or a different router in the same AS, (iii) the forwarding direction, and (iv) whether segment switching occurs. We must account for all combinations of these factors. For instance, whether the segment identifier must be updated depends on (i)–(iii), whether it is updated before or after computing the authenticator depends on (i) and (iii), and depending on (iv) more than one segment identifier may need to be updated. These case distinctions adds substantial complexity to our models over the prior work, which only considered a single factor, namely (i). All of the protocol attacks discovered in this work relate to missing checks in subtle combinations of (i)–(iv) (see Subsection 3.4.4).

In the abstract attacker event, the adversary can create packets consisting of only authorized segments, but combined arbitrarily. In the concrete event, we instead allow the attacker to inject arbitrary derivable packets from the set $\mathsf{DY}(K)$ (anywhere in the network). These events also cover the case of honest senders, as all authorized segments are known to the attacker.

3.4.3 Protocol verification

We express the security properties from Subsection 3.3.1 as invariants of the packets' network traversal history, which we record in the packets themselves as a separate *history* field for each segment. This history is just for specification purposes and cannot be manipulated by the attacker. More precisely, the path-authorization state invariant expresses that for all packets in a state, each of their segment's history belongs to the set of authorized segments. We close this set under prefixing and suffixing to allow for partial traversals. The loop-freedom invariant expresses that each inter-AS link $ext(A, i, B, j)$ traversed in the concatenation of the packet segments' histories is unique, while valley-freedom expresses that these links follow an up-core-down order. These properties hold for honest ASes, protecting them from malicious end hosts colluding with on-path ASes. As in prior work [61], they do not hold for compromised

[61]: Klenze et al. (2022), *IsaNet: A framework for verifying secure data plane protocols*

ASes. For example, for loop freedom to hold, there must be at least one honest AS in the loop.

We prove the above invariants for the abstract model, where they are easiest to prove. Since the abstract attacker can only combine authorized segments, path authorization is trivial. This allows us to focus on valley and loop freedom, proving that each abstract event preserves these two invariants. Throughout the proof, we use technical auxiliary invariants regarding the format of packets and hop fields: for example we prove that the interfaces of adjacent hop field match, unless the ASes at both ends of the link are compromised.

For the refinement, we must relate the abstract and concrete states and events, and show that all concrete events have an abstract counterpart. The challenge here is to show that the refinement preserves intra-segment path authorization, since the concrete attacker can seemingly do much more by sending arbitrary derivable packets instead of just authorized segments. Once established, the refinement preserves all three main properties (actually, all trace properties). Moreover, the protocol uses XOR, which is notoriously difficult to reason about [73, 74]. Below, we first explain how we handle XOR and then describe our refinement proof in more detail.

[73]: Abadi et al. (2006), *Deciding knowledge in security protocols under equational theories*
[74]: Escobar et al. (2012), *Folding variant narrowing and optimal variant termination*

XOR model Rather than representing XOR using equational theories, we adopt a normal form representation of symbolic terms [61], where two terms equal modulo the XOR equational theory have identical normal forms. The normal form of the XOR of different terms is represented as a finite set, and the XOR of such sets corresponds to their symmetric set difference. By ensuring that all terms are in normal form, it becomes sufficient to check for equality of finite sets rather than reasoning over equational theories. In line with prior work [61], we over-approximate the attacker's XOR capabilities by allowing the attacker to extract x and y from $x \oplus y$, which greatly simplifies reasoning. This is a sound approximation that suffices for our setting, since the dataplane protocol uses XOR only as an accumulator function, but not for secrecy.

[61]: Klenze et al. (2022), *IsaNet: A framework for verifying secure data plane protocols*

[61]: Klenze et al. (2022), *IsaNet: A framework for verifying secure data plane protocols*

Refinement proof Our refinement proof proceeds in two steps. First, we observe that while in the concrete model, the attacker can send arbitrary derivable paths, forwarding only occurs if the hop fields are valid. We hence show that we can shorten concrete paths to their longest valid prefix in the packet abstraction function. Second, we show that this shortened cryptographically valid path is authorized for any packet that the adversary can derive. For this, we extract the path from a single hop field's MAC, following its nested structure, and show it is authorized. Our proof involves customized induction schemes, tailored to sequences of hop fields and further auxiliary invariants, e.g., that the intruder's knowledge remains constant across all reachable states of our concrete event system.

3.4.4 Results

Artifacts Overall, we produced several formal protocol models of packet forwarding in SCION, together with their execution environment and attacker model. This Isabelle/HOL development consists of 16,100 LoC, containing over 1,000 lemmas, and substantially extends the existing SCION formalization which consisted of 5,500 LoC [61], with around 450

[61]: Klenze et al. (2022), *IsaNet: A framework for verifying secure data plane protocols*

lemmas. Overall, our Isabelle formalization takes five minutes to verify on a standard laptop. We estimate that this development took two to three person years.

The protocol formalization and proofs guarantee that packet forwarding in SCION as deployed today is secure, even in the presence of a strong attacker. We expect them to be useful also to verify SCION's control plane protocols and during SCION's future evolution, for instance, to assess the impact of protocol changes on the intended security properties.

Attacks found and protocol improvements During our verification effort, we found five previously unknown protocol vulnerabilities, giving rise to four novel attacks, see Table 3.1 and Table 3.2. Three of the vulnerabilities are related to the subtle edge cases in the segment switching logic. All vulnerabilities were confirmed and resolved. The most severe attack (Attack 3 in Table 3.2) allowed a malicious sender to fabricate arbitrary forwarding paths, thus violating all three security properties. This attack exploited missing checks in the way that segments are combined and in the limit of the number of segments. An attacker could hence craft a packet that follows an arbitrary path by using a dedicated segment for each individual hop. This attack could be used to send packets in a loop for a bounded number of iterations, causing congestion and network failures.

Table 3.1: List of protocol vulnerabilities.

ID	Vulnerability description	Fixed
V1	Path segment limit not enforced.	✓
V2	Routers can switch from a segment in the down direction to a segment in the up direction, thus creating a valley.	✓
V3	Non-core ASes can switch between segments in the same direction.	✓
V4	Core ASes fail to enforce segment constraints when switching, allowing multiple *Core*-type links.	✓
V5	No mechanism to stop forwarding after hop fields marked for verification only.	✓

Table 3.2: Protocol attacks and their required vulnerabilities (**V1**–**V5**) and violated properties: **P1**—Path authorization, **P2**—Valley freedom, **P3**—Loop freedom.

Attack	V1	V2	V3	V4	V5	P1	P2	P3
1. Inter-segment path splicing: attacker appends an up-segment to another up-segment (or down-down).			•			**X**		
2. Loop / traffic reflection: attacker sends packets in a loop among core ASes, causing congestion.	•		•	•		**X**		**X**
3. Arbitrary source routing: attacker picks arbitrary forwarding paths.	•	•	•	•		**X**	**X**	**X**
4. Splicing: attacker combines hop fields from different segments.					•	**X**		

The vulnerabilities were resolved in updates to the protocol. The updates added several validation checks in the segment switching logic

(addressing **V2-V4**), removed the need for hop fields used solely for verification purposes (addressing **V5**), and enforced a limit of three segments (addressing **V1**). The updated protocol is the one presented and verified in this paper.

Furthermore, we discovered an additional check that routers could perform (checking link types *within* a segment), which allowed us to prove a stronger valley and loop freedom property. Originally, SCION routers checked for valleys only when switching between segments, assuming that the control plane would only construct valley-free segments. While carrying out our formal proofs, we noticed that two compromised ASes, one at each end of a segment, were enough for an attacker to extend the segment with a valley hop and craft a two-segment packet with a loop passing through the honest ASes in the middle of the segment. Our proposed intra-segment link type checks guarantee that a valley or loop can exist only if *all* on-path ASes are compromised, as opposed to *at least some* AS. The developers implemented our recommendation [75, 76].

[75]: Klenze (2022), *Adding checks to intra-segment forwarding to achieve stronger security*
[76]: Frei (2022), *router: add checks for intra-segment forwarding*

3.5 Linking protocol to code verification

To transition from protocol to code verification, we decompose the concrete protocol model into a model for each router and an environment model. We show that the composition of these models refines the original concrete model. In this refinement, we introduce buffering to separate the router's I/O operations from its internal message processing, in order to subsequently map these operations to the implementation's I/O library calls.

We automatically generate an I/O specification from the router model using a technique by Noah Delius [77], which also generates a correctness proof showing its trace equivalence with the router model. At last, we manually translate this I/O specification into Gobra's specification language. This is a straightforward syntactic translation, but it is trusted (see Section 3.7). Our code-level verification includes proving that the router implementation conforms to this specification, as explained in Section 3.6.

[77]: Delius (2021), *Automatic Generation of I/O Specifications from Component Event Systems in Igloo*

The I/O specification obtained from the router model prescribes which operations the router should do *sequentially*. Importantly, the I/O specification is written independently of how the router implementation parallelizes the different steps of forwarding messages, i.e., receiving, processing, and sending of messages. On one hand, generating a sequential I/O specification complicates the proof that the concurrent code implements the protocol correctly (see Subsection 3.6.3). On the other hand, a sequential specification ensures an adequate separation of concerns: by not prescribing a specific way of parallelizing the forwarding of messages, we ensure that the I/O specification is as simple as possible and that it is reusable across different implementations of SCION, potentially written in different programming languages[2].

2: Earlier versions of the SCION router were written in Python. While they are no longer maintained, one could in principle use the approach outlined in this thesis and the same generated I/O specification to verify these versions using Nagini [78].

3.6 Code verification

In this section, we explain how we verify that the router implementation satisfies the properties listed in Subsection 3.3.1, namely safety, functional correctness, and protocol compliance.

We verify the intended properties by annotating all functions in the router implementation with pre- and postconditions and use Gobra to check that the function implementations satisfy their specifications for all possible executions. Annotations are expressed in special comments that are recognized by the verifier but ignored by other tools like the compiler. Since verification in Gobra is modular, we can split the overall verification effort into small tasks, which can be tackled in parallel (modularity aspect 5 in Subsection 3.3.2).

We verify the code in three main steps (modularity aspect 4 in Subsection 3.3.2). Step 1 focuses on memory safety[3], that is, the implementation does not crash or run into undefined behavior due to memory errors. This step guarantees, in particular, the absence of common coding errors such as de-referencing a `nil` pointer, data races, and accessing an array out-of-bounds. Step 2 verifies functional correctness, that is, that all functions compute the intended results. This entails strengthening the pre- and postconditions from step 1 to express and prove the additional properties. Step 3 proves protocol compliance, that is, that the implementation satisfies the I/O specification obtained from the router model. This step relies on the properties proved in step 2, for instance, to show that each packet forwarded by the router was previously validated and its headers updated according to the SCION protocol.

3: There is a widespread belief that Go programs written without depending on the `unsafe` package are memory safe. This is not true, as data races may lead to undefined behaviour [79].

We perform these steps by progressively strengthening the function specifications in a tight feedback loop with the verification tool. Gobra is integrated into the IDE. Verification errors are reported in terms of the original Go code and its annotations; the programmer is not exposed to the underlying verification logic. We also added Gobra to our continuous integration workflow to re-verify the implementation on every change.

The router implementation uses complex language features that are hard to reason about and, therefore, often avoided when code is written with verification in mind. To be able to verify the existing router implementation, we had to develop novel verification techniques for the features used in the original SCION code base and implement them in Gobra. In particular, we developed a novel automated solution for specifying and verifying closures with captured state and side effects. Additionally, for the first time in a Separation Logic verifier, we added support for modularly specifying invariants on global variables, which are established by the package initializers and maintained by the rest of the program (we describe this addition in Chapter 5). We also extended Gobra to support many features of Go that did not require fundamentally new solutions but did require substantial engineering efforts, e.g., support for the built-in types of Go and its unique control structures (e.g., `defer` statements and `for-range` loops), and we added support for (ghost) mathematical types to allow for cleaner specifications. For these extensions, we greatly benefited from the fact that Gobra is based on an intermediate verification language, Viper [18], such that additional Go features can be supported by devising an encoding into Viper and leveraging its existing proof automation. The extensions are now available in Gobra, substantially strengthening its practical applicability.

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

Below, we provide more details on the three verification steps outlined above, discuss how we mitigate tool performance challenges that we encountered in all three, and summarize our results. We illustrate these steps on the function `process` in Figure 3.4, which is a simplified version[4] of the router implementation's function `process`. It takes as arguments the packet `pkt` to process, a structure `d` representing the router's configuration, as well as the ids of the input and output buffers, `i` and `j`. It is called after

4: In the original implementation, the `process` function is responsible for deserializing, validating, updating the packet headers, and serializing the updated packet. The original version validates and updates the packet gradually, in lockstep, and if any of the enumerated steps fails, `process` generates an error packet. To simplify our presentation, we consider here that `process` receives and returns a parsed packet, and that all validations are performed prior to the call.

```
1 //@ requires PktMem(pkt) && DataPlaneMem(d)
2 //@ requires ifs_valid(Abs(pkt), d, i, j) && crypto_valid(Abs(pkt), d)
3 //@ requires Token(l0) && IOSpec(l0, s0)
4 //@ requires Abs(pkt) ∈ s0.inputBuffer[i]
5 //@ ensures  PktMem(pkt) && DataPlaneMem(d)
6 //@ ensures  update(old(Abs(pkt)), Abs(pkt))
7 //@ ensures  Token(l1) && IOSpec(l1, s1)
8 //@ ensures  Abs(pkt) ∈ s1.outputBuffer[j]
9 func process(pkt *Pkt, d *DataPlane, i uint16, j uint16 /*@, l0 ProtoLoc,
        s0 ProtoSt @*/) /*@ (l1 ProtoLoc, s1 ProtoSt) @*/
```

Figure 3.4: The specification of `process` expresses safety (Lines 1 and 5), functional properties (Lines 2 and 6), the I/O spec (Lines 3 and 7), and properties about the I/O abstract state (Lines 4 and 8). `requires` and `ensures` declare pre- and postconditions; `old` allows a postcondition to refer to a pre-state value.

the interface and cryptographic checks. The function implementation, which we omit for brevity, performs the update of the router model's packet processing event (see Section 3.5). We explain the function's specification below.

3.6.1 Verifying memory safety

[21]: Smans et al. (2009), *Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic*

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

Gobra uses implicit dynamic frames (IDF) [21], a variant of separation logic [31, 32]. The expressiveness of this logic is crucial to tackle the concurrency and complex heap structures in the SCION code base and to obtain the modularity needed for a verification effort of this scale. Using this logic, Gobra checks that programs do not crash and do not exhibit data races, as explained in Chapter 2.

Recall that Gobra expresses ownership of memory by associating a *permission* with each location, which is created during allocation and can be transferred between different function executions (Section 2.3). Gobra checks that a function may access a location only if it holds the corresponding permission; otherwise, verification fails. A function's pre- and postconditions express which permissions the function expects from its caller and which it returns upon termination (see Lines 1 and 5 in Figure 3.4). Predicates such as `PktMem` group together the permissions of entire data structures.

A key difficulty in verifying an existing code base that was not written with verification in mind is the use of complex code patterns that improve performance, but complicate reasoning. We give three examples and explain how we tackle them.

[31]: Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*
[37]: Müller et al. (2016), *Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution*

[31]: Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*
[80]: Schwerhoff et al. (2015), *Lightweight Support for Magic Wands in an Automatic Verifier*
[81]: Dardinier et al. (2022), *Sound Automation of Magic Wands*

First, packets have two data representations, as a bytestring and as a struct. Conceptually, bytestrings are unmarshaled into the struct, then processed, and then marshaled back to the bytestring. However, the details are more complicated because (1) these three steps are interleaved, such that the implementation effectively uses both representations, and (2) struct fields store views of portions of the bytestring (so-called *slices* in Go), such that both representations share memory. As a result, it is difficult to specify permissions and the relevant invariants in a purely recursive manner, as is standard in automated verification tools. Our solution instead uses iterated separating conjunctions (ISCs) [31, 37] and magic wands [31, 80, 81], two separation logic constructs that enable non-recursive specifications. ISCs represent permissions for every value of a quantified variable and are commonly used to specify permissions to random access data structures like arrays (e.g., the ISC `forall i :: 0 <= i < len(a) ==> acc(a[i])` gives permission to every element of the array `a`). Magic wands are separating implications commonly used to express permissions to partial data structures (e.g., the wand `tree(x) --* tree(root)` represents the permissions to the tree starting

at `root` with the exception of the subtree starting at `x`). ISCs allow us to describe how the struct's fields overlap with the bytestring, whereas magic wands enable us to preserve information when there is a short-lived overlap between bytestrings and structs. Both of these features, while standard in the theory of separation logic, are very rarely supported in *automated* separation logic verifiers. We have found them (ISCs in particular) indispensable not only for their traditional use cases like specifying arrays and partial data structures, but also due to the additional flexibility they gave us for specifying other code patterns like these overlapping struct representations.

Second, to avoid memory allocation and reduce the time spent on garbage collection, the router maintains a pool of allocated structs, for instance, to represent SCION paths. Whenever a path is created, a struct is retrieved from the pool and returned afterward. Verification must ensure that structs do not continue to be used after being returned to the pool, and that they are properly initialized before they are re-used. We re-use Gobra's existing permission system to prove this property: When a struct instance is returned to the pool, we prove that all permissions to it are returned to the pool. Thus, it can no longer be used by other code, since that would require those same permissions that are now no longer available.

Third, the router implementation makes extensive use of interfaces for subtype polymorphism. To verify code that uses these interfaces, we have to annotate each interface with specifications that must be satisfied by each implementation of the interface. Due to Go's structural subtyping, one cannot easily determine all implementations of an interface. This makes devising interface specifications challenging, as one must anticipate the behavior of each interface implementation in the entire code base. As a result, we frequently adjusted interface specifications in the SCION libraries as we verified new implementations that were not compatible with the previous version of the specification. Each such change required re-verifying the already-verified interface implementations against the adjusted specification. This highlights a limitation of modular verification when applied to existing code.

3.6.2 Verifying functional correctness

Functional properties describe the result and the state updates performed by a function. Even if a function's *implementation* operates on low-level data structures such as bytestrings, its *specification* needs to express the functional properties in terms of the logical values that the low-level data structures represent. This *data abstraction* is essential to making specifications human-readable and connecting them to the state of the protocol model.

The implicit dynamic frames logic used by Gobra allows programmers to express data abstraction via mathematical functions, written as side-effect-free Go functions, that map a concrete data structure to its logical representation. In contrast to standard separation logic, these *pure* functions decouple permission reasoning from reasoning about the values stored in memory. This allowed us to incrementally *extend* the specifications for safety properties discussed earlier with functional properties.

In our example, the pure function `Abs` abstracts the concrete representation of a packet `pkt` to a message in the protocol model. This function is used in the precondition of `process` (Line 2 in Figure 3.4) to express that

the packet header has been validated. The postcondition at Line 6 states that `process` modifies the packet as prescribed by the router model's *process* event. Crucially, specifying these functional properties in the contract of `process` does not require changing the safety specifications previously added.

This step specified and proved critical properties of functions that parse, serialize, and manipulate the headers of the SCION packets. Many of these functions are part of the API of SCION. As such, our security guarantees apply not only to the router implementation, but also to third-party clients of the SCION libraries.

Abstracting cryptographic material

To relate the messages at the protocol level with those in the implementation, we assume a function α mapping bitstrings to terms, inspired by the work by Sprenger et al. in a verified two-party authentication protocol [23, Sec. 4.2]. The existence of such a function is a standard assumption in symbolic protocol verification. It represents the common abstraction of treating bitstring messages as well-typed terms under a perfect cryptography assumption, which precludes, for instance, collisions between different message types and cryptographic collisions (e.g., MACs). We also assume that the cryptographic operations at the code level correctly implement the corresponding abstract operations by relating them using α.

Formally speaking, α is a surjective function of type

$$\{0,1\}^+ \times \text{TypeTag} \to \text{Term}$$

The type $\{0,1\}^+$ stands for the type of finite bitstrings, TypeTag is the type of *labels*[5] that indicate whether a bitstring was obtained from computing a MAC, or a XOR (an in particular, whether it is a segment ID obtained from XORing the MACs computed at each router), or corresponds to a non-cryptographic field in the packet, and Term stands for the symbolic cryptographic algebra used in the router model.

5: The label of a field of a marshalled SCION packet is inferable from its offset in the packet bitstring.

Similarly to the protocol model's attacker (Subsection 3.4.1), the message forging capabilities of the *bitwise attacker*, i.e., the Dolev-Yao attacker that can observe and inject labeled bitstrings, is modeled as a closure operator $\mathsf{DY}'(K)$, which closes the set of tagged bitstrings that consistute the attacker's initial knowledge (K) under the messages the attacker can derive. It is defined as follows[6]:

$$\mathsf{DY}' = \overline{\alpha^{-1}} \circ \mathsf{DY} \circ \overline{\alpha} \tag{3.3}$$

6: For any function f of type $A \to B$, we use the notation $\overline{f}$ to lift f to a function of type Set$A \to$ Set$B$ that applies f to all elements of its input.

Abstracting over bitstrings using the α function ignores the existence of collisions, i.e., it assumes that every pair of bitstrings and labels corresponds to a single term. We deem this modeling decision realistic. In SCION, MACs are required to use pseudorandom functions (PRFs), making collisions unlikely. The situation is more complex for XOR, as it is trivial for the attacker to craft XOR collisions. However, in SCION, XOR is used only in the segment identifiers, which have the form $MAC_k(a, X) \oplus X$. Since XORing with a random value yields a random result, the problem of finding a collision for such terms again reduces to the PRF security of the underlying MAC scheme. Furthermore, an attacker would need to find a collision such that the resulting hop fields have valid interfaces that can be used for forwarding.

```
1 pred IOSpec(l ProtoLoc, s ProtoSt) {
2   (forall i, pkt :: pkt in s.outputBuffer[i] ==>
3     Send(l, i, pkt, ?l2) &&
4     IOSpec(l2, s)) &&
5   (Recv(l, ?i, ?pkt, ?l2) &&
6     IOSpec(l2, s[inputBuffer[i] += pkt])) &&
7   ...  // other I/O permissions
8 }
```

Figure 3.5: The core predicate of the I/O specification generated from our router model. The predicate includes permissions for all I/O operations prescribed by the model (we show only the `Send` permission here). The recursive predicate application shows how the position in the protocol (`l`) and the model state (`s`) are updated when an I/O operation is performed. Variable $?l_2$ is existentially quantified.

The inability of an attacker to find accidental segment identifier collisions is a fundamental correctness property of SCION's control plane, which we assume to be correctly implemented:

> In principle, in case there is a collision of a segment identifier, splicing multiple segments would become possible. However, as segment identifiers are not under the control of end hosts, they cannot be brute-forced; instead, end hosts would need to wait for accidental collisions. In addition, the probability of an "accidental" match decreases exponentially with the length of the segment identifier, l_{SegID}.
>
> — The Complete Guide to SCION: From Design Principles to Formal Verification [46, Section 5.3.3]

Our proofs are carried out in a symbolic model, which is unable to reason about collisions and their likelihood. Cryptographic proofs avoid this issue, but they are often difficult to formalize and scale for complex protocols. Computational soundness could bridge symbolic and computational models, but this is impossible for most symbolic models that include XOR, including ours [82].

[82]: Unruh (2010), *The impossibility of computationally sound XOR*

3.6.3 Verifying protocol compliance

To verify that the router code implements the protocol correctly, we prove that each I/O operation performed by the router is permitted by the abstract router model. To reason about I/O behavior, we associate a separation-logic permission with each I/O operation that may be performed by the implementation. We annotate the relevant I/O operations of Go's I/O library to require and consume the corresponding permission. Consequently, a caller may perform the I/O operation only if it holds the permission; otherwise, verification fails. This approach was originally proposed by Penninckx et al. [83]; we handle concurrency differently to make the router model and I/O specification independent of the concurrency used in the implementation (see below).

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

I/O specification The I/O specification generated from the router model (see Section 3.5) is a separation logic formula that provides the I/O permissions for the entire router, that is, expresses which I/O operations the router may perform, their arguments, and in which order they may occur. Figure 3.5 shows the definition of the predicate `IOSpec`, the core of the I/O specification. The predicate's parameter `s` represents the router model's state; its relation to the router's concrete data structures is expressed via abstraction functions (Subsection 3.6.2). `IOSpec`'s other parameter, `l`, is a *protocol location* that indicates the execution's current position in the protocol; it is used to specify valid sequences of I/O

```
1 //@ requires SktMem(conn) && PktMem(pkt)
2 //@ requires Token(l1) && Send(l1, conn, Abs(pkt), ?l2)
3 //@ ensures  SktMem(conn) && PktMem(pkt)
4 //@ ensures  Token(l2)
5 func Write(conn uint16, pkt *Pkt /*@, l1 ProtoLoc @*/) /*@ (l2 ProtoLoc) @
      */
```

Figure 3.6: A simplified specification of the `Write` function of Go's I/O library. The function sends packet `pkt`; `conn` is the network socket representing the AS interface that the packet is sent over. Lines 1 and 3 specify permissions for the packet and socket. Lines 2 and 4 specify I/O permissions.

operations and is advanced whenever the router performs a protocol step, such as sending a packet.

An individual I/O permission is parameterized with the protocol locations before and after the operation, as well as the parameters and results of the operation. For instance, Line 3 in Figure 3.5 provides the permission to send packet `pkt` over the AS interface `i`, provided that the packet is in the corresponding output buffer (Line 2) and the protocol is at location `l`. Performing this operation advances the protocol to location l_2. The recursive instance of `IOSpec` at Line 4 lets the router perform the next I/O operation. Its parameters reflect the new protocol location and the fact that the router state remains unchanged. Line 5 provides the permission to receive the next packet `pkt` over the AS interface `i`. The router state is updated so that `pkt` is added to the input buffer. The condition under which an I/O operation is permitted and the modification of the router state are extracted automatically from the guard and update, respectively, of the corresponding event in the router model.

The router's full I/O specification, extracted from the router model, is `Token(l) && IOSpec(l, s0)`, where s_0 is the initial abstract state and `Token(l)` indicates the initial protocol location. This specification, which is a precondition of the router's entry-point, describes the permitted I/O operations of the entire router.

Verifying I/O properties Figure 3.6 shows a specification of the `Write` function from Go's I/O library, which forwards messages to the network. In addition to the memory permissions specified at Line 1, calling `Write` also requires the current protocol location to be l_1 and a `Send`-permission to send the packet `pkt` over the AS interface's network socket `conn` (Line 2). To verify the following call to `Write`

```
/*@ l1, _ := @*/process(pkt, t, conn_in, conn_out /*@, l, s @*/)
Write(conn_out, pkt /*@, l1 @*/)
```

Figure 3.7: Annotated call to the `Write` function.

we use the postconditions at Lines 7 and 8 of `process` (Figure 3.4), together with the definition of `IOSpec`, to obtain the I/O permission required by the `Write` operation.

An attempt to call `Write` *without* first calling `process` would fail for two reasons: First, we could not establish `Token(l1)`, i.e., the execution would be at the wrong protocol location. Second, we could not obtain the I/O permission from `IOSpec` because the condition `pkt in s.outputBuffer[i]` would not hold, i.e., the router would not be in the expected state. This illustrates that verification enforces that both the sequence of I/O operations and the evolution of the router state precisely follow the protocol model.

Concurrency The router implementation spawns multiple threads, each of which receives packets, processes them, and forwards the result. The receive, process, and send operations may occur concurrently, while operations *on the same* packet are sequenced (i.e., a packet is received, processed and sent in sequence by the same thread). The router model, from which we extract our I/O specification, is agnostic to the concurrency used in a concrete implementation. The I/O specification is parametrized with a *global* router state (`s` in Figure 3.5) and describes how this state is affected by each operation.

Due to the global nature of the I/O specification, verification must ensure that each thread may obtain I/O permissions and that all threads collectively maintain the global state used in the I/O specification. To this end, we treat the I/O specification like a shared data structure in concurrent separation logic. This data structure consists of (1) global ghost (that is, verification-only) variables `cL` and `cS` for the current protocol location and the current model state, and (2) an invariant `Token(cL) && IOSpec(cL, cS)` for the current token and I/O specification. We enforce that each protocol step performed by a thread executes atomically and preserves the invariant [84]. In this atomic step, a thread may use the token and `IOSpec` predicate from the invariant (e.g., to perform an I/O operation), update `cL` and `cS`, and then release the token and `IOSpec` predicate to make them available for the next thread.

[84]: Jung et al. (2015), *Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning*

Notice that the `process` function is not atomic, and thus, the invariant may not be open on a call to `process`. As such, no call to it can satisfy the precondition in line 3 of Figure 3.4. To solve this issue, we adapt the specification of `process` as shown in Figure 3.8:

```
1 //@ requires PktMem(pkt) && DataPlaneMem(d)
2 //@ requires ifs_valid(Abs(pkt), d, i, j) && crypto_valid(Abs(pkt), d)
3 //@ requires Invariant(acc(&cL) && acc(&cS) && Token(cL) && IOSpec(cL,
      cS))
4 //@ requires IBufWitness(i, Abs(pkt))
5 //@ ensures  PktMem(pkt) && DataPlaneMem(d)
6 //@ ensures  update(old(Abs(pkt)), Abs(pkt))
7 //@ ensures  OBufWitness(j, Abs(pkt))
8 func process(pkt *Pkt, d *DataPlane, i uint16, j uint16)
```

Figure 3.8: Adapted specification of `process` that allows for concurrent calls.

Notably, the specification of `process` no longer receives the current protocol location l_0 and state s_0 as a parameter, and it does not require the caller to own the resources `Token(`l_0`)` and `IOSpec(`l_0`, `s_0`)`. Instead, it requires that `acc(&cL) && acc(&cS) && Token(cL) && IOSpec(cL, cS)` is an invariant (line 3). Once a property is shown to be an invariant, it will always be an invariant. Thus, repeating the assertion from line 3 as a postcondition is not necessary. The implementation of `process` is now responsible for opening the invariant and updating the local and state of the protocol atomically after processing the packet.

The specification of `process` may not access the current state and location of the protocol, as it does not have permissions to variables `cS` and `cL`, respectively (it may only obtain them by performing an atomic operation). As such, the precondition in line 4 and the postcondition in line 8 of Figure 3.4 cannot be expressed directly. Instead, we define the predicate `IBufWitness(i, pkt)` (resp., `OBufWitness(i, pkt)`) which entails that the packet `pkt` was added to the input (resp., output) buffer of interface `i`. `IOSpec` only allows monotonic additions to the protocol state, i.e., packets may be added to any input or output buffers, but they may never be removed. Thus, the validity of `IBufWitness(i, pkt)` and

`OBufWitness(i, pkt)` is preserved by all updates to the protocol state. Even though Gobra does not have any built-in support for reasoning about monotonicity, we have *soundly*[7] implemented these predicates and reasoning principle in Gobra [85] using a formalization of *ghost locations* and *resource algebras* (in particular, using the *authoritative reource algebra*) from the Iris program logic [28].

The way of verifying I/O properties for concurrent programs shown in this section does not restrict the concurrency used by the implementation and allows the protocol models and the extracted I/O specification to be completely agnostic of the implementation's concurrency structure, which further increases the modularity of our approach.

7: We have implemented each proof rule for resource algebras and ghost locations from Iris in Gobra. Each rule corresponds to a ghost function in Gobra, where its premises occur as preconditions and its conclusions occcur as postcondtions. We have proven each rule sound in Gobra.

[85]: (2025-2026), *`resalgebra`: A library for resource algebras in Gobra*

[28]: Jung et al. (2018), *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*

3.6.4 Results

We equipped the router's implementation with annotations that allow Gobra to prove safety, functional correctness, and compliance with the protocol model. We targeted the current, open-source implementation, which consists of 4,700 lines of Go code (ignoring comments and empty lines), including SCION-specific libraries, but excluding third-party libraries like the Go standard library or the library `gopacket`. We achieved our goal of verifying the deployed, performance-optimized implementation as is, except for three small and local changes to work around limitations of Gobra. First, we rewrote one type declaration that uses a specific combination of Go's interfaces and Go's delegation mechanism that is not supported by Gobra. Second, we split some compound expressions, which allows us to add necessary annotations about intermediate results. Third, we rewrote some `range`-loops into regular `for`-loops, which in some cases simplifies permission-based reasoning.

Altogether, we fully verified 345 functions across 12 packages. For three additional functions, we verified only parts of its implementation. Due to performance problems observed in these functions, their relative lack of importance (the unverified parts deal with error reporting and a SCION extension that is not a focus of our project), and our timing constraints, we never fully verified these functions. Even though these functions are only partially verified, we are convinced that they are correct. Nonetheless, they are a part of our TCB until their proofs are finalized (Section 3.7).

In total, we added 16,700 lines of specifications and annotations, including 1,000 for the I/O specification and the definitions it depends on. We wrote another 2,600 lines of trusted specifications for the Go standard library and third-party libraries. The overhead of 3.6 lines of annotation per line of code [8] is typical for SMT-based deductive verification and would be substantially higher for verification using an interactive theorem prover. Annotating and verifying the code took roughly 2.7 person years (not including the time required to extend Gobra); running Gobra on the implementation takes 46 minutes on a commodity desktop. We expect that our specifications will also facilitate the safe evolution of the code base; after annotating any modified parts, Gobra can check whether the changed implementation is still correct and secure.

8: This ratio is suggestive of the amount of work that goes into verifying the router against its current specification. However, this measurement provides little information otherwise, and in particular, it should not be used to draw comparative conclusions with other projects. Gernot Heiser has an instructive blog post on this point [86].

Discovered implementation bugs We reported fourteen previously unknown issues in the router implementation, all of which were confirmed and fixed by the SCION developers. The detected issues affect the router's safety, functional correctness, and the I/O behavior. We list a subset of the errors we found in Table 3.3. While verifying safety

Issue description	Type
Missing bounds check when reading a host address from a bytestring	S
Missing bounds check in the function that serializes a SCION path	S
Missing synchronization operation in the main loop of the router	S
Latent race condition when processing packets with empty paths	S
`nil`-pointer dereference when reversing an empty SCION path	S
Setter function does not prevent concurrent mutations to values of type `DataPlane`	S
Improper error handling when forwarding to the internal network	F
Missing checks after decoding a SCION path	F
Incorrect error reporting when reversing paths	F
Deserializing an extension header may incorrectly reuse fields from previous deserialization	F
Use of wrong operand in a type comparison leads to a comparison that always yields false	F
Missing checks for the number of hop fields in a SCION path	F
Packets simultaneously originating from, and bound to the internal network are not rejected	P
Missing checks for the validity of the size of path segments	P

Table 3.3: Selected issues reported in the official SCION repository on GitHub.

(referred to as S in Table 3.3), we found missing bounds checks, a `nil`-pointer dereference, and data races. Verification of functional correctness (F) uncovered missing checks after decoding SCION paths, incorrect error handling, and the incorrect accumulation of metrics. Finally, while verifying protocol compliance (P), we discovered a critical bug where the router did not reject packets simultaneously originating from and destined to the internal network. As a consequence, an attacker could craft an ill-formed packet that keeps being redirected to the internal network of an AS, potentially jeopardizing the availability of the router. This behavior is explicitly ruled out by the SCION protocol and our models, but was allowed by the implementation. These findings demonstrate that all steps of the code verification were effective in uncovering bugs. Remarkably, all of these bugs escaped the extensive code reviews, testing, and fuzzing that are continuously performed on the code base, in parallel with formal verification.

In addition to these bugs, we identified a loop that could run for at most one iteration, and proposed a change that replaced the loop with straight-line code. This change had the positive side-effect of simplifying our proofs, eliminating the need for loop invariants and termination measures.

Impact on the Gobra and Viper ecosystems VerifiedSCION is, to date, the biggest program ever verified in a Viper frontend. In the course of this project, we often had to extend Gobra to support previously unsupported features of Go and find new ways to scale verification (Chapter 4). VerifiedSCION had a big impact on the Gobra and Viper ecosystems besides these contributions.

Through VerifiedSCION, we explored less frequently executed paths and configurations of Gobra and Viper. This uncovered various bugs (including soundness and completeness bugs) in both Gobra and Viper, which have since been fixed. Even for the features initially developed independently of VerifiedSCION, like the support for checking *secure information flow* (which we discuss in Subsection 3.6.5), our verified codebase provided a litmus test for their expressiveness and identified various shortcomings of the original implementations. In addition, VerifiedSCION was a big motivator, as well as a judge, for performance optimizations carried out in Gobra and Viper, which led a member of our group to coin the sentence "If it works on SCION, it works".

VerifiedSCION was the product of various collaborators, many of whom were working on verified codebases for the first time. This motivated the development of a GitHub Action [87] to ensure that all contributions to the codebase did not introduce new verification errors, and led to work on new teaching material for Gobra [88, 89] and a linter to detect specification issues [90]. In addition, we developed various utility libraries to reason about slices, concurrency (through resource algebras), and ghost data structures. Many of these utilities were collected in a reusable library that acts as a standard library for Gobra [91]. Finally, we formally specified part of the Go standard library. To reduce our TCB (Section 3.7), we started verifying part of the Go standard library [92].

[87]: Gasser et al. (2022), *`gobra-action`'s Github repository*

[88]: Pereira et al. (2021), *A Tutorial on Gobra*

[89]: Gottschall et al. (2025), *Practical Program Verification in Go with Gobra*

[90]: Herz (2025), *A Specification Linter for Gobra*

[91]: Nezamabadi et al. (2024), *`gobra-libs`'s Github repository*

[92]: Montini (2025), *Specification and Verification of parts of the Go Standard Library using Gobra*

3.6.5 Ongoing work

In addition to the properties previously described, we are currently extending the VerifiedSCION project to verify other kinds of properties. Furthermore, we are currently looking at how to integrate verification in the development process of SCION. Here, we describe these ongoing efforts.

1. Proving finite processing of packets

We are interested in proving that each packet received by the router is processed in finite time. To that end, we need to show that (1) in the absence of deadlocks, each function involved in packet processing terminates, even though the router as a whole may run forever, and (2) there are no deadlocks. So far, we completely achieved the former. To prove termination, we added a *termination measure* (or *ranking function*) to each loop. Gobra checks the following conditions:

1. the termination measure strictly decreases in every loop iteration,
2. the termination measure is always non-negative,
3. the loop body contains only calls and nested loops that terminate.

Together, these three properties imply that the loop terminates. Functions are handled analogously; their termination measures must decrease on every recursive call, including indirect recursive calls.

In order to prove the absence of deadlocks in Gobra, we consider extending Gobra with support for verifying *finite blocking* [93], i.e., proving that no thread is blocked forever. We expect this to be mostly an engineering task, and we believe that we may be able to reuse Nagini's encoding for checking this property [94].

[93]: Boström et al. (2015), *Modular Verification of Finite Blocking in Non-terminating Programs*

[94]: Astrauskas (2016), *Input-Output Verification in Viper*

2. Proving Secure Information Flow

Each border router in an AS A is configured with A's confidential MAC key so that the router can check whether paths are authorized. In principle, the router implementation could leak secrets indirectly by performing output that depends on the MAC key. In such a case, the attacker could obtain secret information by observing the I/O behavior of the system and then use this information to mount an attack.

More recently, we started verifying that the router's implementation does not leak information derived from the key except for the updated *segID* computed in each hop along the forwarding path (Subsection 3.4.2). We assume a threat model where an attacker may observe any messages sent to and from the router, but it may not inspect the memory of the router directly.

We follow the methodology by Wolf and Müller [95] to specify and verify *security policies*, i.e., policies describing under which conditions confidential data may be declassified. We introduce the security policy in two steps:

[95]: Wolf et al. (2024), *Verifiable Security Policies for Distributed Systems*

1. we extend the router's I/O specification with a new *declassify* event, signaling the declassification of a message's *segID*. This event is enabled for every message to be sent by the router. We show, on paper, that the I/O specification extended with declassification refines the previous router I/O specification [96], and thus, it preserves all of its safety properties.
2. we introduce a *classification policy* to describe the expected confidentiality of the inputs and outputs of the events of our I/O specification. For each event `E(out, in)`, where `out` is the output of the action and `in` is the input, we provide a classification specification as a Hoare triple $\{ P \}$ `E(out, in)` $\{ Q \}$, where P and Q describe the lowness of the outputs and inputs, respectively. If an action does not have inputs or outputs, we omit the corresponding argument.
   We provide the following triples for our I/O events:

   $$\{ true \} \texttt{ recv(pkt) } \{ low(\texttt{pkt}) \}$$

   $$\{ low(\texttt{pkt}) \} \texttt{ send(pkt) } \{ true \}$$

   In addition, we provide the following triple for the *declassify* event

   $$\{ true \} \texttt{ declassify(x) } \{ low(x) \}$$

   In the triples above, we use the assertion $low(e)$ to indicate that e has *low* sensitivity, i.e., it is not confidential. Thus, the triples above specify that all messages received by the router are not confidential (as the attacker may have observed them), whereas the router may send only non-confidential messages. Finally, declassifying a value turns that value non-confidential.

[96]: Inndorf (2025), *Verifying Secure Information Flow for the VerifiedSCION Router*

We use Gobra to ensure that the router complies with the security policy by showing (1) that the router complies with the I/O specification extended with the *declassify* event using the approach outlined in Subsection 3.6.3, and (2) the router complies with the classification policy by using Gobra's support for checking secure information flow (SIF), which is specified as a variation of *observational determinism* [97, 98] that accepts non-deterministic I/O behaviours that do not depend on secrets [95].

[97]: McLean (1992), *Proving Noninterference and Functional Correctness Using Traces*
[98]: Roscoe (1995), *CSP and determinism in security modelling*
[95]: Wolf et al. (2024), *Verifiable Security Policies for Distributed Systems*

As done for all properties specified and proven before, we layer the classification annotations on top of the pre-existing specifications and

verify SIF gradually. We start by annotating the methods that perform I/O with their classification specifications. As an example, we add the precondition `Abs(low(pkt))` to the `Write` function (Figure 3.6), requiring that the packet to be sent does not contain confidential information. To verify the call to `Write` in Figure 3.7, we must specify that the `process` function does not leak the key when a packet is processed. We do so by adding `low(Abs(pkt))` as both a precondition and postcondition of `process`, which in turn, requires that we extend the specifications of the methods called from `process`. In this way, SIF annotations propagate to the entire codebase.

So far, we have added SIF annotations to all first-party dependencies of the SCION router, and we verified all but one of these dependencies. While doing so, we identified and fixed various problems with the support for verifying SIF in Gobra, as the original implementation was still experimental and didn't integrate well with features like magic wands, type casts, and closures. We are currently verifying the remaining unverified dependency of the router and the router package itself. Our current progress is slowed down by the following two problems, which we are working to solve:

Performance problems. We are currently experiencing performance and verification stability problems in the currently-unverified packages. We believe these problems originate from the product program construction used by Gobra to verify SIF [99, 100], which significantly increases the size of the Viper programs generated by Gobra and negatively affects verification times (Chapter 4). Similar to the work by Frumin et al. [101], we are currently exploring combinations of the product program transformation with a SIF type system for which there is an efficient type-checking algorithm, so that Gobra applies the product program transformation only when SIF cannot be checked automatically by the type-checker.

[99]: Eilers et al. (2019), *Modular Product Programs*
[100]: Eilers et al. (2021), *Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security*
[101]: Frumin et al. (2021), *Compositional Non-Interference for Fine-Grained Concurrent Programs*

Idiomatic specifications. As mentioned above, we layer the SIF annotations on top of the specifications for all aforementioned properties. In the style of IDF, we would like to abstract over the confidentiality of data structures using pure heap-dependent functions. As an example, we would like to define the following pure function that returns whether the contents of a slice are confidential [9]:

```
1 ghost
2 requires forall i int :: 0 <= i && i < len(s) ==> acc(&s[i])
3 pure func LowSlice(s []int) bool {
4   return low(len(s)) &&
5     forall i int :: 0 <= i && i < len(s) ==> low(s[i])
6 }
```

9: Note that, for a slice `s`, the assertion `low(s)`, implies only that the memory address of `s` is low, not its contents.

By the time we started specifying and verifying SIF, the product program transformation that Gobra depends on did not yet support `low` assertions in pure functions. We have prototyped an extension of the product program transformation that allows for these idioms, and we are currently evaluating its expressiveness on the router implementation.

3. Integration with the upstream SCION repository

In the process of verifying the SCION router and its dependencies, we maintained a separate fork of the SCION repository where we added our

specifications and proofs of correctness. We are currently exploring the migration of the specifications and proof annotations from the Verified-SCION fork to the official SCION repository and the addition of Gobra to the CI workflow of the SCION repository. So far, we have opened two pull requests to the SCION repository [102, 103] that added basic memory safety specifications and proofs to two important packages: `pkg/addr`, which defines the addresses of the SCION hosts, and `pkg/slayers/path`, which defines a common interface implemented by all kinds of paths that the border router supports. These proposals are the first step in transferring the responsibility of proof maintenance to the developers of SCION.

[102]: Pereira (2022), *`scionproto/scion`: Add Gobra to the CI and specify and verify infofield.go and hopfield.go (experimental)*
[103]: Pereira (2025), *experimental: add specifications to packages `pkg/addr` and `pkg/slayers/path`, verify them in the CI with Gobra*

3.7 Assumptions and TCB

Our verification methodology guarantees that the properties we show to hold for the protocol model are preserved all the way down to the routers' implementation. Our guarantees hold under the following assumptions:

Environment. We prove the security of the SCION router based on a model of the environment consisting of a network and an attacker model; we assume that these models cover the behavior of the real environment. We assume the absence of collisions in cryptographic material that justify the adequacy of our mapping α from bitstrings to symbolic cryptographic terms, as explained in Subsection 3.6.2, which in turn depends on the correctness and security of the control plane.

Manual translation of I/O specification. We connect the protocol model to code via an I/O specification. We assume the I/O specification is translated correctly from Isabelle/HOL to Gobra's syntax. This translation is manual, but it can be automated in the future.

Partially verified functions. As mentioned in Subsection 3.6.4, three functions are only partially verified.

Libraries. The router implementation uses external libraries (Go's standard library, Prometheus [104], and GoPacket [105]), whose behaviors we specified using pre- and postconditions, but we did not verify their implementations. We did the same for auxiliary SCION libraries that perform logging, allocation of error values, and declaration of new GoPacket's layer types. Finally, we specified but did not verify the implementation of network operations for sending and receiving packets.

Tool soundness. We assume that the Isabelle/HOL and Gobra verification tools are sound. Moreover, we verify the router's source code, but assume the correctness of the compiler, runtime system, operating system, and hardware.

3.8 Lessons learned

In this section, we reflect on our project and draw lessons learned for future large-scale verification efforts.

3.8.1 Overall approach

The most important success factor for our project was to separate protocol and code verification, which worked remarkably well.

First, this separation proved to be very valuable, since it allowed us to use the right tools for each task. Performing both verification tasks within one framework would have been much harder. For example, to simplify reasoning about path segments, we derived tailor-made induction schemes. We also built protocol models that are parametrized on cryptographic schemes and assumptions about them. These features are not commonly available in SMT-based program verifiers. However, performing the entire verification within an interactive proof assistant would have meant forgoing the high degree of automation offered by SMT-based program verifiers and incurred a significant increase of the verification effort.

Second, our project demonstrated that the linked protocol and code verification processes are *mutually* beneficial, as opposed to the protocol models just serving as a specification of the code. By soundly linking the models to the code, we detected not only discrepancies between the models and the specified security properties, but also instances where the models did not faithfully capture the informal protocol specification. For example, initially our models performed checks on packets upon exiting an AS whereas the implementation correctly conducts checks on entry. In another example, described in Subsection 3.6.4, a discrepancy between models and code turned out to be an error in the implementation instead.

Using two frameworks, Isabelle and Gobra in our case, requires a semantic-preserving translation between them. The existing Igloo methodology soundly links the two formalisms using I/O specifications. We extended it with an automated, sound extraction of I/O specifications in Isabelle/HOL, which made the extraction of the router's I/O specification effortless and reduced its further translation to Gobra to simple syntactic transformations (see Section 3.7). The generated I/O specification is relatively simple and can be reused for different router implementations.

Overall, our success suggests that our verification approach can be used in other verification projects that target protocols and their pre-existing implementations. It is flexible enough to accommodate any protocol modeling approach based on transition systems. This includes, for example, Tamarin protocol models [106], and many program verifiers based on separation logic for different programming languages, including C, Go, Java, Python, and Rust.

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

3.8.2 Continuous development

Like most software, the SCION protocol and code are constantly evolving. In such projects, one cannot defer verification until the software reaches a stable state. Instead, we started our verification effort already when SCION was just a research project and evolved our formalizations and proofs along with the protocol and code.

Verifying the protocol and code already *during* their development has substantial benefits. As described in Subsection 3.4.4, we uncovered vulnerabilities and made suggestions to strengthen the obtained security guarantees, which were implemented early. In contrast, updating already

widely-deployed protocols is hard and costly. Similarly, detecting implementation bugs early reduces the risk of their exploitation by attackers, which would cause economic and reputation damage. This is especially critical for a secure next-generation Internet architecture that is seeking widespread acceptance.

However, the early start also led to significant challenges. We faced multiple protocol changes that affected our models. Evaluating their effects on the security properties would have required laborious modifications to all models in the refinement sequence for each proposed change. To adapt to changes quickly, building on prior work [61, 107], we used parametrization to prove the security of an entire class of data plane protocols. The formalization defines several parameters (such as a cryptographic check function) and conditions that are sufficient for proving security in the parameterized refinement development. Since the proof effort for each instance is minimal, we could easily adapt to protocol changes.

[61]: Klenze et al. (2022), *IsaNet: A framework for verifying secure data plane protocols*
[107]: Klenze et al. (2022), *IsaNet: Formalization of a Verification Framework for Secure Data Plane Protocols*

The development of the router implementation was driven by strict performance and schedule constraints, such that extracting it from a formal model or co-developing code and proofs was not possible. Consequently, we verified an existing implementation that was frequently changing, as developers added features, optimized performance, and fixed bugs. We annotated a fork of the SCION repository and, for a period of 8 months, we ported changes from the original repository weekly to keep both repositories in sync. We eventually fixed the version of SCION to have a stable target for a publication[10]. Porting changes to our code base required adding and often adapting annotations. In this process, we benefitted greatly from using modular verification, which confines the adaptations to a local scope and avoids adapting or re-verifying the unaffected parts of the code base.

10: We have resumed our efforts to keep the verified codebase in sync with the upstream SCION repository. We recently updated the annotations for all dependencies of the SCION router [108], but our efforts to verify the router were stopped by a large ongoing refactoring. We believe this refactoring provides a great case-study for techniques for automated proof repair.

We showed that conducting quality assurance *during* development is feasible, but requires strong modularity to limit the impact of changes. The reward is a higher chance of fixing found problems.

3.8.3 Limits of automated code verification

It was initially unclear whether SMT-based, automated verification would scale to the SCION router's Go implementation. While we hoped automation would make verifying a large, complex implementation a manageable task, SMT-based verification has two known limitations.

First, automated verification gives users less control over the proof, with limited means for debugging when the tool cannot automatically prove a property. We often found strategies to work around this limitation by phrasing annotations in a way that is beneficial for the SMT solver. In some cases, this required a good understanding of the tool's inner workings.

Second, proof automation may be slow, hindering the workflow of writing and incrementally improving specifications. As explained in Chapter 4, verification performance is a central challenge when SMT-based verifiers are used, and scaling them requires developing new strategies and improving the tools.

We conclude that automated verification of real-world code bases in complex languages is feasible. We are confident that automated verification greatly reduced the effort of verifying the code base compared

to a more manual approach, but there were times when fine-grained control over the proof would have been very helpful to debug proof failures and to prevent the SMT solver from diverging. We believe that an hybrid approach like that of Loom [109] which combines automation by default with manual proof development when needed is an exciting and promising way to deal with the shortcommings of SMT-based verifiers.

[109]: Gladshtein et al. (2026), *Foundational Multi-Modal Program Verifiers*

3.9 Related work

We discussed operating system and compiler verification in the introduction. We focus here on the verification of networking systems, security protocols, and other distributed systems.

Networking-related verification As noted in the introduction, our work differs from verifying network configurations [49–52], which only considers the model level, and network functions [53–57], which focuses on the fully automated verification of local properties of network component implementations. In contrast, we combine protocol model and code verification to establish both network-wide security properties and local properties of the router. Moreover, the complexity of the existing implementation mandates deductive verification, requiring substantially more human effort.

[49]: Khurshid et al. (2013), *VeriFlow: Verifying Network-Wide Invariants in Real Time*
[50]: Kozen (2014), *NetKAT – A Formal System for the Verification of Networks*
[51]: Weitz et al. (2016), *Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver*
[52]: Beckett et al. (2017), *A General Approach to Network Configuration Verification*

[53]: Pirelli et al. (2022), *Automated Verification of Network Function Binaries*
[54]: Zhang et al. (2020), *Automated Verification of Customizable Middlebox Properties with Gravel*
[55]: Zaostrovnykh et al. (2019), *Verifying software network functions with no verification expertise*
[56]: Liu et al. (2018), *p4v: practical verification for programmable data planes*
[57]: Zaostrovnykh et al. (2017), *A Formally Verified NAT*

As mentioned previously, we build on an existing protocol verification of a simplified version of SCION [61, 107]. We adopted their refinement-based approach, but extended the models substantially. Chen et al. [110] model S-BGP and an early version of SCION in a Prolog-style declarative language for networking protocols, and verify route authenticity (control plane) and data path authenticity (data plane, for SCION only). The latter is weaker than path authorization, as it considers each hop separately instead of relating successive hops. Hence, it does not rule out path splicing attacks.

[61]: Klenze et al. (2022), *IsaNet: A framework for verifying secure data plane protocols*
[107]: Klenze et al. (2022), *IsaNet: Formalization of a Verification Framework for Secure Data Plane Protocols*

[110]: Chen et al. (2015), *A Program Logic for Verifying Secure Routing Protocols*

Arnaud et al. [111, 112] model routing protocols in a process calculus and propose two decision procedures for their analysis. One analyzes a protocol for any topology and the other works for a given topology. All of these works are limited to protocol design verification, whereas we also provide guarantees for the implementation.

[111]: Arnaud et al. (2014), *Modeling and verifying ad hoc routing protocols*
[112]: Arnaud et al. (2011), *Deciding Security for Protocols with Recursive Tests*

Verified security protocols The Everest project has verified several security protocols, including TLS [9, 10] and QUIC [11] at the code level. They implement the TLS and QUIC protocols in F# [9] and F* [10, 11], respectively, and use the associated refinement type checker (F7 and F*, respectively) for the cryptographic security proofs. As they use a computational attacker model, their proofs yield stronger security guarantees than are possible in a Dolev-Yao model. The resulting protocol implementations, written in F# or in OCaml extracted from F*, are reference implementations, which were designed for verification and achieve a significantly lower performance than OpenSSL. The authors also provide more efficient implementations written in language fragments that can be compiled to C (e.g., the Low* fragment of F* [14]), which achieve a performance similar to an optimized implementation in some cases, but require an additional proof of equivalence with the reference implementation [10, 11]. In contrast, we verified a pre-existing, optimized

[9]: Bhargavan et al. (2013), *Implementing TLS with Verified Cryptographic Security*
[10]: Delignat-Lavaud et al. (2017), *Implementing and Proving the TLS 1.3 Record Layer*

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

[14]: Protzenko et al. (2017), *Verified low-level programming embedded in F**

protocol implementation and soundly connect it to a protocol model using the Igloo methodology.

Both DY* [113] and Arquint et al. [106] verify security protocols at the code level by proving invariants over the protocol traces and showing that these invariants entail the desired security properties. Our refinement-based approach increases modularity, which is crucial for verifying systems of the size and complexity of SCION. Arquint et al. [114] extend the Tamarin prover with the automated generation of each protocol role's I/O specification and verify the official Go implementation of the WireGuard key exchange protocol. As said earlier, Tamarin is not expressive enough for our purposes.

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

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

[114]: Arquint et al. (2023), *Sound Verification of Security Protocols: From Design to Interoperable Implementations*

Distributed system verification IronFleet [6], Verdi [7, 115], and Velisarios [8] verify respectively the Paxos, Raft, and PBFT consensus protocols. Chapar [116] proves causal consistency of key-value stores. In IronFleet, models and implementation are written in Dafny [17], which does not allow for concurrent code, and are verified with an SMT solver. The other approaches model algorithms in Coq and rely on code extraction from Coq, which is unsuitable for verifying existing implementations in commonly-used programming languages. Anvil [16] is a framework for building and verifying Kubernetes controllers. It restricts the structure of the implementation, making it unsuitable for verifying pre-existing code.

[6]: Hawblitzel et al. (2015), *IronFleet: Proving Practical Distributed Systems Correct*

[7]: Wilcox et al. (2015), *Verdi: a framework for implementing and formally verifying distributed systems*

[115]: Woos et al. (2016), *Planning for change in a formal verification of the Raft consensus protocol*

[8]: Rahli et al. (2018), *Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq*

[116]: Lesani et al. (2016), *Chapar: certified causally consistent distributed key-value stores*

[17]: Leino (2010), *Dafny: An Automatic Program Verifier for Functional Correctness*

[16]: Sun et al. (2024), *Anvil: Verifying Liveness of Cluster Management Controllers*

3.10 Conclusion

We verified the SCION router from its high-level design down to its performance-optimized implementation. A key success factor was our highly modular verification approach, which allowed us to reduce the verification complexity, work in parallel on different aspects of the problem, and confine the impact of protocol and implementation changes. We discovered both design and implementation errors, critically affecting the router's security, that evaded all prior reviews and testing and have all been subsequently fixed. Future work includes a tighter integration of code development and verification, finding automated ways of repairing specifications and proof annotations when the code inevitably changes, incorporating upcoming features of the SCION router, and verifying the control plane.

Taming Verification Performance | 4

One of the greatest obstacles to using SMT-based verifiers in large verification developments is their proneness to unpredictable, bad, and unstable verification performance. In the VerifiedSCION project, the size and complexity of both the verified code and its specifications repeatedly led to scalability issues with Gobra, resulting in verification times that were too long to make effective progress. In fact, performance problems were, by far, the biggest practical hurdle we faced when using Gobra in the VerifiedSCION project.

In this chapter, we distill important lessons about the performance of Gobra. Furthermore, we present and evaluate various effective strategies to improve verification performance that we curated while working on VerifiedSCION. While the analysis and strategies presented in this chapter focus on Gobra and its intermediate verification language (Viper), we observe that the performance issues discussed here are prevalent in SMT-based program verifiers and believe they are of wider applicability.

4.1 Introduction 55
4.2 Symbolic execution of Gobra 57
4.3 Improving verification performance 60
4.4 Verification performance in VerifiedSCION 72
4.5 Related work 82
4.6 Conclusion 85

4.1 Introduction

Program verifiers based on SMT solvers like Dafny [17], F* [117], Raven [118], Verifast [119], Viper [18], and its family of verifiers (which includes Gobra [29], Nagini [78], Prusti [120], and Vercors [121]) extract *proof obligations*, i.e., logical conditions that, together, imply the correctness of the program, from annotated programs. Then, most[1] of these conditions are encoded into satisfiability problems that are discharged by SMT solvers. Because they rely on SMT solvers to prove complex properties, these verifiers often allow for surprisingly short and mostly automated proofs of correctness. Unfortunately, the reliance on SMT solvers, which allows for a high level of automation, is also the cause of two major usability problems that appear to be endemic to SMT-based verifiers and that severely delay and block larger verification efforts: *unpredictable and long verification times*, and *proof instability* [122].

First, unpredictable and long verification times disrupt interactive proof development, which proceeds iteratively as discussed in Section 3.6. If a single run of the verifier takes many minutes, by the time it is over, the proof authors have moved on to a different task. If verification errors are found, the authors have to switch contexts to understand the issue. In some cases, verifiers may even appear to diverge (i.e., they time out after a few hours spent verifying a single member) without finding any verification errors. These cases are particularly frustrating, as they provide very little information about the correctness of the program or the state of the proof.

Second, proof instability, which is characterized by seemingly innocuous changes to the source program or its specification altering the verification results (e.g., changing the identifiers used in the program), sidetracks proof authors by forcing them to revisit and stabilize old proofs. To make matters worse, the causes of instability are hard to isolate because (1) their effects are sometimes non-local, i.e., verification errors are reported spuriously in members that do not contain the root cause of the

[17]: Leino (2010), *Dafny: An Automatic Program Verifier for Functional Correctness*

[117]: Swamy et al. (2016), *Dependent types and multi-monadic effects in F**

[118]: Gupta et al. (2025), *Raven: An SMT-Based Concurrency Verifier*

[119]: Jacobs et al. (2010), *A Quick Tour of the VeriFast Program Verifier*

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

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

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

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

[121]: Armborst et al. (2024), *The VerCors Verifier: A Progress Report*

1: Not all proof obligations need to be dispatched to SMT solvers. Program verifiers may discharge them through syntactic checks or via their own decision procedures, e.g., the symbolic execution engine for Viper keeps track of the state of the heap in a dedicated data-structure and is able to discharge some permission-related proof obligations without the SMT solver.

[122]: Brugger et al. (2026), *Toward Practical Deductive Verification: Insights from a Qualitative Survey in Industry and Academia*

instability, and (2) they are highly dependent on the internal state of the verifier, which makes these issues difficult to minimize and reproduce. A particularly puzzling example of this was reported in the issue tracker of Gobra [123]. In this example, adding empty lines to the program caused it to either verify or be rejected, depending on the number of added empty lines. This issue happened in an older version of Gobra, where the variable identifiers used in the SMT queries depended on the position of the corresponding member in the file. Thus, adding new lines in the Gobra program changed the identifiers (and nothing else) of the generated SMT queries and the SMT solver reported different verification results based on that. Similar issues have been reported in Dafny [124].

[123]: Berling (2022), *Whitespace-dependent bug: "Assertion unknown might not hold"*

Both kinds of issues frustrate and bamboozle even the most experienced proof engineers and verifier developers, who have grown to see them as a fact of life. However, in the context of verifying large pre-existing projects where software engineers without extensive expertise in formal methods might eventually be expected to maintain the proofs, these problems are absolute show-stoppers.

While using Gobra on VerifiedSCION, we consistently experienced both classes of issues. In fact, performance issues were, by far, the biggest roadblock to making steady progress towards our verification goals. Most of the performance problems we encountered could be attributed to one or both of the following causes:

1. *Long functions:* Gobra verifies code using symbolic execution, which verifies each path in the control flow and each case in the specification (e.g., for disjunctions) separately, leading to a potentially large number of *branches* (Section 4.2). The time required to verify a function depends primarily on the size of the function, the number of branches, and the complexity of the properties to be verified. The SCION router implementation contains several large functions (e.g., the function `Run` has 96 lines of Go code and `prepareSCMP` has 101 lines, excluding verifier annotations) with many branches, for which verification was slow.
2. *Large verification context:* While we verify each function separately, the effort for the verification tool is affected by the entire program's *verification context*, which includes user-provided type declarations and invariants, as well as mathematical axioms generated by Gobra to model Go's built-in types such as slices, structs, and interfaces. Large verification contexts have a severe impact on verification performance [125] and stability [126] because the resulting global declarations and (quantified) axioms in the proof obligations can overwhelm the SMT solver by causing too many quantifier instantiations.

[125]: Gopinathan et al. (2025), *Accelerating Automated Program Verifiers by Automatic Proof Localization*

[126]: Zhou et al. (2024), *Context Pruning for More Robust SMT-based Program Verification*

Besides limiting our ability to verify code quickly, poor verification performance often discouraged us from making large changes to the proofs (e.g., to make them more readable), as the cost of re-verifying and stabilizing the new code was too high in some cases. Furthermore, it made it hard for people to contribute to the project, especially our Bachelor's and Master's students, who had no prior verification experience and found the performance problems inscrutable.

In this chapter, we analyze the performance of Gobra on VerifiedSCION and we discuss and evaluate the strategies adopted to improve Gobra's performance. Our goal is to provide an overview of the current performance bottlenecks in Gobra that may serve as a starting point for further research in verification performance and (in)stability. We organize the remainder of this chapter as follows: In Section 4.2, we summarize the

background on the symbolic execution engine of Gobra that is relevant for understanding later sections. In Section 4.3, we present the most important strategies we adopted to overcome the performance problems and discuss their effectiveness on VerifiedSCION. These solutions are complementary and focus on minimizing and managing the proof context and on tweaking the verification algorithms to change the structure of the proof obligations emitted by Gobra. Incidentally, the solutions we propose for managing the context seem not only to improve verification time but also to help with proof stability. In Section 4.4, we present data collected during the verification of the latest version of VerifiedSCION, and we distill important observations about how the time is spent by Gobra in the latest version of VerifiedSCION. Our goal with this section is to clearly show the bottlenecks of the verifier, in order to guide future research in verification performance. In Section 4.5, we describe the existing research addressing the performance and stability of SMT-based verifiers. Finally, we conclude this chapter in Section 4.6 and discuss future research directions.

While we focus specifically on Gobra in this chapter, other SMT-based verifiers are likely to encounter some of the same issues described here given sufficiently complex codebases and specifications. We believe that the strategies presented here (especially those based on context management) apply to other verifiers as well.

4.2 Symbolic execution of Gobra

The Gobra verifier translates annotated Go programs to Viper [18] programs that are verified using Silicon [33], the symbolic execution engine of Viper. During this translation, non-pure functions and methods in Go/Gobra are translated to Viper methods, pure functions and methods are translated to Viper functions, and Gobra predicates are translated to Viper predicates[2], and Viper domains axiomatizing built-in types from Go and Gobra are generated. Understanding the performance problems in Gobra thus requires some understanding of Silicon. In this section, we provide a quick tour of Silicon, which provides the background required to understand the remaining sections. We assume some familiarity with Viper; unfamiliar readers may consult its tutorial [127].

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

[33]: Schwerhoff (2016), *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*

2: The translation scheme used by Gobra is subtler than presented here, e.g., the use of some features like closures and `outline` statements (Subsection 4.3.2) may cause a single function to be translated into various Viper methods; the one presented here is only a good approximation.

[127]: (n.d.), *Viper Tutorial*

Silicon performs verification in two separate steps. In the first, functions and predicates are verified sequentially for well-formedness (e.g., that the preconditions of all callees and partial operators like division hold). To this end, Silicon symbolically *inhales* the bodies of predicates and discharges the proof obligations computed from them (e.g., that the preconditions of all function calls hold at the places where they are called). Likewise, Silicon checks the well-formedness of functions by *evaluating* their bodies, assuming their preconditions, and proving that the postcondition is established. In the second step, Silicon verifies each method by symbolically *executing* its body, starting from a state that satisfies the precondition. Silicon is *method modular*, i.e., it verifies each method separately from the others, and each method call is verified with respect to the specification of the callee and not its implementation. For each method, Silicon discharges the proof obligations generated by the execution of each statement (e.g., that the preconditions of a method hold when it is called), and checks that the postcondition is guaranteed to hold by the end of the execution. In this step, functions are assumed to have been verified and their behavior is axiomatized via two (universally

quantified) axioms per function: one that equates the value of a function call to its body, and another that establishes the postcondition on every call.

4.2.1 Symbolic state and symbolic execution

During the verification of all functions, predicates, and methods, Silicon maintains a *symbolic state* consisting of three components: the *store*, the *path condition*, and the *symbolic heap*.

The *store* γ is a function that maps local program variables to *symbolic values*, i.e., expressions that depend on symbolic variables.

The *path condition* π is a stack that contains all assumptions made on the current path, which includes the branch conditions taken on conditional control flow and the assumptions introduced by explicit `assume` statements and method calls (i.e., their postconditions). When Silicon executes an `if`-statement or evaluates a conditional expression (`b ? e1 : e2`) or implication (`b ==> e`), it *branches*: it forks the current symbolic state into two copies (one where the `b` is assumed, and one where its negation is assumed) and verifies each path independently to completion. Given that each verification path may take a non-negligible amount of time to verify, by default, Silicon stops executing a path when it is able to prove that the path condition entails `false`, as this implies the current path is infeasible. In addition, Silicon joins paths right after evaluating *pure* conditional expressions and implications, i.e., those that do not contain accessibility predicates for memory locations or predicates. By default, Silicon *does not* attempt to merge the paths resulting from an `if`-statement or from evaluating *impure* conditionals and expressions. Silicon may also branch *temporarily* when evaluating `unfolding` expressions: evaluating `unfolding P(x) in e` expressions cause multiple paths to be created for evaluating `e`, which are joined immediately afterwards.

Finally, the *symbolic heap* h is a multiset of *heap chunks*, each representing a fractional permission to access a heap location or a predicate instance, along with the values stored in the corresponding locations. For example, a heap chunk for the assertion `acc(x.f, 1/2)` records the receiver `x`, the field `f`, the symbolic value of the field, and the fractional permission held `1/2`. Silicon uses the symbolic heap to record the resources currently owned.

Discharging heap-related proof obligations Heap-related proof obligations, such as checking permissions during field accesses, result in checks performed on the symbolic heap.

Various algorithms for discharging proof obligations in separation logic have been proposed [128]. By default, Silicon uses the *SE-PS* algorithm [128]: To prove that an assertion such as `acc(x.f, p)` holds in the current state, Silicon traverses the heap until it finds a chunk for the memory location `x.f` with a permission amount greater or equal to `p`. It may be the case that no single chunk contains permission `p`, but two chunks for `x.f` exist with permissions `p1` and `p2` such that `p1 + p2` $\geq$ `|p|`. Because of this, when Silicon cannot find a corresponding chunk to prove the ownership of a resource, it performs *heap consolidation*, i.e., it merges the chunks for the same receiver into a single chunk by adding their permission amounts, and then tries once more to find a suitable chunk. Importantly, not all heap-related proof obligations may result in

[128]: Eilers et al. (2024), *Verification Algorithms for Automated Separation Logic Verifiers*

queries to the SMT solver. For example, Silicon may be able to discharge a proof obligation `acc(x.f, p)` without resorting to the SMT solver by finding a chunk whose memory location and permission amount syntactically match `x.f` and `p`, respectively. The other proof obligations, including functional properties, result in queries to the SMT solver.

The SE-PS algorithm suffers from a fundamental incompleteness in the presence of *disjunctive aliasing*, i.e., when a reference may alias at least two different references, but the reference that it aliases is not fully determined by the path condition.Figure 4.1 illustrates disjunctive aliasing:

```
1 inhale acc(x.f) && acc(y.f)
2 inhale z == x || z == y
3 assert acc(z.f)
```

Figure 4.1: Example of disjunctive aliasing.

With SE-PS, the assertion in line 3 can never be proven: by the time the assert statement is reached, there are two chunks in Silicon's representation of the heap, one providing full permission to `x.f`, and the other providing full permission to `y.f`. Because no single chunk is enough to establish the full permission to `z.f`, this example fails to verify.

To overcome this incompleteness, Silicon may be configured to use the *SE-PC* algorithm [128]. Using this algorithm, to prove that the assertion `acc(x.f, p)` holds, Silicon sums the total permission amounts of all chunks for fields that may alias with `x.f`, and checks that it is at least `p`. When SE-PC is used in the example above, the sum of the permissions to `z.f` is given by the expression `(z == x ? 1 : 0) + (z == y ? 1 : 0)`, which together with the path condition, implies that the permission amount to `z.f` is 1. The increased completeness of SE-PC comes at the cost of more complex heap-related proof obligations, which may take longer to be discharged by the SMT solver.

Finally, *quantified permissions* [37], which are Viper's implementation of the iterated separating conjunction, are handled differently in Silicon: the verifier uses specialized heap chunks to represent them, with dedicated algorithms for inhaling, exhaling, and checking permissions. The details of these algorithms are not critical for this thesis and we defer them to the original paper [37]. The relevant aspect for this thesis is that, as Vytautas Astrauskas suggests in his doctoral thesis [129, Appendix A], programs with quantified permissions are prone to performance problems: the computation of the permission amounts for memory locations occurring in quantified permissions may cause the SMT solver to do a case analysis where the number of explored cases is exponential in the number of heap chunks that may provide permission to a memory location. We discuss the potential impact of this in the next sections.

[37]: Müller et al. (2016), *Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution*

[129]: Astrauskas (2024), *Leveraging Uniqueness for Modular Verification of Heap-Manipulating Programs*

4.2.2 Verification context

Having summarized how Silicon performs symbolic execution, we may now define the *verification context* as the set of facts assumed and provided to the SMT solver to establish a given proof obligation. Given a proof obligation in a Viper method generated by Gobra, the verification context contains the environment axiomatizations of Gobra's data structures defined in terms of uninterpreted functions, the universally quantified axioms generated for each function defined in the current program, and the current symbolic state. The context of proof obligations occurring in predicates and functions is similar, but contains only the axioms for the functions verified up to that point.

An important part of the context is the set of assumed universally quantified assertions. Silicon relies on the *e-matching* algorithm [130] of the SMT solver to determine when to instantiate universal quantifiers based on syntactic *triggers*—when a ground term matching a trigger appears, the solver produces a new instance of the quantified formula. Each instantiation may introduce new ground terms that match other triggers, potentially leading to long chains of quantifier instantiations and, in the worst case, to a *matching loop*, i.e., an infinite loop of instantiations where each instantiation produces a new expression that matches the trigger. In practice, the number of quantifier instantiations is a common cause of long verification times in SMT solvers, and the selection of triggers has consequences for both the performance and stability of SMT-based verifiers [131].

[130]: Moskal (2009), *Programming with triggers*

[131]: Leino et al. (2016), *Trigger Selection Strategies to Stabilize Program Verifiers*

4.3 Improving verification performance

In this section, we describe the most effective strategies we identified to overcome the performance problems found in VerifiedSCION. Our strategies can be divided into two groups: those for managing the proof context (as defined in Section 4.2), and those that change the verification algorithms used to generate the proof obligations.

On the side of context management, we extended Gobra with two novel features to eliminate unnecessary context. First, we introduce a slicing algorithm that automatically prunes unnecessary definitions from the context (Subsection 4.3.1). Second, we introduce `outline` statements, which allow verifying a sequence of statements against a contract, and treat that code as opaque in the place where it occurs (Subsection 4.3.2). Together with the information hiding features of Gobra (`opaque` functions and abstract predicates), which we discuss in Subsection 4.3.3, proof authors have a high degree of control over the proof context, with slicing handling the coarse-grained pruning of unrelated package-level declarations and the manual features giving users finer control where the automatic analysis is too conservative.

On the side of algorithms, we explore the trade-offs involved in choosing algorithms for generating proof obligations in separation logic—in particular, joining paths more frequently to reduce the number of paths explored by the symbolic execution verifier, at the cost of making the symbolic states more complex—which may lead to vastly different verification times.

These approaches are highly complementary: the techniques to manage proof context simplify the proof obligations by eliminating redundant or unnecessary context, which may reduce the verification time of each SMT query, whereas the choice of the verification algorithm can lead to fewer paths, and thus, to fewer SMT queries.

These strategies improved Gobra's performance substantially and enabled the successful verification of the router. All of these strategies can now also be applied in other verification projects.

4.3.1 Automatically minimizing the context in Gobra

As discussed in Section 4.2, every SMT query that Silicon issues is discharged against the entire verification context, which includes background theories, defined as sets of uninterpreted functions and their

axioms (which often contain quantified assertions), and the definitions of all functions and predicates of the Viper program under verification.

Gobra's encoding emits a single Viper program per package, and it heavily relies on background theories (i.e., Viper's *domains*), uninterpreted functions and fields to define the semantics of various data-structures, including Go's arrays and slices, structs, strings and interfaces. All of these definitions are part of the proof context of every member verified with Gobra, even though the proof obligations of any individual member typically depend only on a small fraction of them. The irrelevant declarations increase the size of the formulas dispatched to the SMT solver, which directly impacts its verification performance.

To minimize the irrelevant context available at each member, we extended Gobra with the *slicing algorithm* proposed by Wolf [132] which conservatively identifies and prunes irrelevant definitions and axioms in the verification context prior to verification. Verification slicing is an automatic, sound pre-processing step implemented at the Viper level that, for each Viper member, computes a *slice* of the Viper program which over-approximates the declarations that the member's proof obligations can depend on, and then verifies the member against that slice rather than the full program.

[132]: Wolf (2024), *Automated Verification of Advanced Correctness and Security Properties*

The slicing algorithm can be used in two modes:

1. The user can explicitly select a subset of members of the program under verification to be verified. The analysis then generates a Viper program containing the selected Viper members and its relevant context. This enables a workflow where the user does not need to transform the source program to only focus on the proof obligations of the current method[3].
2. The user may split a program into a set of smaller programs with smaller contexts such that if all programs in the set are correct, then the original program is correct. The user must provide the maximum number of programs that the program should be split into as a configuration option, and the slicing algorithm generates multiple Viper programs with only part of the context. This works in three steps: (1) Viper identifies all members that have proof obligations (methods, functions and predicates); (2) for each member with proof obligations M, Viper generates a slice containing M and its dependencies, computed using the algorithm described below; (3) Viper merges each slice until the number of slices remaining is at most the user-provided bound. To make effective usage of the context, Viper uses heuristics to group members with similar dependencies in the same Viper program.

3: One may wonder about the interplay between this feature and the caching of verification results. Various verifiers that are method-modular, like Dafny and Viper, support caching verification results. Thus, when a member changes, only the members that directly depend on what was changed need to be reverified. For example, if the specification of a Viper method changes, then only the bodies of the callers of that method need to be re-verified (and nothing else). If only the body of a method changes, no other member needs to be reverified. Thus, it may seem like the slicing algorithm is redundant in the presence of caching. The reason this is not the case is that the slicing algorithm enables workflows where the verifier hasn't run through the entire codebase once. This is often the case in packages that are still going through verification, or when there are different members of the team verifying different parts of the same package, or when the verifier is updated which might lead to changes in the verification results.

To compute a program slice for a Viper member (field declaration, function[4], method, predicate, or domain axiom) `M` in the first mode, Viper performs a conservative *may-depend* analysis that identifies all the members that may logically affect the verification results of `M`. In particular, Viper detects dependencies between *parts of Viper members*. There are eight kinds of parts considered: field declarations, function signatures (including their specifications), function bodies, method signatures (including their specifications), method bodies, predicate signatures, predicate bodies, and axioms. Viper finds the following dependencies:

4: We treat domain functions and functions declared outside domains equally for the purposes of this analysis

1. a function body may depend on its signature. Likewise for predicates and methods.

2. M may depend on the specification of a function F that it calls; if F is not opaque (Subsection 4.3.3), then M may depend on its body too. If F is opaque, M may depend on F if it explicitly *reveals* a call to F.
3. M may depend on the specification of a method N if it calls it. M may never depend on the body of N;
4. M may depend on a predicate P that it explicitly mentions. M may depend on the body of P if it ever folds or unfolds an instance of P.
5. an axiom M may depend on any function called in its body.
6. a function M may depend on an axiom A if either (1) A is not a quantified assertion and it contains a call to M, or (2) A is a quantified assertion and a call to M occurs in the triggers of A.
7. M may depend on the declaration of a field F if M contains a field access to F.

The slice computed for a part of a member M is given by the transitive closure of the *may-depend* relation. Any part of a member not occurring in the slice cannot logically affect the verification results of M. We exemplify the slicing algorithm in the snippet below:

Listing 4.1: Example of Viper's automated slicing — Full program.

```
field next: Ref
field val: Int

predicate list(l: Ref) {
    l != null ==> acc(l.next) && acc(l.val) && list(l.next)
}

@opaque() function length(l: Ref): Int
    requires list(l)
    ensures  0 <= result
{
    unfolding list(l) in l == null ? 0 : 1 + length(l.next)
}

@opaque() function get(l: Ref, i: Int): Int
    requires list(l)
    requires 0 <= i < length(l)
{
    unfolding list(l) in
    let len == (@reveal() length(l)) in
    i == 0 ? l.val : get(l.next, i - 1)
}

method store(l: Ref, i: Int, v: Int)
    requires list(l)
    requires 0 <= i < length(l)
    ensures  list(l)
    ensures  length(l) == old(length(l))
    ensures  get(l, i) == v
    ensures  forall j: Int :: 0 <= j < length(l) && j != i ==>
        get(l, j) == old(get(l, j))
{
    // implementation
    ...
}

method swap(l: Ref, i: Int, j: Int)
    requires list(l)
    requires 0 <= i < length(l) && 0 <= j < length(l)
    ensures  list(l)
    ensures  length(l) == old(length(l))
    ensures  get(l, i) == old(get(l, j)) && get(l, j) == old(get(l, i))
    ensures  forall k: Int ::  0 <= k < length(l) &&
        k != i && k != j ==> get(l, k) == old(get(l, k))
{
    var tmp: Int := get(l, i)
    store(l, i, get(l, j))
    store(l, j, tmp)
}
```

This snippet defines two fields, `next` and `val`, to store the next element in the linked list, and the value of the current element, respectively. In addition, it defines the predicate `list`, which stands for the ownership of an entire linked list, functions `length` to compute the length of a list and `get` to access an element at a given valid index, and finally, methods `store` to modify the value in the list at a given index, and `swap` to swap two elements of the list.

Computing the slice for method `swap` results in a much smaller program:

```
predicate list(l: Ref)

function length(l: Ref): Int
    requires list(l)
    ensures  0 <= result

function get(l: Ref, i: Int): Int
    requires list(l)
    requires 0 <= i < length(l)

method store(l: Ref, i: Int, v: Int)
    requires list(l)
    requires 0 <= i < length(l)
    ensures  list(l)
    ensures  length(l) == old(length(l))
    ensures  get(l, i) == v
    ensures  forall j: Int :: 0 <= j < length(l) && j != i ==>
        get(l, j) == old(get(l, j))

method swap(l: Ref, i: Int, j: Int)
    requires list(l)
    requires 0 <= i < length(l) && 0 <= j < length(l)
    ensures  list(l)
    ensures  length(l) == old(length(l))
    ensures  get(l, i) == old(get(l, j)) && get(l, j) == old(get(l, i))
    ensures  forall k: Int :: 0 <= k < length(l) &&
        k != i && k != j ==> get(l, k) == old(get(l, k))
{
    var tmp: Int := get(l, i)
    store(l, i, get(l, j))
    store(l, j, tmp)
}
```

Listing 4.2: Example of Viper's automated slicing — Slice for the `swap` method.

As method `swap` directly references all other members, their definitions must all be included in the slice. However, the body of method `store` need not be included, as Viper verifies methods modularly. Even though `swap` depends on functions `length` and `get`, their bodies are never revealed, and thus, their body may not influence the verification of `swap`. For a similar reason, the body of `list` need not be included, as this predicate is never unfolded in the body of `swap` or in its other dependencies. The slicing algorithm performs more effectively in projects with strict information hiding disciplines (Subsection 4.3.3). This point is demonstrated by the use of opaque functions here. If `get` was not opaque, the slice of `swap` would also include the body of `get`, as well as the bodies of `list` and `length`, as they are unfolded and revealed (respectively) in the body of `get`.

Gobra lifts the slicing algorithm from Viper to Gobra programs. Importantly, the algorithm is completely transparent to the user, both at the Viper level and at the Gobra level. The user does not need to provide annotations to identify each program slice, as this is done completely automatically. We defer the full technical description of the algorithm to Wolf's thesis [132].

Evaluation. We evaluated the effectiveness of this strategy by automatically slicing the `router` package from VerifiedSCION into 10 Viper files. We ran Viper with the best configuration we identified (which we describe in Subsection 4.4.1) on all generated files. We verified all files 5 times, and averaged their total execution times. On average, the total verification time was 2424 seconds (with a standard deviation of 82ms), which represents a speed-up of 11% compared to the average execution time of the original program. Inspecting an arbitrary execution, which we assume to be representative due to the low standard deviation, we observe that the total number of SMT queries has increased — 1,734,664 queries, compared to 1,316,837 in the unsliced version — due to redundancy, as Viper predicates, functions, and method signatures may be included in multiple slices. Nonetheless, the total time spent in queries to the SMT solver decreases to approximately 1600 seconds, compared to the 2000 seconds spent without slicing,and the total number of SMT queries taking 1 second or longer decreased to 91, compared to 162 in the unsliced version. These speed-ups suggest that the unnecessary context in a large verification project is noticeably detrimental to the verification time.

4.3.2 Manually managing the context with `outline` statements

Verification slicing prunes the proof context coarsely, at the granularity of package-level declarations, but it is ineffective at pruning the current symbolic state, i.e., it cannot remove unnecessary facts from the path condition and unnecessary heap chunks from the current heap. Long functions therefore remain a problem: every assertion is discharged against a context that accumulates everything established earlier in the body. A natural answer is to refactor a long function into several shorter ones, each with its own contract, so that verification proceeds method by method. This option, however, was not available to us. Because our goal was to verify the existing SCION router as is, we didn't have the freedom to refactor the code to split functions.

To obtain the same effect without changing the code, we designed a new code annotation, the *outline statement*, which causes verification to handle a sequence of statements as if they had been extracted into a separate function. Analogously to functions, `outline` statements have pre- and postconditions and are verified modularly, so they split the verification of a function into smaller logical parts without altering the implementation. Unlike an actual extracted function, however, an `outline` statement transparently shares the scope of its enclosing member: the statements inside the `outline` statement can read and update the same local variables, parameters, and receiver as the surrounding code, without the user having to thread anything through an argument list or a return tuple. The construct can be seen as a localized application of separation logic's *frame rule* [20] at the statement level: the precondition *explicitly* selects the resources the outlined statement needs, and everything else is framed around it. As a concrete (contrived) example, consider the following code:

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

Listing 4.3: Example of Gobra's `outline` statement — Program without `outline` statement.

```
1  x.f = 5
2  y.f = 6
3  assert x.f < y.f
```

Pretending that the second statement was difficult to verify, we could extract it as follows:

```
1  x.f = 5
2  requires acc(y.f)
3  ensures acc(y.f) && y.f == 6
4  outline (
5  y.f = 6
6  )
7  assert x.f < y.f
```

Listing 4.4: Example of Gobra's `outline` statement — Program with an `outline` statement.

This feature allows for fine-grained control over the proof context, as (1) the preconditions of `outline` statements can be used to prune the proof context for the outlined statement, and (2) quantified assertions that are established and used only in an `outline` statement are not included in the proof context of the entire function.

In addition, `outline` statements helped us in two other ways during the verification of the SCION router. They made it easier to isolate verification errors: a verification query that fails to terminate inside a large function will, once the offending region is wrapped in an `outline` statement, often terminate with a localised error message scoped to that statement. They also expose additional opportunities for parallel verification, since each outlined statement is verified independently of its enclosing method and can be dispatched to a separate verifier instance. Finally, `outline` statements may be used to reduce the number of paths explored during verification by outlining `if` statements and other branching statements.

Implementation. `outline` statements are replaced in a compilation step that moves the outlined statements into a fresh Viper method whose contract is taken from the pre- and postconditions written immediately before the `outline` keyword, and the `outline` statement is replaced by a call to that method. The user does not have to declare which variables the new method takes or returns: Gobra infers its input parameters by scanning the outlined statement for variables that are defined outside and read inside, and infers its output parameters from the local variables that are defined inside the outlined statement, together with variables that are defined outside but modified inside the statement. By the time Gobra emits Viper, an `outline` statement looks like an ordinary method call. The current implementation imposes restrictions on the control flow inside outlined statements: currently, `return` statements are disallowed, as well as `break` and `continue` statements whose target loop lies outside the outlined statement. These features can easily be supported by extending the methods generated for outlined statements with boolean return parameters that identify whether any of these kinds of statements was executed, together with corresponding support on the caller (e.g., the surrounding method should return if the outlined statement returned), and by extending the assertion language for contracts in `outline` statements with boolean expressions so that postconditions may be conditional on the control-flow that was executed.

Evaluation. In total, by the end of the VerifiedSCION project, there were 13 `outline` statements in the `router` package, spread across 7 Gobra functions, and 17 more in the rest of the codebase. To evaluate the effectiveness of this solution, we measure the average verification times of the program slices obatained for each member with an `outline` statement (using the slicing algorithm described in Subsection 4.3.1) across 3 executions with the configuration described in Subsection 4.4.1, and we compare them against the average verification time of each slice

with the `outline` statements replaced by their bodies. We present the results in Table 4.1:

Table 4.1: Verification time for members that contain outline statements in the `router` package.

Member	#outlines	VTO	SDO	VT	SD	SU
AddSvc	2	53.53	0.77	54.03	1.54	1.01
Run	1	145.56	1.43	197.23	2.76	1.35
closureCall_Run	6	532.24	99.89	585.28	125.10	1.10
initMetrics	1	154.95	1.96	152.34	2.04	0.98
prepareSCMP	1	109.94	2.70	154.45	6.03	1.40
processOHP	1	162.83	0.70	192.95	3.61	1.18
testRun	1	110.53	0.48	111.90	1.23	1.01

In this table, the column **#outlines** stands for the number of outline statements in the corresponding member, **VTO** and **SDO** stand for the average verification time and standard deviation of the executions *with* `outline` statements respectively, and **VT** and **SD** stand for the average verification time and standard deviation of the executions *without* `outline` statements, respectively. **SU** stands for the speed-up, computed as **VT** / **VTO**.

These metrics show that `outline` statements had noticeable effects on verification times, with various members taking tens of seconds less to verify. The most noticeable improvements occur in `prepareSCMP` and `Run`, with speedups of 1.40 and 1.35, respectively. At the same time, `outline` statements may not lead to significant speed-ups and can even cause small slowdowns (e.g., `initMetrics`). As such, it may appear odd that some of these `outline` statements exist, given that each such statement introduces additional annotations for the contract, without performance benefits. These `outline` statements were first introduced in earlier versions of the codebase, when the respective members were still under verification and had verification errors. In our experience, the verification of members with errors often took longer to terminate and even diverged without providing useful debugging information. The `outline` statements allowed us to verify the problematic parts of the code in isolation, which made it easier to localize the sources of the verification errors in a timely fashion. Most of these uses of `outline` statements were erased from the codebase when the corresponding member was fully verified, while a few others survived to the current version of the codebase.

4.3.3 Manually managing the context through information hiding

In this subsection, we discuss two idioms we relied on in VerifiedSCION to manage the definitions in scope: opaque functions and abstract predicates. They are both manual mechanisms, in the sense that the proof author decides where a fact is revealed.

Gobra inherits from Viper the notion of an *opaque* function: a pure function whose body is treated by the verifier as uninterpreted unless the user explicitly reveals it at a given program point. At all other program points, the only facts known about the function are its signature and its contract; in particular, none of the quantified axioms implied by its body enter the proof context. This is the standard idiom for information hiding in SMT-based verifiers, and similar mechanisms are available in various other SMT-based verifiers (see Section 4.5). In VerifiedSCION, we use

`opaque` for pure helper functions whose body is large or complex, but whose precise definition is needed only in a few places. In total, there are 56 pure functions in the entire VerifiedSCION codebase, including the `router` package but also all other verified packages [5]. In the codebase of the `router` package alone, we reveal definitions 173 times. This level of fine-grained control was absolutely essential for keeping verification times under control: removing all `opaque` and `reveal` annotations leads verification to time out ater 6 hours.

5: Given that the `router` package imports all other verified packages, the pure functions defined in these packages are part of the verification context of any proof obligation of the `router`.

A different idiom that emerged during the verification of the SCION router is to use Viper's *isorecursive semantics* for separation logic predicates [38] not only to abstract over resourceful assertions, but as a deliberate way of controlling the information provided to the SMT solver. Under this interpretation, owning a predicate instance `P(x)` does not automatically entail ownership of the resources abstracted away in the predicate. Instead, the proof author must explicitly convert between a predicate instance and its body with `fold`/`unfold` statements. The main example of this is how we dealt with permissions for slices of bytes in VerifiedSCION, which are used extensively to store packets and packet headers. Rather than writing specifications for members that access and manipulate slices with quantified permissions for the elements of a slice `s` (i.e., `forall i int :: 0 <= i < len(s) ==> acc(&s[i])`, or `acc(s)` for short), we defined and used the predicate `Bytes(s)`, which abstracts over the same permissions. This strategy was one of the earliest and most consequential strategies we found, as it effectively reduces the number of quantified permission chunks in the context. This is particularly important when multiple slices of bytes are in the context. In such cases, Silicon needs to consider the possibility that the slices overlap, which leads to complex permission expressions for the memory locations in the quantified permission and to the performance problems discussed in Section 4.2. While a more thorough performance analysis is still required, we observed in an informal experiment that substituting every instance of the `Bytes` predicate with its body — and removing the corresponding `fold` and `unfold` statements and `unfolding` expressions — caused verification to time out after 6 hours.

[38]: Summers et al. (2013), *A Formal Semantics for Isorecursive and Equirecursive State Abstractions*

Finally, predicates compose naturally with `outline` statements (Subsection 4.3.2), leading to `outline` statements with the following shape:

```
1 requires Bytes(s)
2 ensures  Bytes(s)
3 ensures  ... // functional spec
4 outline (
5 unfold Bytes(s)
6 ... // statements that read and modify the slice
7 fold Bytes(s)
8 )
```

Listing 4.5: Example of the combination of `outline` statements with predicates for precise managing of the verification context.

We relied on this combination in our proofs to limit the scope in which quantified permissions are known to the verifier. Likewise, `outline` statements can be combined with `opaque` functions to limit the context in which a function definition is revealed, but we didn't rely on this combination.

4.3.4 Picking and tweaking verification algorithms

All strategies shown so far focus on managing and reducing the verification context available to the verifier, but they do not change the verification algorithms ultimately used by Gobra and Viper. In addition

to these strategies, we experimented with several additional optimizations and configurations of Silicon to explore the trade-offs between performance and completeness. We discuss these experiments here.

Complexity of SMT queries and number of paths

As seen in Section 4.2, Silicon creates a new verification path every time it executes an if-statement, or when it evaluates implications (of the form `b ==> e`) and conditional expressions (of the form `b ? e1 : e2`), where b is an expression of type `bool` and `e`, `e1`, and `e2` are pure boolean expressions (i.e., boolean expressions that do not provide permissions to resources) or impure expressions providing permissions to resources (e.g., `acc(x.f)`).

In verification based on symbolic execution, excessive branching may lead to long verification times. A reason for this is that branching has the potential to generate redundant queries to the SMT solver, as shown below:

Listing 4.6: Example of branching leading to redundant SMT queries.

```
1  pred P()
2  pred Q()

4  func m(b1 bool, b2 bool, x int)
5      requires 1 < x
6  {
7      inhale b1 ==> P() // <-- branching due to impure implication
8      inhale b2 ==> Q() // <-- branching due to impure implication

10     assert 0 < x // <-- 4 branches reach this assertion
11 }
```

Here, the two impure implications inhaled at the beginning of the program cause Silicon to create 4 paths that reach assertion `0 < x`. Given that the assertion does not depend on the inhaled resources, all four paths verify essentially the same proof obligation. In contexts where the assertion may be costly to prove (e.g., it falls in the theory of non-linear integer arithmetic) or where each new path may be long-lived, the time spent on redundant queries may become unacceptable.

To reduce the number of paths, Silicon is able to join paths by merging the symbolic states of each path. By default, Silicon merges verification paths *only* immediately after evaluating pure implications and pure conditional expressions — because evaluating pure expressions cannot change the symbolic state, the joined paths continues from the current symbolic state.

There is ample opportunity to optimize verification times by reducing the number of paths even further. To this end, we explored various configurations of Silicon and we found two options to particularly improve the verification performance of SCION. First, Silicon can be configured to perform more joins [133]. In particular, there are two modes for this configuration:

[133]: Bösiger (2021), *Performance Improvements of a Program Verifier*

- *Join when evaluating impure expressions* - under this option, Silicon joins paths immediately after evaluating impure implications and conditionals.
- *Always join* - under this option, Silicon performs all the joins performed in the previous mode. In addition, it also joins paths after if-statements.

When Silicon performs more joins, it needs to compute the new symbolic state for the new joined path. To that end, Silicon employs a *state merging algorithm* to merge the symbolic states of the paths being joined. Symbolic states (γ_1, π_1, h_1) and (γ_2, π_2, h_2) are joined as follows:

- Merging stores: If $x \in dom(\gamma_1)$ but $x \notin \mathsf{dom}(\gamma_2)$ or vice-versa, we omit x from the new store γ_3: since Viper has the usual lexical scoping, x cannot be referenced later on the verification path. If $\gamma_1(x) = v_1$ and $\gamma_2(x) = v_2$, we set $\gamma_3(x) \mapsto ite(b_1, v_1, v_2)$, where $ite(b, e_1, e_2)$ evaluates to e_1 if b holds and to e_2 otherwise.
- Merging path conditions: the path conditions collected on each path are guarded by an implication from the corresponding branch condition.
- Merging heaps: Let $r.f \mapsto v \# p$ denote a heap chunk[6] with receiver r, location f, symbolic value v, and permission amount p. Chunks present in both h_1 and h_2 are carried over to the new heap h_3 unchanged. A chunk $r.f \mapsto v \# p$ present only in h_1 is added to h_3 with its permissions guarded by the branch condition: $r.f \mapsto v \# ite(b_1, p, 0)$. Likewise for chunks only in h_2.

6: For simplicity, we only consider chunks for fields here. Predicate chunks are handled similarly.

Joining paths can thus be seen as a way to reduce the number of verification paths that are explored, at the cost of making the symbolic states more complex, which in turn leads to more complex SMT queries that may take longer to be discharged as the SMT solver may perform more case splits.

Second, Silicon may be configured to perform a Viper-to-Viper transformation that conditionalizes permission expressions to potentially reduce the number of symbolic execution paths. More precisely, under this configuration, Silicon rewrites implications of the form `b ==> acc(e, p)` to `acc(e, b ? p : none)` when doing so is safe. This rewrite is subject to strict syntactic restrictions that prevent a well-formed expression from being rewritten into a non-well formed one, e.g., if the resource `e` includes an access to a field like in `P(x.f)`, then the implication cannot be rewritten, as the permissions to `x.f` may only be available in case `b` holds.

```
1  field f: Int

3  predicate P(x: Int)

5  method m(b: Bool, x: Ref)
6      requires b ==> acc(x.f)
7  {
8      // The following impure implication cannot
9      // be conditionalized. Rewriting it to
10     // acc(P(x.f), b ? 1 : 0) results in an
11     // ill-formed expression, as the permission
12     // to x.f is not available
13     inhale b ==> P(x.f)
14 }
```

Given that the expression denoting the permission amount must be pure, as it denotes a permission amount rather than a resource, Silicon can immediately join them without resorting to the state merging algorithm. In comparison, enabling more joins leads to more complex symbolic states due to the merging of the heap chunks.

Evaluation. We evaluate the effectiveness of the different configurations of Silicon discussed in this section by verifying each of the five slowest functions in the `router` package (which are listed in Subsection 4.4.2) over

all possible six combinations of the option to conditionalize permissions and joining paths. We verify each member in each configuration three times, with a timeout of 20 minutes. Our results are shown in Table 4.2:

Table 4.2: Verification times of the slowest functions of the `router` package with different Silicon configurations.

Member	CP	MJ	MVT (timeouts)	TP
closureCall_Run	False	No	- (3)	-
closureCall_Run	False	IC	- (3)	-
closureCall_Run	False	All	- (3)	-
closureCall_Run	True	No	- (3)	-
closureCall_Run	True	IC	- (3)	-
closureCall_Run	True	All	600 (1)	1
process	False	No	- (3)	-
process	False	IC	736 (0)	119
process	False	All	- (3)	-
process	True	No	- (3)	-
process	True	IC	829 (0)	119
process	True	All	- (3)	-
processOHP	False	No	71 (0)	118
processOHP	False	IC	20 (0)	16
processOHP	False	All	194 (0)	1
processOHP	True	No	321 (0)	78
processOHP	True	IC	108 (0)	16
processOHP	True	All	- (3)	-
processPkt	False	No	- (3)	-
processPkt	False	IC	37 (0)	23
processPkt	False	All	- (-)	-
processPkt	True	No	- (3)	-
processPkt	True	IC	68 (0)	23
processPkt	True	All	- (-)	-
doXover	False	No	28 (0)	4
doXover	False	IC	19 (0)	2
doXover	False	All	67 (0)	1
doXover	True	No	72 (0)	4
doXover	True	IC	43 (0)	2
doXover	True	All	104 (0)	1

In this table, the **CP** column stands for whether permission conditionalization is enabled, **MJ** stands for whether the options for more joins are enabled (No – default Silicon joining behaviour, IC – Silicon joins immediately when evaluating impure expressions, All – Silicon joins paths after evaluating impure expressions and `if` statements), **MVT** stands for the mean verification time (in seconds) taken to verify the corresponding method, measured over the executions that do not time out (together with a count how many of these executions timed out), and **TP** stands for the number of *terminal paths*, i.e., the number of paths that are not terminated early and reach the end of the execution of the member. We highlight the fastest configuration for each member in yellow, and we highlight in orange the two configurations that introduced spurious verification errors.

Our measurements illustrate the trade-off between the number of verification paths and the complexity of the symbolic state on each path. Three of the five functions (`closureCall_Run`, `process`, and `processPkt`) time out under the default configuration (CP disabled, MJ set to No). Enabling joins for impure expressions (IC) yields the fastest configuration for four of the five functions: `process`, `processOHP`, `processPkt`, and `doXover`.

For `processOHP`, for example, joining paths from impure branches reduces the number of terminal paths from 118 to 16 and the verification time from 71 seconds to 20, a speedup of 3.5×.

Reducing the number of paths further — by also joining paths after if-statements (MJ set to All) — does not always improve performance and can even severely degrade it. For `processOHP`, going from 16 terminal paths (IC) to a single path (All) increases the verification time from 20 seconds to 194 seconds, as the merged symbolic state becomes significantly more complex. For `process`, it causes all three runs to time out, despite IC being able to verify the function. For `processPkt`, it introduces spurious verification errors. On the other hand, `closureCall_Run` can *only* be verified with the most aggressive joining (CP enabled, MJ set to All).

Conditionalizing permissions (CP) shows mixed results. While it contributes to the termination of `closureCall_Run` (together with MJ set to All), it often increases verification times when combined with IC, as seen for `processOHP` and `doXover`.

These results show that there is no single configuration that is universally optimal. Instead, finding the right balance between the number of paths and the complexity of each path condition is function-dependent. In practice, joining on impure expressions without enabling permission conditionalization offered the best trade-off for the functions in our evaluation. However, in the context of the entire `router` package, we found that joining on impure expressions and enabling permission conditionalization, together with per-function configuration in problematic functions like `closureCall_Run`, led to the best global verification times. A reason for this is that the implementation of more joins in Silicon is currently not applied during the verification of predicates and pure functions, which misses out on optimization opportunities.

Completeness of the decision procedures for reasoning about the heap

Given that various functions of the router code exhibit disjunctive aliasing, we initially verified the router with the SE-PC algorithm. In an attempt to reduce the verification times, we extended Gobra to support annotating each member with the verification algorithm to be used for that member, with the goal of using the SE-PS algorithm whenever possible, as the proof obligations generated by it are simpler.

In the end, the approach of selecting an algorithm per member became too onerous, and not useful until a member was fully verified: we often could not predict beforehand whether a function exhibits disjunctive aliasing. Thus, we would be confident switching to the more efficient algorithm only once the method is verified, to make sure the heap-related verification errors we run into during verification are not possibly caused by the incompleteness of SE-PS.

Finally, we tried a hybrid verification algorithm that discharges heap-related proof obligations first using the SE-PS algorithm, and if it fails, it consolidates the heap and re-attempts to discharge the proof obligation using SE-PC. Unfortunately, this strategy was not effective either: we enabled this configuration and ran it five times on the most recent version of the router (Subsection 4.4.1). Out of these executions, two timed-out after two hours, a sign of verification instability. The three remaining executions took, on average, 3814 seconds compared to the 2648 seconds that it takes on average with the SE-PC algorithm (Subsection 4.4.2), even though the 99th percentile for the duration of each query went down to 23

milliseconds. We attribute this increase of verification time to the increase in the number of heap-related SMT queries generated by the hybrid algorithm, which nearly tripled — 336,608 queries compared to 118,911 — compared to the number of queries performed using SE-PC exclusively (Subsection 4.4.3), while the numbers of other query types remained roughly the same. We believe these additional queries result from (1) more common failures when discharging proof obligations due to the incompleteness of SE-PS, which makes Silicon consolidate the heap and re-attempt the failed proof obligation once more, and (2) the additional queries performed during state consolidation, after failed first attempts, which attempts to merge aliased heap chunks in the symbolic heap until a fix-point is reached [33, 3.4.2. Overcoming Heap Incompletenesses].

[33]: Schwerhoff (2016), *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*

In addition to trying out different variations of the symbolic execution engine, we also attempted to run Gobra with the verification condition generation backend for Viper (which implements the *VCG-TA* algorithm [128]), but we were discouraged from continuing due to performance issues in simple examples in the Gobra test suite and lack of support for features supported in Silicon (e.g., magic wands with quantified permissions on the right operand) which were used at some point in our codebase. As such, we leave the exploration of the performance of VerifiedSCION with verification condition generation for future work.

[128]: Eilers et al. (2024), *Verification Algorithms for Automated Separation Logic Verifiers*

4.4 Verification performance in VerifiedSCION

Silicon verifies each member by symbolically executing it: it explores the branches of the member's control flow and, along each path, discharges proof obligations by querying the SMT solver. Verification time is therefore determined by the number of queries issued and by the cost of each query. The former grows with the number and depth of paths explored during symbolic execution. The latter depends on the two components of a query: the verification context presented to the solver — where quantified assertions are especially costly, as they may give rise to large numbers of quantifier instantiations — and the proof obligation itself. To understand which of these dimensions drives the performance problems in VerifiedSCION, we must first determine where verification time is actually spent. We therefore collect metrics during verification that serve as proxies for these factors, and structure our analysis around the following research questions:

Q1 How is verification time split between different members?
Q2 How much time is spent by the verifier on queries to the SMT solver?
Q3 Which queries to the SMT solver are the slowest?

By answering these questions, we illuminate the performance bottlenecks that *currently* exist on VerifiedSCION, after applying the techniques discussed in Section 4.3, which we hope can guide future research in verification performance.

4.4.1 Experimental setup

In this chapter, we analyze metrics collected from the `router` package of VerifiedSCION, which implements the SCION border router. This is the largest package in the entire project and the slowest to verify.

For simplicity, our analysis is performed on the Viper files extracted from the latest version of the Gobra files from VerifiedSCION (commit

`fdec91c9`). This removes one level of indirection from our discussion of the results, i.e., we do not map the results of verifying the Viper program back to the original program but instead discuss them at the Viper level.

The Viper files analyzed here (as well as in the rest of the chapter) were obtained from Gobra (commit `a208a792`) and verified with a version of Silicon based on commit `475f5e42` from the `master` branch of Silicon, extended to collect metadata about each verified member and SMT query. We use Z3 [19] (version `4.8.7`) as our SMT solver. Our measurements are collected on a 2022 Macbook Pro with an M2 Max CPU and 64GB of RAM. Unless stated otherwise, we run Silicon with the standard configuration, with the exception that (1) we disable per-method parallelism so that the effects of the proposed optimizations are not masked by threads verifying slow members, (2) we use the SE-PC algorithm for discharging heap-related proof obligations due to the presence of disjunctive aliasing (Section 4.2), and (3) we enable joins on impure branches and permission conditionalization globally, which significantly speeds up verification, but provide function-specific configurations for functions that do not terminate in a timely fashion otherwise (e.g., `closureCall_Run` (Subsection 4.3.4).

[19]: De Moura et al. (2008), *Z3: An efficient SMT solver*

4.4.2 Q1: How is verification time split between different members?

Using Gobra to generate a Viper file from the `router` package yields a Viper program with 2313 total members, split as shown in Table 4.3:

	Abstract	Non-abstract	Total
Functions	370	517	887
Predicates	57	97	154
Methods	802	470	1272
Total	1229	1084	2313

Table 4.3: Distribution of generated members per kind.

In this table, we distinguish between *abstract members*, i.e., members whose bodies are not provided to the verifier, and non-abstract ones. Abstract members are, in general, generated for three reasons: (1) they are used to encode built-in constructs in Go or Gobra, (2) they are used in the specifications of trusted members from trusted packages (e.g., the Go standard library specifications), and (3) they correspond to the specifications of functions from imported packages and methods that should not be re-verified. The large number of abstract methods here can mostly be attributed to the large number of packages imported by the router package, which come either from first-party libraries that we verified, or from trusted packages.

Verifying the `router` package in Silicon usually takes around 45 minutes in total. We measured the verification time of 5 executions of Silicon on the Viper program extracted from the `router` package. On average, an execution takes 2691 seconds, with a standard deviation of 164 seconds. This suggests that the verification times obtained in different executions are stable. For the sake of simplicity, we look at a single execution picked arbitrarily in the rest of this section, which we assume to be representative, given the low variance in verification times. In this execution, verification takes 2648 seconds in total, of which only 16 are spent directly verifying abstract members[7]. **The direct overhead of re-verifying the contracts of**

7: Even abstract members have proof obligations. For example, the specifications of abstract functions and methods are checked to be self-framing and well-formed, i.e., the preconditions of all their operations hold.

the imported members is negligible relative to the total verification time, but the increased context may still hurt verification times in less direct ways. We analyze this possibility in Subsection 4.3.1.

We look at the performance of each kind of member in isolation. We list the slowest members of each kind, focusing on predicates that take longer than 1s to verify and functions and methods that take more than 15s to verify. We report multiple dimensions that may be indicative of high verification times:

1. The number of terminal paths.
2. The number of lines of code and specification, which we take as an imperfect proxy for the complexity of the proof obligations[8], including the complexity of the path condition.
3. The total time spent on SMT verification, and the total verification time.
4. The longest SMT query performed during that member's verification.

8: This is an imperfect proxy because it does not take into account the complexity of predicates that are unfolded and of the contracts and bodies of functions and contracts of methods that are called in the body. As such, an unfold statement for a complex predicate has the same weight, in this metric, as an unfold statement of a predicate whose body is just `true`.

Predicate verification. All predicates verify quickly, as shown in Figure 4.2:

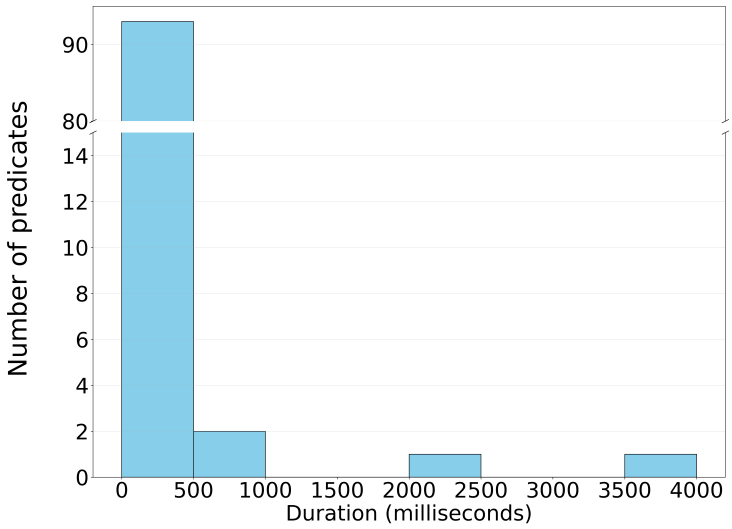


Figure 4.2: Distribution of verification times of non-abstract predicates

Only two predicates take more than one second to verify, and none takes more than four seconds. In Table 4.4, we present various performance metrics relative to these predicates: the number of terminal paths (**TP**), the total verification time (**Time**), the number of lines of Viper code comprising the body of the predicate (**LoC**), the total time spent in SMT queries (**SMT**), and the duration of the slowest SMT query (**SQ**). All times are reported in milliseconds.

Table 4.4: Performance metrics of the slowest predicates.

Name	TP	Time	LoC	SMT	SQ
Mem_DataPlane	512	3,542	54	2,967	7
dynamic_pred_1	4	2,237	185	1,200	5

The predicate `Mem_DataPlane` is the invariant of the `DataPlane` data-structure, which contains a configuration of a SCION border router, and it provides the ownership of all its fields and the data reachable from them if they are already initialized. The predicate `dynamic_pred_1` is a predicate generated by the interface encoding that captures the invariant

of all implementations of an interface type, depending on its runtime type. `Mem_DataPlane` is by far the predicate that leads to the highest number of terminal paths in the entire codebase (the predicate with the second highest number of paths has 4). This high number results from having 9 impure implications of the form

```
d.fieldX ==> acc(PredX(d.fieldX))
```

in its definition, where `fieldX` is a placeholder for a field of DataPlane, and `PredX` is a predicate that depends on the value of the field[9]. The overwhelming majority of the time spent verifying this predicate (2801ms) is spent on checking the infeasibility of paths in order to prune unreachable ones, which is a consequence of the high number of paths explored during its verification. `dynamic_pred_1` is the predicate with the longest body (the one in second place having only 133 lines). Interestingly, in both cases, **the time spent on queries to the SMT solver is more than half the verification time, but each individual query finishes in less than 10 milliseconds**. The cost therefore accumulates over many small queries, not from a few heavy ones.

9: Passing a field as a parameter to `PredX` here prevents this implication from being conditionalized (Subsection 4.3.4).

Function verification. Verification times for functions are low in general. In fact, 95% of the (non-abstract) functions verify in less than 2s, as shown in Figure 4.3.

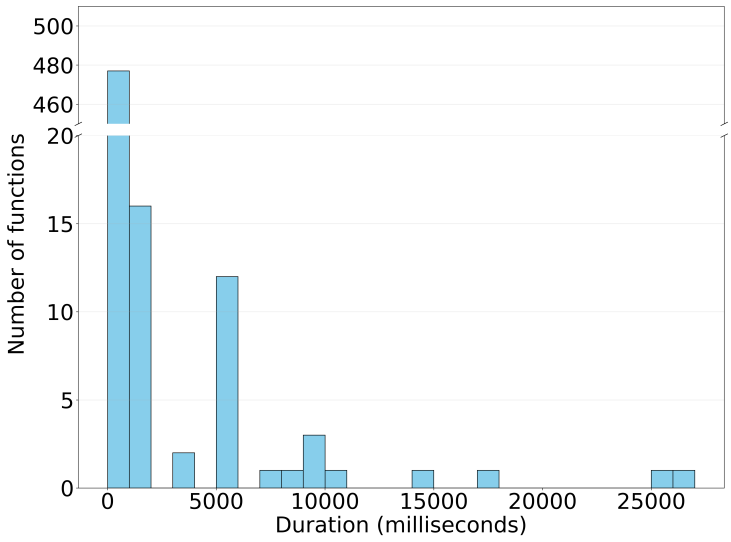


Figure 4.3: Distribution of verification times of non-abstract functions

In total, only 3 functions take more than 15 seconds, and none takes more than 27 seconds. The functions that take the longest to verify are listed in Table 4.5. This table has the same columns as Table 4.4, with the addition of **LoS** which stands for the number of lines of specification in Viper.

Name	TP	Time	LoS	LoC	SMT	SQ
DstIsLocalIngressID	1	26,188	6	9	14,361	35
LastHopLen	1	25,874	8	10	13,905	35
DpAgreesWithSpec	1	17,760	2	4	12,989	8

Table 4.5: Performance metrics of the slowest functions.

These functions take the longest to verify, despite their short contracts and bodies. `DstIsLocalIngressID` and `LastHopLen` unfold the predicates that take the longest to verify (`Mem_DataPlane` and `dynamic_pred_1`),

and `DpAgreesWithSpec` unfolds `Mem_DataPlane`, which causes many new paths to be executed. In all these functions, there is only one terminal path, as none of their preconditions contain impure implications or conditional expressions, and the predicates are unfolded using `unfolding P(x) in e` expressions, which cause multiple paths to be explored for evaluating `e`, which are joined immediately afterwards (Section 4.2). Similarly to the verification of `Mem_DataPlane`, the vast majority of the verification time in these functions (more than 11 seconds in each function) is spent checking which paths are feasible, which is a direct consequence of the number of paths temporarily created to evaluate the bodies of `unfolding` expressions.

Like the predicates shown above, in all three cases, the time spent on queries to the SMT solver is more than half the verification time, but **each individual query finishes quickly**, in no more than 35 milliseconds.

Method verification. In general, methods take the longest to verify, but the large majority of them verify quickly; in fact, 95% of the methods verify in less than 9 seconds, as shown in Figure 4.4:

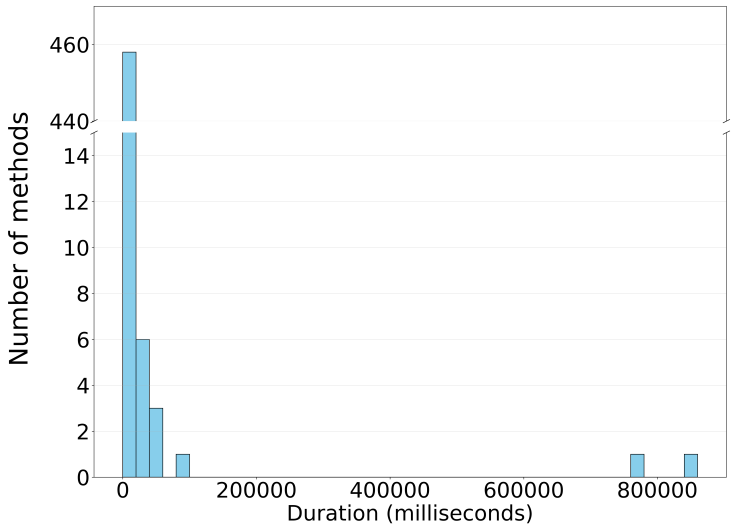


Figure 4.4: Distribution of verification times of non-abstract methods

At the same time, there are 16 methods that take longer than 15 seconds to verify, which we show in Table 4.6. Of these methods, **two of them take a prohibitively long time**: the method `closureCall_Run` implements the receive-process-send loop that the router runs on each of its connections (and thus, it contains various proof obligations related to protocol compliance), and it takes 849 seconds (more than 14 minutes) to verify. The method `process` implements the main logic for updating the headers of SCION packets and it takes 770 seconds (more than 12 minutes) to verify. These methods are a fundamental part of the router implementation and their verification performance was a major bottleneck for making progress, not only due to their long verification times, but also due to the tendency for verification to diverge when it was not yet complete and the methods contained verification errors.

Name	TP	Time	LoS	LoC	SMT	SQ
closureCall_Run	1	848,992	21	870	803,098	10,571
process	119	770,336	207	1,112	638,274	3,894
processOHP	16	99,127	92	1,007	72,465	129
processPkt	23	55,877	69	915	39,364	76
doXover	2	44,741	214	587	34,657	844
prepareSCMP	13	40,327	44	974	22,106	51
initMetrics	6	36,556	17	320	21,221	558
updateElem...	1	36,466	11	172	35,686	10,621
validateSrcDstIA	7	36,366	156	187	22,575	19
decodeLayers	8	35,594	123	502	10,104	448
processEgress	5	31,998	106	184	22,579	34
parsePath	5	23,658	108	266	15,707	79
verifyCurrentMAC	3	19,121	221	387	12,041	41
handleSCMPTrace...	8	17,728	222	387	10,580	12
validateEgressID	48	16,261	224	509	9,533	238
handleEgress...	5	16,056	216	203	10,536	67

Table 4.6: Performance metrics of the slowest methods.

Out of these 16 methods, 11 of them are in at least one of the top ten for the highest number of terminal paths, longest specification, and longest implementation, which suggests that the metrics reported here are indicative of verification performance. Except for one outlier (`decodeLayers`), the time spent on the SMT solver is above 50% of the total verification time, with the two longest methods spending more than 80% doing SMT queries. Here too, **the duration of the slowest SMT query is very small compared to the total time spent in the SMT solver**, but the absolute times of the slowest queries reach dozens of seconds, compared to dozens of milliseconds when verifying functions. Finally, compared to the slowest predicates and functions, the relative time spent on infeasibility checks is lower in methods, which is a consequence of the more aggressive joining of paths in method verification (Subsection 4.3.4).

4.4.3 Q2: How much time is spent by the verifier on queries to the SMT solver?

The measurements so far suggest that most of the time is spent in calls to the SMT solver. This is true not only for the members which we discussed so far, but for the entire codebase, where **the verification time is dominated by the time spent on SMT solving**. In total, Silicon performs 1,316,837 queries to the SMT solver, which are responsible for 77% of the total time spent by Silicon.

In this section, we explore how this verification time is distributed between different kinds of proof obligations. To that end, we instrumented Silicon to report the kind of each SMT query. In general, every query to the SMT solver performed by Silicon can be classified as one of the following categories:

1. **Heap**: Heap-related proof obligations (e.g., does the current state have enough permissions to access a memory location).
2. **Functional**: Queries that discharge non-heap-related assertions like arithmetic properties or conditions on the values of variables.
3. **Infeasibility**: Queries that check whether a program point is unreachable in the current path. Silicon uses these to prune infeasible paths before continuing symbolic execution.

4. **Consistency**: Side-conditions checked by Silicon to guarantee the soundness of its decision procedures (e.g., injectivity constraints on quantified permissions [37]).
5. **Scope**: `push/pop` commands that Silicon emits at branching points and at the boundaries of scoped operations (e.g., `unfolding`) to snapshot and restore Z3's assertion stack in incremental mode [134]. The commands themselves carry no logical content.

[37]: Müller et al. (2016), *Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution*

[134]: Bjørner et al. (n.d.), *Programming Z3 - Incrementality*

Table 4.7 shows the total amount of time spent in each category of SMT queries (in seconds), and the percentage of time spent in SMT solving per kind of query:

Table 4.7: Distribution of total time in SMT solving per kind of SMT query.

	Heap	Functional	Infeasibility	Consistency	Scope
Total time (s)	1,578	70	349	4	45
Percentage of the time in the SMT solver	77.09%	3.42%	17.06%	0.22 %	2.21%
#Queries	118,911	12,797	73,479	7,742	1,103,908

In total, **the majority of the time is spent on discharging proof obligations related to the heap**. These proof obligations take longer, on average, than any other kind of query. Surprisingly, there are more queries performing infeasibility checks than queries about functional correctness. The high number of infeasibility checks suggests that Silicon is verifying many paths. This is due to the `Mem_DataPlane` predicate (Subsection 4.4.2) being used and unfolded often in the `router` package, causing the verifier to branch due to the impure implications in its body. In turn, this may duplicate the work done by the SMT solver (e.g., expensive heap-related proof obligations may need to be checked in each path resulting from inhaling an impure implication, even if the resources provided by the implication are unrelated to the current proof obligation). We believe that `Mem_DataPlane` could be rewritten such that all of its impure implications are moved to separate predicates, which would be unfolded on a per-need basis. This would prevent the automatic branching at the points where this predicate is unfolded. We considered this optimization, but we decided against it, as this would increase the annotation burden even more, and because the optimizations discussed in Subsection 4.3.4 are already effective at decreasing the number of paths executed when verifying methods.

4.4.4 Q3: Which queries to the SMT solver are the slowest?

In general, each query to the SMT solver is discharged very quickly: the 99th percentile of the verification time for queries is below 27 milliseconds, but some queries require multiple seconds to verify, with the slowest taking 11 seconds. In Figure 4.5, we show how the verification times of the slowest queries (those lasting longer than 1 second) are distributed: In total, only 162 queries take this long to verify, and they are distributed across four methods (three of them, `closureCall_Run`, `updateElemWitnessAuxLemma`, and `process` are listed in the table of the slowest methods, Table 4.6). Out of these, 148 are heap-related proof obligations and 141 are heap-related proof obligations related to quantified permissions. **Proof obligations related to quantified permissions are the costliest in VerifiedSCION**: while there are only 17,519 queries related to quantified permissions, compared to 101,392 heap-related queries that are not about quantified permissions, the total amount of time spent on the former is 683 seconds, compared to 895 seconds for the latter.

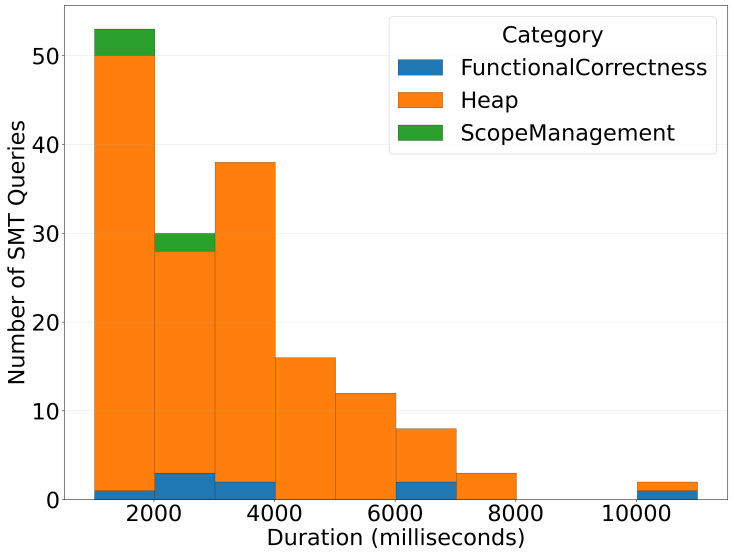


Figure 4.5: Distribution of verification times of SMT queries that take longer than 1s

Quantifier instantiations. The choice of triggers in universally quantified assertions can have a big impact on the verification performance and stability. As discussed in Section 4.2, a common cause of performance and stability issues is the occurrence of matching loops, which translates to large numbers of quantifier instantiations and, possibly, solver divergence. To understand whether matching loops can be a cause of the performance issues in the slowest members, we measured the number of quantifier instantiations reported by Z3 on *program slices computed from each of the slowest members* reported in Subsection 4.4.2 as described in Subsection 4.3.1 and we ran SMTscope [135] on traces obtained by running Z3 on each slice to automatically identify matching loops. We opted to measure quantifier instantiations on program slices rather than the entire program, as tracing the entire execution of Z3 for the entire `router` package would have exceeded the memory restrictions of SMTscope. We report our results in Table 4.8, Table 4.9, and Table 4.10.

[135]: Fiala et al. (2026), *SMTScope: Automated and Efficient Analysis of SMT Traces*

Predicate	QI	TO	UP	ADT	SA
Mem_DataPlane	14,234	No	No	No	No
dynamic_pred_1	9,565	No	No	No	No

Table 4.8: Measures related to quantifier instantiations of the slowest predicates.

Function	QI	TO	UP	ADT	SA
DstIsLocalIngressID	95,014	No	No	No	No
LastHopLen	98,459	No	No	No	No
DpAgreesWithSpec	36,329	No	No	No	No

Table 4.9: Measures related to quantifier instantiations of the slowest functions.

Table 4.10: Measures related to quantifier instantiations of the slowest methods.

Method	QI	TO	UP	ADT	SA
closureCall_Run	12,985,041	Yes	No	No	No
process	27,759,279	Yes	No	Yes	No
processOHP	1,378,729	No	No	Yes	No
processPkt	1,127,768	Yes	No	Yes	No
doXover	804,105	No	No	Yes	No
prepareSCMP	586,311	No	No	Yes	No
initMetrics	2,099,649	Yes	No	No	Yes
updateElem...	57,776	No	Yes	No	No
validateSrcDstIA	488,440	Yes	No	No	No
decodeLayers	909,231	No	Yes	Yes	No
processEgress	1,487,637	No	No	Yes	No
parsePath	500,590	No	No	Yes	No
verifyCurrentMAC	436,207	No	No	Yes	No
handleSCMPTrace...	147,907	No	No	Yes	No
validateEgressID	282,350	No	No	Yes	No
handleEgress...	303,561	No	No	Yes	No

In column **QI** of each table, we report the number of quantifier instantiations reported by Z3 (with the flag `-st` for reporting solver statistics) when verifying each program slice. **The number of quantifier instantiations range from a few dozens of thousands of instantiations for the slices of the slowest predicates and functions, to dozens of millions for those of the slowest methods**[10]. **TO** reports whether SMTscope times-out (i.e., its analysis does not finish in 10 minutes) when analysing the full trace produced by Z3 when verifying a slice. When a trace leads to a time-out, we populate the columns to the right on a best-effort basis, based on prefixes of the trace for which the analysis terminates. **UP** reports whether a matching loop caused by user-provided assertions was found. Only two members exhibit this kind of matching loop: `updateElemWitnessAuxLemma` and `decodeLayers`. These correspond to overlooked issues that should be fixed, especially the one that occurs in member `updateElemWitnessAuxLemma`, where the slowest query to the SMT solver takes more than 10 seconds (Table 4.6). **ADT** reports matching loops caused by the theory solver for algebraic datatypes in Z3 [136], which encodes the various symbols related to algebraic datatypes (constructors, accessors, recognizers) as uninterpreted functions and emits the axioms for these functions lazily. We speculate this may be caused by an issue in Z3 that causes non-termination when checking satisfiability in the presence of a recursive data type and a condition that is unsatisfiable for any finite instance of it [137]. Finally, **SA** reports whether matching loops caused by Silicon's built-in axiomatizations was found. Only one member suffers from a matching loop of this kind, whose possibility is documented in the map axiomatization file [138].

In conclusion, **matching loops are detected in the verification of almost all of the slowest methods**. Surprisingly though, only two methods suffer from matching loops introduced by user-provided assertions. Matching loops are also introduced by Z3's theory solver for algebraic datatypes and the built-in axiomatizations of Silicon, and they cannot be easily curbed with additional annotations on the input program.

10: The program slice of a member may include other members from the list. For example, the program slice for `closureCall_Run` includes the predicate `Mem_DataPlane`, as this predicate is a direct dependency of the method. Thus, the number of quantifier instantiations computed for a member should be taken as a loose upper-bound on the quantifier instantiations that occur in that method alone.

[136]: Bjørner et al. (n.d.), *Z3 Internals (Draft) - Algebraic Datatypes*

[137]: McPeak (n.d.), *Infinite loop making model for recursive datatype in presence of quantifiers*

[138]: (n.d.), *Silicon's axiomatization of maps*

4.4.5 A note on verification instability

We experienced proof instability occasionally, especially with certain configurations of the verifier (e.g., when branch parallelism is enabled, as discussed below) or when certain features are used, like the support for SIF (Subsection 3.6.5), which leads to large proof obligations. While instability was an obstacle to the long-term maintainability of the project, it was a lesser problem for us than verification performance. We believe that this was due to four main reasons:

1. First, Gobra quickly adopted an encoding where the identifiers in the generated Viper program do not depend on global state or their position in the source code. This made instability less likely to be observed.
2. Apart from the extensive use of universal quantifiers, the proof obligations found in VerifiedSCION rarely fall on theories that are known to cause instability in the Z3 verifier (e.g., non-linear integer arithmetic [139]).
3. Gobra fixes the random seeds of the solver, and verifies the proof obligations within each member in a deterministic order.
4. In the beginning of the project, before most proof obligations were proven, slow verification times were a greater impediment than instability, which manifests itself in previously-verifying proof obligations. Thus, we invested early into techniques to decrease verification times by reducing the proof context (Section 4.3). We speculate that these steps contributed en passant to the stabilization of our proof obligations, given that reducing irrelevant proof context may increase proof stability [126]. Intuitively, if instability is the consequence of the SMT solver being highly sensitive to its input in unpredictable ways, then removing unnecessary context from a proof obligation (like ambient formalizations that are not used in a proof) means that the proof obligations passed to the SMT solver become less likely to change when unrelated parts of the program under verification change.

[139]: (n.d.), *Dafny Documentation - Verification optimization guide*

[126]: Zhou et al. (2024), *Context Pruning for More Robust SMT-based Program Verification*

Enabling branch parallelism. By default, Silicon allocates an instance of the SMT solver per logical core present in the machine, and it parallelizes method verification. To verify a method, Silicon picks an SMT solver instance that is currently not in use and it dispatches all proof obligations to that instance until completion. In situations where the verification time is dominated by a few methods, this strategy may lead to inefficient resource allocation: after some time, only a small number of SMT solver instances are busy verifying the proof obligations of slow methods sequentially, while many other instances sit idle.

To improve resource utilization in these cases, Silicon can be configured to verify branches in parallel rather than sequentially. Despite improving verification performance, this feature causes verification instability, which translates to spurious errors occurring occasionally in verifier runs. We believe the most plausible reason for this is that enabling branch parallelization causes the names of variables used in the encoding of each proof obligation in SMT-LIB2 (the format for logical formulas accepted by the SMT solver) to change between runs more frequently than when this feature is disabled. To understand why renamings can happen more often with branch parallelization, it is necessary to understand that Silicon generates *fresh variables*, i.e., variables whose identifiers syntactically differ from all identifiers used so far by the solver, to name terms in proof

obligations. These identifiers are computed based on a per-solver counter. Enabling branch parallelization causes more renamings of variables in the generated inputs to the SMT solver to be observable across different executions of the verifier: when branch parallelization is disabled, the names of the fresh variables used in all proof obligations for a given member depend exclusively on the initial value of the counter of the solver assigned with verifying that member; enabling branch parallelization makes the names of the fresh variables used in proof obligations from different branches unrelated, given that any branch may be assigned to any solver instance.

We believe this may result in increased observable instability, as the operation of SMT solvers is sensitive to variable renamings [140]. We resolved this trade-off in favor of verification stability because it is essential for making steady progress in the verification.

[140]: Zhou et al. (2023), *Mariposa: Measuring SMT Instability in Automated Program Verification*

4.5 Related work

Verification performance is a common problem for practitioners of formal verification [122], motivating various approaches to deal with slow verification times. Verifier instability (also called *brittleness*), a related problem, has also received considerable attention. In the remainder of this section, we discuss the most relevant lines of work; the discussion is not meant to be exhaustive.

[122]: Brugger et al. (2026), *Toward Practical Deductive Verification: Insights from a Qualitative Survey in Industry and Academia*

Identifying the sources of poor performance. Various solutions have been proposed to identify the sources of performance issues. The authors of the VCC verifier [141] report using the *Z3 Axiom Profiler* [130] and the *Z3 Inspector* [130] to identify the sources of long verification times.

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

The Z3 Axiom Profiler analyzes the quantifier instantiation patterns obtained during a run of the verifier to identify problematic behaviors like matching loops. It was later extended by Becker et al. [142] with features like a visualization of the causal relationships between instantiations, and detailed justifications for each quantifier instantiation. Recently, the Axiom Profiler was replaced by SMTscope [135], a novel tool with more effective analysis for detecting matching loops and multiplicative quantifiers.

[142]: Becker et al. (2019), *The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations*

[135]: Fiala et al. (2026), *SMTScope: Automated and Efficient Analysis of SMT Traces*

The Z3 Inspector monitors the progress of Z3 to pinpoint the verification conditions that take too long to be discharged. It displays the source code of the original VCC program and annotates each line with possible errors that could be reported for that line if a proof obligation is not proven. As the SMT solver works through the formula, the error message corresponding to the current proof obligation is highlighted, providing visual feedback on which SMT queries take the longest. Similarly, Dafny [17] provides visual information for how long each assertion takes to be discharged by the SMT solver in its IDE [143].

[17]: Leino (2010), *Dafny: An Automatic Program Verifier for Functional Correctness*

[143]: Mayer (2023), *Making Verification Compelling: Visual Verification Feedback for Dafny*

In the context of program verifiers based on Symbolic Execution, Arquint [144] proposed a profiler for Silicon which records the duration of individual symbolic execution steps.

[144]: Arquint (2019), *Profiling Symbolic Execution*

Reducing the verification times through context management. Verification times in SMT-based verifiers are highly sensitive to the verification context passed to the SMT solver when discharging proof obligations. As such, various tools have evolved to allow for careful management and simplification of the verification context.

Dafny provides `assert-by` statements [145], which allow proving an assertion using additional statements as proof hints. Its encoding ensures that the provided hints do not pollute the proof context of other assertions. In addition, Dafny provides various features to manage the definitions that are in scope, including the ability to label and hide assertions [146] and opaque functions with proof annotations to reveal and hide function definitions in selected scopes of the program. More recently, Dafny introduced `opaque` blocks [147] which are similar to Gobra's `outline` statements. Unlike `outline` statements, the context at the start of an `opaque` block is the full context accumulated up to the block definition, whereas the starting context of an `outline` statement is provided explicitly by a precondition. `outline` statements provide more control over the context, at the cost of requiring more annotations. Besides user-facing features, Dafny's implementation underwent changes to reduce the number of quantified axioms in context. For example, Bordis and Leino [148] introduced *free facts*, i.e., quantified axioms that are automatically pre-instantiated by Dafny based on strict syntactic clues rather than being emitted to the SMT solver.

[145]: The dafny-lang community (n.d.), *Dafny Reference Manual: assert ... by*

[146]: The dafny-lang community (n.d.), *Dafny Reference Manual: Labeling and revealing assertions*

[147]: The dafny-lang community (n.d.), *Dafny Reference Manual: Opaque Block*

[148]: Bordis et al. (2024), *Free facts: An alternative to inefficient axioms in Dafny*

Like Dafny, Verus [149] provides various features for context management including `assert-by` statements and opaque functions. Verus's `assert-by` statements also allow for careful control of which decision procedures are enabled: by default, Verus does not use bit-blasting and the decision procedures for non-linear arithmetic, as both are known to lead to proof instability and slow-downs. However, these features may be selectively enabled in the proof hints of an `assert-by` statement. Furthermore, Verus allows for explicit management of the verification context through *broadcast groups* [150], i.e., sets of ambient formalizations (possibly containing quantified facts) that are made available to the verifier at the module, function, or proof context level. Their approach lets library authors provide different predefined automation levels, depending on the quantified assertions they expose.

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

[150]: Bai et al. (2025), *Tunable Automation in Automated Program Verification*

Very recently, Dross et al. [151] reported various strategies to manage the verification context in the SPARK verifier for Ada [152]. First, they implement algorithms to prune unused symbols and hypotheses in the generated verification conditions, to replace the encodings of Ada records that are never accessed with abstract types, and to hide definitions based on the access modifiers and visibility rules of the language. In their experiments, these optimizations lead to improvements in verification times. Furthermore, they introduce features to manually manage the verification context, like revealing and hiding function definitions on-demand, and the `Assert_And_Cut` statement, which is used to assert a property P. After the assertion, P becomes the only fact known about the program up to that point. A similar functionality can be obtained with `outline` statements in Gobra.

[151]: Dross et al. (2026), *Managing the Proof Context in SPARK*

[152]: Barnes (2012), *Spark: The Proven Approach to High Integrity Software*

Techniques for reasoning about the heap. As seen in this chapter, reasoning about the heap and aliasing may be a source of verification performance issues. It is often practical, therefore, to switch between different verification algorithms that offer different trade-offs between completeness and verification performance for dispatching heap-related

proof obligations. In the context of Separation Logic, Eilers et al. [128] provide a comprehensive analysis of the performance and completeness characteristics of various algorithms for automated verifiers. Another common way to trade off completeness for verification performance is to dispatch heap-related proof obligations through efficient type checkers by adopting linear type systems [153, 154], rather than dispatching them through SMT solvers. More recently, it was shown that uniqueness and immutability annotations may also be introduced in a type system to avoid or simplify heap-related proof obligations dispatched to the SMT solver [155].

[128]: Eilers et al. (2024), *Verification Algorithms for Automated Separation Logic Verifiers*

[153]: Bierhoff (2011), *Automated program verification made SYMPLAR: symbolic permissions for lightweight automated reasoning*

[154]: Li et al. (2022), *Linear types for large-scale systems verification*

[155]: Haak et al. (2026), *Scalable Deductive Verification of Data-Level Parallel Programs*

Caching and re-verification. In a verification project, proof obligations are checked multiple times throughout the proof engineering workflow: first, writing a proof requires verifying various versions of the annotated program until the verifier accepts the proof; second, the verifier or its dependencies (e.g., the SMT solver) may be updated in ways that break pre-existing proofs; finally, verification may be integrated into continuous integration pipelines. As such, optimizing re-executions of the verifier and caching of verification results has received some attention.

Gopinathan et al. [125] propose an approach to reduce the time it takes to re-verify a program by reducing the possible quantifier instantiations and thus, the search space for the SMT solver. Starting from a successful verification, they extract the *proof essence* of each assertion, which includes the quantified axioms used by the SMT solver to prove its validity. Based on the proof essence, they generate an optimized program (with the same semantics) such that re-verifying an assertion in the optimized program uses only the axioms in its proof essence.

[125]: Gopinathan et al. (2025), *Accelerating Automated Program Verifiers by Automatic Proof Localization*

Leino and Wüstholz [156] reduced the response times of the Dafny IDE by adopting an algorithm for caching verification results that precisely identifies which parts of the program need reverification when the program changes. Of particular interest, this algorithm is able to partially cache proof obligations of a method even when that method changes. For example, adding a new postcondition to a method causes the body of the method to be verified only against the new postcondition, rather than all postconditions. In addition, callers that were verified previously to the addition of the postcondition are not re-verified, even though the specification of one of their callees just changed. Sadykov et al. [157] propose an algorithm to cache and re-use SMT solver's unsatisfiable cores to reduce the number of solver calls in symbolic execution.

[156]: Leino et al. (2015), *Fine-grained caching of verification results*

[157]: Sadykov et al. (2025), *Cache-a-lot: Pushing the Limits of Unsatisfiable Core Reuse in SMT-Based Program Analysis*

Measuring and reducing verifier instability. Much work on the performance of SMT-based verifiers targets proof instability, i.e., small changes in the input program leading to noticeable differences in the verification time and result.

Verifier instability is still poorly understood. Cebeci et al. [158] analyzed unstable queries reported in the Z3 repository and identified bugs, solver misconfigurations, and misunderstandings of how triggering works as the root causes of the observed instability. Furthermore, they conjecture that verifier instability is often caused by engineering mistakes.

[158]: Cebeci et al. (2025), *A conjecture regarding SMT instability*

Zhou et al. [140] built Mariposa, a tool that measures instability by creating semantics-preserving mutants of an SMT query (obtained by renaming variables, reordering assertions, and changing random seeds of the solver) and running the solver on each mutant. A statistical analysis is used to determine if a query is unstable based on how many of the

[140]: Zhou et al. (2023), *Mariposa: Measuring SMT Instability in Automated Program Verification*

mutants are solvable within a time limit. Using Mariposa on proof obligations obtained from verification projects written in Dafny and F*, they found that 2.6% of the SMT queries are unstable, and this number is likely to be larger for projects still under active development. As such, it is likely that someone doing a large proof development in these tools will face instability.

To reduce the likelihood that instability will be found due to semantics-preserving transformations, Amrollahi et al. [159] propose normalizing the SMT solver inputs.

[159]: Amrollahi et al. (2025), *Towards SMT Solver Stability via Input Normalization*

Zhou et al. [126] showed that irrelevant context may greatly contribute to instability, and proposed a technique to filter out less relevant axioms. Many of the aforementioned approaches for managing verification context may also be used to stabilize verification [148, 150, 151], and some code-level strategies to reduce the verification context, like using extrinsic specifications rather than intrinsic (i.e., specifying properties of functions as separate lemmas, rather than through their contracts) may also help avoid instability [160].

[126]: Zhou et al. (2024), *Context Pruning for More Robust SMT-based Program Verification*

[160]: Tomb et al. (2023), *Avoiding verification brittleness in Dafny*

Finally, Zhou et al. [161] propose Cazamariposas, a tool to debug instability problems caused by problematic quantifiers which proposes fixes based on whether a quantifier is over- or under-instantiated.

[161]: Zhou et al. (2025), *Cazamariposas: Automated Instability Debugging in SMT-Based Program Verification*

4.6 Conclusion

In this chapter, we examined how verification time is spent in a large verification project carried out in Gobra. We observed that, for the vast majority of members, the verification time is very short. In fact, the verification time is dominated by a small number of members that take a very long time to verify. In addition, we described the strategies adopted to keep verification time manageable. Our strategies allow for fine-grained control over the verification context, which ultimately leads to a reduction of irrelevant information in the proof obligations. We also explored how picking different verification algorithms for heap-related proof obligations may lead to different trade-offs between completeness and performance, and between indicators of verification time like the number of terminal paths explored by the verifier and the complexity of each query.

This chapter shows that, in the current state of the art for automated separation logic verifiers, understanding and optimizing the verification performance of a large verification project in a verifier like Gobra and Viper requires familiarity with the underlying algorithms used to generate the proof obligations, and often requires a deliberate trade-off between completeness and performance and a delicate balance between the duration of individual SMT queries and the number of paths that are executed. As a consequence, optimizing verification times is still out of reach for most casual users of these tools, and only accessible to power users. Rather than hiding these algorithms under the hood as an implementation detail, power users of these tools may desire explicit control over which verification algorithms are used for a given program, member, or even block. In addition, we find that the deep integration between these verifiers and SMT solvers is sometimes an impediment to effective progress in verification—rather than spending time focusing on the logically relevant parts of the proof, we find ourselves establishing hypotheses about the causes of poor performance and experimentally

testing them, usually by selectively hiding different parts of our program from the SMT solver. We believe that program verifiers aimed at power users ought to allow them to control when to rely on the SMT solver (perhaps as a tactic), but should also allow them to fall back to manual proofs when SMT solvers behave unpredictably. Velvet [162] is a good first-step in this direction: it is a foundational verifier for an imperative *heapless* language embedded in Lean [163] that allows for *multi-modal* proofs which combine Lean's automation tactics, SMT solvers, and manual proving. We believe this is a worthwhile direction for separation logic verifiers too—different algorithms for generating and verifying heap-related proof obligations, whether based on symbolic execution like the ones discussed here or based on verification condition generation, could be provided as tactics for reasoning about heap-manipulating programs rather than as standalone programs.

[162]: Gladshtein et al. (2026), *Velvet: A Foundational Multi-Modal Verifier for Imperative Programs in Lean*

[163]: Moura et al. (2021), *The Lean 4 Theorem Prover and Programming Language*

4.6.1 Future work

This chapter provides a starting point for future work on understanding the sources of poor verification performance in large projects carried out in Viper-based verifiers. While the measurements shown here focus on a fully verified program, it is seldom the case that we are verifying a program that is certainly free of verification errors (e.g., when verification is still ongoing), and both our experience and the folklore wisdom indicate that SMT-based verifiers behave more unpredictably in the presence of errors. Future experiments should thus focus on measuring how the verification times of the components of a project evolve over time and on identifying strategies to deal with solver divergence early on.

In addition, the experiments so far show how onerous it is to discharge heap-related proof obligations, especially those related to quantified permissions. It is thus important that we develop new algorithms for reasoning about the heap and investigate their trade-offs (e.g., the work by Markus Limbeck on a greedy algorithm for quantified permissions in Silicon [164]) and that we find ways to soundly integrate substructural (especially linear or affine) type systems for which there are efficient type checkers with separation logic verifiers, which could significantly reduce the number of heap-related proof obligations discharged by the SMT solver.

[164]: Limbeck (2025), *Enhancing Deductive Verification of Unbounded Heap Data Structures*

Finally, we believe there is still opportunity for optimizing Silicon to skip redundant proof obligations. In particular, Silicon currently performs branching eagerly: when an impure implication is inhaled, Silicon branches right away, which may duplicate checks. Conditionalizing the permission amount (Subsection 4.3.4) often helps reduce the number of paths, but the rewriting of the permission amount is conservative and often skips rewritings that are safe, but whose safety cannot be syntactically decided. Likewise, performing more joins (Subsection 4.3.4) can reduce the number of paths but it relies on a heavy-handed algorithm to merge the symbolic states of different paths, which complicates the proof obligations. We believe that yet another optimization that may reduce the number of paths and the redundant checks is to perform branching lazily. For that, we can introduce a new kind of heap chunk, a *lazy conditional chunk*, that represents an impure implication. Executing the inhales at the beginning of the program causes such a chunk to be produced without branching. When discharging a proof obligation that may depend on the resources granted by the right-hand side of an impure implication stored as a lazy conditional chunk (which could

be determined with a syntactic check based on the fields and predicate instances that occur in the proof obligation), Silicon would branch and replace the lazy chunks by their non-lazy counterpart.

Modular Reasoning about Global Variables and Their Initialization

5

5.1 Introduction 89
5.2 Key ideas 93
5.3 Formalization 101
5.4 Implementation 110
5.5 Evaluation 114
5.6 Related work 120
5.7 Conclusion 121

In this chapter, we present a practical verification technique to reason formally and modularly about global state and its initialization. Our technique is based on separation logic and uses *module invariants* to specify ownership and values of global variables. A partial order on modules and methods allows us to reason modularly about when a module invariant may be soundly assumed to hold, irrespective of when exactly the module initializer establishing it runs. Our technique supports both thread-local and shared global state. We formalize it as a program logic in Iris and prove its soundness in Rocq. We make only minimal assumptions about the initialization semantics, making our technique applicable to a wide range of programming languages. We implemented our technique in existing verifiers for Java and Go and demonstrate its effectiveness on typical uses cases of global state, on the codebase of the VerifiedSCION project (Chapter 3), and on the development of verification libraries.

The implementation of our technique in exising verifiers, the instructions on how to reproduce our evaluation, and our formalization in Iris were provided as an artifact to OOPSLA 2025 [165].

[165]: Pereira et al. (2025), *Artifact for "Modular Reasoning about Global Variables and Their Initialization"*

5.1 Introduction

Many programming languages support *global state* to implement patterns such as caches, dispatch tables, and global counters. To list a few, C, C++, Go, and Rust support global variables, Java and C# have static fields, and Scala has singleton objects. Global variables have two defining characteristics: First, their scope corresponds, at least, to the module[1] in which they are declared; often, they are accessible to the entire program. Second, they have a *static lifetime*, and thus, they may be accessed at any point during the execution of the program. In practice, global variables are widely used for storing both shared immutable values (e.g., the OpenJDK implementation of Java's `Byte` class [166] has a static field named `cache` that stores an array of interned instances of `Byte` to be returned by the method `Byte.valueOf`) and mutable state (e.g., Java's `Logger` class [167] stores the hierarchical namespace of loggers in static fields).

1: While we do not formally define *module*, we note that it corresponds to *classes* in Java, *packages* in Go, and *compilation units* in `C/C++`.

[166]: (2024), *OpenJDK's implementation of Java's Byte class*

[167]: (2024), *Java™ Platform, Standard Edition 8 API Specification - Class Logger*

Both characteristics pose challenges for modular verification. The global scope complicates local reasoning: global variables are shared among different methods (and different threads in concurrent executions), such that ownership-based verification as in separation logic [20] is not directly applicable. A standard solution to this problem is to maintain invariants over the shared state. We adopt this approach by introducing *module invariants* over the values of global variables. Such invariants need to be established during the initialization of global variables; *from then on*, they may be assumed and need to be preserved by code that accesses global state. However, this simple inductive argument is complicated by the second characteristic of global variables, their static lifetimes. Real programming languages have a complex initialization semantics; for instance, in Java, C#, Go, and Rust[2], global variables are initialized by *module initializers*. Module initializers are guaranteed to *start* executing

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

2: Initializers are not part of the Rust language, but a similar functionality is provided by the `lazy_static!` macro defined in the `lazy-init` crate [168], which precedes all accesses to a global variable by code that runs the initialization code if it hasn't been run yet.

before the global variables of their modules are accessed for the first time, but are in general *not* guaranteed to *finish*, for instance, when the access occurs *during* the initialization or when module initializers are mutually recursive. Consequently, the module invariant may not have been established by the time of the first access. As a result, this language design poses three challenges for modular verification.

1. *Validity of module invariants:* A verification technique must determine when module invariants may soundly be assumed to hold. This requires a way to determine when each module initializer has definitely finished (and, thus, established the module invariant) and when a module invariant may be temporarily violated, for instance, during an update of global data through another thread. This challenge is complicated by the fact that the order in which module initializers run is not known modularly (e.g., in Go [169]) or is even determined dynamically (e.g., in C# and Java [170, 171]).
2. *Side effects of module initializers:* In mainstream languages, module initializers may perform arbitrary side effects on all memory locations reachable from global variables. Consequently, the program behavior depends on when module initializers are executed, and in which order. Since this information is not known modularly (or even statically), a verification technique must take into account all valid timings of the module initializers.
3. *Cyclic initialization dependencies:* Some languages like Java and C# allow two or more modules to mutually access each other's global variables from their module initializers (possibly indirectly via method calls). Such cyclic dependencies usually indicate latent bugs, such as reading uninitialized global variables and deadlock [172, 173]. Bugs like these plague many concurrent codebases, including the previously mentioned `Logger` class in the Java standard library. They need to be prevented by a verification technique.

[169]: (2025), *The Go Programming Language Specification, Package initialization*

[170]: (2024), *Java Language Specification. Chapter 12: Execution*

[171]: (2024), *Static Constructors (C# Programming Guide)*

[172]: Börger et al. (2000), *Initialization problems for Java*

[173]: Liu et al. (2023), *Initializing Global Objects: Time and Order*

Despite the prevalence of global state, to our knowledge, there are no verification techniques that provide a practical solution to these challenges. Leino and Müller [174] propose a verification technique for invariants over global state. In their technique, modules are initialized eagerly according to a user-defined validity ordering. It is, thus, not applicable to mainstream languages, which have a different initialization semantics. Summers et al. [175] also use invariants to reason about static fields, but require programs to follow a strict, static ownership scheme [176], which is too restrictive for many practical use cases. Both of the above approaches support only sequential programs. Jacobs et al.'s technique [177] targets concurrent programs by requiring *all* accesses to global variables to be synchronized with a lock. They do not support programs that safely access mutable global variables without explicit synchronization. Moreover, like both previous approaches, they impose a restrictive ownership regime. Liu et al. [173] present a static analysis to identify common issues in the initialization of Scala's singleton objects. However, their technique does not allow one to verify functional correctness. Finally, prior to our work, no deductive verifier like Gobra [29], KeY [178], VerCors [121], or VeriFast [119] supports modular reasoning about global state.

[174]: Leino et al. (2005), *Modular Verification of Static Class Invariants*

[175]: Summers et al. (2009), *Universe-Type-Based Verification Techniques for Mutable Static Fields and Methods*

[176]: Dietl et al. (2005), *Universes: Lightweight Ownership for JML*

[177]: Jacobs et al. (2007), *A Simple Sequential Reasoning Approach for Sound Modular Verification of Mainstream Multithreaded Programs*

[173]: Liu et al. (2023), *Initializing Global Objects: Time and Order*

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

[178]: Ahrendt et al. (2016), *Deductive Software Verification - The KeY Book - From Theory to Practice*

[121]: Armborst et al. (2024), *The VerCors Verifier: A Progress Report*

[119]: Jacobs et al. (2010), *A Quick Tour of the VeriFast Program Verifier*

In this chapter, we present the first modular verification technique for programs with global state that is sufficiently expressive to handle practical implementations and languages. To handle mutable state and concurrency, our technique builds on concurrent separation logic [20, 71]. In particular, we treat global variables as separation logic resources that can

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

[71]: O'Hearn (2007), *Resources, concurrency, and local reasoning*

be accessed only when the executing methods owns the corresponding permission. This treatment allows us to rule out data races and reason about side effects on global variables.

We specify modules with *module invariants*, separation logic assertions that may contain permissions to the global variables of a module as well as constraints on their values. A proof obligation ensures that the invariant of each module is established by the module's initializer.

Similar to Iris invariants [28], module invariants are represented as separation logic resources called *module tokens* (or *tokens* for short). A method may *open* a module invariant to obtain the contained permissions if it owns the corresponding token. Opening a module invariant is sound only at program points where the initialization of the corresponding module has definitely finished; otherwise, one would potentially assume the module invariant before it has been established. Therefore, opening an invariant is guarded by two proof obligations: (1) An invariant may be opened when the subsequent execution step (or sometimes an earlier operation) ensures that initialization must have started; for instance in Java, initialization of a class is triggered by object creation, static field access, and static method calls and, thus, an invariant may be opened at these operations. This proof obligation guarantees that by the time we attempt to use the invariant, initialization has at least started. (2) We check that the current method is not executed *during* the initialization; the semantics of the programming language then implies that module initialization must also have finished before the current method is executed. To enable this second proof obligation, we introduce a novel specification construct called *initialization levels* that establishes a partial order on the modules and methods of a program. Proof obligations ensure that the initializer of a module with level m may call only methods with a strictly smaller level. Consequently, a method with level l cannot be part of the initialization of any module with a level of at most l, such that their initialization, if started, must also have finished.

[28]: Jung et al. (2018), *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*

The approach described so far already solves the three challenges above. A method may soundly assume the invariant of a module if it holds the corresponding token and the method's level is at least the module's level (Challenge 1). Since module initializers do not own any module tokens, they cannot cause side effects beyond initializing the global variables of the enclosing module (and newly-allocated memory). Consequently, verification does not depend on their exact execution time (Challenge 2). For the same reason, a module initializer cannot read global variables of other modules, making it impossible to create cyclic initialization dependencies (Challenge 3).

The technique described so far is sound, but overly restrictive for two reasons. First, it is common for global variables to store *immutable* data such as constants or global look-up tables. Once initialized, such data can be safely accessed (also concurrently), such that the verification overhead of passing around module tokens is unnecessary. The same applies to global data that is protected by a synchronization primitive such as a lock. Second, not allowing the initializer of one module to access global variables of another module prevents common idioms, such as printing debug messages during initialization using Java's `System.out` stream.

To remove these limitations, our technique supports *duplicable* module invariants, in addition to the *non-duplicable* invariants described so far. Duplicable invariants may contain permission to read memory locations as well as value constraints to capture properties that, once established, hold for the rest of the program execution. Therefore, opening a duplicable

invariant does *not* require the presence of a module token; every method or module initializer may open the duplicable invariant of every module with at most the same level. Verification that a duplicable invariant has been established is analogous to non-duplicable invariants.

Duplicable invariants provide a convenient way to access immutable data. They can also be used to share a lock among different threads. By including permissions in the lock invariant, threads can get synchronized write access to global state. Duplicable invariants also let the initializer of a module read global variables of other modules, irrespective of their levels, and even modify them if they are protected by a synchronization primitive (we will present a further relaxation for sequential code in Subsection 5.4.1).

For duplicable invariants, Challenge 1 and 2 are solved analogously to non-duplicable invariants. Challenge 3 is solved by the level requirement. Since a module initializer may only read global variables of modules with a strictly smaller level, cyclic dependencies are not possible.

Our technique is applicable to any programming language that satisfies the following three conditions:

C1 We call a point in the execution of a program *definitely initialized for a module M* if it occurs (1) right before an access to a global variable of M or a call to a static method[3] of M, and (2) outside of M's module initializer and methods directly or indirectly called from it. The module initializer of M starts executing at the latest when reaching the first definitely-initialized execution point for M. It also finishes before executing the next operation after the first definitely-initialized execution point for M unless (1) it does not terminate or (2) M is part of a cyclic initialization dependency. In case (1), all threads block at the next definitely-initialized execution point for M.

C2 In case the language includes operations that trigger module initialization (e.g., in Java, those mentioned in the previous condition) then only the first occurrence of such an operation in a program execution triggers. No other operations wait for module initialization to finish.

C3 The module initializer of a module M runs at most once during the entire execution of the program.[4]

3: In our examples, we treat constructors analogously to static methods.

4: Conditions **C2** and **C3** may seem redundant at first. However, **C2** ensures that triggering module initialization does not yield an error. Moreover, for languages where initialization may happen only when triggered, **C3** follows from **C2**, but is necessary for our technique to be sound for other languages.

These conditions are satisfied by a wide range of mainstream programming languages with different initialization semantics, including Java, Go, C#, and Scala. In particular, the conditions do not require that the initializer of a module runs, unless an access to one of its variables occurs. This allows for both eager and lazy initialization.

In the rest of this chapter, we present our technique informally and illustrate it on examples (Section 5.2), and then formalize our techniques as a program logic in the Iris framework [84] and prove its soundness in Rocq (Section 5.3). To demonstrate that our technique is practical, we instantiate it to two mainstream languages with different initialization semantics, Java and Go, and implement it in two state-of-the-art deductive verifiers, VerCors [121] and Gobra [29] (Section 5.4). Our evaluation demonstrates that the technique is expressive and can be automated effectively (Section 5.5).

[84]: Jung et al. (2015), *Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning*

[121]: Armborst et al. (2024), *The VerCors Verifier: A Progress Report*

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

Contributions We summarize the main contributions of this chapter:

- We present the first practical, modular verification technique for programs with global state, written in mainstream programming languages. Our technique is applicable to a wide range of programming languages with different initialization semantics including eager and lazy initialization, such as C++, C#, Go, Java, and Scala.
- We formalize our technique as a separation logic. Our soundness proof is mechanized in the Iris framework and applies to languages whose initialization semantics satisfies conditions **C1-C3**.
- We implement our technique in two state-of-the-art verifiers for languages with different initialization semantics: Gobra for Go and VerCors for Java.
- We demonstrate the effectiveness of our approach on a wide range of challenging examples. Our experiments show that our technique imposes little annotation overhead and can be automated in SMT-based verifiers.

Collaboration To reiterate, this chapter results from a collaboration between me, Marco Eilers, Isaac van Bakel, Patricia Firlejczyk, and Peter Müller. I designed and implemented various solutions to support this feature in Gobra and observed that we should be able to generalize it to any language that supports global variables and static initializers, as long as we impose a few restrictions on the semantics of the language. To that end, I and Marco Eilers supervised the bachelor's thesis of Patricia Firlejczyk, who came up with a language-agnostic solution for sequential programs. I generalized this solution to multi-threaded programs, after realizing that any program could be modeled as a multi-threaded program where each static initializer is modeled as a separate thread and every access to global variables may potentially synchronize the thread performing the access and the initializer thread. I implemented the approach described in this chapter in Gobra, and Marco implemented it in VerCors. Isaac van Bakel joined the project to formalize our approach in the Iris separation logic and prove its soundness. He also came up with the formal statement of the absence of initialization cycles and identified some unsoundnesses in the different versions of the program logics.

5.2 Key ideas

In this section, we present our technique informally and illustrate its key ideas on several examples; the formal proof rules are presented in the next section.

Our examples are written in Java syntax, but we do not assume Java's lazy initialization semantics. As explained in the introduction, our technique is applicable to any language whose module initialization semantics satisfies conditions **C1–C3**. Therefore, we assume that a module initializer may start non-deterministically at any moment prior to the first access to a global variable or call to a static method of the module, and its execution may interleave with the threads of the program. This treatment of module initializers over-approximates both lazy and eager initialization semantics, thus providing general reasoning principles that are language agnostic.

Listing 5.1: Class `Fib` provides a memoized implementation of the Fibonacci sequence. `cache` is a global variable; the code inside the `static`-block is the module initializer.

```
1  class Fib {
2    private final static HashMap<Integer, Integer> cache;
3    static {
4      cache = new HashMap<>();
5      cache.put(0, 0);
6      cache.put(1, 1);
7    }
8    public static int fib(int n) {
9      boolean hasKey = cache.containsKey(n);
10     if (hasKey) {
11       return cache.get(n);
12     }
13     int fibN = fib(n-1) + fib(n-2);
14     cache.put(n, fibN);
15     return fibN;
16   }
17 }
```

5.2.1 Module invariants

The method `fib` in Listing 5.1 recursively computes the `n`-th element of the Fibonacci sequence, and uses the global variable `cache` to memoize all prior results.

Proving that this method computes the `n`-th Fibonacci number when `n` is non-negative requires that the following properties hold for each access to the global variable `cache`:

- **P1** `cache` holds a non-null value, otherwise a `NullPointerException` may be thrown;
- **P2** `cache` is not accessed concurrently by another thread, otherwise data races may occur;
- **P3** `cache` contains the keys `0` and `1`, otherwise the method may not terminate;
- **P4** `cache` maps all of its keys `k` to the `k`-th Fibonacci number, otherwise the method may produce wrong results.

More generally, we must show for each access to a global variable that the access is *safe*, *i.e.*, it does not lead to `null` pointer dereferences or race conditions (properties **P1** and **P2**), and that the values stored in global variables satisfy certain functional properties (**P3** and **P4**).

We use *module invariants* to make both kinds of properties explicit and formal. For our example in Java notation, module invariants can be specified at the level of classes; however, unlike traditional *class invariants* [179], they constrain global variables, not the fields of instances of the class.

We express module invariants in the assertion language of separation logic; thus, they provide information about the ownership of global variables and the values stored in them. A method may access a global variable if it owns it. We use *fractional permissions* [36] to allow concurrent reads, while ensuring exclusive writes. Ownership of a memory location `x` (which might be a global variable) is denoted with the *points-to* assertion $x \mapsto_p v$, where p is a rational number in the interval $(0, 1]$, and v is the value stored in location x (or _ if the value is irrelevant). A fractional amount p strictly less than 1 provides read-only access, whereas write

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

access requires full ownership ($p = 1$). As is standard, the ownership of a memory location may be split and re-combined:

$$x \mapsto_{p1+p2} v \equiv x \mapsto_{p1} v * x \mapsto_{p2} v$$

In our example, we express properties **P1**–**P4** with the following module invariant:

$$\begin{aligned} Inv_{\texttt{Fib}} \triangleq \exists m. \exists a. \, \texttt{cache} \mapsto_1 m \, * \\ m \neq null * is_map(m, a) * \\ \{0, 1\} \subseteq dom(a) * \\ (\forall k. \, k \in dom(a) \implies 0 \leq a(k) \wedge a(k) = \mathcal{F}(k)) \end{aligned}$$

The first conjunct provides exclusive access (full ownership) to `cache` and, thus, rules out data races (property **P2**). The second conjunct guarantees property **P1**. The third conjunct uses the separation logic predicate $is_map(m, a)$ to express full ownership of the hash map (property **P2**) and to abstract the state of the hash map to a partial function from keys to values, a; the concrete definition of is_map is irrelevant here. The remaining two conjuncts express property **P3** and **P4**, where $\mathcal{F}$ denotes the mathematical Fibonacci function.

Establishing module invariants

The module initializer $Initializer_M$ of a module M must establish M's module invariant, that is, it must satisfy the following triple:

$$\left\{ \text{Init}_M \; * \ast_{g \, \in \, \text{vars}(M)} g \mapsto_1 _ \right\} \; \texttt{Initializer}_M \; \{\text{Inv}_M\}$$

where Init_M indicates that module M has started initializing, as we will discuss in Subsection 5.2.2, $\ast$ is a syntactic shorthand for the separating conjunction over a set of locations, $\text{vars}(M)$ is the set of the identifiers of all global variables declared in module M, and Inv_M is M's module invariant. The precondition allows the module initializer to write to all global variables of the module, but does not directly permit accesses to global variables of other modules. Such accesses are in general not safe because they might introduce data races. We will see in Subsection 5.2.3 how we can allow such accesses if they are properly synchronized. Note that providing ownership to the module initializer cannot duplicate permissions because, according to condition **C3**, an initializer runs at most once.

Verifying this triple for the module initializer of class `Fib` is straightforward, provided we have appropriate specifications for the constructor and the `put` method of class `HashMap`.

Note that while termination of module initializers is generally desired, it is not required for the soundness of our technique. According to condition **C1**, a non-terminating initializer for a module M causes all thread executions to block at the next definitely-initialized execution points for M, such that any code that might use M's invariant is unreachable.

Using module invariants

The module invariant of a module M may be soundly used at any program point where the two following conditions hold:

IC1 M's initializer has started and finished, that is, its module invariant has been established;
IC2 No other thread concurrently relies on or invalidates the module invariant of M.

We explain how to check condition **IC1** in Subsection 5.2.2.

To establish condition **IC2**, we introduce a separation logic resource $[\text{Token } M]_p$ for every module M, where p denotes fractional ownership. Conceptually, one can think of this resource as an isorecursive predicate [38] that abstracts over the module invariant; like other isorecursive separation logic predicates, it can be unfolded to obtain the module invariant (thereby multiplying ownership amounts with fraction p, as usual) and get access to M's global variables. However, unfolding the predicate is sound only if condition **IC1** holds; otherwise, one might use an invariant that has not (yet) been established. To emphasize this difference to standard separation logic predicates, we use a different terminology. We call $[\text{Token } M]_p$ the *module token* (or *token* for short) of M; exchanging the token for the module invariant is called *opening* the invariant and comes with proof obligations that ensure **IC1** (see Subsection 5.2.2). *Closing* an invariant is analogous to folding a predicate, without extra proof obligations.

[38]: Summers et al. (2013), *A Formal Semantics for Isorecursive and Equirecursive State Abstractions*

The program entry point (*i.e.*, the `main` method) gets the module token $[\text{Token } M]_1$ for every module M in the program[5]. The `main` method can transfer tokens like any other separation logic resource, for instance, pass them to other methods or threads, or include them in predicates and module invariants. The standard separation logic proof rules ensure that neither a token nor the corresponding module invariant can be duplicated or forged, which guarantees condition **IC2**.

5: The set of all modules is generally not known during modular verification. However, it can be soundly approximated by including all tokens that are mentioned in the specifications of methods called from the main module, or in its module invariant, predicates, lock invariants, etc. Tokens of other modules could not be used during verification of `main` anyway.

The only way for a method to obtain ownership of a global variable is via a module invariant. Hence, the module invariant of a module M typically contains ownership of all global variables of M. A method may then either obtain M's token (for instance, via its precondition) and open the invariant to obtain ownership of the global variables, or it may require ownership directly in its precondition, pushing the task of opening the invariant to its caller. Our example uses the latter option, as expressed in the following specification for method `fib`:

$$\{\text{Inv}_{\texttt{Fib}} \land 0 \leq \texttt{n}\}\ \texttt{fib(n)}\ \{v.\ \text{Inv}_{\texttt{Fib}} \land v = \mathcal{F}(\texttt{n})\}$$

Verification of the implementation from Listing 5.1 against this specification is straightforward. Any caller of `fib` must hold $\text{Inv}_{\texttt{Fib}}$, for instance, by obtaining the token $[\text{Token } \texttt{Fib}]_1$ from `main` and opening the invariant, or by acquiring a lock whose lock invariant provides $\text{Inv}_{\texttt{Fib}}$. By building on separation logic, our technique is completely agnostic to how resources are manipulated and how threads are synchronized.

5.2.2 Definite initialization

In the previous subsection, we saw that a module invariant may be opened only at points, where the corresponding module invariant has

been established (condition **IC1**). This is the case when the module initializer (1) has started to execute and (2) has terminated.

To check the first condition, we use a dedicated separation logic resource Init_M indicating that M's initialization has started. This resource is obtained when accessing one of M's global variables or calling one of M's static methods. According to condition **C1** from the introduction, module initialization must start there at the latest. This approach soundly models both lazy initialization as in Java and eager initialization as in Go.

Checking the second condition is more involved. We need to make sure that the execution point at which we attempt to open the invariant for module M cannot be reached (directly or transitively) from M's initializer (otherwise, the execution point would be *during* M's initialization). To check this property modularly, we introduce *initialization levels* (or *levels* for short), a novel specification construct, which will later also allow us to rule out cyclic initialization dependencies (Challenge 3 in the introduction).

We require programmers to annotate each module, module initializer, and method with an initialization level, *i.e.*, a value from a pre-selected partial order. We impose the following *level requirements*:

LC1 The level of the module initializer of a module M is *strictly less than* M's level.

LC2 A method or module initializer with level l may call only methods with a level of *at most* l.

Note that requirement **LC2** allows recursive implementations (including mutual recursion if all involved methods have the same level).

The level requirements imply that, in a module M with level l, the initializer can (directly or indirectly) call only methods with a strictly smaller level. Importantly, *all other methods* may safely assume that they are *not* executed while M's module initializer runs. Consequently, according to condition **C1**, if M's initializer has been started, it must also have terminated and, therefore, it is sound to open M's invariant. So in summary, our proof obligations ensure condition **IC1** by checking that (1) the Init_M resource has been obtained and (2) the level of the current method is above M's level[6]. Inside the module initializer of a module N, check (1) is sufficient to establish **IC1**. As we will see in the next section, Init_M cannot be transferred into a module initializer (e.g., via a lock). So the only way for N's initializer to obtain Init_M is to access a field or call a static method of M. Since we prove that cyclic initialization dependencies do not occur (see Subsection 5.2.4), condition **C1** guarantees that M's initializer has terminated before execution proceeds. When both **IC1** and **IC2** hold, we can soundly open the invariant.

6: Technically, it would be sufficient to check that the method's level is *not below* M's level. The stronger check here will be necessary to avoid cyclic initialization dependencies, as we will explain in Subsection 5.2.4.

To verify the `Fib` module (Listing 5.1), the assigned initialization levels must satisfy the following constraints: the level of `HashMap`'s constructor and `put` method must be at most that of `Fib`'s initializer; the levels of `HashMap`'s `containsKey`, `get`, and `put` must all be at most `fib`'s. These constraints are needed to allow the various calls (**LC2**).

The need to assign levels may seem laborious at first, but is not difficult in practice. First, many modules do not call any methods from their initializers, because they do not have global variables or their global variables are initialized to constants. All those modules can have an arbitrarily small level. For the other modules, one can choose the smallest level that is greater than the levels of all methods called by their initializer.

Listing 5.2: Class `Counter`, which implements a global, thread-safe counter.

```
1  class Counter {
2    private final static Lock cl;
3    private static int count;
4
5    static {
6      count = 0;
7      cl = new Lock();
8    }
9
10   public static int getAndInc() {
11     cl.lock();   // acquire lock invariant
12     int result = count++;
13     cl.unlock(); // release lock invariant
14     return result;
15   }
16 }
```

[180]: Leino (2023), *Program Proofs*

Second, levels for methods are related to the termination measures used by modular verifiers to prove termination. For instance, Dafny's termination measures [180] are lexicographically-ordered tuples that include the module name as first component, ordered by the import relation. The same approach works for our technique and ensures that it is always possible to call methods in imported modules, such as those of `HashMap` in our example.

5.2.3 Duplicable invariants

As explained in Subsection 5.2.1, module tokens are necessary to ensure condition **IC2**, that is, to prevent different threads from performing unsynchronized accesses to global variables, potentially leading to data races and situations where an invariant is used by one thread while it has been temporarily violated by another. However, requiring the module token to open an invariant is overly restrictive in two common cases: when an invariant constrains only immutable global variables (e.g., global constants) and when access to (mutable) global variables is synchronized by a concurrency primitive such as a lock. In the former case, concurrent read accesses do not cause data races and the invariant cannot be violated after it has been established. In the latter, synchronization prevents data races and confines temporary violations to synchronized critical sections.

The example in Listing 5.2 illustrates both cases. The global variable `cl` is declared `final`, meaning that it cannot be modified after the module initializer has terminated. The global variable `count` is mutable, but protected by the non-reentrant lock in `cl`.

To handle such cases, we complement the module invariants presented so far with *duplicable module invariants*. A separation logic assertion P is *duplicable* if it satisfies the equivalence $P \equiv P * P$. A duplicable invariant cannot provide full ownership (or even a fixed fraction) of a resource. Consequently, once the invariant has been established, it cannot be violated by writing to the global variables it mentions, nor can there be data races on these variables. Hence, condition **IC2** holds automatically and does not have to be enforced via module tokens. This allows us to use the following, weaker proof obligations for duplicable invariants:

(1) They must be established by the module initializer. (2) To ensure condition **IC1**, the duplicable invariant of a module M may be opened only if the current method or initializer owns Init_M and its level is at least M's level. (3) The level requirements **LC1** and **LC2** from Subsection 5.2.2 hold.

In our example, we associate the lock `cl` with the lock invariant lockInv, which grants full ownership to `count`:

$$\text{lockInv} \triangleq \texttt{count} \mapsto_1 _$$

As usual in concurrent separation logics, a predicate $\text{isLock}(v, I)$ expresses that lock v has the lock invariant I. Calls to `lock` require this predicate and provide the lock invariant, whereas calls to `unlock` do the reverse. The predicate $\text{isLock}(o, P)$ is duplicable, such that all threads may compete for the lock by calling `lock`.

We can then specify the following duplicable module invariant for `Counter`:

$$\text{Inv}_{\texttt{Counter}} \triangleq \exists o.\, \exists p.\, 0 < p * \texttt{cl} \mapsto_p o \; * \text{isLock}(o, \text{lockInv})$$

This allows us to discharge the proof obligations mentioned above: (1) The module initializer establishes the duplicable invariant; in particular, `cl`'s lock invariant holds. (2) Method `getAndInc` may open the duplicable module invariant because it obtains $\text{Init}_{\texttt{Counter}}$ when accessing `cl`. Moreover, we can assign levels such that `getAndInc`'s is at least `Counter`'s. (3) By assigning levels to all lock operations below those of `Counter`'s, all method and constructor calls are permitted.

Interestingly, these proofs do not require the specification of `getAndInc` to include any permissions to global variables nor the predicates our technique introduces (Init, tokens). The handling of `Counter`'s global state is hidden from clients and does not complicate their verification (except assigning suitable levels). In particular, this example demonstrates that duplicable invariants allow the module initializer of a module N to get read and write access to the global variables of another module M by using suitable synchronization. This synchronization is necessary to prevent data races. For immutable global variables of M, N's module initializer may directly open M's duplicable invariant (without synchronizing), as long as the level assignment allows that.

5.2.4 Cyclic initialization dependencies

The technique introduced so far is sound unless the program contains *cylic initialization dependencies*, that is, two or more modules where each module's initializers accesses a global field or static method of the next module on the cycle. The following example illustrates the problem.

In an execution where `A` is initialized first, the call to `B.foo` will start initialization of `B`, such that the assertion in `B`'s initializer is reached *before* `f` has been initialized and `A`'s invariant has been established. Nevertheless, this program verifies using the technique introduced so far! In particular, there is a level assignment that allows the call to `B.foo` and opening `A`'s invariant to prove the assertion in `B`'s initializer, just pick `B.foo` $<$ `A` $<$ `B` in the level ordering. This problem is not surprising, given that our technique so far builds on condition **C1** from the introduction, which fully applies only in programs without cyclic initialization dependencies.

Listing 5.3: Classes `A` and `B` form a cyclic initialization dependency.

```
1  class A {
2    // dup invariant: ∃p. 0 < p ∧ f ↦_p 1
3    static int f;
4    static {
5      B.foo();
6      f = 1;
7    }
8  }
9
10 class B {
11   static {
12     // open A's invariant
13     assert A.f == 1;
14   }
15
16   static void foo() {}
17 }
```

Listing 5.4: Client program for the `Fib` module from Listing 5.1.

```
1 class Main {
2   void main(String args[]) {
3     int x = Fib.fib(3);
4     assert x == 2;
5   }
6 }
```

Our level requirements **LC1** and **LC2** ensure that any *method* that executes during the initialization of a module M has a level strictly smaller than M and, thus, we can conclude when a method is *not* part of the initialization and may soundly open M's invariant. However, these requirements do not provide the same guarantee for *module initializers*. In the above example, `B`'s initializer runs during `A`'s initialization, but the level requirements do not prevent it from opening `A`'s invariant.

Intuitively, one can consider accesses to global variables and calls to static methods of a module M as "calling" M's initializer in case the initializer has not been started before. Therefore, we introduce an additional level requirement, which is the analog of **LC2** for these conceptual calls:

LC3 A method or module initializer with level l may access a global field or call a static method of a module M only if (1) it owns Init_M or (2) M's level is *at most* l.

Together, our three level requirements rule out cyclic initialization dependencies, which makes our technique sound as we will prove later. In particular, the above example now correctly fails to verify because **LC3** introduces an additional constraint for the call to `B.foo`, $\texttt{B} < \texttt{A}$, leading to an unsatisfiable set of level constraints.

5.2.5 A complete example

Having presented the key concepts of our approach, we can now complete our example by adding a client for the `Fib` module from Listing 5.1: In order to verify this client program, we need to establish the following proof obligations:

1. The level requirements **LC1**, **LC2**, and **LC3** hold for the entire program.
2. The precondition of `fib` holds before the call on line 3.
3. The conditions **IC1** and **IC2** hold at every program point where the invariant is opened.
4. The assertion in line 4 holds.

(1) The level requirement **LC1** trivially holds for class `Main` given that it does not have a module initializer. We can satisfy requirement **LC2** by choosing the level L of the method `main` to be higher than that of method `fib`; similarly, we can satisfy requirement **LC3** by choosing L such that it is also higher than that of the `Fib` module.

(2) Recall that the precondition of `fib` is that the invariant of `Fib` holds when the call is performed. We can thus satisfy this precondition by opening the invariant immediately before the call in line 3.

(3) Opening the invariant of `Fib` requires proving that the conditions **IC1** and **IC2** hold right before the call. As explained in Subsection 5.2.2, the condition **IC1** requires that initialization of the `Fib` module has started and finished. The former can be established from the call to `Fib.fib` itself, which yields the resource $\mathrm{Init}_{\texttt{Fib}}$ to indicate that initialization has started. The latter follows from the fact that the `main` method's level is above `Fib`'s. Finally, condition **IC2** follows from the fact that method `main` holds the module token of `Fib` ($[\text{Token } \mathit{Fib}]_1$) at the time of the call, since, as explained in Subsection 5.2.1, the `main` method initially owns the module token $[\text{Token } M]_1$ for every module M in the program.

(4) After the call to `fib`, the assertion in line 4 follows directly from the method's postcondition.

5.3 Formalization

We formalize our technique as a program logic for an example concurrent language with global state using Iris [84], a separation logic framework. We establish its soundness using our key ideas, proved using the Iris proof mode [181, 182] in the Rocq proof assistant [183]. Our soundness result establishes the correctness of our treatment of invariants, and the absence of initialization cycles.

Our soundness result is shown for an ML-like language with global modules (Subsection 5.3.1). It is given a semantics by translation to HeapLang, another ML-like language with existing Iris support but no module system (Subsection 5.3.2). We model initializers using concurrent preemptive HeapLang threads which can execute at any time, overapproximating the different initialization semantics of real-world languages. We formally define the assertion language for global state (Subsection 5.3.3) and give its embedding into Iris' separation logic (Subsection 5.3.4).

Initialization cycle freedom in particular is shown by instrumenting the semantics of our ML-like language with a notion of runtime level, and asserting that all module accesses are strictly monotonic according to this level (Subsection 5.3.5). These assertions are then shown to always succeed for verified programs, so they can be elided from the semantics.

[84]: Jung et al. (2015), *Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning*

[181]: Krebbers et al. (2017), *Interactive proofs in higher-order concurrent separation logic*

[182]: Krebbers et al. (2018), *MoSeL: a general, extensible modal framework for interactive proofs in separation logic*

[183]: The Rocq Development Team (2025), *The Rocq Prover Reference Manual – Release 9.0.0*

$$
\begin{aligned}
v \in Val &::= () \mid n \in \mathbb{Z} \mid b \in \{\mathsf{True}, \mathsf{False}\} \mid \ell \in Loc \mid \ldots \\
x \in Var &::= \ldots \\
f \in Field &::= \ldots \\
m \in Method &::= \ldots \\
M \in Id &::= \ldots \\
e \in Expr &::= v \mid x \mid \lambda x.\, e \mid e\ e \mid \texttt{let}\ x := e\ \texttt{in}\ e \\
&\quad \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{while}\ e\ \texttt{do}\ e \\
&\quad \mid \texttt{ref}\ e \mid !e \mid e \leftarrow e \mid !M.f \mid M.f \leftarrow e \\
&\quad \mid \texttt{call}\ M.m \mid \texttt{fork}\ e \mid \ldots
\end{aligned}
$$

$$
\phi \in Modules \triangleq Id \rightharpoonup \left\{ \begin{array}{r} \textsc{Fields} : \overline{Field} \\ \textsc{MethodDefns} : Method \rightharpoonup Expr \\ \textsc{Init} : Expr \end{array} \right\}
$$

$$
e_\phi \in Program \triangleq Expr \times Modules
$$

Figure 5.1: The syntax of our source language. The highlighted terms are the significant novelties w.r.t. HeapLang.

5.3.1 Syntax

In this subsection, we provide the syntax for our core language, which we use as a minimal example for our soundness results. This syntax is presented in Figure 5.1. Effectively, our language is defined by augmenting Iris' HeapLang with modules and global initialization.

Expr. The three significant additions to the expression language with respect to HeapLang are: loads from module fields $!M.f$; stores to module fields $M.f \leftarrow e$; and calls to module methods $\texttt{call}\ M.m$. Module methods can take arguments by being declared as lambdas.

Module. A module consists of a set of field and method declarations, together with the module initializer expression. Method declarations consist of a module-unique method name, along with an expression body. Field declarations are simply a module-unique field name. Field initialization can be performed in the module initializer.

Program. A program e_ϕ consists of an expression e, together with a set of accompanying named modules ϕ. The expression e is the program's entry point, i.e., it plays the role of the main function of the program. The modules in the context can refer to each other's fields in their initializers and methods. These references can potentially be cyclic.

5.3.2 Translational semantics

The semantics of our language is given by an embedding into HeapLang (shown in Figures 5.2 to 5.4), with its usual program state definition. This is done to simplify the formalization overhead: instantiating Iris with an entirely new language requires significant proof and mechanization effort. By instead using a translational semantics, we are able to re-use the existing work for HeapLang.

Since our language is effectively an extension to HeapLang, most expressions are translated directly in the expected style (*i.e.* variables to variables, lambdas to lambdas, and so on), as shown in Figure 5.2. The only exceptions are those language features not present in HeapLang - namely, global fields, static methods, and module initializers. Giving

Translation of expressions. $\boxed{[\![e]\!]^{\omega}_{\tau}}$

$$
\begin{array}{rlrl}
[\![()]\!]^{\omega}_{\tau} &\triangleq () & [\![n]\!]^{\omega}_{\tau} &\triangleq n \\
[\![b]\!]^{\omega}_{\tau} &\triangleq b & [\![\ell]\!]^{\omega}_{\tau} &\triangleq \ell \\
[\![x]\!]^{\omega}_{\tau} &\triangleq x & [\![\lambda x.\ e]\!]^{\omega}_{\tau} &\triangleq \lambda x.\ [\![e]\!]^{\omega}_{\tau} \\
[\![e\ e']\!]^{\omega}_{\tau} &\triangleq [\![e]\!]^{\omega}_{\tau}\ [\![e']\!]^{\omega}_{\tau} & [\![\texttt{ref}\ e]\!]^{\omega}_{\tau} &\triangleq \texttt{ref}\ [\![e]\!]^{\omega}_{\tau} \\
[\![!e]\!]^{\omega}_{\tau} &\triangleq !\,[\![e]\!]^{\omega}_{\tau} & [\![e \leftarrow e']\!]^{\omega}_{\tau} &\triangleq [\![e]\!]^{\omega}_{\tau} \leftarrow [\![e']\!]^{\omega}_{\tau} \\
& \vdots & &
\end{array}
$$

Figure 5.2: The translational semantics of basic terms in our language. These terms don't interact with our translation structures ω, τ. We omit the syntax of HeapLang terms for brevity, but the languages are very similar, which is why most terms translate in a direct way [184].

$$\omega \in \mathit{LevelAssignment} \triangleq \mathit{Id} \rightharpoonup L \quad (L \text{ is a partially-ordered set})$$

$$\tau \in \mathit{TranslationCtx} \triangleq \left\{ \begin{array}{rl} \textsc{LockVars} : \mathit{Id} & \rightharpoonup \mathit{Var} \\ \textsc{FieldVars} : \mathit{Id} \times \mathit{Field} & \rightharpoonup \mathit{Var} \\ \textsc{MethodVars} : \mathit{Id} \times \mathit{Method} & \rightharpoonup \mathit{Var} \\ \textsc{ThreadIdVar} : \mathit{Var} & \\ \textsc{LevelVar} : \mathit{Var} & \end{array} \right\}$$

Figure 5.3: Structures used for the translational semantics. ThreadIdVar stores the identifier of the module the current thread is initializing, if any. LevelVar stores the runtime level of the current thread.

these a semantics requires additional book-keeping structures, shown in Figure 5.3. Global fields are translated to HeapLang variables of heap-allocated values, which can be mutated. The mapping of global field to HeapLang variable is recorded in the FieldVars of the translation context τ. Similarly, static methods also become HeapLang variables of heap-allocated stored expressions (i.e. the method body), which can be loaded and executed. These are also recorded in τ as the MethodVars.

We can overapproximate both eager and lazy initialization approaches by encoding whole programs using HeapLang's support for concurrency. In this encoding, the program expression is executed in one thread, and each module initializer is executed in a separate thread forked from the main one. A single thread per initializer suffices because initializers execute at most once (**C3**). All threads share access to the translation variables which model the fields and methods of modules, and HeapLang's concurrent semantics allow each thread to take steps at any point.

Module access synchronization

Our model restricts executions to those satisfying **C1** (where initialization starts, and potentially finishes, before the first definitely-initialized execution point). We do this by inserting synchronization code in our translational semantics, leveraging mutual exclusion (mutex) locks already provided in HeapLang. When creating a thread for each module initializer, we allocate and acquire a mutex for that thread until the initializer has completed execution (recording it in LockVars.) Once the initializer thread has completed, we release the mutex.

Since the mutex is taken while the initializer is executing, it then suffices to make a module's field accesses (both reads and writes) and method calls wait on that module's mutex, before performing the corresponding operation on the appropriate HeapLang variable (Figure 5.4). This prevents field and method use until the initializer has finished execution. By **C2**, this is all we can do - once the mutex is released by the initializer, no

Translation of module accesses. $\boxed{[\![e]\!]^{\omega}_{\tau}}$

$$
\begin{aligned}
\textsc{Sync}(M)^{\omega}_{\tau} \triangleq\ & \texttt{if}\ \tau.\textsc{ThreadIdVar} = M \\
& \texttt{then skip} \\
& \texttt{else if}\ \omega(M) < \tau.\textsc{LevelVar} \\
& \texttt{then} \\
& \quad \texttt{lock}\ \tau.\textsc{LockVars}(M); \\
& \quad \texttt{unlock}\ \tau.\textsc{LockVars}(M) \\
& \texttt{else fail}
\end{aligned}
$$

$$
\begin{aligned}
[\![!M.f]\!]^{\omega}_{\tau} \triangleq\ & \textsc{Sync}(M)^{\omega}_{\tau}; \\
& !(\tau.\textsc{FieldVars}(M, f))
\end{aligned}
$$

$$
\begin{aligned}
[\![M.f \leftarrow e]\!]^{\omega}_{\tau} \triangleq\ & \texttt{let}\ tmp = [\![e]\!]_{\tau}\ \texttt{in} \\
& \textsc{Sync}(M)^{\omega}_{\tau}; \\
& \tau.\textsc{FieldVars}(M, f) \leftarrow tmp
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{call}\ M.m]\!]^{\omega}_{\tau} \triangleq\ & \textsc{Sync}(M)^{\omega}_{\tau}; \\
& !(\tau.\textsc{MethodVars}(M, m))\ \tau.\textsc{ThreadIdVar}\ \tau.\textsc{LevelVar}
\end{aligned}
$$

Figure 5.4: The translational semantics of global state in our language. Sync is a meta-function for code generation that enforces the correct semantics for module accesses - access from the initializer is always allowed, and definitely-initialized execution points both assert monotonicity of the runtime level and block until initialization succeeds.

further synchronization occurs. We note that the mutex doesn't restrict *simultaneous* field accesses, since the mutex does not guard the access - it only ensures that the thread for the corresponding initializer has finished executing.

The only exception to this mutex-taking is in the initializer thread of a module itself, since **C2** forbids accesses to a module from inside that module's initializer to block. Therefore, our translation has access to a *thread identifier* variable, which marks the thread as either the unique initializer thread for a particular module, or a non-initializer thread. When performing an access, the module name is compared against the thread identifier at runtime. If there is a match, the mutex is ignored, which allows module initializers to access their own module fields and methods without blocking. This logic is implemented in the Sync code generation function shown in Fig. 5.4.

Acyclic module accesses

Finally, for module accesses we also assert that the access is *acyclic*. This is done by assigning threads a *runtime level*, which can be inspected by the semantics[7]. We then make each initializer's runtime level the initialization level of the module it initializes. When a module access occurs, the current thread's runtime level is asserted to be strictly above the initialization level (and thus the runtime level) of the module being blocked on (Figure 5.4). For non-blocking accesses, no such check is necessary.

7: This comparison must match the partial order for used for initialization levels. We therefore require that partial order to be representable and comparable at runtime. We believe this is not onerous.

5.3.3 Program logic

We formally describe our reasoning principles for global state as a Hoare logic over our core language. Our program logic is defined using a

$$P, Q \in Asst ::=$$

$$\top \mid \bot \mid \ell \mapsto_p v \mid M.f \mapsto_p v \mid [\text{Token } M]_p \mid \text{Init}_M \mid \bigcirc_M P \mid P * Q \mid \ldots$$

Figure 5.5: Assertions in our separation logic.

separation logic assertion language. We also define how to specify whole programs, including those with module dependencies.

Assertions

Our assertion language is shown in Figure 5.5. As well as the typical constructor of a (fractional) heap points-to ($x \mapsto_p v$) and logical operators, we also define the specialized assertions:

- $M.f \mapsto_p v$, the (fractional) points-to for the field f of the module M.
- $[\text{Token } M]_p$, the module token of M.
- $\bigcirc_M P$, an assertion under the *initialization-guarded modality* $\bigcirc_M$. The guarded assertion P might not hold in the current heap, but *will* hold once initialization has completed for M. In particular, $\bigcirc_M \bot$ would require that the module initializer for M does not terminate.
- Init_M, the fact that M's initialization has started. This assertion holds no resources and is duplicable. It is used to eliminate the initialization-guarded modality $\bigcirc_M$ (see GuardElim).

The modality $\bigcirc_M$ has two roles: it allows us to soundly reason about accessing module invariants which may not yet have been established (because the corresponding module initializer has not run), and it allows us to fearlessly access a module invariant after initialization is known to have completed. It is often useful to be able to "unfold" the module token of a module M into a copy of its module invariant even before Init_M is known. Once Init_M is known, we can then open the invariant of M and soundly eliminate the modality at *any time* (even outside of definitely-initialized execution points).

Hoare logic

The triple $\{P\}\ e\ |^{l}_{\Phi}\ \{v.\ Q\}$ for an expression e is parameterized by an initialization level l[8] and set of module specifications Φ. The rules for these triples are given in Figure 5.6. All triples derivable for a set of module specifications are also derivable for any superset of those specifications. As a consequence, our logic is *modular* in terms of program modules.

8: We implicitly fix the partial order L used for initialization levels.

A program triple $\{P\}\ e_\phi\ \{v.\ Q\}$ is a specification for the whole program e_ϕ. Its definition is shown in Figure 5.7. A program is specified if there is a set of module specifications which allows for acyclically specifying the module initializers and methods of all program modules. Each module initializer is required to establish its duplicable and non-duplicable module invariant. Furthermore, all methods are required to establish their specifications, though they may do so at any level.

Level requirements

Perhaps surprisingly, accessing fields of a module M for reading or writing does not directly impose any restrictions on the initialization level. Instead, the user is required to prove that initialization has already started (or the current thread is the initializer) using Init_M. The initializer of M starts with access to this fact, and can thus justify all of its own field accesses. In order to prove that initialization has started, and thus obtain Init_M, outside the initializer of M, a level check *is* required (as in InitializedI1). This can be proved immediately before a definitely-initialized execution point for M. In the case of InitializedI2, this requires that we can only learn Init_M once the right-hand side is a value, since initialization is triggered only at this point.

Methods are allowed to have levels below that of their module, to allow for methods which are called from the module initializer. However, calling these methods from outside the module initializer *would still* trigger initialization, so they can only be safely called at the module level or higher. To call a method at a level below the module level, one must know that initialization will not trigger, which is implied by the Init_M requirement in GlobMethodCall. Similarly to fields, in order to use method calls to justify that initialization will be triggered, the initialization level must be at the module level or higher (InitializedI3).

5.3.4 Soundness

We show our program logic to be sound in only allowing access to established invariants, and preventing non-strictly-monotonic initializer blocking. Both of these would result in runtime errors in HeapLang under our translational semantics. Therefore, it suffices to show the absence of errors in HeapLang terms generated from source expressions specified by our program logic.

We do this by using the HeapLang weakest-precondition definition generated by Iris for HeapLang terms; these weakest preconditions are automatically sound by construction. Program triples in our program logic are encoded to weakest-precondition statements on the translated output term, which are then proven. As a result, our program logic ensures source terms are safe to execute under the translational semantics.

5.3.5 Freedom of initialization cycles

We assert that blocking module accesses are strictly monotonic in terms of the runtime levels involved - so that each initializer can block only on a thread of a strictly lower level. From the soundness of our logic, we know these assertions always succeed for verified programs. By **C2**, we also know that blocking due to initialization only occurs after these assertions. Therefore, at any point during execution, we can strictly order all runtime threads according to which other thread they are blocked on, if any. It is a well-known result that this implies that the directed graph of runtime threads (where edges indicate blocking due to module accesses) is always acyclic [185]. Therefore, our program logic prevents initialization cycles. We do not mechanize this proof beyond showing that accesses are monotonic.

[185]: Meyer et al. (2015), *Partial Orders*

Moreover, since we prove the absence of failed monotonicity assertions in any verified program, we know that our soundness result does not rely on the behavior of the model in the case of an initialization cycle. As such, our result applies to languages with *any* initialization cycle behavior, even if it does not result in a program error. The runtime levels themselves can even be elided from the semantics, since their only purpose is to check the assertions, which always succeed.

Hoare triples of assertions. $\boxed{\{P\}\ e\ |^{l}_{\Phi}\ \{v.\ Q\}}$

$$\textsc{OpenMInv}\ \frac{0 < p \qquad \left\{P * \bigcirc_M \Phi(M).\text{MInv}(p)\right\}\ e\ |^{l}_{\Phi}\ \left\{v.\ Q * \Phi(M).\text{MInv}(p)\right\}}{\left\{P * [\text{Token}\ M]_p\right\}\ e\ |^{l}_{\Phi}\ \left\{v.\ Q * [\text{Token}\ M]_p\right\}}\ \Phi(M).\textsc{Level} \le l$$

$$\textsc{OpenDupMInv}\ \frac{\{P * \bigcirc_M \Phi(M).\text{DupMInv}\}\ e\ |^{l}_{\Phi}\ \{v.\ Q\}}{\{P\}\ e\ |^{l}_{\Phi}\ \{v.\ Q\}}\ \Phi(M).\textsc{Level} \le l$$

$$\textsc{GuardElim}\ \frac{\{P\}\ e\ |^{l}_{\Phi}\ \{v.\ Q\}}{\{\bigcirc_M P * \text{Init}_M\}\ e\ |^{l}_{\Phi}\ \{v.\ Q\}} \qquad \textsc{LevelDrop}\ \frac{\{P\}\ e\ |^{l'}_{\Phi}\ \{v.\ Q\}}{\{P\}\ e\ |^{l}_{\Phi}\ \{v.\ Q\}}\ l' \le l$$

$$\textsc{GlobFieldRead}\ \frac{0 < p}{\left\{M.f \mapsto_p v * \text{Init}_M\right\}\ !M.f\ |^{l}_{\Phi}\ \left\{v'.\ M.f \mapsto_p v * v = v'\right\}}$$

$$\textsc{GlobFieldStore}\ \frac{}{\left\{M.f \mapsto _ * \text{Init}_M\right\}\ M.f \leftarrow v\ |^{l}_{\Phi}\ \left\{v'.\ v' = () * M.f \mapsto v\right\}}$$

$$\textsc{GlobMethodCall}\ \frac{}{\begin{array}{l}\{\Phi(M).\textsc{Methods}(m).\textsc{Pre}(v) * \text{Init}_M\} \\ \quad \texttt{call}\ M.m\ v\ |^{l}_{\Phi} \\ \{v'.\ \Phi(M).\textsc{Methods}(m).\textsc{Post}(v, v')\}\end{array}}\ \Phi(M).\textsc{Methods}(m).\textsc{MLevel} \le l$$

$$\textsc{InitializedI1}\ \frac{\{P * \text{Init}_M\}\ !M.f\ |^{l}_{\Phi}\ \{v.\ Q\}}{\{P\}\ !M.f\ |^{l}_{\Phi}\ \{v.\ Q * \text{Init}_M\}}\ \Phi(M).\textsc{Level} \le l$$

$$\textsc{InitializedI2}\ \frac{\{P * \text{Init}_M\}\ M.f \leftarrow v\ |^{l}_{\Phi}\ \{Q\}}{\{P\}\ M.f \leftarrow v\ |^{l}_{\Phi}\ \{Q * \text{Init}_M\}}\ \Phi(M).\textsc{Level} \le l$$

$$\textsc{InitializedI3}\ \frac{\{P * \text{Init}_M\}\ \texttt{call}\ M.m\ v\ |^{l}_{\Phi}\ \{Q\}}{\{P\}\ \texttt{call}\ M.m\ v\ |^{l}_{\Phi}\ \{Q * \text{Init}_M\}}\ \Phi(M).\textsc{Level} \le l$$

Figure 5.6: Selected Hoare logic rules for the core language.

$$\Phi \in \mathit{ModuleSpecs} \triangleq \mathit{Id} \rightharpoonup$$

$$\left\{ \begin{array}{l} \textsc{Level} : L \\ \textsc{MInv} : \mathbb{Q} \to \mathit{Asst} \\ \textsc{DupMInv} : \mathit{Asst} \\ \textsc{Methods} : \mathit{Method} \rightharpoonup \left\{ \begin{array}{l} \textsc{MLevel} : L \\ \textsc{Pre} : \mathit{Val} \to \mathit{Asst} \\ \textsc{Post} : \mathit{Val} \times \mathit{Val} \to \mathit{Asst} \end{array} \right\} \end{array} \right\}$$

$$\{P\}\ e_\phi\ \{v.\ Q\} \triangleq$$

$$\left\{ \begin{array}{rl} \textsc{MSpecs} : & \mathit{ModuleSpecs} \\ \textsc{MainSpec} : & \exists l.\ \{P\}\ e\ |^{l}_{\textsc{MSpecs}}\ \{v.\ Q\} \\ \textsc{InitSpecs} : & \forall M \in \mathsf{dom}(\phi).\ \exists l' < \textsc{MSpecs}(M).\textsc{Level}. \\ & \left\{ \ast_{f \in \phi(M).\textsc{Fields}} M.f \mapsto () * \mathrm{Init}_M \right\} \\ & \quad \phi(M).\textsc{Init}\ |^{l'}_{\textsc{MSpecs}} \\ & \{\textsc{MSpecs}(M).\textsc{MInv}(1) * \textsc{MSpecs}(M).\textsc{DupMInv}\} \\ \textsc{MethodSpecs} : & \forall M \in \mathsf{dom}(\phi).\ \forall m \in \mathsf{dom}(\phi(M).\textsc{MethodDefns}). \\ & \forall v \in \mathit{Val}. \\ & \{\textsc{MSpecs}(M).\textsc{Methods}(m).\textsc{Pre}(v) * \mathrm{Init}_M\} \\ & \quad \phi(M).\textsc{MethodDefns}(m)\ v\ |^{\textsc{MSpecs}(M).\textsc{Methods}(m).\textsc{MLevel}}_{\textsc{MSpecs}} \\ & \{v'.\ \textsc{MSpecs}(M).\textsc{Methods}(m).\textsc{Post}(v, v')\} \end{array} \right\}$$

Figure 5.7: Program specifications in our core language.

5.4 Implementation

Our technique is applicable to a large set of real languages that support global state, as we show in this section on two complex, real-world languages with very different initialization systems, Java and Go. We also demonstrate that our technique can be integrated into existing automated deductive verification tools, VerCors [121], a verifier for Java, and Gobra [29] for Go.

[121]: Armborst et al. (2024), *The VerCors Verifier: A Progress Report*

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

Applying our technique to a real-world language requires soundly modeling all constructs in the language that are relevant for initialization and not present in our model. Additionally, if a language provides stronger guarantees (e.g., about the time when initializers execute) than we assume in our logic, we may incorporate these additional guarantees to make our technique more expressive and easier to apply.

5.4.1 Application to Java

First, we show how our technique can be applied to a language like Java, which performs initialization of static fields lazily on a class level. With very minor differences in some details, everything here also applies to initialization in C# and some other JVM-based languages like Scala.

Additional language features and guarantees

Java offers several important language features that do not exist in the model described so far, but can be easily incorporated:

First, as stated previously, Java's constructors can be modeled as static methods of the same class, reflecting the fact that invoking a constructor will trigger initialization of the new object's class.

Second, Java supports subclassing in a way that is relevant for initialization. In particular, its runtime system guarantees that any class' superclasses are initialized before its own initialization starts. As a result, it is possible to have indirect cyclic initialization dependencies, if for example the initializer of class `A` reads a static field of class `Sub`, and `Sub`'s superclass `Super`'s initializer reads a field of class `A`. Since the read in `A` triggers initialization of `Super` indirectly by triggering the initialization of `Sub`, there is a danger of it causing a deadlock or reading uninitialized data in a similar way as with direct cyclic initialization dependencies. While not necessary for the soundness of our technique, we can easily prevent these indirect cyclic dependencies by enforcing an additional level constraint: *the initialization level of a subclass must be at least that of its superclass*. As a result, subclass initializers can potentially use (duplicable) invariants of their superclass but not vice versa, as one would expect.

Third, for method overrides in subclasses, as part of ensuring behavioral subtyping [39], we have to ensure that overriding methods open at most the invariants known to hold when the original overridden method is called. This constraint can again be enforced by requiring overridden method's levels to be at least that of their overrides.

[39]: Liskov et al. (1994), *A Behavioral Notion of Subtyping*

Java's initialization system offers some guarantees that go beyond the minimal conditions **C1-C3**. In particular, Java guarantees that initialization of a class is performed lazily and will start only when encountering the static field access or method invocation of said class, or before the

similarly-triggered initialization of a subclass. Thus, Java would in principle allow identifying the exact point at which an initializer will execute, and therefore, e.g., concluding more precisely what the resulting state of the module is. However, such precise reasoning is inherently non-modular, since it requires knowing what other code has or has not been executed previously.

Sequential Java

While Java is a concurrent language, many existing Java programs are sequential. Many patterns of accessing global state that are not modularly correct in a concurrent setting are unproblematic in a sequential one. For example, in a concurrent setting, when opening some module A's non-duplicable invariant, we have to make sure that this invariant is not currently being used or broken by other threads (**IC2**). We enforce this condition by requiring the token [Token A]. As a result, initializers generally cannot open non-duplicable invariants of other modules unless they obtain the required token via a lock or other synchronization primitive.

However, in sequential Java, no such concurrent accesses are possible and condition **IC2** holds trivially: There is only one thread, and initializers are triggered at clearly defined points in the execution and running in the same thread. Thus, in this setting, assuming all other conditions are satisfied, it is always sound (including from initializers) to open any module invariants that are not already opened *by the current thread*.

We can adapt our system to this setting as follows: We no longer use tokens to ensure condition **IC2**, and instead rely on initialization levels to track which invariants may be opened. To do this, we observe that since class initialization in Java is triggered only upon the first use of a class, which in turn is required by our logic to only happen in a context whose initialization level is higher than that of the class, throughout the entire call stack of the single thread (including implicitly-triggered static initializers), the initialization level only ever decreases. We can now enforce a simple additional condition, namely that code that executes while the non-duplicable invariant of some class with level l is opened must be correct (according to the Hoare triple definition from Sec. 5.3.3) with a level *strictly less than* l, i.e., said code may only call methods with a level less than l, open invariants of other modules with a level less than l, and so on. Effectively, this condition is equivalent to using LevelDrop to temporarily reduce the current level whenever an invariant is opened. We also enforce that initializers close all non-duplicable invariants they have opened before they terminate. This condition suffices to maintain the invariant that *if the current level is greater or equal that of any class A, then the invariant of A holds and can be opened*.

As a result, static initializers can directly open the static invariants of classes with lower levels, which means that initializers may access static fields of (some) other classes without requiring any additional synchronization, as one would expect in a setting where no other threads can interfere. Note that this additional freedom does not require changing our core technique and instead directly results from enforcing an additional condition and incorporating the additional guarantees provided by the language (Java) and the setting (the absence of multithreading).

Implementation in VerCors

[121]: Armborst et al. (2024), *The VerCors Verifier: A Progress Report*

We have implemented our technique for Java in VerCors [121], an automated, SMT-based verifier for a number of concurrent languages including Java. Our implementation supports advanced Java features like subclassing as described in Subsection 5.4.1; additionally, we have implemented a mode (enabled via a command line flag) that further simplifies reasoning for sequential code, as described in Subsection 5.4.1.

We were able to easily integrate our technique into VerCors, adding the required specification constructs and a single encoding pass that translates specifications and adds all relevent proof obligations. In total, we added or modified a total of ca. 1000 lines of code in VerCors.

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

We introduced ghost [186] statements that allow users to explicitly open and close module invariants, or open duplicable invariants. Our implementation is able to automatically eliminate the initialization-guarded modality whenever initialization of the relevant class has provably finished.

We require users to annotate classes with duplicable and non-duplicable invariant assertions (with the obvious defaults), and similarly allows annotating methods, constructors, and classes with initialization levels (in the form of natural number literals) as described in Section 5.2. We choose default values for initialization levels to be 1 for classes and methods, and $l - 1$ for the initializer of a class with level l. This choice ensures that in code that does not perform or interact with module initialization or static fields, all level constraints are always satisfied, meaning that no additional annotation overhead is imposed for code that does not require it.

5.4.2 Application to Go

Initialization in Go works fundamentally differently from Java; Go offers additional guarantees not present in Java, but also poses a new challenge. Nevertheless, our technique applies.

Additional language features and guarantees

First, unlike Java's lazy initialization, Go performs initialization eagerly. As a result, one may soundly assume that all *packages* (Go's notion of modules) have initialized at the beginning of the `main` function.

Second, the Go compiler guarantees that the package import relation forms a DAG. Thus, cyclic initialization dependencies between packages are ruled out by the compiler. We can automatically and modularly infer an assignment of initialization levels per package from the import relation: if package A imports B, the level of B must be lower than A's level. In addition, *by default*, all methods in a package may adopt the package level. With these conditions, level annotations are necessary only in methods that may be called from the package initializer, which greatly reduces the number of required annotations.

Third, since all packages are guaranteed to be initialized when `main` starts, there is no longer a need to open and close non-duplicable invariants; instead, conceptually, all non-duplicable invariants can be opened at the beginning of `main` and remain open, which also fully eliminates the need for module tokens. Finally, package initialization in Go is guaranteed

to run sequentially, package by package, in reverse order of imports. Thus, similarly to sequential programs in Java (Subsection 5.4.1), package initializers in Go may open the invariants of imported packages, as long as they close them by the end of the initializer. As a result, when applying our logic to Go, we end up with significantly simpler reasoning just by incorporating language guarantees into the existing technique.

Finally, Go supports a form of structural subtyping, where values of a type T may be assigned to variables of an *interface type* I if T has at least all methods declared in I. Unlike Java, subtyping plays no role in the order in which modules are initialized. We adopted a simple design for dealing with level constraints in Gobra's interfaces: we eschew any level annotations on interface methods, and we consider that the level of an interface method (i.e., the level of a method whose receiver type is the interface type, rather than a concrete implementation) is the level of the module in which the interface is declared. While doing so is not sound in general, we identified and implemented a simple syntactic restriction that enforces the soundness of this solution.

On the other hand, Go also poses a new challenge by allowing multiple initializers in the same package, spread across different files of the package. This leads to the infamous *Static Initialization Order Fiasco* [187], originally identified in C++, where the order in which global variables are initialized is determined by the order in which files are passed to the compiler. This may cause unexpected problems as programs evolve. Files `file2.go` (Listing 5.5) and `file3.go` (Listing 5.6) provide a contrived, yet demonstrative example. Together, these files constitute the package `main`, and each of these files has an initializer (function `init`). If we follow the Go team's recommendation of passing files in the lexical name order to the compiler [169], the initializer in `file2.go` runs before the one in `file3.go` and the program runs successfully to completion. If we rename `file3.go` to `file1.go` and pass the files in lexicographic order, the two initializers run in the reversed order, and a `panic` is caused in line 4 of `file1.go`, as we try to put a key/value pair in a `nil` map.

[187]: (2023), *Static Initialization Order Fiasco*

[169]: (2025), *The Go Programming Language Specification, Package initialization*

Listing 5.5: File `file2.go`

```
1 package main
2
3 var m map[int]int
4
5 func init() {
6   m = make(map[int]int)
7 }
```

Listing 5.6: File `file3.go`

```
1 package main
2
3 func init() {
4   m[0] = 1
5 }
6
7 func main() {}
```

A simple solution (in the spirit of CSL [71]) is to over-approximate all possible orders in which the initializers of a package may run by modeling each of them as a thread that executes in parallel with the other initializers in the same package. This treatment requires that the initializer of a file does not access global variables declared in other files, which can be enforced by defining the precondition of the initializer of each file to contain only the permissions to the global variables declared in that file. One can then simply ensure that the package invariant of a package is

[71]: O'Hearn (2007), *Resources, concurrency, and local reasoning*

established after all its initializer threads conclude.

Implementation in Gobra

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

We implemented our technique in Gobra [29], a deductive program verifier for Go. In total, our extension comprises around 900 new lines of code. Our implementation incorporates the simplifications afforded by the additional guarantees listed in Subsection 5.4.2.

For every package, we introduce two initialization levels: the package level and the level of the package initializer. The latter is constrained to be lower than the former and higher than all the package levels of the imported packages. We introduce an annotation `mayInit` for methods that should be verified with the level of the initializer, as they may be called during initialization. All other methods are verified at the package level.

[188]: (2022), *cmd/compile: global variable initialization done in unexpected order*

We tested our implementation in Gobra to guarantee it implements the Go language specification correctly. In the process, we identified a mismatch between the intended semantics for initialization and the compiled program, which led us to find a previously unknown bug in the Go compiler [188]. This bug was caused by an optimization that rewrites the initializing expression for a global variable x without taking into account the effect that functions called during initialization might have on the value of x. In some cases, this bug caused the compiled program to store unexpected values in global variables after package initialization.

5.5 Evaluation

To demonstrate both the flexibility and the real-world applicability of our technique, we have evaluated it in three steps. First, we verified a number of small case studies representing important usage patterns for global state and initialization, shown in Tables 5.1 and 5.2, using our implementations in VerCors and Gobra, respectively. Second, we applied our technique to the VerifiedSCION project (Chapter 3), for which we specified and verified all uses of global state using our technique as implemented in Gobra. Third, we applied our technique to develop new verification (ghost) libraries. These libraries rely extensively on ghost global state.

5.5.1 Case studies

We collected a set of 11 examples representing important coding patterns or showcasing specific aspects of initialization to demonstrate that our technique is sufficiently flexible to verify different idiomatic uses of global state and initializers. We manually translated all applicable examples to both Java and Go. Since, for example, Go does not have support for subclassing, we only verified a Java version of that example. For examples marked with an asterisk in Table 5.1, we verified the Java code using the version of our technique adapted for sequential Java discussed in Subsection 5.4.1, since the code is not thread-safe and thus not correct in a concurrent setting.

Name	LOC	Ann.	MInv.	Time [s]
Byte	18	17	7	1.91
Client*	13	18	11	1.58
Contexts	38	21	5	1.34
Counter	43	59	15	2.03
Fib	19	18	6	1.41
Fib Conc.	25	28	9	1.79
Finalizer	57	59	21	21.35
Names*	35	24	11	2.26
Singleton*	24	20	10	4.24
Subclass	26	10	9	1.35
Theory*	27	23	11	1.69

Table 5.1: Verified case studies in Java with total lines of code (LOC), lines of proof annotations (Ann.), the number of those lines that is specific to the declaration of module invariants, levels and opening and closing module invariants (MInv.), as well as the verification time in seconds (Time).

Name	LOC	Ann.	MInv.	Time [s]
Byte	22	27	8	1.77
Client	15	26	9	1.02
Contexts	35	20	5	1.17
Counter	24	55	8	1.44
Fib	14	21	6	0.88
Fib Conc.	21	29	9	1.37
Finalizer	47	75	21	33.46
Names	42	25	4	2.25
Singleton	21	24	2	0.61

Table 5.2: Verified case studies in Go with total lines of code (LOC), lines of proof annotations (Ann.), the number of those lines that is specific to the declaration of module invariants, levels and opening and closing module invariants (MInv.), as well as the verification time in seconds (Time).

Fib is the example from Listing 5.1, specified and verified exactly as shown, and **Fib Conc.** is a thread-safe version of `Fib`. **Counter** is the example from Listing 5.2, adapted so that all calls to `getAndInc` are specified to return a fresh value. **Byte** and **Finalizer** are (simplified) versions of code found in the Java standard library and the OpenJDK Java runtime, respectively. The `Byte` class allocates a global pool of (immutable) `Byte` objects (one for every possible `byte` value) upon initialization that can then be used by all clients via a duplicable invariant. The `Finalizer` class maintains a global doubly-linked list of `Finalizer` instances for objects that are being garbage collected and whose `finalize` methods are to be invoked before they are deleted, which we verify using a non-duplicable invariant. **Singleton** is a generic implementation of (a non-thread-safe version of) the singleton pattern, using a static method to initialize a static field on first access. We use a duplicable invariant to give all clients read access. The **Subclass** example declares several classes and subclasses, and illustrates that our technique correctly rejects (using initialization levels) classes which introduce cyclic dependencies between static initializers even when they are indirect via superclass initialization, but allows similar uses that do not lead to cycles. **Client** and **Theory** are examples from Leino and Müller's work [174]. The former uses a global counter that is read and incremented every time a new instance of the class is allocated, thus providing a unique identifier for each instance. The latter shows that their system can be used in a setting with subclassing where subclasses must satisfy behavioral subtyping [39]; we show that we can support the same setting. **Contexts** and **Names** are two examples from the evaluation of Liu et al.'s static analysis [173]. They are simplified extracts from the Scala compiler which are rejected by their static analysis because they do not satisfy its initialization-time irrelevance property, but which (as stated by their paper) constitute cases where the program can nevertheless be regarded as correct. We verify both examples, i.e., prove that module invariants are established and maintained correctly, showing that the presence of user-defined invariants allows us to be

[174]: Leino et al. (2005), *Modular Verification of Static Class Invariants*

[39]: Liskov et al. (1994), *A Behavioral Notion of Subtyping*

[173]: Liu et al. (2023), *Initializing Global Objects: Time and Order*

more precise than a static analysis.

Tables 5.1 and 5.2 show the size, number of overall proof annotations, and number of annotations specific to initialization (i.e., module invariants, levels, and ghost code for opening and closing module invariants) for each example, as well as the verification time. Verification time was measured on a laptop with an Apple M2 Max and 64GB of RAM and averaged over 10 runs. The amount of initialization-specific annotations is low throughout: for our case studies, which emphasize complex uses of global state, they are, at most, 22% of the total code for the case studies in Go, and at most 25% for those in Java, with the exception of the `Client` class, where 35% of the total code consists of initialization-specific annotations. The higher annotation count for this example results from having to manually provide levels to this class and all its methods, as the default levels do not satisfy the level requirements. For the other case studies in Java, a substantial amount of the proof can be performed using the default values for levels.

Our Go case studies tend to require fewer initialization-specific annotations than the ones written in Java: Gobra requires fewer level annotations and no annotations to open and close non-duplicable invariants. Apart from these differences, the overall proof structure and the module invariants for the Go and Java versions of the same code are generally similar, in spite of the different initialization semantics of both languages, showing that our technique offers a general way of reasoning about global state and initialization that effectively abstracts over semantic details.

For the most part, our examples verify in a few seconds, which is in line with the verification time in both VerCors and Gobra for programs of similar size and complexity that do not use global state and initialization. This suggests that the proof obligations imposed by our technique are not more difficult to prove than those that are not related to module initialization. In our evaluation, the `Finalizer` example stands out for taking multiple tens of seconds to verify in both VerCors and Gobra. The cause of the long verification time is a method using a module invariant that makes heavy use of quantifiers and sequences, which are challenging for SMT solvers to reason about. This is a known issue unrelated to initialization-specific proof obligations.

5.5.2 Real-world Go code base

To demonstrate that our technique scales beyond small case studies, we used it to specify and verify *all uses* of global state in the Go implementation of the router for the SCION internet architecture (Chapter 3).

The code base uses global state mainly for two purposes: (1) to store, in a mutable slice, the metadata about the types of network paths the router is aware of, and (2) to store immutable values that are reused throughout the codebase, like error messages. We capture properties about the former using non-duplicable invariants and properties about the later using duplicable invariants.

In total, we specified properties of global state and package initializers for eight packages, using duplicable invariants for all eight and non-duplicable invariants for three of them. The resulting annotation overhead was low, with a total of 67 lines of duplicable invariants and 27 lines of non-duplicable invariants. Due to the simplifications described in Subsection 5.4.2, little additional ghost code was required, with only 30 additional lines of initialization-related ghost code or annotations.

Applying our technique to this code base allowed us to replace previously unjustified assumptions of properties of the global state by sound, principled reasoning. Compared to the previous version of the code base, our changes did not slow down verification significantly. In our experiments, verifying the Go router took on average 45m46s without verifying the proof obligations related to initialization and opening module invariants, and 47m05s with them enabled, an increase of less than 3%. This demonstrates our approach scales to larger verification efforts.

5.5.3 Ghost global state

We have found that using ghost global variables is useful to provide models for verification libraries, i.e., libraries containing only ghost code. Logically speaking, ghost global variables are often useful to name witnesses of existentially quantified assertions. Doing so allows us to rewrite these assertions without existential quantifiers, as we can refer to the global variable directly. This, in turn, is useful to stabilize proofs and verification times (Chapter 4). In addition, naming the witness allows us to refer to the same witness throughout the program.

We demonstrated the usefulness of ghost global variables in two libraries implemented in Gobra. The first is a library that implements a general interface for reasoning about concurrent programs using ghost locations and resource algebras [85] in the style of Iris [84] and uses global state to define a model for each proof rule from Iris that manipulates ghost locations. It uses module invariants to state the fundamental correctness property of the library: each element of a resource algebra stored in a ghost location is valid, and composing the elements of ghost locations with the same name at any point in the program yields a valid element.

[85]: (2025-2026), *`resalgebra`: A library for resource algebras in Gobra*

[84]: Jung et al. (2015), *Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning*

The second library [189], which we discuss in length here, implements the refinement methodology from Bílá et al. [30] for verifying programs by refinement against an abstract model defined as a state-transition system.This library comprises roughly 700 lines of Gobra code, including comments and examples, and it verifies in 12 seconds. The implementation does not rely on verifier assumptions and abstract members. As such, it constitutes the first *mechanized* soundness proof of this methodology.

[189]: Pereira (2026), *Refinement with Guards and Locally Inductive Invariants (LIIs)*

[30]: Bílá et al. (2025), *A Refinement Methodology for Distributed Programs in Rust*

Using this library, it is possible to define state-transition systems in ghost code by implementing the following interface:

```
1  type TransitionSystem interface {
2    ghost
3    ensures r ==> Valid(s)
4    pure Init(s State) (r bool)

6    ghost
7    pure Valid(s State) bool

9    ghost
10   pure Enabled(s State, e Event) bool

12   ghost
13   requires Valid(s)
14   requires Enabled(s, e)
15   ensures  Valid(n)
16   pure Update(s State, e Event) (n State)
17 }
```

Listing 5.7: Definition of the `TransitionSystem` interface. It contains four methods: `Init` determines which states are acceptable initial states, `Valid` specifies the invariant of the state-transition system, `Enabled` determines which events are enabled at each state, and `Update` computes the state after an event.

When this library is imported, the `main` function (or a function called from it) may select *once* the state-transition system that the program under verification must refine, as well as the initial state of the abstract

model. After the state-transition system and initial state are selected, its current state may be updated by performing an *event*, as long as it is enabled.

The refinement library supports *event guards*, i.e., resources that must be owned by the thread that performs an event. If a thread tries to perform an event `e` but it does not hold the guards required by `e`, our methodology ensures that the program is rejected, even if `e` is enabled. The guard requirements for each event are expressed by implementing the following interface:

Listing 5.8: Definition of the `EventGuard` interface. It contains two methods: `Guards` returns the set of all guards, and `Requires` identifies which guards must be owned when performing an event.

```
type EventGuard interface {
  ghost
  decreases
  pure Guards() (ghost set[GuardV])

  ghost
  ensures  res ==> g elem Guards()
  decreases
  pure Requires(e Event, g GuardV) (res bool)
}
```

Event guards are a core contribution of the methodology from Bílá et al. They allow for a useful reasoning principle: if a thread owns a guard, no other thread may have performed the corresponding events. As such, it is possible to derive *locally inductive invariants (LIIs)*, i.e., stronger inductive predicates about the abstract states that were reached since the thread acquired the ownership of the guard[9].

9: While we do not cover LIIs in depth, we have proved this reasoning principle sound in Gobra.

Having explained the core components of the methodology, we may now cover this library's core API:

Listing 5.9: Public interface of the refinement library. To simplify, we omit the postconditions and lemmas that are related to LIIs.

```
pkgInvariant Init()
package refinementguard

ghost var TS TransitionSystem
ghost var EG EventGuard

pred Init()

pred Own()

pred GuardR(v GuardV)

ghost
requires Own()
requires acc(&TS) && TS != nil
ensures  TS.Valid(st)
decreases
pure func CurrState() (st State)

ghost
requires Init()
requires t != nil && eg != nil
requires t.Init(st)
ensures  Own()
ensures  acc(&TS, _) && TS === t
ensures  acc(&EG, _) && EG === eg
ensures  CurrState() === st
ensures  forall g GuardV :: g elem eg.Guards() ==> GuardR(g)
decreases
func InitTS(t TransitionSystem, eg EventGuard, st State)

ghost
requires Own()
requires acc(&TS, _) && TS != nil
requires acc(&EG, _) && EG != nil
requires TS.Enabled(CurrState(), e)
```

```
37 requires forall g GuardV :: EG.Requires(e, g) ==> GuardR(g)
38 ensures  Own()
39 ensures  acc(&TS, _) && TS != nil
40 ensures  acc(&EG, _) && EG != nil
41 ensures  CurrState() === old(TS.Update(CurrState(), e))
42 ensures  forall g GuardV :: EG.Requires(e, g) ==> GuardR(g)
43 decreases
44 func UpdateTS(e Event)
```

The public interface of the refinement library contains two (public) global variables: `TS` stores the state-transition system against which we are refining the program, and `EG` stores the `EventGuard` that associates each event with guards. When this package is imported, the `main` function (or a function called from it) may start the refinement proof by calling `InitTS`. This function takes the `TransitionSystem` and `EventGuard` against which refinement is proven, as well as an acceptable initial state. Importantly, *only one* call to `InitTS` is allowed: in order to satisfy the precondition in line 21, the current thread must own the resource `Init()` passed to the `main` function from the module invariant of the package (line 1). The resource `Init()` is consumed by the function `InitTS`, which in turn returns the ownership of the current abstract state (resource `Own()` in line 24), as well as the ownership of every guard `g`, which is represented as `GuardR(g)` (line 28). The current abstract state may change only through calls to the method `UpdateTS`. To perform an event `e`, the caller must own the current state (line 33) and all the guards required by it (line 37), and it must show that `e` is enabled for the current state (line 36).

The implementation of this library is enabled by Gobra's support for global variables and their initialization: First, the semantics of global state and package initialization guarantee that at most one instance of the `Init()` resource will ever be allocated. This guarantees that all threads have a consistent view of the current state of the protocol: in order for a thread to perform a protocol step, it must first acquire the only instance of `Init()` by synchronizing with all other threads that may perform transitions of the protocol concurrently. This guarantees that, at any moment in time, a proof of refinement that has started will never be invalidated by another thread that obtains ownership of a second instance of `Init()` which it uses to perform steps of the protocol without synchronizing with all other threads. Second, in order to prove the soundness of this library, we maintain various ghost data structures stored in global variables, including the current abstract state, as well as metadata per each guard of the state-transition system, so that we can prove the soundness of LIIs in a way that resembles the on-paper proof [30].

The success we had with implementing the refinement library and resource algebras in Gobra demonstrates how powerful this feature is. We are confident that ghost global state may be used to fully implement other advanced reasoning techniques in Gobra, like prophecy variables [190].

[190]: Jung et al. (2019), *The future is ours: prophecy variables in separation logic*

Finally, we believe that the refinement library shown here is very versatile. In the future, we would like to experiment with embedding the SCION router model directly in the program using this library, rather than generating the I/O specification described in the Subsection 3.6.3 via refinement as a way to demonstrate the versatility and scalability of this library.

5.6 Related work

Module invariants have been suggested as a way to specify the global variables of modules in various projects. The Java Modeling Language (JML), a specification language for Java adopted by various deductive program verifiers including KeY [178], Krakatoa [191] and OpenJML [192], proposes the use of *static invariants* to specify the expected values in static fields [193]. However, they do not prescribe a way to modularly check the establishment of these invariants or state at what program points they may be soundly assumed. To the best of our knowledge, no deductive verifier for mainstream programming languages supported module invariants prior to this work, including the aforementioned ones based on JML, other Java verifiers like VerCors [121] and VeriFast [119], and verifiers for other languages like Gobra [29] for Go, and Leon [194] for Scala.

[193]: (2013), *JML Reference Manual - Type Specifications*

Leino and Müller [174] propose using static invariants to reason about static fields in a programming language based on a *sequential* subset of Java and C#. They combine a strict ownership model for objects (see below) with a partial order on classes, the *validity ordering*, which prescribes the order in which classes are initialized. A notion of transitive validity allows them to write concise specifications. Unfortunately, this validity ordering requires that the initialization of a class runs *before* that of its superclasses. This strong restriction is not satisfied by the semantics of languages like Java and C#. Their approach does not handle concurrency and uses a restrictive ownership model that rules out referencing an object from two classes, even when sharing is safe, e.g., because the object is immutable. In contrast, our approach handles subclassing as it occurs in mainstream languages, supports concurrency, and leverages the full flexibility of separation logic to allow sharing when it is safe.

[174]: Leino et al. (2005), *Modular Verification of Static Class Invariants*

Jacobs et al. [177] propose a technique for modular verification of static invariants in multi-threaded programs written in a Java-like language. Their approach requires that *all accesses* to static fields are guarded by a class-level lock, unless *all fields* of that class are immutable. Our technique, by contrast, supports a wide variety of synchronization primitives and requires none when unsynchronized accesses are safe. Jacobs et al.'s approach prevents deadlocks, including those caused by cyclic initialization dependencies, by imposing an acquisition order on all locks. Their approach requires that module imports are acyclic, which is enforced by runtime checks. In contrast, we use initialization levels to prevent cyclic initialization dependencies statically, and allow for cyclic module imports.

[177]: Jacobs et al. (2007), *A Simple Sequential Reasoning Approach for Sound Modular Verification of Mainstream Multithreaded Programs*

Summers et al. [175] propose an extension to the *Visibility Technique* from Müller et al. [195] for verifying static invariants in a *sequential* Java-like programming language. Their approach imposes *effect annotations* on methods, consisting of all classes whose static methods may be transitively called from that method. To alleviate the need for these annotations, they introduce levels, which are similar to our initialization levels, but only apply to classes. In their approach, classes at a lower level may not mention others at a higher level, which prevents mutually defined classes.

[175]: Summers et al. (2009), *Universe-Type-Based Verification Techniques for Mutable Static Fields and Methods*
[195]: Müller et al. (2006), *Modular invariants for layered object structures*

All approaches mentioned so far [174, 175, 177] are based on restrictive ownership disciplines [195–197]. To the best of our knowledge, our technique is the first based on concurrent separation logic [71, 84], which is more expressive than the aforementioned disciplines, and for which there is an extensive body of work on reusable reasoning principles for

[195]: Müller et al. (2006), *Modular invariants for layered object structures*
[196]: Barnett et al. (2004), *Verification of Object-Oriented Programs with Invariants*
[197]: Leino et al. (2004), *Object Invariants in Dynamic Contexts*

different concurrency primitives like locks [198–200] and channels [201–203].

The Boogie verifier [204] supports global variables, which may be declared with `where`-clauses [205] that specify properties that are assumed whenever the variable is havocked. However, these clauses play a different role than our module invariants. They are used to encode trusted assumptions, rather than to express verified properties. Mutations of global variables are tracked via `modifies`-clauses in method specifications, whereas we rely on separation logic to reason about side effects.

[205]: (2023), *The Boogie IVL Language Reference - Global Variables*

Liu et al. [173] propose a static analysis to identify common issues in the initialization of Scala's singleton objects based on the principle of *initialization-time irrelevance*, which expresses that the state of a global singleton object at the end of its initialization should not depend on when that object is initialized. In contrast, we allow a module's state after initialization to depend on the initialization time *only if* its invariant is guaranteed to hold. Their technique does not consider user-provided invariants and imposes cumbersome restrictions, e.g., the static initializer of a singleton object must not perform any (even synchronized!) accesss to mutable global state of another.

[173]: Liu et al. (2023), *Initializing Global Objects: Time and Order*

Internally, Liu et al.'s analysis makes use of ideas commonly used when reasoning about object initialization and instance invariants, for which various solutions have been proposed [206–208]. These works do not address module initialization and instead identify conditions under which objects may safely escape constructors and non-initialized objects may be accessed. While there are commonalities with module initialization, reasoning about the latter poses distinct challenges: Module initialization starts and ends at modularly unknown execution points (unlike constructors, which are invoked explicitly and eventually return), and global state can be accessed from anywhere in the code (unlike objects under construction, which must be passed around explicitly).

[206]: Summers et al. (2011), *Freedom before commitment: a lightweight type system for object initialisation*
[207]: Liu et al. (2020), *A type-and-effect system for object initialization*
[208]: Blaudeau et al. (2022), *A conceptual framework for safe object initialization: a principled and mechanized soundness proof of the Celsius model*

5.7 Conclusion

In this chapter, we propose a practical technique for specifying and verifying programs that make use of global data. Our technique can be applied to concurrent programs, and to a wide range of programming languages with varying semantics, including lazy and eager initialization. It allows specifying modules with invariants, and uses initialization levels, a novel specification construct that can be used to prevent cyclical initialization cycles and to identify which module invariants are guaranteed to hold. Furthermore, we show it can be automated in state-of-the-art deductive program verifiers.

6 Conclusions and Future Work

This thesis presents our contributions towards the general goal of verifying pre-existing software systems. A key success factor was our modular verification approach, which allowed us to reduce the verification complexity and work in parallel on different aspects of the problem. We demonstrated the feasibility of our approach on the VerifiedSCION project, where we verified the SCION router from its high-level model, where we proved system-wide security properties, down to its performant implementation, which we showed to correctly implement the protocol. This is, to the best of our knowledge, the largest project verified with a separation-logic SMT-based verifier to date. In the process, we discovered both design and implementation errors, critically affecting the router's security, that evaded all prior reviews and testing and have all been subsequently fixed.

The biggest obstacle to our productivity was verification performance. The size of the codebase, the size of the proof context of some proof obligations, and the complexity of others (e.g., those related to quantified permissions) led to unpredictably long verification times. We identified various strategies that successfully decreased the verification time and made the proofs more stable. We split these strategies into two groups: (1) strategies to carefully manage the verification context, which reduce the amount of irrelevant information in the proof obligations, and (2) different configurations of the symbolic execution algorithm used by Gobra for generating proof obligations, which lead to completeness trade-offs and trade-offs between the number of explored branches and the complexity of each SMT query. In addition, we provide a thorough analysis of the verification performance in the latest version of VerifiedSCION, which we hope serves as an important case study for future research on the performance of SMT-based verifiers. An important conclusion of our work on verification performance is that understanding and addressing performance issues still requires good familiarity with the algorithms used to generate the proof obligations (e.g., how permission amounts are computed, when and how verification branches are joined). Different algorithms provide different trade-offs, and no single configuration seems to be optimal.

Carrying out the work on VerifiedSCION required extending Gobra to support modular reasoning about all the features used by the original open-source router implementation. A particularly challenging feature is global variables, as their initializers may be side-effectful and the exact timing and order in which they run cannot be determined modularly. We propose the first practical technique for specifying and verifying *concurrent* programs that make use of global data that not only applies to Go (which immediately solves the problem of verifying the SCION codebase), but also generalizes to other programming languages with different initialization semantics (e.g., lazy initialization as found in Java instead of the eager initialization as found in Go). Our technique allows specifying modules with invariants, and uses initialization levels, a novel specification construct that can be used to prevent cyclic initialization dependencies and to identify which module invariants are guaranteed to hold. We demonstrated the versatility of our approach on challenging examples, and we applied it to the VerifiedSCION codebase.

Future work

Even though we verified a real-world codebase using the methodology and techniques discussed in this thesis, there are still various challenges that prevent verification from being practical, which we will explore in future work: First, the experiments described in Chapter 4 show that the proof obligations related to heap reasoning (especially those related to quantified permissions) are extremely onerous, and we ought to experiment with new designs that explore different trade-offs between completeness and performance in Separation Logic, in the spirit of the paper *Verification Algorithms for Automated Separation Logic Verifiers* [128]. The differences between algorithms matter, and rather than hiding them from the users, I believe it would be productive to provide a proof environment for separation logic where different (configurable) algorithms (e.g., verification condition generation, symbolic execution, and model checking) are provided as tactics (or metaprograms that generate each proof obligation as a lemma and automatically dispatch them to the SMT solver, exposing the unsolved goals to the user similarly to Velvet [162]), and where users may define new procedures and prove them sound.

[128]: Eilers et al. (2024), *Verification Algorithms for Automated Separation Logic Verifiers*

[162]: Gladshtein et al. (2026), *Velvet: A Foundational Multi-Modal Verifier for Imperative Programs in Lean*

Second, estimating the effort and duration of verification projects is extremely hard. We ought to understand and reduce the costs of program verification. An illuminating step in this direction was taken by Matichuk et al. [209], who analyzed proofs from the sel4 kernel [1] and two large software verification proof developments in the Archive of Formal Proof and identified a consistent quadratic relation between the size of the formal statement of a property and the final size of its formal proof in the interactive theorem prover Isabelle. In turn, Staples et al. [210] identified a linear relation between effort (in person-weeks) and proof size (in lines of proof script). This suggests an extremely high cost for formal verification in most settings. To the best of our knowledge, no existing work provides such an analysis of the effort required for verification projects carried out in SMT-based verifiers. As I write this thesis, an optimism that Large Language Models will lower the costs of proofs defines the current zeitgeist. This optimism is embodied by the various research projects to generate program correctness proofs using LLMs in verifiers like Dafny [211–213] and Verus [214–217], as well as by the various startups developing specialized models and tools for program verification.

[209]: Matichuk et al. (2015), *Empirical Study Towards a Leading Indicator for Cost of Formal Software Verification*

[1]: Klein et al. (2009), *seL4: formal verification of an OS kernel*

[210]: Staples et al. (2014), *Productivity for proof engineering*

[211]: Misu et al. (2024), *Towards AI-Assisted Synthesis of Verified Dafny Methods*

[212]: Mugnier et al. (2025), *Laurel: Unblocking Automated Verification with Large Language Models*

[213]: Sun et al. (2024), *Clover: Closed-Loop Verifiable Code Generation*

[214]: Yang et al. (2025), *AutoVerus: Automated Proof Generation for Rust Code*

[215]: Chen et al. (2026), *Automated Proof Generation for Rust Code via Self-Evolution*

[216]: Zhong et al. (2025), *RAG-Verus: Repository-Level Program Verification with LLMs using Retrieval Augmented Generation*

[217]: Liu et al. (2026), *KVerus: Scalable and Resilient Formal Verification Proof Generation for Rust Code*

Finally, software is subject to change. In contexts where the code is developed separately from its correctness proof, it would be desirable for the proof to be updated (semi-)automatically based on the changes to the code. Relying on SMT solvers helped us in this regard, as it allowed us to quickly incorporate small changes from the developers with little effort on our part to change the specification and proof annotations. In the presence of large refactorings (e.g., introducing concurrency in a sequential codebase, adopting different concurrency primitives, or changing fundamental data strucures of the program), it is not clear how this can be done.It would thus be instructive to apply the techniques from *proof repair* [218] to large verified codebases, with the goal of making proof maintenance as painless for developers as possible. However, in the presence of large changes, the relation between intermediate computations of old and new programs may become very complex, preventing the use of approaches that rely on tracing executions to align the computations of both versions [219].

[218]: Ringer (2021), *Proof Repair*

[219]: Gopinathan et al. (2023), *Mostly Automated Proof Repair for Verified Libraries*

Bibliography

[1] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220. DOI: 10.1145/1629575.1629596 (cited on pages 1, 124).

[2] Dirk Leinenbach and Thomas Santen. "Verifying the Microsoft Hyper-V Hypervisor with VCC". In: *FM 2009: Formal Methods*. Ed. by Ana Cavalcanti and Dennis R. Dams. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 806–809 (cited on page 1).

[3] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels". In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 2016, pp. 653–669 (cited on page 1).

[4] Xavier Leroy. "Formal verification of a realistic compiler". In: *Commun. ACM* 52.7 (2009), pp. 107–115 (cited on page 1).

[5] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML: a verified implementation of ML". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 179–192 (cited on page 1).

[6] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. "IronFleet: Proving Practical Distributed Systems Correct". In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. 2015 (cited on pages 1, 24, 53).

[7] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. "Verdi: a framework for implementing and formally verifying distributed systems". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015*. 2015 (cited on pages 1, 53).

[8] Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. "Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq". In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Proceedings*. Ed. by Amal Ahmed. Vol. 10801. Springer, 2018 (cited on pages 1, 53).

[9] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. "Implementing TLS with Verified Cryptographic Security". In: *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 2013, pp. 445–459 (cited on pages 1, 52).

[10] 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: *2017 IEEE Symposium on Security and Privacy, SP 2017, CA, USA, 2017*. IEEE Computer Society, 2017 (cited on pages 1, 52).

[11] 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: *42nd IEEE Symposium on Security and Privacy, SP 2021, 2021*. IEEE, 2021, pp. 1162–1178 (cited on pages 1, 52).

[12] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. "HACL*: A Verified Modern Cryptographic Library". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, 2017*. ACM, 2017, pp. 1789–1806 (cited on page 1).

[13] 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: *2020 IEEE Symposium on Security and Privacy, SP 2020, 2020*. IEEE, 2020 (cited on page 1).

[14] 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: *PACMPL* 1.ICFP (2017) (cited on pages 1, 52).

[15] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. "Refinement through restraint: bringing down the cost of verification". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. Nara, Japan: Association for Computing Machinery, 2016, pp. 89–102. DOI: 10.1145/2951913.2951940 (cited on page 1).

[16] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. "Anvil: Verifying Liveness of Cluster Management Controllers". In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, July 2024, pp. 649–666 (cited on pages 1, 24, 53).

[17] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20 (cited on pages 1, 53, 55, 82).

[18] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016. Proceedings*. 2016 (cited on pages 2, 9, 37, 55, 57).

[19] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008 (cited on pages 2, 9, 73).

[20] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Logic in Computer Science (LICS)*. IEEE Computer Society Press, 2002 (cited on pages 2, 64, 89, 90).

[21] Jan Smans, Bart Jacobs, and Frank Piessens. "Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic". In: *ECOOP*. Vol. 5653. Lecture Notes in Computer Science. Springer, 2009, pp. 148–172 (cited on pages 2, 9, 12, 38).

[22] Xin Zhang, Hsu-Chun Hsiao, Geoffrey Hasker, Haowen Chan, Adrian Perrig, and David Andersen. "SCION: Scalability, Control, and Isolation On Next-Generation Networks". In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2011 (cited on page 3).

[23] 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 (cited on pages 3, 24, 28, 29, 40).

[24] J. Pereira, T. Klenze, S. Giampietro, M. Limbeck, D. Spiliopoulos, F. Wolf, M. Eilers, C. Sprenger, D. Basin, P. Müller, and A. Perrig. "Protocols to Code: Formal Verification of a Secure Next-Generation Internet Router". In: *Computer and Communications Security (CCS)*. CCS '25. Taipei, Taiwan: Association for Computing Machinery, 2025. DOI: 10.1145/3719027.3765104 (cited on page 5).

[25] Linard Arquint, Peter Müller, Wytse Oortwijn, João Pereira, and Felix Wolf. "Code-Level Verification". In: *The Complete Guide to SCION: From Design Principles to Formal Verification*. Ed. by Laurent Chuat, Markus Legner, David Basin, David Hausheer, Samuel Hitz, Peter Müller, and Adrian Perrig. Springer, 2022. Chap. 23, pp. 519–562 (cited on page 5).

[26] Linard Arquint, David Basin, Tobias Klenze, Si Liu, Peter Müller, João Pereira, Christoph Sprenger, and Felix Wolf. "Current Status and Plans". In: *The Complete Guide to SCION: From Design Principles to Formal Verification*. Ed. by Laurent Chuat, Markus Legner, David Basin, David Hausheer, Samuel Hitz, Peter Müller, and Adrian Perrig. Springer, 2022. Chap. 24, pp. 563–572 (cited on page 5).

[27] João Pereira, Isaac van Bakel, Patricia Firlejczyk, Marco Eilers, and Peter Müller. "Modular Reasoning about Global Variables and Their Initialization". In: *Proc. ACM Program. Lang.* 9.OOPSLA2 (Oct. 2025). DOI: 10.1145/3763133 (cited on page 5).

[28] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28 (2018), e20. DOI: 10.1017/S0956796818000151 (cited on pages 6, 44, 91).

[29] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. "Gobra: Modular Specification and Verification of Go Programs". In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 367–379. DOI: 10.1007/978-3-030-81685-8_17 (cited on pages 6, 9, 55, 90, 92, 110, 114, 120).

[30] Aurea Bílá, João Pereira, and Peter Müller. "A Refinement Methodology for Distributed Programs in Rust". In: *Proc. ACM Program. Lang.* 9.OOPSLA2 (Oct. 2025). DOI: 10.1145/3763119 (cited on pages 6, 117, 119).

[31] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *LICS*. IEEE Computer Society, 2002, pp. 55–74 (cited on pages 9, 24, 38).

[32] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *CSL*. Vol. 2142. Lecture Notes in Computer Science. Springer, 2001, pp. 1–19 (cited on pages 9, 38).

[33] Malte Schwerhoff. "Advancing Automated, Permission-Based Program Verification Using Symbolic Execution". PhD thesis. ETH Zurich, 2016 (cited on pages 9, 57, 72).

[34] Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. "Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions". In: *ECOOP 2013 – Object-Oriented Programming*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 451–476 (cited on page 9).

[35] Edsger W. Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs". In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. DOI: 10.1145/360933.360975 (cited on page 9).

[36] John Boyland. "Checking Interference with Fractional Permissions". In: *Static Analysis, 10th International Symposium, SAS 2003, 2003, Proceedings*. Ed. by Radhia Cousot. Vol. 2694. Lecture Notes in Computer Science. Springer, 2003, pp. 55–72 (cited on pages 12, 13, 94).

[37] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Proceedings, Part I*. Vol. 9779. Lecture Notes in Computer Science. Springer, 2016, pp. 405–425 (cited on pages 14, 38, 59, 78).

[38] Alexander J. Summers and Sophia Drossopoulou. "A Formal Semantics for Isorecursive and Equirecursive State Abstractions". In: *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*. Ed. by Giuseppe Castagna. Vol. 7920. Lecture Notes in Computer Science. Springer, 2013, pp. 129–153. DOI: 10.1007/978-3-642-39038-8_6 (cited on pages 15, 67, 96).

[39] Barbara Liskov and Jeannette M. Wing. "A Behavioral Notion of Subtyping". In: *ACM Trans. Program. Lang. Syst.* 16.6 (1994), pp. 1811–1841. DOI: 10.1145/197320.197383 (cited on pages 18, 110, 115).

[40] S. Kent, C. Lynn, and K. Seo. "Secure Border Gateway Protocol (S-BGP)". In: *IEEE Journal on Selected Areas in Communications* 18.4 (2000) (cited on page 23).

[41] R. Bush. *Origin Validation Operation Based on the Resource Public Key Infrastructure (RPKI)*. RFC 7115. 2014 (cited on page 23).

[42] Matthew Lepinski and Kotikalapudi Sriram. *BGPsec Protocol Specification*. RFC 8205. Sept. 2017. URL: https://rfc-editor.org/rfc/rfc8205.txt (cited on page 23).

[43] Danny Cooper, Ethan Heilman, Kyle Brogle, Leonid Reyzin, and Sharon Goldberg. "On the risk of misbehaving RPKI authorities". In: *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*. 2013, pp. 1–7 (cited on page 23).

[44] Qi Li, Yih-Chun Hu, and Xinwen Zhang. "Even rockets cannot make pigs fly sustainably: Can BGP be secured with BGPsec?" In: *Proceedings of the NDSS Workshop on Security of Emerging Networking Technologies (SENT)*. Internet Society. 2014 (cited on page 23).

[45] Benjamin Rothenberger, Daniele E. Asoni, David Barrera, and Adrian Perrig. "Internet Kill Switches Demystified". In: *Proceedings of the European Workshop on Systems Security (EuroSec)*. 2017 (cited on page 23).

[46] Laurent Chuat, Markus Legner, David Basin, David Hausheer, Samuel Hitz, Peter Müller, and Adrian Perrig. *The Complete Guide to SCION: From Design Principles to Formal Verification*. Springer International Publishing, 2022 (cited on pages 23, 41).

[47] François Wirz, Marten Gartner, Jelte van Bommel, Elham Ehsani Moghadam, Grace H. Cimaszewski, Anxiao He, Yizhe Zhang, Henry Birge-Lee, Flix Kottmann, Cyrill Krähenbühl, Jonghoon Kwon, Kyveli Mavromati, Liang Wang, Daniel Bertolo, Marco Canini, Buseung Cho, Ronaldo A. Ferreira, Simon. P. Green, David Hausheer, Junbeom Hur, Xiaohua Jia, Heejo Lee, Prateek Mittal, Omo Oaiya, Chanjin Park, Adrian Perrig, Jerry Sobieski, Yixin Sun, Cong Wang, and Klaas Wierenga. "Scaling SCIERA: A Journey Through the Deployment of a Next-Generation Network". In: *Proceedings of ACM SIGCOMM*. Sept. 2025 (cited on page 23).

[48] SIX. *Secure Swiss Finance Network*. URL: https://www.six-group.com/en/products-services/banking-services/ssfn.html (cited on page 23).

[49] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and Philip Brighten Godfrey. "VeriFlow: Verifying Network-Wide Invariants in Real Time". In: *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*. 2013, pp. 15–27 (cited on pages 24, 52).

[50] Dexter Kozen. "NetKAT – A Formal System for the Verification of Networks". In: *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014*. 2014, pp. 1–18 (cited on pages 24, 52).

[51] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. "Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver". In: *Proc. ACM Program. Lang.* OOPSLA 2016. Association for Computing Machinery, 2016, pp. 765–780 (cited on pages 24, 52).

[52] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. "A General Approach to Network Configuration Verification". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017*. ACM, 2017, pp. 155–168 (cited on pages 24, 52).

[53] Solal Pirelli, Akvile Valentukonyte, Katerina J. Argyraki, and George Candea. "Automated Verification of Network Function Binaries". In: *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, 2022*. Ed. by Amar Phanishayee and Vyas Sekar. USENIX Association, 2022 (cited on pages 24, 52).

[54] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. "Automated Verification of Customizable Middlebox Properties with Gravel". In: *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, 2020*. USENIX Association, 2020, pp. 221–239 (cited on pages 24, 52).

[55] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh R. Iyer, Matteo Rizzo, Luis Pedrosa, Katerina J. Argyraki, and George Candea. "Verifying software network functions with no verification expertise". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, 2019*. ACM, 2019, pp. 275–290 (cited on pages 24, 52).

[56] Jed Liu, William T. Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Calin Cascaval, Nick McKeown, and Nate Foster. "p4v: practical verification for programmable data planes". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Hungary 2018*. Ed. by Sergey Gorinsky and János Tapolcai. ACM, 2018 (cited on pages 24, 52).

[57] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina J. Argyraki, and George Candea. "A Formally Verified NAT". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017*. ACM, 2017 (cited on pages 24, 52).

[58] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Springer, 2002 (cited on pages 24, 29).

[59] Danny Dolev and Andrew Chi-Chih Yao. "On the security of public key protocols". In: *IEEE Trans. Information Theory* 29.2 (1983) (cited on pages 24, 31).

[60] Tobias Klenze, Christoph Sprenger, and David Basin. "Formal Verification of Secure Forwarding Protocols". In: *2021 IEEE 34rd Computer Security Foundations Symposium (CSF)*. IEEE, 2021 (cited on pages 24, 27, 31).

[61] Tobias Klenze, Christoph Sprenger, and David Basin. "IsaNet: A framework for verifying secure data plane protocols". In: *Journal of Computer Security* (2022) (cited on pages 24, 26, 27, 31, 33, 34, 51, 52).

[62] J. Pereira, T. Klenze, S. Giampietro, M. Limbeck, D. Spiliopoulos, F. Wolf, M. Eilers, C. Sprenger, D. Basin, P. Müller, and A. Perrig. *Artifact for the paper "Protocols to Code: Formal Verification of a Secure Next-Generation Internet Router"*. https://doi.org/10.5281/zenodo.16891069. 2025. DOI: 10.5281/zenodo.16891069 (cited on page 25).

[63] Sofia Giampietro. "Advances in the Symbolic Verification of Cryptographic Protocols". PhD thesis. ETH Zurich, 2026 (cited on page 25).

[64] Giacomo Giuliari, Dominik Roos, Marc Wyss, Juan Angel García-Pardo, Markus Legner, and Adrian Perrig. "Colibri: a cooperative lightweight inter-domain bandwidth-reservation infrastructure". In: *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 104–118 (cited on page 28).

[65] Benedikt Schmidt, Simon Meier, Cas J. F. Cremers, and David A. Basin. "Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties". In: *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. Ed. by Stephen Chong. IEEE Computer Society, 2012, pp. 78–94 (cited on page 29).

[66] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. "The TAMARIN Prover for the Symbolic Analysis of Security Protocols". In: *Computer Aided Verification - 25th International Conference, CAV 2013, 2013. Proceedings*. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 696–701 (cited on page 29).

[67] Bruno Blanchet. "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules". In: *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 2001, Canada*. IEEE Computer Society, 2001, pp. 82–96 (cited on page 29).

[68] Bruno Blanchet. "Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif". In: *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. Ed. by Alessandro Aldini, Javier López, and Fabio Martinelli. Vol. 8604. Lecture Notes in Computer Science. Springer, 2013, pp. 54–87 (cited on page 29).

[69] Martín Abadi and Leslie Lamport. "The Existence of Refinement Mappings". In: *Theor. Comput. Sci.* 82.2 (1991) (cited on page 29).

[70] Nancy A. Lynch and Frits W. Vaandrager. "Forward and Backward Simulations: I. Untimed Systems". In: *Inf. Comput.* 121.2 (1995) (cited on page 29).

[71] Peter W. O'Hearn. "Resources, concurrency, and local reasoning". In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 271–307 (cited on pages 30, 90, 113, 120).

[72] 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 (1)*. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 367–379 (cited on page 30).

[73] Martín Abadi and Véronique Cortier. "Deciding knowledge in security protocols under equational theories". In: *Theoretical Computer Science* 367.1 (2006), pp. 2–32 (cited on page 34).

[74] Santiago Escobar, Ralf Sasse, and José Meseguer. "Folding variant narrowing and optimal variant termination". In: *The Journal of Logic and Algebraic Programming* 81.7 (2012), pp. 898–928 (cited on page 34).

[75] Tobias Klenze. *Adding checks to intra-segment forwarding to achieve stronger security*. https://github.com/scionproto/scion/issues/4148. 2022 (cited on page 36).

[76] Matthias Frei. *router: add checks for intra-segment forwarding*. https://github.com/scionproto/scion/pull/4294. 2022 (cited on page 36).

[77] Noah Delius. "Automatic Generation of I/O Specifications from Component Event Systems in Igloo". MA thesis. Department of Computer Science, ETH Zurich, Switzerland, Sept. 2021 (cited on page 36).

[78] Marco Eilers and Peter Müller. "Nagini: A Static Verifier for Python". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Springer, 2018 (cited on pages 36, 55).

[79] Ralf Jung. *There is no memory safety without thread safety*. https://www.ralfj.de/blog/2025/07/24/memory-safety.html. 2025 (cited on page 37).

[80] Malte Schwerhoff and Alexander J. Summers. "Lightweight Support for Magic Wands in an Automatic Verifier". In: *29th European Conference on Object-Oriented Programming, ECOOP 2015*. Vol. 37. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 614–638. DOI: 10.4230/LIPICS.ECOOP.2015.614 (cited on page 38).

[81] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. "Sound Automation of Magic Wands". In: *Computer Aided Verification - 34th International Conference, CAV 2022, Proceedings, Part II*. Ed. by Sharon Shoham and Yakir Vizel. Vol. 13372. Lecture Notes in Computer Science. Springer, 2022, pp. 130–151. DOI: 10.1007/978-3-031-13188-2_7 (cited on page 38).

[82] Dominique Unruh. "The impossibility of computationally sound XOR". In: *IACR Cryptology ePrint Archive* 2010 (Dec. 2010), p. 389 (cited on page 41).

[83] Willem Penninckx, Bart Jacobs, and Frank Piessens. "Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs". In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, 2015. Proceedings*. 2015 (cited on page 41).

[84] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India: Association for Computing Machinery, 2015, pp. 637–650. DOI: 10.1145/2676726.2676980 (cited on pages 43, 92, 101, 117, 120).

[85] *resalgebra: A library for resource algebras in Gobra*. https://github.com/viperproject/gobra-libs/tree/main/resalgebra. 2025-2026 (cited on pages 44, 117).

[86] Gernot Heiser. *Benchmarking Crimes Meet Formal Verification*. https://microkerneldude.org/2025/04/27/benchmarking-crimes-meet-formal-verification/. 2025 (cited on page 44).

[87] Johannes Gasser, João Pereira, and Linard Arquint. *gobra-action's Github repository*. https://github.com/viperproject/gobra-action. 2022 (cited on page 46).

[88] João Pereira and Felix Wolf. *A Tutorial on Gobra*. https://github.com/viperproject/gobra/blob/master/docs/tutorial.md. 2021 (cited on page 46).

[89] Ali Gottschall and João Pereira. *Practical Program Verification in Go with Gobra*. https://viperproject.github.io/gobra-book/. 2025 (cited on page 46).

[90] Amos Herz. *A Specification Linter for Gobra*. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Amos_Herz_PW_Report.pdf. 2025 (cited on page 46).

[91] Daniel Nezamabadi and João Pereira. *gobra-libs's Github repository*. https://github.com/viperproject/gobra-libs. 2024 (cited on page 46).

[92] Axel Montini. *Specification and Verification of parts of the Go Standard Library using Gobra*. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Axel_Montini_PW_Project_Report.pdf. 2025 (cited on page 46).

[93] Pontus Boström and Peter Müller. "Modular Verification of Finite Blocking in Non-terminating Programs". In: *29th European Conference on Object-Oriented Programming, ECOOP 2015, Prague, Czech Republic, July 5-10, 2015*. Ed. by John Tang Boyland. Vol. 37. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 639–663. DOI: 10.4230/LIPICS.ECOOP.2015.639 (cited on page 46).

[94] Vytautas Astrauskas. "Input-Output Verification in Viper". MA thesis. ETH Zurich, 2016 (cited on page 46).

[95] Felix A. Wolf and Peter Müller. "Verifiable Security Policies for Distributed Systems". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS '24. Salt Lake City, UT, USA: Association for Computing Machinery, 2024, pp. 4–18. DOI: 10.1145/3658644.3690303 (cited on page 47).

[96] Henri Inndorf. *Verifying Secure Information Flow for the VerifiedSCION Router*. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Henri_Inndorf_BSc_Report.pdf. 2025 (cited on page 47).

[97] John McLean. "Proving Noninterference and Functional Correctness Using Traces". In: *J. Comput. Secur.* 1.1 (Jan. 1992), pp. 37–57 (cited on page 47).

[98] A.W. Roscoe. "CSP and determinism in security modelling". In: *Proceedings 1995 IEEE Symposium on Security and Privacy*. 1995, pp. 114–127. DOI: 10.1109/SECPRI.1995.398927 (cited on page 47).

[99] Marco Eilers, Peter Müller, and Samuel Hitz. "Modular Product Programs". In: *ACM Trans. Program. Lang. Syst.* 42.1 (Nov. 2019). DOI: 10.1145/3324783 (cited on page 48).

[100] Marco Eilers, Severin Meier, and Peter Müller. "Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security". In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 718–741 (cited on page 48).

[101] Dan Frumin, Robbert Krebbers, and Lars Birkedal. "Compositional Non-Interference for Fine-Grained Concurrent Programs". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1416–1433. DOI: 10.1109/SP40001.2021.00003 (cited on page 48).

[102] João Pereira. *`scionproto/scion`: Add Gobra to the CI and specify and verify infofield.go and hopfield.go (experimental)*. https://github.com/scionproto/scion/pull/4187. 2022 (cited on page 49).

[103] João Pereira. *experimental: add specifications to packages `pkg/addr` and `pkg/slayers/path`, verify them in the CI with Gobra*. https://github.com/scionproto/scion/pull/4837. 2025 (cited on page 49).

[104] Prometheus Authors 2014-2024. *Prometheus Go client library*. https://github.com/prometheus/client_golang/ (cited on page 49).

[105] Google. *GoPacket*. https://github.com/google/gopacket (cited on page 49).

[106] Linard Arquint, Malte Schwerhoff, Vaibhav Mehta, and Peter Müller. "A Generic Methodology for the Modular Verification of Security Protocol Implementations". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Denmark, 2023*. Ed. by Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda. ACM, 2023 (cited on pages 50, 53).

[107] Tobias Klenze and Christoph Sprenger. "IsaNet: Formalization of a Verification Framework for Secure Data Plane Protocols". In: *Archive of Formal Proofs* (June 2022). https://isa-afp.org/entries/IsaNet.html, Formal proof development (cited on pages 51, 52).

[108] Aaron Bojarski. *Updating the proof annotations to the latest version of SCION*. https://github.com/aaronbojarski/VerifiedSCION/pull/4. 2024–2025 (cited on page 51).

[109] Vladimir Gladshtein, George Pîrlea, Qiyuan Zhao, Vitaly Kurin, and Ilya Sergey. "Foundational Multi-Modal Program Verifiers". In: *Proc. ACM Program. Lang.* 10.POPL (Jan. 2026). DOI: 10.1145/3776719 (cited on page 52).

[110] Chen Chen, Limin Jia, Hao Xu, Cheng Luo, Wenchao Zhou, and Boon Thau Loo. "A Program Logic for Verifying Secure Routing Protocols". In: *Logical Methods in Computer Science* 11.4 (2015) (cited on page 52).

[111] Mathilde Arnaud, Véronique Cortier, and Stéphanie Delaune. "Modeling and verifying ad hoc routing protocols". In: *Inf. Comput.* 238 (2014), pp. 30–67 (cited on page 52).

[112] Mathilde Arnaud, Véronique Cortier, and Stéphanie Delaune. "Deciding Security for Protocols with Recursive Tests". In: *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*. Ed. by Nikolaj Bjørner and Viorica Sofronie-Stokkermans. Vol. 6803. Lecture Notes in Computer Science. Springer, 2011, pp. 49–63 (cited on page 52).

[113] 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: *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, 2021, pp. 523–542 (cited on page 53).

[114] 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: *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 1077–1093 (cited on page 53).

[115] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. "Planning for change in a formal verification of the Raft consensus protocol". In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, 2016*. 2016 (cited on page 53).

[116] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. "Chapar: certified causally consistent distributed key-value stores". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016 (cited on page 53).

[117] 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: *Principles of Programming Languages (POPL)*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016 (cited on page 55).

[118] Ekanshdeep Gupta, Nisarg Patel, and Thomas Wies. "Raven: An SMT-Based Concurrency Verifier". In: *Computer Aided Verification: 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part I*. Zagreb, Croatia: Springer-Verlag, 2025, pp. 81–106. DOI: 10.1007/978-3-031-98668-0_4 (cited on page 55).

[119] Bart Jacobs, Jan Smans, and Frank Piessens. "A Quick Tour of the VeriFast Program Verifier". In: *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*. Ed. by Kazunori Ueda. Vol. 6461. Lecture Notes in Computer Science. Springer, 2010, pp. 304–311. DOI: 10.1007/978-3-642-17164-2_21 (cited on pages 55, 90, 120).

[120] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. "Leveraging Rust Types for Modular Specification and Verification". In: *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. Vol. 3. OOPSLA. 2019, 147:1–147:30 (cited on page 55).

[121] Lukas Armborst, Pieter Bos, Lars B. van den Haak, Marieke Huisman, Robert Rubbens, Ömer Şakar, and Philip Tasche. "The VerCors Verifier: A Progress Report". In: *Computer Aided Verification*. Ed. by Arie Gurfinkel and Vijay Ganesh. Cham: Springer Nature Switzerland, 2024, pp. 3–18 (cited on pages 55, 90, 92, 110, 112, 120).

[122] Lea Salome Brugger, Xavier Denis, and Peter Müller. *Toward Practical Deductive Verification: Insights from a Qualitative Survey in Industry and Academia*. 2026. URL: https://arxiv.org/abs/2510.20514 (cited on pages 55, 82).

[123] Nico Berling. *Whitespace-dependent bug: "Assertion unknown might not hold"*. https://github.com/viperproject/gobra/issues/418. 2022 (cited on page 56).

[124] Remy Willems. *Verification behavior changes when making unrelated changes*. https://github.com/dafny-lang/dafny/issues/1585. 2021 (cited on page 56).

[125] Kiran Gopinathan, Dionysios Spiliopoulos, Vikram Goyal, Peter Müller, Markus Püschel, and Ilya Sergey. "Accelerating Automated Program Verifiers by Automatic Proof Localization". In: *Computer Aided Verification*. Ed. by Ruzica Piskac and Zvonimir Rakamarić. Cham: Springer Nature Switzerland, 2025, pp. 153–174 (cited on pages 56, 84).

[126] Yi Zhou, Jay Bosamiya, Jessica Li, Marijn Heule, and Bryan Parno. "Context Pruning for More Robust SMT-based Program Verification". In: *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD) Conference*. Oct. 2024 (cited on pages 56, 81, 85).

[127] *Viper Tutorial*. https://viper.ethz.ch/tutorial/ (cited on page 57).

[128] Marco Eilers, Malte Schwerhoff, and Peter Müller. "Verification Algorithms for Automated Separation Logic Verifiers". In: *Computer Aided Verification*. Ed. by Arie Gurfinkel and Vijay Ganesh. Cham: Springer Nature Switzerland, 2024, pp. 362–386 (cited on pages 58, 59, 72, 84, 124).

[129] Vytautas Astrauskas. "Leveraging Uniqueness for Modular Verification of Heap-Manipulating Programs". PhD thesis. ETH Zurich, 2024 (cited on page 59).

[130] Michał Moskal. "Programming with triggers". In: *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. SMT '09. Montreal, Canada: Association for Computing Machinery, 2009, pp. 20–29. DOI: 10.1145/1670412.1670416 (cited on pages 60, 82).

[131] K. R. M. Leino and Clément Pit-Claudel. "Trigger Selection Strategies to Stabilize Program Verifiers". In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Cham: Springer International Publishing, 2016, pp. 361–381 (cited on page 60).

[132] Felix A. Wolf. "Automated Verification of Advanced Correctness and Security Properties". PhD Thesis. ETH Zurich, 2024 (cited on pages 61, 63).

[133] Fabian Bösiger. "Performance Improvements of a Program Verifier". Bachelor's Thesis. ETH Zurich, 2021 (cited on page 68).

[134] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph Wintersteiger. *Programming Z3 - Incrementality*. https://theory.stanford.edu/~nikolaj/programmingz3.html#sec-incrementality (cited on page 78).

[135] Jonáš Fiala and Peter Müller. "SMTScope: Automated and Efficient Analysis of SMT Traces". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Sebastian Junges and Guy Katz. Cham: Springer Nature Switzerland, 2026, pp. 233–251 (cited on pages 79, 82).

[136] Nikolaj Bjørner, Clemens Eisenhofer, Arie Gurfinkel, Nuno P. Lopes, Leonardo de Moura, Lev Nachmanson, and Christoph Wintersteiger. *Z3 Internals (Draft) - Algebraic Datatypes*. https://z3prover.github.io/papers/z3internals.html#sec-algebraic-datatypes (cited on page 80).

[137] Scott McPeak. *Infinite loop making model for recursive datatype in presence of quantifiers*. https://github.com/Z3Prover/z3/issues/578 (cited on page 80).

[138] *Silicon's axiomatization of maps*. https://github.com/viperproject/silicon/blob/59339ac7bd1e40171ec2101cadbc src/main/resources/dafny_axioms/maps.vpr (cited on page 80).

[139] *Dafny Documentation - Verification optimization guide*. https://dafny.org/v4.5.0/VerificationOptimization/VerificationOptimization (cited on page 81).

[140] Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. "Mariposa: Measuring SMT Instability in Automated Program Verification". In: *2023 Formal Methods in Computer-Aided Design (FMCAD)*. 2023, pp. 178–188. DOI: 10.34727/2023/isbn.978-3-85448-060-0_26 (cited on pages 82, 84).

[141] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. "VCC: A Practical System for Verifying Concurrent C". In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 23–42 (cited on page 82).

[142] Nils Becker, Peter Müller, and Alexander J. Summers. "The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tomáš Vojnar and Lijun Zhang. Cham: Springer International Publishing, 2019, pp. 99–116 (cited on page 82).

[143] Mikael Mayer. *Making Verification Compelling: Visual Verification Feedback for Dafny*. https://dafny.org/blog/2023/04/19/making-verification-compelling-visual-verification-feedback-for-dafny. 2023 (cited on page 82).

[144] Linard Arquint. "Profiling Symbolic Execution". MA thesis. ETH Zurich, 2019 (cited on page 82).

[145] The dafny-lang community. *Dafny Reference Manual: assert ... by*. https://dafny.org/latest/DafnyRef/DafnyRef#sec-verification-debugging-assert-by (cited on page 83).

[146] The dafny-lang community. *Dafny Reference Manual: Labeling and revealing assertions*. https://dafny.org/latest/DafnyRef/DafnyRef#sec-labeling-revealing-assertions (cited on page 83).

[147] The dafny-lang community. *Dafny Reference Manual: Opaque Block*. https://dafny.org/latest/DafnyRef/DafnyRef#sec-opaque-block (cited on page 83).

[148] Tabea Bordis and Rustan Leino. "Free facts: An alternative to inefficient axioms in Dafny". In: (2024) (cited on pages 83, 85).

[149] 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 (Apr. 2023). DOI: 10.1145/3586037 (cited on page 83).

[150] Alexander Y. Bai, Chris Hawblitzel, and Andrea Lattuada. *Tunable Automation in Automated Program Verification*. 2025. URL: https://arxiv.org/abs/2512.03926 (cited on pages 83, 85).

[151] Claire Dross, Joffrey Huguet, and Johannes Kanig. *Managing the Proof Context in SPARK*. Presented at the Dafny Workshop 2026, co-located with POPL 2026. 2026 (cited on pages 83, 85).

[152] John Barnes. *Spark: The Proven Approach to High Integrity Software*. Altran Praxis, 2012 (cited on page 83).

[153] Kevin Bierhoff. "Automated program verification made SYMPLAR: symbolic permissions for lightweight automated reasoning". In: *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2011. Portland, Oregon, USA: Association for Computing Machinery, 2011, pp. 19–32. DOI: 10.1145/2048237.2048242 (cited on page 84).

[154] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. "Linear types for large-scale systems verification". In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (Apr. 2022). DOI: 10.1145/3527313 (cited on page 84).

[155] Lars B. van den Haak, Anton Wijs, and Marieke Huisman. "Scalable Deductive Verification of Data-Level Parallel Programs". In: *Computer Aided Verification*. Ed. by Eva Darulova, Anthony W. Lin, and Philipp Rümmer. Cham: Springer Nature Switzerland, 2026, pp. 90–112 (cited on page 84).

[156] K Rustan M Leino and Valentin Wüstholz. "Fine-grained caching of verification results". In: *International Conference on Computer Aided Verification*. Springer. 2015, pp. 380–397 (cited on page 84).

[157] Rustam Sadykov, Azat Abdullin, and Marat Akhin. *Cache-a-lot: Pushing the Limits of Unsatisfiable Core Reuse in SMT-Based Program Analysis*. 2025. URL: https://arxiv.org/abs/2504.07642 (cited on page 84).

[158] Can Cebeci, Nikolaj Bjørner, George Candea, and Clément Pit-Claudel. *A conjecture regarding SMT instability*. 2025. URL: https://dslab.epfl.ch/pubs/smt-instability-conjecture.pdf (cited on page 84).

[159] Daneshvar Amrollahi, Mathias Preiner, Aina Niemetz, Andrew Reynolds, Moses Charikar, Cesare Tinelli, and Clark Barrett. *Towards SMT Solver Stability via Input Normalization*. 2025. URL: https://arxiv.org/abs/2410.22419 (cited on page 85).

[160] Aaron Tomb and Jean-Baptiste Tristan. *Avoiding verification brittleness in Dafny*. https://dafny.org/blog/2023/12/01/avoiding-verification-brittleness/. 2023 (cited on page 85).

[161] Yi Zhou, Amar Shah, Zhengyao Lin, Marijn Heule, and Bryan Parno. "Cazamariposas: Automated Instability Debugging in SMT-Based Program Verification". In: *Automated Deduction – CADE 30: 30th International Conference on Automated Deduction, Stuttgart, Germany, July 28-31, 2025, Proceedings*. Stuttgart, Germany: Springer-Verlag, 2025, pp. 75–94. DOI: 10.1007/978-3-031-99984-0_5 (cited on page 85).

[162] Vladimir Gladshtein, Vitaly Kurin, Yueyang Feng, Dipesh Kafle, George Pîrlea, Qiyuan Zhao, and Ilya Sergey. "Velvet: A Foundational Multi-Modal Verifier for Imperative Programs in Lean". In: *Computer Aided Verification, CAV 2026*. 2026 (cited on pages 86, 124).

[163] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 625–635. DOI: 10.1007/978-3-030-79876-5_37 (cited on page 86).

[164] Markus Limbeck. "Enhancing Deductive Verification of Unbounded Heap Data Structures". MA thesis. ETH Zurich, 2025 (cited on page 86).

[165] João Pereira, Isaac van Bakel, Patricia Firlejczyk, Marco Eilers, and Peter Müller. *Artifact for "Modular Reasoning about Global Variables and Their Initialization"*. 2025. DOI: 10.5281/zenodo.15756716 (cited on page 89).

[166] *OpenJDK's implementation of Java's Byte class*. 2024. URL: https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/lang/Byte.java (cited on page 89).

[167] *Java™ Platform, Standard Edition 8 API Specification - Class Logger*. 2024. URL: https://docs.oracle.com/javase/8/docs/api/java/util/logging/Logger.html (cited on page 89).

[168] *Crate `lazy_static`*. 2023. URL: https://docs.rs/lazy_static/latest/lazy_static/ (cited on page 89).

[169] *The Go Programming Language Specification, Package initialization*. 2025. URL: https://go.dev/ref/spec%5C#Program_initialization (cited on pages 90, 113).

[170] *Java Language Specification. Chapter 12: Execution*. 2024. URL: https://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html (cited on page 90).

[171] *Static Constructors (C# Programming Guide)*. 2024. URL: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-constructors (cited on page 90).

[172] Egon Börger and Wolfram Schulte. "Initialization problems for Java". In: *Software-Concepts & Tools* 19 (2000), pp. 175–178. DOI: 10.1007/s003789900003 (cited on page 90).

[173] Fengyun Liu, Ondřej Lhoták, David Hua, and Enze Xing. "Initializing Global Objects: Time and Order". In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). DOI: 10.1145/3622844 (cited on pages 90, 115, 121).

[174] K. Rustan M. Leino and Peter Müller. "Modular Verification of Static Class Invariants". In: *FM 2005: Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 26–42. DOI: 10.1007/11526841_4 (cited on pages 90, 115, 120).

[175] A. J. Summers, S. Drossopoulou, and P. Müller. "Universe-Type-Based Verification Techniques for Mutable Static Fields and Methods". In: *Journal of Object Technology (JOT)* 8.4 (June 2009). DOI: 10.5381/jot.2009.8.4.a4 (cited on pages 90, 120).

[176] Werner Dietl and Peter Müller. "Universes: Lightweight Ownership for JML". In: *J. Object Technol.* 4.8 (2005), pp. 5–32. DOI: 10.5381/JOT.2005.4.8.A1 (cited on page 90).

[177] Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. "A Simple Sequential Reasoning Approach for Sound Modular Verification of Mainstream Multithreaded Programs". In: *Electronic Notes in Theoretical Computer Science* 174.9 (2007). Proceedings of the Thread Verification Workshop (TV 2006), pp. 23–47. DOI: 10.1016/j.entcs.2007.04.005 (cited on pages 90, 120).

[178] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, eds. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. Lecture Notes in Computer Science. Springer, 2016 (cited on pages 90, 120).

[179] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992 (cited on page 94).

[180] K. Rustan M. Leino. *Program Proofs*. MIT Press, 2023 (cited on page 98).

[181] Robbert Krebbers, Amin Timany, and Lars Birkedal. "Interactive proofs in higher-order concurrent separation logic". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '17. Paris, France: Association for Computing Machinery, 2017, pp. 205–217. DOI: 10.1145/3009837.3009855 (cited on page 101).

[182] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. "MoSeL: a general, extensible modal framework for interactive proofs in separation logic". In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018). DOI: 10.1145/3236772 (cited on page 101).

[183] The Rocq Development Team. *The Rocq Prover Reference Manual – Release 9.0.0*. https://rocq-prover.org/doc/V9.0.0/refman. 2025 (cited on page 101).

[184] *HeapLang overview*. 2025. URL: https://gitlab.mpi-sws.org/iris/iris/-/blob/master/docs/heap_lang.md (cited on page 103).

[185] Albert R. Meyer and Adam Chlipala. "Partial Orders". In: *Mathematics for Computer Science—MIT Course No. 6.042J*. Accessed March 25, 2025. MIT OpenCourseWare: Massachusetts Institute of Technology, 2015, pp. 336–338 (cited on page 106).

[186] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. "The Spirit of Ghost Code". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 1–16. DOI: 10.1007/978-3-319-08867-9_1 (cited on page 112).

[187] *Static Initialization Order Fiasco*. 2023. URL: https://en.cppreference.com/w/cpp/language/siof (cited on page 113).

[188] *cmd/compile: global variable initialization done in unexpected order*. 2022. URL: https://github.com/golang/go/issues/51913 (cited on page 114).

[189] João Pereira. *Refinement with Guards and Locally Inductive Invariants (LIIs)*. https://github.com/viperproject/gobra-libs/pull/37. 2026 (cited on page 117).

[190] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. "The future is ours: prophecy variables in separation logic". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371113 (cited on page 119).

[191] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. "The KRAKATOA tool for certificationof Java/Java Card programs annotated in JML". In: *J. Log. Algebraic Methods Program.* 58.1-2 (2004), pp. 89–106. DOI: 10.1016/J.JLAP.2003.07.006 (cited on page 120).

[192] David R. Cok. "OpenJML: JML for Java 7 by Extending OpenJDK". In: *NASA Formal Methods*. Ed. by Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 472–479. DOI: 10.1007/978-3-642-20398-5_35 (cited on page 120).

[193] *JML Reference Manual - Type Specifications*. 2013. URL: https://www.cs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_8.html%5C#SEC68 (cited on page 120).

[194] Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. "An overview of the Leon verification system: verification by translation to recursive functions". In: *Proceedings of the 4th Workshop on Scala*. SCALA '13. Montpellier, France: Association for Computing Machinery, 2013. DOI: 10.1145/2489837.2489838 (cited on page 120).

[195] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. "Modular invariants for layered object structures". In: *Sci. Comput. Program.* 62.3 (Oct. 2006). DOI: 10.1016/j.scico.2006.03.001 (cited on page 120).

[196] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. "Verification of Object-Oriented Programs with Invariants". In: *Journal of Object Technology* 3.6 (June 2004). Workshop on Formal Techniques for Java-like Programs (FTfJP), ECOOP 2003, pp. 27–56. DOI: 10.5381/jot.2004.3.6.a2 (cited on page 120).

[197] K. Rustan M. Leino and Peter Müller. "Object Invariants in Dynamic Contexts". In: *ECOOP 2004 – Object-Oriented Programming*. 2004. DOI: 10.1007/978-3-540-24851-4_22 (cited on page 120).

[198] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. "Local Reasoning for Storable Locks and Threads". In: *Programming Languages and Systems*. Ed. by Zhong Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 19–37. DOI: 10.1007/978-3-540-76637-7_3 (cited on page 121).

[199] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. "Oracle semantics for concurrent separation logic". In: *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems*. ESOP'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 353–367. DOI: 10.1007/978-3-540-78739-6_27 (cited on page 121).

[200] Alexandre Buisse, Lars Birkedal, and Kristian Støvring. "Step-Indexed Kripke Model of Separation Logic for Storable Locks". In: *Electronic Notes in Theoretical Computer Science* 276 (2011). Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII), pp. 121–143. DOI: 10.1016/j.entcs.2011.09.018 (cited on page 121).

[201] Jules Villard, Étienne Lozes, and Cristiano Calcagno. "Proving Copyless Message Passing". In: *Programming Languages and Systems*. Ed. by Zhenjiang Hu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 194–209. DOI: 10.1007/978-3-642-10672-9_15 (cited on page 121).

[202] K. Rustan M. Leino, Peter Müller, and Jan Smans. "Deadlock-Free Channels and Locks". In: *Programming Languages and Systems*. Ed. by Andrew D. Gordon. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 407–426. DOI: 10.1007/978-3-642-11957-6_22 (cited on page 121).

[203] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. "Actris: session-type based reasoning in separation logic". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371074 (cited on page 121).

[204] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. "Boogie: A modular reusable verifier for object-oriented programs". In: *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4*. Springer. 2006, pp. 364–387. DOI: 10.1007/11804192_17 (cited on page 121).

[205] *The Boogie IVL Language Reference - Global Variables*. 2023. URL: https://boogie-docs.readthedocs.io/en/latest/LangRef.html%5C#global-variables (cited on page 121).

[206] Alexander J. Summers and Peter Mueller. "Freedom before commitment: a lightweight type system for object initialisation". In: *SIGPLAN Not.* 46.10 (Oct. 2011), pp. 1013–1032. DOI: 10.1145/2076021.2048142 (cited on page 121).

[207] Fengyun Liu, Ondřej Lhoták, Aggelos Biboudis, Paolo G. Giarrusso, and Martin Odersky. "A type-and-effect system for object initialization". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428243 (cited on page 121).

[208] Clément Blaudeau and Fengyun Liu. "A conceptual framework for safe object initialization: a principled and mechanized soundness proof of the Celsius model". In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (Oct. 2022). DOI: 10.1145/3563314 (cited on page 121).

[209] Daniel Matichuk, Toby Murray, June Andronick, Ross Jeffery, Gerwin Klein, and Mark Staples. "Empirical Study Towards a Leading Indicator for Cost of Formal Software Verification". In: *International Conference on Software Engineering*. Firenze, Italy, Feb. 2015 (cited on page 124).

[210] Mark Staples, Ross Jeffery, June Andronick, Toby Murray, Gerwin Klein, and Rafal Kolanski. "Productivity for proof engineering". In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '14. Torino, Italy: Association for Computing Machinery, 2014. DOI: 10.1145/2652524.2652551 (cited on page 124).

[211] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. "Towards AI-Assisted Synthesis of Verified Dafny Methods". In: *Proc. ACM Softw. Eng.* 1.FSE (July 2024). DOI: 10.1145/3643763 (cited on page 124).

[212] Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Zhou Yuanyuan. "Laurel: Unblocking Automated Verification with Large Language Models". In: *Proc. ACM Program. Lang.* 9.OOPSLA1 (Apr. 2025). DOI: 10.1145/3720499 (cited on page 124).

[213] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. *Clover: Closed-Loop Verifiable Code Generation*. 2024. URL: https://arxiv.org/abs/2310.17807 (cited on page 124).

[214] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. "AutoVerus: Automated Proof Generation for Rust Code". In: *Proc. ACM Program. Lang.* 9.OOPSLA2 (Oct. 2025). DOI: 10.1145/3763174 (cited on page 124).

[215] Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng Cheng, Fan Yang, Shuvendu K Lahiri, Tao Xie, and Lidong Zhou. *Automated Proof Generation for Rust Code via Self-Evolution*. 2026. URL: https://arxiv.org/abs/2410.15756 (cited on page 124).

[216] Sicheng Zhong, Jiading Zhu, Yifang Tian, and Xujie Si. *RAG-Verus: Repository-Level Program Verification with LLMs using Retrieval Augmented Generation*. 2025. URL: https://arxiv.org/abs/2502.05344 (cited on page 124).

[217] Yuwei Liu, Xinyi Wan, Yanhao Wang, Minghua Wang, Lin Huang, and Tao Wei. *KVerus: Scalable and Resilient Formal Verification Proof Generation for Rust Code*. 2026. URL: https://arxiv.org/abs/2605.03822 (cited on page 124).

[218] Talia Ringer. "Proof Repair". English. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; People - Simpson, Anne (1956- ; Last updated - 2025-06-18; SubjectsTermNotLitGenreText - Simpson, Anne (1956-. PhD thesis. 2021, p. 163 (cited on page 124).

[219] Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. "Mostly Automated Proof Repair for Verified Libraries". In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591221 (cited on page 124).

Glossary

AS Autonomous System. 23, 26, 27, 47

BGP Border Gateway Protocol. 23

CSL concurrent separation logic. 30

IDF implicit dynamic frames. 38, 48

LII locally inductive invariant. 117, 118

SIF secure information flow. 48, 81

TCB Trusted Computing Base. 44, 46, 49

VerifiedSCION Project with the goal of formally verifying the SCION dataplane.. i, iii, 3, 4, 6, 7, 23, 46, 55, 57, 60, 64–67, 72, 73, 75, 77, 79, 81, 121

João C. Pereira

CONTACT INFORMATION

email: joao.pereira@inf.ethz.ch
github: github.com/jcp19

EDUCATION

2020–2026 **ETH Zurich**, Zürich, Switzerland
Doctoral student
- Developed tools and foundations for practical *formal verification* of concurrent and distributed systems.
- Contributed to *Gobra*, a deductive program verifier for *Go* based on *Separation Logic* and on the *Viper* verification infrastructure.
- Used Gobra to verify the reference implementation of the router for *SCION*, a new internet architecture for inter-AS routing.
- Advised by Prof. Peter Müller (Programming Methodology Group).

2014–2020 **University of Minho**, Braga, Portugal
Integrated Master's Degree in Informatics Engineering
- Average Grade: 18.28 out of 20
- Excellence Award (2016): highest average grade of all 1st-year students.
- Excellence Award (2017): highest average grade of all 2nd-year students.

Master's Thesis
Scalable Trace Analysis of Distributed Systems: Finding Data Races
Advised by Prof. Jorge Sousa Pinto

CONFERENCE PUBLICATIONS

1. **Protocols to Code: Formal Verification of a Secure Next-Generation Internet Router**.
João C. Pereira, Tobias Klenze, Sofia Giampietro, Markus Limbeck, Dionysios Spiliopoulos, Felix Wolf, Marco Eilers, Christoph Sprenger, David Basin, Peter Müller, Adrian Perrig in *Conference on Computer and Communications Security (CCS)* , 2025
Distinguished Artifact Award
2. **A Refinement Methodology for Distributed Programs in Rust**.
Aurea Bílá, João C. Pereira, Peter Müller in *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2025
Distinguished Artifact Award
3. **Modular Reasoning about Global Variables and Their Initialization**.
João C. Pereira, Isaac van Bakel, Patricia Firlejczyk, Marco Eilers, Peter Müller in *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2025
4. **Gobra: Modular Specification and Verification of Go Programs**.
Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, Peter Müller in *International Conference on Computer Aided Verification (CAV)*, 2021
5. **Testing for Race Conditions in Distributed Systems via SMT Solving**.
João C. Pereira, Nuno Machado, Jorge Sousa Pinto in *14th International Conference on Tests and Proofs (TAP)*, 2020

Book Chapters

1. **Code-Level Verification**.
   Linard Arquint, Peter Müller, Wytse Oortwijn, João C. Pereira, Felix A. Wolf in *The Complete Guide to SCION*. Information Security and Cryptography. Springer, Cham. 2022
2. **Current Status and Plans**.
   Linard Arquint, David Basin, Tobias Klenze, Si Liu, Peter Müller, João C. Pereira, Christoph Sprenger, Felix A. Wolf in *The Complete Guide to SCION*. Information Security and Cryptography. Springer, Cham. 2022

Professional Employment

Microsoft Research, Redmond, WA, USA.
Research Intern
June 2023 – August 2023.
- Developed techniques to translate C programs to *idiomatic* Safe Rust using LLMs. We used the Verus verifier to prove that the LLM translation matches the semantics of the original program.
- Mentor: Chris Hawblitzel

Amazon Web Services, New York, NY, USA.
Applied Scientist Intern in the Automated Reasoning Group.
February 2020 – April 2020.
- Worked on detecting AWS API misusages in clients using static analysis.
- Mentor: Martin Schäf

National Institute of Informatics, Tokyo, Japan.
Intern at the Programming Research Laboratory.
February 2019 – May 2019.
- Experimented with applying bidirectional transformations to re-add syntactic sugar to desugared code.
- Mentor: Prof. Zhenjiang Hu

INESC-Tec, Braga, Portugal.
Research Initiation Grant Holder at the High-Assurance Software Lab (HASLab).
April 2018 – September 2018.
- Worked on the detection of race conditions from traces of distributed systems.
- Mentor: Nuno Machado

INESC-Tec, Braga, Portugal.
Research Initiation Grant Holder at the High-Assurance Software Lab (HASLab).
February 2017 – July 2017.
- Looked for common error patterns in packages developed for the *Robot Operating System (ROS)* platform.
- Developed simple ROS applications and tested them in *TurtleBot*, a low-cost personal robot kit.
- Mentor: Prof. Alcino Cunha

Academic Activities

Service

- *CAV 2026*, Subreviewer
- *CAV 2021*, Artifact Evaluation Committee

Conference Volunteering

- Student Volunteer, *3rd World Congress on Formal Methods (FM 2019)*, Porto, Portugal
- Student Volunteer, *46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2019)*, Cascais, Portugal

Training Schools

- *Marktoberdorf Summer School (MOD) 2024*, Herrsching am Ammersee, Germany.
- *Oregon Programming Languages Summer School (OPLSS) 2024*, Boston, MA, USA.
- *Oregon Programming Languages Summer School (OPLSS) 2021*, Remote.
- *SAT/SMT/AR Summer School 2019*, Lisbon, Portugal.
- *Central-European Functional Programming School (CEFP) 2019*, Budapest, Hungary.
- *VMCAI Winter School 2019*, Lisbon, Portugal.
- *Multi-Paradigm Modelling for Cyber-Physical Systems (MPM4CPS) Training School 2018*, Pisa, Italy.
- *DeepSpec Summer School 2018*, Princeton, NJ, USA.

Teaching Experience

ETH Zurich, Zürich, Switzerland

- Head TA for the MSc class *Concepts of Object-Oriented Programming*. 2022 and 2023.
- TA for the MSc class *Concepts of Object-Oriented Programming*. 2021.
- TA for the BSc class *Formal Methods and Functional Programming*. 2021, 2022, and 2025.

University of Minho, Braga, Portugal

- TA for the BSc class *Software Development Laboratories III*. 2018.
  Subjects: Software Development in C and Java.

Supervised Students

1. Axel Montini, Practical Work project, *Verification of Mutex implmentation from the Go Standard Library*, 2025
2. Amos Herz, Practical Work project, *A Specification Linter for Gobra*, 2025
   Co-supervised with Lea Brugger.
3. Andrea Keusch, MSc thesis, *Analyzing Assumptions with Automated Verification Tools*, 2025
   Co-supervised with Lea Brugger. **Earned an ETH Medal for Outstanding Master's Thesis.**
4. Florian Marek, Internship, *An empirical assessment of proof strategies in large-scale verification projects*, 2025
5. Henri Inndorf, BSc thesis and internship, *Verifying Secure Information Flow for the VerifiedSCION Router*, 2025

6. Aaron Bojarski, Practical Work project, *Evolving the VerifiedSCION Codebase*, 2024
7. Ali Gottschall, BSc thesis, *The Gobra Book: Creating a Didactic Online Resource for Gobra*, 2024
8. Conradin Laux, BSc thesis, *Verifying Parts of the Go Standard Library*, 2024
9. Micha Buri, BSc thesis, *Translating Cryptographic Code from C to Rust using LLMs*, 2024
10. Patricia Firlejczyk, BSc thesis, *Specifying and Verifying Static Initialization in Deductive Program Verifiers*, 2024
    Co-supervised with Marco Eilers.
11. Daniel Nezamabadi, Practical Work project, *A Standard Library for Gobra*, 2024
12. Markus Limbeck, Internship and Practical Work project, *Verifying the IO Behaviour of the SCION Router*, 2023
    Co-supervised with Dionysios Spiliopoulos.
13. Liming Han, MSc thesis, *A Linear Typesystem for a Go Verifier*, 2023
    Co-supervised with Felix Wolf.
14. Benjamin Frei, BSc thesis, *Translating Pedagogical Verification Exercises to Viper*, 2023
    Co-supervised with Nicolas Klose.
15. Jan Schär, MSc thesis, *Proving Refinement in a Rust Verifier*, 2022
    Co-supervised with Aurea Bílá.
16. Gavin Gray, Internship, *Verifying memory safety of the SCION Router*, 2022
17. Adrian Jenny, Practical Work project, *Verifying Go's Standard Library*, 2022
18. Timon Egli, BSc thesis, *Translating Pedagogical Verification Exercises to Viper and its Front-ends*, 2022
19. Daneshvar Amrollahi, Internship, *Modularly Verifying and Specifying Initialization Code*, 2022
20. Lino Telschow, MSc thesis, *Specifying and Verifying the IO Behavior of the SCION Border Router*, 2021
    Co-supervised with Felix A. Wolf.
21. Johannes Gasser, BSc thesis, *Program Verification in Continuous Integration Workflows*, 2021
    Co-supervised with Linard Arquint.
22. Cheng Xuan, BSc thesis, *Verifying Termination of Go Programs*, 2021
    Co-supervised with Linard Arquint.
23. Dennis Buitendijk, BSc thesis, *Cloud-based Verification IDE*, 2021
    Co-supervised with Linard Arquint.
24. Luca Halm, BSc thesis, *Verification of Practical Go Programs*, 2021
    Co-supervised with Felix A. Wolf.

Other Activities

- Attended a *Mental Health First Aid* course provided by Ensa (`www.ensa.swiss`).
- Presented *Proving Programs Correct with Gobra* at the Zürich Gophers' Go Meetup XXI. The talk introduces Go developers to deductive program verification of Go programs using Gobra.
- Delegate at the *European Youth Parliament Academic Forum.* Committee on International Trade, discussed the viability of a Digital Single Market in the EU. Porto, Portugal, 2015.
- Participant at the *National Robotics Challenge 2014* (Robot Rescue Contest), Espinho, Portugal, 2014.