Automated Verification of Advanced Correctness and Security Properties

Felix Alexander Wolf

DISS. ETH. NO. 30619

DISS. ETH NO. 30619

Automated Verification of Advanced Correctness and Security Properties

A thesis submitted to attain the degree of

Doctor of Sciences
(Dr. sc. ETH Zurich)

presented by

Felix Alexander Wolf
MSc ETH CS, ETH Zurich
born on 02.01.1995

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner
Prof. Dr. David Basin, co-examiner
Prof. Dr. Bart Jacobs, co-examiner
Prof. Dr. Gidon Ernst, co-examiner

2024

Abstract

Software has become ubiquitous, ranging from apps for toasters to important infrastructure such as transportation, finance, and healthcare. Ensuring that software actually behaves as intended is critical in cases where implementation errors may lead to various forms of damage, such as financial losses, losses of private or confidential data, or even threats to human safety. To apply program verification to real-world programs a high degree of automation is vital. In this thesis, we expand the state-of-the-art in two areas where techniques for automated program verification remain challenging, namely the automated verification of modern logics with advanced correctness properties such as linearizability and the verification of language-agnostic information flow properties.

Modern separation logics allow one to prove rich properties of intricate code, *e.g.* functional correctness and linearizability of non-blocking concurrent code. However, this expressiveness leads to a complexity that makes these logics difficult to apply. Manual proofs or proofs in interactive theorem provers consist of a large number of steps, often with subtle side conditions. On the other hand, automation with dedicated verifiers typically requires sophisticated proof search algorithms that are specific to the given program logic, resulting in limited tool support that makes it difficult to experiment with program logics, *e.g.* when learning, improving, or comparing them.

In the context of secure information flow, security policies express the classification and declassification of data. Existing policy frameworks that support automated reasoning about code are tightly linked to a programming language, which limits their flexibility and complicates reasoning, for instance, during audits. Language-agnostic policy frameworks, which abstract the concrete behavior of programs, lack automated program verification.

First, we present Gobra, a modular, deductive program verifier for Go that proves memory safety, crash safety, data-race freedom, and user-provided specifications. Gobra is based on separation logic and supports a large subset of Go. Its implementation translates an annotated Go program into the Viper intermediate verification language and uses an existing SMT-based verification backend to compute and discharge proof obligations. We build our verification techniques for security properties on top of Gobra to apply them to a practical, real-world programming language.

Second, we present a framework for the specification and verification of security policies for distributed systems, where attackers may observe the I/O performed by a program, but not its memory. Our policies are expressed over the I/O behaviors of programs and, thereby, language-agnostic. We present techniques to reason formally about policies, and to verify that an implementation satisfies a given policy. We formalize these verification techniques in Isabelle/HOL. An evaluation on several case studies, including an implementation of the WireGuard VPN key exchange protocol, demonstrates that our policies are expressive, and that verification is amenable to SMT-based verification.

Third, we systematically develop a proof outline checker for the TaDA logic, a complex program logic for the verification of block-free code. Proof outline checkers take as input a program annotated with the most essential proof steps and then check automatically that this outline represents a valid proof in the program logic. Our proof outline checker reduces the verification of the TaDA logic to a simpler verification problem, for which automated tools exist. Our approach leads to proof outline checkers that provide substantially more automation than interactive provers, but are much simpler to develop than custom automatic verifiers.

Zusammenfassung

Software ist mittlerweile allgegenwärtig und reicht von Apps für Toaster bis hin zu wichtigen Infrastrukturen für Verkehr, Finanzen, und Gesundheitswesen. Es ist wichtig sicherzustellen, dass sich Software tatsächlich wie beabsichtigt verhält, vor allem in Fällen, in denen Implementierungsfehler zu verschiedenen Formen von Schäden führen können, wie zum Beispiel finanziellen Verlusten, dem Verlust privater oder vertraulicher Daten, oder sogar Gefahren für die menschliche Sicherheit. Ein hoher Automatisierungsgrad ist für die Anwendung der Programmverifikation auf reale Programme unerlässlich. In dieser Arbeit erweitern wir den Stand der Technik in zwei Bereichen, in welchen Techniken zur automatisierten Programmverifikation nach wie vor eine Herausforderung darstellen. Diese Bereiche sind die automatisierte Verifikation moderner Logiken mit fortgeschrittenen Korrektheitseigenschaften wie Linearisierbarkeit und die Verifikation von programmiersprachunabhängigen Informationsflusseigenschaften.

Moderne Separationslogiken ermöglichen die Verifikation umfangreicher Eigenschaften von kompliziertem Code, zum Beispiel die funktionale Korrektheit und die Linearisierbarkeit von nicht blockierendem, nebenläufigem Code. Diese Ausdruckskraft führt jedoch zu einer Komplexität, die die Anwendung dieser Logiken erschwert. Manuelle Beweise oder Beweise in interaktiven Theorembeweisern bestehen aus einer großen Anzahl von Schritten, oft mit subtilen Nebenbedingungen. Andererseits erfordert die Automatisierung mit dedizierten Verifizierern in der Regel hochentwickelte Algorithmen zur Beweissuche, die spezifisch für die gegebene Programmlogik sind. Das führt zu einer begrenzten automatisierten Unterstützung, die Experimente erschwert, zum Beispiel beim Lernen, Verbessern, oder Evaluieren einer Logik.

Im Kontext des sicheren Informationsflusses definieren Sicherheitsrichtlinien die Klassifizierung und gültige Deklassifizierung von Daten. Bestehende Frameworks für Sicherheitsrichtlinien, die automatisierte Verifikation von Code unterstützen, sind eng mit einer Programmiersprache verknüpft, was ihre Flexibilität einschränkt und die Validierung von Richtlinien erschwert. Sprachunabhängige Frameworks, die das konkrete Verhalten von Programmen abstrahieren, verfügen nicht über eine automatische Programmverifikation.

Als erstes präsentieren wir Gobra, ein modularer, deduktiver Programmverifizierer für Go, der Memory Safety, Crash Safety, Data-Race-Freedom, und vom Benutzer definierte Spezifikationen beweist. Gobra basiert auf Separationslogik und unterstützt eine große Teilmenge von Go. Seine Implementierung übersetzt ein annotiertes Go Programm in die Zwischenverifikationssprache Viper und verwendet ein bestehendes SMT-basiertes Verifikations-Backend, um Beweisverpflichtungen zu berechnen und zu überprüfen. Wir bauen unsere Verifikationstechniken für Sicherheitseigenschaften auf Gobra auf, um unsere Techniken auf eine praktische, reale Programmiersprache anzuwenden.

Zweitens stellen wir ein Framework für die Spezifikation und Verifikation von Sicherheitsrichtlinien für verteilte Systeme vor, bei denen Angreifer zwar die von einem Programm ausgeführten I/O Operationen beobachten können, nicht aber den Speicher eines Programms beobachten können. Unsere Richtlinien werden über das I/O-Verhalten von Programmen ausgedrückt und sind somit sprachunabhängig. Wir stellen Techniken vor, um formal über Richtlinien zu argumentieren und um zu überprüfen, ob eine Implementierung eine gegebene Richtlinie erfüllt. Wir formalisieren diese Verifikationstechniken in Isabelle/HOL. Eine Evaluierung mehrerer Fallstudien, einschließlich einer Implementierung des WireGuard VPN-Schlüsselaustauschprotokolls, zeigt, dass unsere Richtlinien aussagekräftig sind und dass die Verifikation zugänglich für SMT-basierte Verifikation ist.

Drittens entwickeln wir systematisch einen Proof Outline Checker für die TaDA-Logik, eine komplexe Programmlogik für die Verifikation von blockfreiem Code. Proof Outline Checker nehmen als Eingabe ein mit den wichtigsten Beweisschritten annotiertes Programm und prüfen dann automatisch, ob diese Schritte einen gültigen Beweis in der Programmlogik darstellen. Unser Proof Outline Checker reduziert die Verifikation der TaDA-Logik auf ein einfacheres Verifikationsproblem, für das automatisierte Verifizierer existieren. Unser Ansatz führt zu Proof Outline Checkern, die wesentlich mehr Automatisierung bieten als interaktive Beweiser, aber einfacher zu entwickeln sind als benutzerdefinierte automatische Verifizierer.

Acknowledgements

There have been three pivotal experiences in my life that have majorly shaped my decision to pursue a doctorate. The first experience was the movie "Terminator 3". The movie's profound message had inspired me to study computer science. The second experience was my bachelor's formal methods course. There I gained the enlightenment that program verification is the coolest research field in the history of humankind. The last experience was a panel discussion at one of ETH's job fairs. While only a minority of the panelists advocated for a doctorate, everyone agreed that if one is considering a doctorate at any point in their life, then now is the right time to do so. Of course, there are many more experiences that I could list. However, instead of the past, I want to dedicate the following paragraphs to the people who have shaped and twisted my path through the doctorate.

I start with my supervisor, Peter Müller. Despite leading such a large research group, he always managed to devote his time to us. He showed me the qualities of good research and taught me the difference between academic text and a soup of words. I feel lucky to have had such a great supervisor. Many thanks also to my co-examiners David Basin, Bart Jacobs, and Gidon Ernst. I appreciate that such great researchers have taken the time to review and evaluate my thesis.

Next, I want to mention all my precious coworkers and collaborators. I start with the Gobra team, which includes Linard Arquint, João Pereira, Dionisios Spiliopoulos, as well as the past members Wytse Oortwijn and Martin Clochard. Before my doctorate, I had the goal to create a beautiful program verifier that is involved in many exciting projects. I can confidently say that we have achieved this goal together, admittedly once in a while missing the beauty. A lot of my time has been involved with the Verified Scion project, where I collaborated, among other people, with Christoph Sprenger, Ralf Sasse, Joseph Lallemand, Tobias Klenze, Sofia Giampietro, Adrian Perrig, Markus Limbeck, and Sven Wiesner. It was a valuable experience to be part of such a large project, involving so many different research groups, and exchanging thoughts with people from other research areas. Another endeavor has been my role as head teaching assistant for the Introduction to Programming course. Thomas Gross, Remi Meier, Michael Faes, and all later members have been a great team. A doctorate at ETH is long, so I want to thank all my seniors, Arshavir Ter-Gabrielyan, Alexandra Bugariu, Marco Eilers, Jérôme Dohrau, Vytautas Astrauskas, and Federico Poli, who have shown me, although a bit delayed, that a doctorate does in fact end eventually. Our postdocs, Alex Summers, Caterina Urban, Malte Schwerhoff, Wytse Oortwijn, Christoph Matheja, Martin Clochard, Xavier Denis, Michael Sammler, and Marco Eilers have helped me find my footing in the academic microcosm. I explicitly thank Malte for helping me join the research group and for supporting my first steps in academia. I also thank the venerable sage of research, Marco, for always being willing to listen to my shenanigans. Gaurav Parthasarathy has been a wonderful officemate. He has been the best conversation partner and his delicate play of the keyboard has always been a pleasure to indulge in. I also enjoyed the discussions with my office-neighbor Thibault Dardinier, despite his incorrect opinions about secure information flow. At some point being the senior myself, it has been a pleasant experience seeing our junior researchers grow, including Nicolas Klose, Anqi Li, Jonás Fiala, Aurel Bílý, Lea Brugger, Hongyi Ling, and Yushuo Xiao. I also want to mention the members of the institute's new research group, namely Ralf Jung, Isaac van Bakel, Johannes Hostert, and Max Vistrup. I am sure that their radical ideas about formal verification will be a positive influence in the future.

I thank Marlies Weissert, Sandra Schneider, Bernadette Gianesi, and Denise Spicher for being a great support for bureaucratic matters.

All my coworkers have taught me a plethora of life lessons, some of which I want to honor explicitly. I thank Vytautas for teaching me that sometimes no answer is the best answer. I thank João for teaching me how to program. I thank Marco for his many bits of wisdom, including the benefits of a healthy work routine and the importance of focused work.

Lastly, I thank my friends and family. I want to especially thank my parents, Karin Taube and Manfred Wolf, who have always spared no effort to support me throughout my studies. Their dog is also ok.

Contents

1. Introduction

"I intend to live forever. So far, so good."

— Stephen Wright

Software has become ubiquitous, ranging from apps for toasters to important infrastructure such as transportation, finance, and healthcare. Ensuring that software actually behaves as intended is critical in cases where implementation errors may lead to various forms of damage, such as financial losses, losses of private or confidential data, or even threats to human safety. Traditional techniques for quality assurance such as testing or bug finders cannot guarantee the absence of bugs since these techniques are only able to reason about a limited number of program executions. In contrast, program verification aims to formally ensure the correctness of software for all its executions, *i.e.* for all inputs, all thread schedules, and all interactions with the environment.

To apply program verification to real-world code, the automation of program verification is a crucial requirement. *Deductive program verifiers* take a program together with a specification describing the intended behavior and then verify whether the program satisfies the given specification. To verify programs, such tools typically require hints by the user in the form of proof annotations added to a program or interactive clues. The amount of required proof annotations or clues then determines the degree of automation of the verification technique. Without a sufficiently high degree of automation, verifying large-scale or complex software systems is intractable due to the amount of work and difficulty of the task.

The focus of this thesis is on advancing the state-of-the-art in terms of the kind of code and the kind of properties that can be verified with automated verification.

Regarding the targeted code, our work focuses on the verification of code with fine-grained concurrency, where threads, instead of locks, use atomic language primitives such as `compare-and-swap` to interact with memory. Verifying rich properties about code with fine-grained concurrency requires the verification of advanced correctness properties about the code: Consider a small program `z = *x; *x = z + 1` that first reads the value of some pointer `x` into a temporal variable `z` and then increases the value of the pointer by one. If the program is executed sequentially, then we are able to verify that if the pointer `x` was not null before the program execution, then after the program the value of the pointer has been increased by one. However, for concurrent programs, the specification does not hold if other treads are able to change the pointer value between the read `z = *x` and write `*x = z + 1` of the program. To recover the ability to verify strong specifications, program logics specialized for fine-grained concurrency introduce advanced correctness properties such as linearizability. These specialized logics are very expressive, but challenging to apply and automate because they often comprise complex proof state and many complex proof rules.

Regarding the targeted properties, in addition to the advanced correctness properties that are necessary to verify strong specifications for code with fine-grained concurrency, our work focuses on the verification of advanced security properties about code, more specifically, on information flow properties. Information flow properties specify how specific data

may influence the behavior of a program. Flow properties have a variety of applications: In the context of confidentiality, flow properties are used to specify that information about confidential data may not influence the publicly observable behavior of a program. For integrity, we are able to specify that untrusted data does not taint trusted data, which is used to reason about attack vectors such as injection attacks. Lastly, different variations of access control specify who is permitted to access specific data. Note that this is not an exhaustive list. A limitation of modern verification approaches for information flow is that their guarantees are tightly linked to a programming language. As a consequence, reasoning about the guarantees established by code verification happens at the level of the programming language and, thereby, faces the full complexity of the language. This poses a challenge when aiming to reason about real-world programs that are written in established programming languages with a plethora of complex language features. Reasoning about the guarantees established by code verification is important, especially for security, where the specification of information flow properties can become large and we want to ensure, potentially with formal audits, that the verified specification correctly reflects the intended security requirements for an application. In contrast to flow properties that are tied to a programming language, which we refer to *language-based* flow properties, *system-based* flow properties express language-agnostic specifications of security. However, in contrast to language-based flow properties, automated verification techniques to verify system-based flow properties have not been sufficiently explored.

1.1. State of the Art

Deductive program verifiers achieve automation in several different ways. SMT-based program verifiers take programs that are annotated with proof statements together with a specification of the intended behavior and then produce either a single or set of SMT formulas such that if the SMT formulas are satisfied, then the program satisfies the given specification. These SMT formulas are then checked with high-performant SMT solvers such as Z3 [1] or CVC5 [2]. Instead of a tool producing these SMT formulas itself, several works [3–6] propose the use of *intermediate verification languages (IVLs)* to simplify the creation of program verifiers. IVLs offer basic programming languages together with verification backends for their language. Analogous to the encoding into SMT formulas, program verifiers then encode their input into an IVL program such that if the IVL program verifies successfully (using the verification backend provided by the IVL), then the input program satisfies its given specification. More recently, program verifiers based on interactive theorem provers such as Coq [7] have achieved a high degree of automation by developing sophisticated proof search algorithms [8–10]. These proof search algorithms, often referred to as tactics, help to construct formal proofs by automatically inferring the proof rules that are necessary to reduce a proof goal into smaller sub-goals. An advantage of using interactive theorem provers is that verification does not have to trust the correctness of SMT solvers.

These advancements in the automation of program verifiers have made

[3]: Barnett et al. (2006), *Boogie: A Modular Reusable Verifier for Object-oriented Programs*
[4]: Filliâtre et al. (2013), *Why3 - Where Programs Meet Provers*
[5]: Müller et al. (2016), *Viper: A Verification Infrastructure for Permission-Based Reasoning*
[6]: Maksimovic et al. (2021), *Gillian, Part II: Real-World Verification for JavaScript and C*
[7]: consortium (n.d.), *The Coq proof assistant*
[8]: Sammler et al. (2021), *RefinedC: automating the foundational verification of C code with refined ownership types*
[9]: Gäher et al. (2024), *RefinedRust: A Type System for High-Assurance Verification of Rust Programs*
[10]: Mulder et al. (2022), *Diaframe: automated verification of fine-grained concurrent programs in Iris*

it possible to develop program verifiers for real-world programming languages. Examples are Verifast [11], Krakatoa [12], Frama-C [13], SecC [14], VCC [15], RefinedC [8], and VerCors [16] for C and Java; Spec# [17] for C#, Nagini [18] for Python; and Prusti [19], Creusot [20], and RefinedRust [9] for Rust. With the exception of Verifast, SecC, RefinedC, and RefinedRust, all of these program verifies are built on top of an IVL.

Verifying that a program satisfies an information flow property typically requires reasoning about multiple executions of the program since implicit flows of information may not be observable from a single execution. Properties involving multiple executions of a program are referred to as *hyperproperties* [21]. A successful approach for the automated verification of hyperproperties is the use of *product programs* [22, 23]. A product program construction composes a program with itself such that a single execution of the constructed product program corresponds to a fixed number of executions of the original properties. Hyperproperties are then verified by applying the standard automated verification techniques to the product program. So far, these automated verification techniques have been mostly used for the verification of language-based information flow properties. Existing work on system-based information flow properties typically verifies programs either through refinements [24] or unwinding techniques [25, 26], both of which are hard to automate.

[21]: Clarkson et al. (2008), *Hyperproperties*

[22]: Barthe et al. (2011), *Secure information flow by self-composition*

[23]: Barthe et al. (2011), *Relational Verification Using Product Programs*

[24]: Mantel (2003), *A uniform framework for the formal specification and verification of information flow security*

[25]: Goguen et al. (1984), *Unwinding and Inference Control*

[26]: Popescu et al. (2021), *Bounded-Deducibility Security (Invited Paper)*

Under the hood, deductive program verifiers justify that a program satisfies a given specification through the use of a program logic. Standard Hoare logic [27] proves specifications of the form $\{P\}c\{Q\}$, describing that if an execution of the program c starts in a program state s that satisfies the *precondition* P and ends in a program state s' after the execution, then the program state s' satisfies the *postcondition* Q. A successful approach to reason about memory and concurrency is to use *separation logic* [28]. In separation logic, each memory location is associated with a *permission*, guarding access to the memory location. Permissions are created when a memory location is allocated. In particular, permissions cannot be duplicated or forged. Permissions are held by method executions and transferred between methods upon call and return. Which permissions to transfer upon call and return is specified in the callee's method's pre- and postcondition, respectively. A method may access a memory location only if it holds the associated permissions. Since permissions are not duplicable and cannot be forged, separation logic ensures that accesses to memory locations do not cause segfaults, since permissions only exist for allocated memory locations, or data-races, since at most one thread has exclusive access to a memory location. Code that uses locks to synchronize accesses to memory is verified by defining *lock invariants*. A lock invariant specifies permissions that are conceptually held by a lock and defines functional properties that have to hold whenever the lock is not currently acquired. When a method acquires a lock, the permissions of the lock invariant are transferred to the method and the functional properties specified by the lock invariant may be assumed. Conversely, when a lock is released, the permissions are returned and the functional specification has to hold.

[27]: Hoare (1969), *An Axiomatic Basis for Computer Programming*

More recently, numerous separation logics have been proposed that enable the verification of fine-grained concurrency by incorporating ideas from concurrent separation logic, Owicki-Gries [29], and rely-guarantee [30]. Examples include CAP [31], iCAP [32], CaReSL [33],

[29]: Owicki et al. (1976), *An Axiomatic Proof Technique for Parallel Programs I*

[30]: Jones (1983), *Specification and Design of (Parallel) Programs*

CoLoSL [34], FCSL [35], Iris [36], GPS [37], RSL [38], and TaDA [39] (see Brookes et al. [40] for an overview). An important concept present in some of these logics is *abstract atomicity*. Instead of specifying the state when a method returns, abstract atomic specifications describe the state at the linearization point of a method. This technique enables the verification of richer functional specifications for fine-grained concurrent code since specifications about the linearization point do not have to account for the interference caused by other threads happening after the linearization point. This expressiveness comes at the price of more complex proof state and proof rules. While some automated verifiers for fine-grained concurrency logics have existed before our work, they usually have strong limitations. For instance, Caper [41] supports an improved version of CAP [31], a predecessor logic of TaDA. Caper achieves an impressive degree of automation, but it cannot verify abstract atomicity. SmallfootRG [42] can prove memory safety, but not functional correctness. CAVE [43] can prove linearizability, but cannot reason about non-linearizable code.

[40]: Brookes et al. (2016), *Concurrent separation logic*

[41]: Dinsdale-Young et al. (2017), *Caper - Automatic Verification for Fine-Grained Concurrency*

[31]: Dinsdale-Young et al. (2010), *Concurrent Abstract Predicates*

[42]: Calcagno et al. (2007), *Modular Safety Checking for Fine-Grained Concurrency*

[43]: Vafeiadis (2010), *Automatically Proving Linearizability*

As for fine-grained concurrency, a plethora of logics enable us to reason about secure information flow [14, 44–47]. A key challenge for the verification of concurrent code is that the thread scheduler may influence the behavior of a program. Therefore, data that influences the thread schedule, for instance, by having an effect on the execution time, may influence the rest of the program as well. In the context of secure information flow, this issue is typically addressed by preventing confidential data from influencing the execution time. Such influences are avoided by enforcing that executions do not branch on confidential data, *e.g.* the conditions of while- and if-statement may not depend on confidential data. Some works permit branches on secrets, if they are able to prove that both executions starting from the branch have an equal execution time. A more recent approach [46] is to verify that the results computed by a program are the same regardless of how threads are scheduled, entailing that the results are not influenced by the thread scheduler.

[14]: Ernst et al. (2019), *SecCSL: Security Concurrent Separation Logic*

[44]: Murray et al. (2018), *COVERN: A Logic for Compositional Verification of Information Flow Control*

[45]: Yan et al. (2021), *SecRSL: security separation logic for C11 release-acquire concurrency*

[46]: Eilers et al. (2023), *CommCSL: Proving Information Flow Security for Concurrent Programs using Abstract Commutativity*

[47]: Frumin et al. (2021), *Compositional Non-Interference for Fine-Grained Concurrent Programs*

[46]: Eilers et al. (2023), *CommCSL: Proving Information Flow Security for Concurrent Programs using Abstract Commutativity*

1.2. Challenges

The core goal of this thesis is to advance the state-of-the-art in terms of the programs and in terms of the properties that can be verified with automated verification techniques.

We tackle the following concrete challenges:

Challenge 1: Automation of Complex Logics. As discussed in Sec. 1.1, modern separation logics allow one to prove rich properties of intricate code, *e.g.* functional correctness and linearizability of non-blocking concurrent code. However, this expressiveness leads to a complexity that makes these logics difficult to apply. Automation with dedicated verifiers typically requires sophisticated proof search algorithms that are specific to the given program logic, resulting in a limited tool support and making it difficult to experiment with program logics. A systematic approach to automate the verification of such complex logics with advanced correctness properties remains challenging.

Challenge 2: Automation of System-Based Flow Properties. In contrast to language-based information flow properties, there are only a few modern works that focus on verifying that code satisfies system-based flow properties. Language-based approaches define and verify flow properties on top of a fixed programming language with a fixed language semantic. Conversely, system-based approaches define flow properties on a higher level, typically on top of some form of general traces that provide an abstract representation of program behavior. The abstract representation of program behavior introduced two challenges: (1) The loss of information caused by the abstraction of program behavior makes it challenging to define information flow properties that are expressive enough to support modern programs with concurrency and heap effects. For instance, decisions made by the thread scheduler are not visible at the level of the abstracted program behavior and, thereby, hard to distinguish from unintended flows of information. (2) Code verification has to bridge the abstraction gap between the abstract program behavior and the concrete operations performed by the code.

Existing system-based approaches are based on the notion of *non-deducibility* [48]. These approaches are able to handle point (1), but require the explicit construction of proof witnesses, which is hard to automate. An approach that solves point (1) and is amicable to automated verification remains challenging.

[48]: Sutherland (1986), *A model of information*

Challenge 3: Reasoning about Practical Programming Languages. The ultimate goal of program verification is to verify relevant real-world code. Logics introduced to reason about advanced correctness or security properties are typically defined for simple canonical programming language devoid of important features of practical programming languages such as exceptions, interfaces, and closures. Conversely, program verifiers that are able to verify practical programming languages such as Java or Rust typically only support the verification of less advanced properties. The aim of this challenge is the automated verification of advanced properties for programs written in practical languages.

1.3. Contributions and Outline

We address the challenges with the following contributions (in reverse order):

Contribution 1: Modular Verification of Go Code. In Chapter 2, we present Gobra, a modular, deductive program verifier for Go that proves memory safety, crash safety, data-race freedom, and user-provided specifications. Go is a popular systems programming language targeting, especially, concurrent and distributed systems. Go differentiates itself from other imperative languages by offering structural subtyping and lightweight concurrency through goroutines with message-passing communication. This combination of features poses interesting challenges for static verification, most prominently the combination of a mutable heap and advanced concurrency primitives. Gobra is based on separation logic and supports a large subset of Go. Its implementation translates an annotated Go program into the Viper intermediate verification language and uses an existing SMT-based verification backend to compute and

discharge proof obligations. Gobra is intended for the verification of substantial, real-world code, and was, for instance, used to verify the Go implementation of the SCION internet architecture [49, 50]. We build our verification techniques for security properties on top of Gobra to apply them to a practical, real-world programming language.

[49]: Zhang et al. (2011), *SCION: Scalability, control, and isolation on next-generation networks*
[50]: Pereira et al. (2024), *Protocols to Code: Formal Verification of a Next-Generation Internet Router*

Contribution 2: Verifiable Security Policies for Distributed Systems. In Chapter 3, we present a framework for the specification and verification of system-based security policies for distributed systems, where attackers may observe the I/O performed by a program, but not its memory. Security policies belong to the group of secure information flow hyperproperties, expressing that confidential data is not leaked during a program's execution. In this context, security policies classify the sensitivity of data and specify when *declassification*, *i.e.* the intentional release of information, is permitted. We achieve system-based security policies by expressing policies over traces of *I/O actions*, the basic building blocks of communication, such as sending or receiving a message. This language-independent representation is well-suited for distributed systems, where attackers observe the I/O behavior of a program, but not the content of the memory. To demonstrate the benefits of system-based properties, we introduce a technique to reason formally about the guarantees provided by a policy, enabling formal audits of security policies independent of a specific program implementation or language. For code verification, we introduce a technique to verify that an implementation satisfies a policy using a combination of established verification techniques that are well-suited for automation. We fully formalize our approach in the theorem solver Isabelle/HOL. We implement our code verification on top of Gobra to target real programs. An evaluation on several case studies, including an implementation of the WireGuard VPN key exchange protocol, demonstrates that our policies are expressive, and that code verification is amenable to automated verification.

Contribution 3: A Proof Outline Checker for TaDA. In Chapter 4, we systematically develop an automated *proof outline checker* for the TaDA logic [39]. Proof outline checkers take as input a *proof outline*, a formal proof skeleton that contains the key proof steps but omits most of the details, and then check automatically that it represents a valid proof in the program logic. Proof outline checkers provide automation for proof steps for which good proof search algorithms exist, and can, as we demonstrate, support expressive logics by requiring annotations for complex proof steps. Due to this flexibility, proof outline checkers are especially useful for experimenting with a logic. Our work goes beyond existing proof outline checkers and automated verifiers by supporting the substantially more complex program logic TaDA, which handles fine-grained concurrency, linearizability, abstract atomicity, and other advanced features. We believe that our systematic development of Voila generalizes to other complex logics.

[39]: Rocha Pinto et al. (2014), *TaDA: A Logic for Time and Data Abstraction*

1.4. Publications

The chapters of this thesis are based on the following publications:

Chapter 2 is based on

> *F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller.*
> *Gobra: Modular Specification and Verification of Go Programs*
> *In CAV 2021 [51]*

Chapter 3 is based on a paper that is currently under submission:

> *F. A. Wolf and P. Müller.*
> *Verifiable Security Policies for Distributed Systems*
> *In CCS 2024 [52]*

Chapter 4 is based on

> *F. A. Wolf, M. Schwerhoff, and P. Müller.*
> *Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA*
> *In FM 2021 [53]*

including the journal version of this paper

> *F. A. Wolf, M. Schwerhoff, and P. Müller.*
> *Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA*
> *In Formal Methods Syst. Des. 2022 [54]*

1.5. Further Contributions

During the thesis work, the following additional contributions to the scientific community have been made:

> *C. Sprenger, T. Klenze, M. Eilers, F. A. Wolf, P. Müller, M. Clochard, and D. A. Basin.*
> *Igloo: soundly linking compositional refinement and separation logic for distributed system verification*
> *In OOPSLA 2020 [55]*

> *L. Arquint, F. A. Wolf, J. Lallemand, R. Sasse, C. Sprenger, S. N. Wiesner, D. A. Basin, and P. Müller.*
> *Sound Verification of Security Protocols: From Design to Interoperable Implementations*
> *In SP 2023 [56]*

> *J. C. Pereira, T. Klenze, S. Giampietro, M. Limbeck, D. Spiliopoulos, F. A. Wolf, M. Eilers, C. Sprenger, D. A. Basin, P. Müller, and A. Perrig.*
> *Protocols to Code: Formal Verification of a Next-Generation Internet Router*
> *In CoRR 2024 [50]*

Gobra: Modular Specification and Verification of Go Programs

2.

"Beware of bugs in the above code; I have only proved it correct, not tried it."

— Donald Knuth

Go is an increasingly popular systems programming language targeting, especially, concurrent and distributed systems such as web applications. It combines standard features of imperative languages, such as mutable heap data structures, with less common concepts, such as structural subtyping and lightweight concurrency through goroutines with message-passing communication.

This combination of features poses interesting challenges for static verification, most prominently the combination of a mutable heap and advanced concurrency primitives. Prior research on Go verification handles some of these features, but not their combination. For instance, Lange et al. [57, 58] verify safety and liveness of Go's message-passing, but do not consider functional properties about the heap state, whereas Perennial [59] supports heap data structures, but neither channels nor interfaces.

[57]: Lange et al. (2017), *Fencing off Go: liveness and safety for channel-based programming*

[58]: Lange et al. (2018), *A static verification framework for message passing in Go using behavioural types*

[59]: Chajed et al. (2019), *Verifying concurrent, crash-safe systems with Perennial*

We present Gobra, an automated, modular verifier for heap-manipulating, concurrent Go programs. Gobra supports a large subset of Go, including Go's interfaces and primitive data structures, both of which have not been fully supported in previous work. Gobra verifies memory safety, crash safety, data-race freedom, and user-provided specifications. It takes as input a Go program annotated with assertions such as pre and postconditions and loop invariants. Verification proceeds by encoding the annotated programs into the intermediate verification language Viper [5] and then applying an existing SMT-based verifier. In case verification fails, Gobra reports at the level of the Go program which assertions it could not verify.

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

Gobra's assertion language builds on established concepts: Gobra uses separation logic style permissions [28] to reason locally about heap data structures. It supports recursive predicates and specification methods to abstract over (possibly unbounded) data structures and their contents. In particular, Gobra has first-class predicates that enable a natural specification of concurrency primitives, for instance, to parameterize a lock by an invariant.

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

Gobra is intended for the verification of substantial, real-world code, and was, for instance, used to verify the Go implementation of the SCION internet architecture [49, 50]. This chapter makes the following technical contributions:

[49]: Zhang et al. (2011), *SCION: Scalability, control, and isolation on next-generation networks*

[50]: Pereira et al. (2024), *Protocols to Code: Formal Verification of a Next-Generation Internet Router*

- We present the Gobra tool, an automated modular verifier for annotated Go programs. Our evaluation demonstrates that Gobra can verify non-trivial examples with good performance. Our artifact is available online [60].
- We define a specification language for functional properties of Go programs. Our specification language provides a consistent abstraction at the level of Go and does not leak details of the underlying encoding.
- We present the first specification and verification technique for structural subtyping via Go interfaces.

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

- Our Viper encoding supports, among other features, Go's broad collection of built-in data types, such as slices and channels. A lightweight annotation allows it to apply separation logic to reason soundly about addressable memory locations, but use a more efficient encoding for others.

Outline. We demonstrate key features of Gobra on examples (Sec. 2.1), give an overview of the encoding into Viper (Sec. 2.2), and provide an experimental evaluation of Gobra (Sec. 2.4). Lastly, Sec. 2.5 discusses related work and concludes.

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

This chapter is based on a previous publication about Gobra [51].

2.1. Gobra in a Nutshell

This section illustrates Gobra's specification language on simple examples and shows how we handle interfaces, closures, and concurrency.

2.1.1. Basics

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

Gobra uses a variant of separation logic [28] in order to reason about mutable heap data structures and concurrency. Separation logics associate an access permission with each heap location. Access permissions are held by method executions and transferred between methods upon call and return. A method may access a location only if it holds the associated permission. Permission to a shared location `v` is denoted in Gobra by `acc(&v)`, which is analogous to separation logic's $v \mapsto \_$. Gobra provides an expressive permission model supporting fractional permissions [61] to allow concurrent read accesses while still ensuring exclusive writes, (recursive) predicates to denote access to unbounded data structures, and quantified permissions (also called iterated separating conjunction) to express permissions to random-access data structures such as arrays and slices.

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

The example in Fig. 2.1 illustrates the use of permissions. Method `incr` increases all elements of a given slice `s` by an amount `n`. (Slices are data types that can intuitively be seen as shared arrays of variable length.) The method requires permission to all slice elements (via its precondition) and returns them to the caller (via its first postcondition).

Functional properties are expressed via standard assertions, which include side-effect free Go expressions (including calls to pure methods, as we explain below) as well as universal quantification and old-expressions to refer to the value an expression had in the pre-state of a method. In our example, the second postcondition uses these assertions to express the functional behavior of the method. The loop invariants are analogous to the method contracts and are needed for verification.

We refer to memory locations as either *shared* or *exclusive*. Shared locations reside on the heap and, thus, are accessible by multiple methods and threads; reasoning about shared locations requires permissions to ensure race freedom and to enable framing, *i.e.* preserving information across heap changes. On the other hand, exclusive locations are accessed exclusively by one method execution and may be allocated on the stack; they

```
1  requires ∀ k int :: 0 ≤ k < len(s)  ⟹  acc(&s[k])
2  ensures  ∀ k int :: 0 ≤ k < len(s)  ⟹  acc(&s[k])
3  ensures  ∀ k int :: 0 ≤ k < len(s)  ⟹  s[k] == old(s[k]) + n
4  func incr (s []int, n int) {
5
6    invariant 0 ≤ i ≤ len(s)
7    invariant ∀ k int :: 0 ≤ k < len(s)  ⟹  acc(&s[k])
8    invariant ∀ k int :: i ≤ k < len(s)  ⟹  s[k] == old(s[k])
9    invariant ∀ k int :: 0 ≤ k < i  ⟹  s[k] == old(s[k]) + n
10   for i := 0; i < len(s); i += 1 {
11     s[i] = s[i] + n
12   }
13 }
```

Figure 2.1.: A simple Gobra example showing method and loop contracts.

can be reasoned about as local variables. The Go compiler determines automatically whether a location is shared or exclusive, for instance by determining whether its address is taken at some point of the execution. To make verification independent of a particular compiler analysis, Gobra requires variables to be decorated with an extra annotation `@` at the declaration point to treat the variable as a shared location, as illustrated by the following client of `incr`:

```
1  a@ := [4]int { 1, 2, 4, 8 }
2  incr(a[2:], 2)
3  assert a == [4]int { 1, 2, 6, 10 }
```

The first line declares a Go array `a` of fixed length 4, with values 1, 2, 4, and 8. This array is sliced on Line 2 using the syntax `a[2:]`, thereby omitting the first two elements of `a` from the created slice. Since `a` is used in a context in which it is sliced, it is a shared location, which is made explicit via the `@` annotation. Consequently, the array creation will produce permissions to the array elements, which are required by `incr`'s precondition. Omitting the `@` annotation will cause a verification error.

2.1.2. Interfaces

Go supports polymorphism through *interfaces*, named sets of method signatures. Subtyping for interfaces is structural: a type implements an interface iff every method of the interface is implemented by the type. The subtype relationship is determined by the type checker, without any declarations from the programmer*.

Calls on an interface value are dynamically dispatched. In settings with nominal subtyping, dynamic dispatch is handled by proving behavioral subtyping [62]: each subtype declaration requires a proof that the specifications of subtype methods refine the specifications of the corresponding supertype methods. Since structural subtypes are not declared explicitly, we adapt this approach as follows.

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

Whenever a Go program assigns a value to a variable of an interface type, Gobra requires an *implementation proof*, that is, a proof that each method of the subtype satisfies the specification of the corresponding method in the interface. Implementation proofs are inferred automatically by

* For the sake of simplicity, we omit *embeddings*, Go's construct for delegation; an extension is straightforward.

```
1  type stream interface{
2    pred memory()
3
4    requires acc(memory(), _) // arbitrary fraction of memory()
5    pure hasNext() bool
6
7    requires memory() && hasNext()
8    ensures  memory()
9    next() interface{}
10 }
11 ---------------------------------------------------------------
12 type counter struct{ f int; max int }
13
14 requires acc(&x.f, _) && acc(&x.max, _)
15 pure func (x *counter) hasNext() bool { return x.f < x.max }
16
17 requires acc(&x.f) && acc(&x.max, 1/2) && x.hasNext()
18 ensures  acc(&x.f) && acc(&x.max, 1/2) && x.f == old(x.f)+1
19 ensures  typeOf(y) == int && y.(int) == old(x.f)
20 func (x *counter) next() (y interface{}) { x.f++;return x.f-1 }
21
22 ---------------------------------------------------------------
23 pred (x *counter) memory() { acc(&x.f) && acc(&x.max) }
24
25 (*counter) implements stream {
26
27   pure (x *counter) hasNext_Proof() bool {
28     return unfolding acc(x.memory(), _) in x.hasNext()
29   }
30
31   (x *counter) next_Proof() (res interface{}) { ... }
32 }
33
```

Figure 2.2.: An interface specification for a stream (Lines 1–10) together with an implementation (Lines 12–20) and an implementation proof (Lines 23–32). We write `acc(p, _)` to denote an arbitrary, positive amount of predicate `p`, and simply `p` for `acc(p, 1/1)`. At Line 18, the fractional permission to `&x.max` entails that `x.max` is not modified.

Gobra in simple cases; user-provided implementation proofs are required especially when they include ghost operations, for instance, to manipulate predicates.

The example in Fig. 2.2 illustrates this approach. Interface `stream` (Lines 1–10) declares an interface with two methods, `hasNext` and `next`. The latter may return values of an arbitrary type, which is denoted by an empty interface. Since interfaces do not contain an implementation, their specification must be fully abstract. To this end, `stream` introduces an abstract predicate `memory`, whose definition is provided by the subtypes of the interface. The functional behavior of interface methods can be expressed in terms of pure (that is, side-effect free) abstract methods, here, `hasNext`, which will also be defined in subtypes.

Next, Lines 12–20 show an implementation of the interface in the form of a counter. The counter has a current `f` and maximum `max` value. As long as the maximum value is not reached, `next` will increase the current value. At Line 20, an integer can be assigned to the empty interface since behavioral subtyping holds trivially. The specification at Line 19 expresses that the returned interface value contains an integer with the old value of the `f` field.

The counter implementation is completely independent of the `stream` interface. Their connection is established only in the implementation proof (Lines 23–32). This proof defines the `memory` predicate from the `stream` interface for receivers of type `counter` (Line 23). Moreover, an implementation proof verifies that the specification of each method implementation refines the specification of the corresponding interface

method. This proof checks that, assuming the precondition of an interface method, a call to the implementation method with identical arguments establishes the postcondition of the interface method. This format is enforced syntactically and permits ghost operations before and after the call to manipulate predicates. For instance, the proof on Line 28 for `hasNext` temporarily unfolds the `memory` predicate to obtain permission to `x`, which is required by the implementation method, and conversely after the call.

Implementation proofs can be written explicitly, imported from other packages, and also inferred automatically when no explicit proof exists in the current scope. Currently, Gobra does not infer ghost operations such as the `unfolding` on Line 28; our experiments suggest that already simple heuristics can deal with many cases occurring in practice. For instance, many implementation proofs we have encountered follow the same pattern: First, the interface predicate instances of the precondition are unfolded. Second, the implementation method is called. Lastly, the interface predicate instances of the postcondition are folded. This pattern can be generated automatically to alleviate the annotation burden.

Gobra's implementation proofs enable one to reason about interfaces without enforcing subtype declarations in either the interface or the declaration, which would defeat the purpose of structural subtyping. This solution allows one to reason about dynamically-dispatched calls. For instance, the following code snippet verifies in Gobra:

```
1 x := &counter{0, 50}
2 var y stream = x
3 fold y.memory()
4 var z interface{} = y.next()
```

In particular, Gobra is able to determine that `next`'s precondition `hasNext()` holds because `y.hasNext()` is equal to `x.hasNext()`, and the latter follows from the definition of `hasNext` (Line 15) and the initial value of `x.f`. This intuitive reasoning is enabled by an intricate underlying encoding, which is not exposed to users. Users do not have to know how interface predicates are encoded and can treat interface predicates the same as any other separation-logic predicate.

2.1.3. Closures

As for other modern programming languages, Go supports first-class functions, *i.e.* functions may be used like any other value, in particular, assigned to variables, passed as arguments, and returned as results. Function values are obtained either by referencing declared methods or by defining *closures*. Closures are function literals that may reference their surrounding variables. E.g., the closure `func (n int) { x = n }` takes an integer `n`, which the closure then assigns to the variable `x`.

We reason about closures similar to interfaces with four ingredients: (1) As for interface methods, we annotate the behavior of function values via specifications consisting of pre- and postconditions. (2) To specify that a, potentially dynamically created, function value `f` satisfies a specification `s`, we introduce an *implementation assertion* `f implements s`. (3) To derive an implementation assertion, as for interfaces, we require an implementation proof, showing that the function value satisfies a

```
1  ghost
2  requires c.inv()
3  ensures  c.inv() && r >= 0
4  func positive(c companion, a int) (r int)
5
6  type companion interface{
7    pred inv()
8  }
9
10 func main() {
11   x@ := 0
12   f := requires acc(&x)
13        ensures  acc(&x) && m == max(old(x),n) && x == m
14        func f_Spec(n int) (m int) {
15          if n > x {
16            x = n
17          }
18          return x
19        }
20
21   proof f implements positive{f_comp{&x}} {
22     unfold f_comp{&x}.inv()
23     r = f(a) as f_Spec
24     fold f_comp{&x}.inv()
25   }
26
27   fold f_comp{&x}.inv()
28   y := hof(f_comp{&x}, f)
29 }
30
31 ghost type f_comp struct{ x *int }
32 pred (c f_comp) inv() { acc(c.x) && *c.x >= 0 }
33
34 requires c.inv() && f implements positive{c}
35 ensures  c.inv() && res >= 0
36 func hof(ghost c companion, f func(int) int) (res int) {
37   res = f(42) as positive{c}
38   return
39 }
```

Figure 2.3.: A toy example verifying the use of a closure. Lines 36–39 show a higher-order method taking a function `f` as argument. The implementation assertion `f implements positive{c}` specifies which specification the method `hof` expects the function to satisfy, where Lines 1–4 define the pre- and postconditions. The predicate `inv` of the interface `companion` expresses which permissions and properties must hold so that the specification expected by `hof` is satisfied by the closure. Lastly, Lines 10–29 show a client method that creates a closure and then passes it to `hof`.

particular specification. (4) When calling a function value, we specify which particular specification we want to use to reason about the call.

Because closures may capture their surrounding state, whether a closure satisfies a specification may depend on the current program state. This dependence makes it harder to provide closures with abstract specifications that do not involve specialized knowledge about the closure itself. We address this challenge by enabling us to parameterize specifications with interfaces. As discussed in Sec. 2.1.2, interfaces abstract the state of concrete implementations. Analogously, we are able to use interfaces to abstract over the state captured by a closure. By combining points (1-4) with our support for interfaces, we are able to reason about closures capturing arbitrary state.

We illustrate our approach with the example shown in Fig. 2.3. The method `main` (Lines 10–29) assigns to a variable `f` a closure that takes an integer and returns the maximum of all the arguments that the function was called with (Lines 14–19). The closure stores the largest argument encountered so far in a local variable `x`. After creating the closure, the method `main` passes the closure to a higher-order function `hof`, which calls the passed closure with the argument 42 and returns the result (Lines 34–39).

In the example, we specify the behavior of the function stored in `f`

in two ways, namely directly at the closure (Lines 12–13) and in the implementation assertion `f implements positive{c}` (Line 34). In the closure specification at Lines 12–13, the precondition expresses that calling the closure requires write permissions to the captured variable `x`. Gobra requires that all variables captured in closures are annotated as shared. The postcondition specifies that the method returns the permissions to the variable, that the returned value is the maximum of the argument and the old value of `x`, and that the new value of `x` is equal to the returned maximum. For the sake of brevity, we assume that a pure method `max`, returning the maximum of two integers, is defined. In Gobra, the direct specification of closures (*e.g.* Lines 12–13), which we refer to as a *local specification*, may be used only within the method that the closure is defined in. We enforce this restriction as a design choice since reading the specification requires looking at the method that the closure was defined in and local specifications may depend on local variables.

The implementation assertion used in the specification of the method `hof` specifies a different behavior than the specification of the closure `f`. To specify the behavior of a closure outside of the method that the closure is defined in, we introduce *specification expressions*. Specification expressions are of the form `s{x1,x2,...}` where `s` is the name of the method whose pre- and postcondition specify the behavior of the function value. The arguments `x1,x2,...` partially apply arguments of `s`, effectively parameterizing the specification of `s`. For instance, the specification expression `positive{c}` specifies that the result of a call is positive as annotated in the postcondition of the method `positive`. The interface `companion` abstracts over the state captured by the function values satisfying the specification. In our example, the interface defines a single predicate `inv`, abstracting the permissions and properties of the captured state that have to hold to satisfy the specification. We do not add a special feature to express when a specification is satisfied. Instead, by adding the predicate instance `c.inv()` to the pre- and postcondition of `positive`, we straightforwardly specify that calls require the predicate instance and that calls preserve the predicate instance. The closure `f` only satisfies that the result is non-negative if permissions to the captured variable are held and if the value of the captured variable is non-negative. We express this requirement with the specification argument `f_comp{&x}` (Line 21), implementing the interface `companion`. As for the method `positive`, `f_comp` is defined only for the purpose of verification to handle the captured state of the closure `f`. The type `f_comp` defines a struct with a field `x`, used to store the reference to the captured local variable `x`. The predicate `inv` then specifies which assertion the local variable has to satisfy as mentioned before.

Lines 21–25 show the implementation proof that verifies that the closure satisfies the specification expression `positive{f_comp{&x}}`, which is necessary to call `hof`. Analogous to the implementation proofs for interfaces, the implementation proofs for closures verify that a specification satisfied by the closure entails the provided specification expression. In our example, the proof requires only to unfold and fold the predicate instance of the interface. As per our ingredients, calling the closure requires us to specify which specification we want to use to reason about the call. We denote with `as s` that we use the specification `s` to reason about the call, *i.e.* the call requires and ensures the pre- and

postcondition of `s`, respectively. At Line 23, $\texttt{f}_{\texttt{spec}}$ is the name of the local specification. After the implementation proof, we obtain the assertion `f implements positive{f_comp{&x}}`. Before calling `hof`, we fold the predicate instance to satisfy the precondition of `hof`. Lastly, inside of the `hof` method, we annotate that we use `positive{c}` to reason about the call (Line 37), making it possible to show `hof`'s postcondition asserting that the result is positive.

2.1.4. Concurrency

Go supports concurrency through *goroutines*, lightweight threads started by prefixing a method call with the `go` keyword. Go offers the usual synchronization primitives, but goroutines idiomatically synchronize via *channels*. Buffered channels provide asynchronous communication, where sending a message blocks only when the buffer is full. Unbuffered channels offer rendez-vouz communication.

Gobra enables verification of concurrent programs by associating Go's synchronization primitives with predicates that do not only express properties of data but also express how permissions to shared memory get transferred between threads. For instance, lock invariants may include properties as well as permissions to the data protected by the lock, and channel invariants include properties and permissions of the data sent over a channel. These invariants are specified via ghost operations when the synchronization primitive is initialized.

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

Fig. 2.4 illustrates Gobra's concurrency support using an excerpt from a parallel search-and-replace algorithm (see the full paper [63] for the complete example). Method `searchAndReplace` spawns a series of worker threads and then sends each of them a chunk of the input slice to process. The worker threads are joined via a wait group `wg`. Method `worker` implements the worker threads.

Gobra associates channels (like `c` in the example) with a predicate to specify properties and permissions of the sent data. The call `c.Init(...)` on Line 11 takes this predicate as an argument. As expressed on Line 2, it includes permissions to the chunk a worker operates on. For synchronous channels, an additional predicate can specify permissions transferred in the opposite direction, from the receiver to the sender. Initializing a channel also creates send and receive permissions for the channel, which are used to control which threads may access it. In our example, we transfer a fraction of the receive permission to each worker (Line 32).

The workers receive permission to the chunk they operate on via a message sent on Line 27 and received on Line 39. The transfer back is orchestrated through a wait group, which implements an abstract shared counter. Wait groups are used as follows: The main thread adds to the counter the number of units of work to be done in spawned goroutines (Line 25). Each spawned goroutine decreases the counter each time a unit of work is done (via a call to `Done`, Line 42). The master can await the counter to reach 0 via a call to `Wait` (Line 29). Gobra uses dedicated permissions to express the obligation of a thread to perform units of work before decreasing the counter; each time this happens, permissions are transferred to the wait group and, eventually to the main thread calling `Wait`. We omit the details here for brevity.

```
1  pred messagePerm(wg *sync.WaitGroup, chunk []int, x, y int) {
2    ( ∀ i int :: 0 ≤ i < len(chunk)  ⟹  acc(&chunk[i]) ) && ...
3  }
4
5  requires ∀ i int :: 0 ≤ i < len(s)  ⟹  acc(&s[i])
6  func searchAndReplace(s []int, x, y int) {
7    var wg@ sync.WaitGroup
8    ghost wg.Init()
9    c := make(chan []int,4)
10   // predicate-name{..., _, ...} is syntax for partial application
11   ghost c.Init(messagePerm{&wg, _, x, y})
12
13   // Spawn workers
14   invariant acc(c.RecvChannel(), _)
15   invariant c.RecvGotPerm() == messagePerm{&wg, _, x, y}
16   for i := 0; i < numOfWorkers; i++ { go worker(c, wg, x, y) }
17
18   // Split slice into chunks, which are sent to workers
19   invariant c.SendChannel()
20   invariant c.SendGivenPerm() == messagePerm{&wg, _, x, y}
21   invariant ∀ i int :: offset ≤ i < len(s)  ⟹  acc(&s[i])
22   invariant ... // constraints on offset and nextOffset
23   for offset := 0; offset != len(s); offset = nextOffset {
24     nextOffset = ...
25     wg.Add(1)
26     fold messagePerm{&wg, _, x, y}(s[offset:nextOffset])
27     c <- s[offset:nextOffset]
28   }
29   wg.Wait()
30 }
31
32 requires acc(c.RecvChannel(), _)
33 requires c.RecvGotPerm() == messagePerm{wg, _, x, y};
34 func worker(c <- chan []int, wg *sync.WaitGroup, x, y int) {
35
36   invariant acc(c.RecvChannel(), _)
37   invariant c.RecvGotPerm() == messagePerm{wg, _, x, y};
38   invariant ok  ⟹  messagePerm{wg, _, x, y}(chunk)
39   for chunk, ok := <- c; ok; chunk, ok = <-c {
40     unfold messagePerm{wg, _, x, y}(chunk)
41     ... // replace x with y in chunk
42     wg.Done() // same as wg.Add(-1)
43   }
44 }
```

Figure 2.4.: Excerpt showing goroutines, channels, and wait groups. The code spawns workers (Line 16), sends slice chunks through a channel to the workers (Line 27), and then waits on a wait group (Line 29). A worker receives a chunk (Line 39), processes it, and then notifies the wait group (Line 42). For the sake of simplicity, some details were omitted.

In our example, this mechanism allows the main thread to recover the permissions to the entire slice once the workers have terminated. The example in Fig. 2.4 illustrates only the permission aspect of the verification. Functional correctness can be verified easily based on the explained machinery, by specifying a stronger channel invariant that includes the work obligation for each worker. We omit the details here, but see the full paper [63] for the complete example.

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

2.2. Encoding

Gobra encodes an annotated Go program into a Viper program verifying only if the input program is correct. Viper provides a simple imperative language, where a program, as for Gobra, consists of methods (with pre- and postconditions), pure methods, and predicates. Since many features of Gobra are also present in Viper, parts of the encoding are straightforward. All topmost structural members of a Go program, namely methods, pure methods, and predicates, are encoded to their

Viper counterparts. Furthermore, Viper supports permissions-based reasoning, including fractional and quantified permissions. However, Viper does not offer most of Go's types, such as slices, interfaces, and closures; and memory is represented differently in Viper. In contrast to Go's pointer-based memory model, Viper's memory is object-based, *i.e.* only objects are stored in the memory, and thereby may be aliased, where objects have user-declared fields to store values.

Our encoding then has two main ingredients: First, we define a type encoding that specifies how we represent Go's program state and Gobra's ghost state in Viper. Second, we encode operations on Go's program state and Gobra's ghost state into Viper operations that preserve the encoded operation's behavior with respect to the type encoding.

A key aspect of the type encoding is how we model whether data is stored on the heap or not. As discussed in Sec. 2.1.1, we use the notion of *shared* and *exclusive* to determine where data resides. Shared memory locations reside in the heap and, thereby, require permission-based reasoning. Conversely, exclusive memory locations do not reside in the heap and, thereby, do not require permission-based reasoning. For the encoding, to capture whether an expression represents a shared or exclusive memory location, we extend Go's types with a shared and exclusive modifier, denoted as $t@$ and $t\bullet$, respectively, for some type t. We then encode expressions with type $t\bullet$ and $t@$ to a Viper expression representing a value of t and a memory location storing values of t, respectively. For the latter, we use Viper's objects to get access to Viper's permission-based reasoning. We refer to expressions with a shared and exclusive type as shared and exclusive expressions, respectively.

Importantly, our classification of shared and exclusive memory locations may differ from how programs actually store data. In Sec. 2.2.5, we show that our memory model, *i.e.* our treatment of shared and exclusive, is sound. More specifically, we show that verified properties hold for the actual executions of a program, too, even if data is stored differently.

In Sec. 2.2.1, we present Gobra's augmented type system. In Sec. 2.2.2, we present the core of Gobra's encoding, detailing how we bridge the gap between our and Viper's memory model. Afterwards, we introduce the encodings of more specific features, namely interfaces (Sec. 2.2.3) and closures (Sec. 2.2.4).

2.2.1. Gobra's Augmented Type System

The purpose of our augmented type system is to identify whether an expression is shared or not. The type system does not enforce restrictions beyond the restriction that only shared variables, *i.e.* variables annotated with `@`, may be referenced. This restriction includes implicit references, namely capturing a variable in a closure or taking a slice of an array. This restriction ensures that exclusive expressions never have aliases, and thereby, do not require permission-based reasoning. Every Go program where only shared variables are referenced is well-typed in our augmented type system. Every Go program is trivially well-typed if all variables are annotated as shared.

$$\frac{}{\Gamma, \mathtt{x}\square : t \vdash \mathtt{x} : t\square}\ (\textsc{Var})$$

$$\frac{\Gamma \vdash e : t@}{\Gamma \vdash e : t\bullet}\ (\textsc{Load}) \qquad \frac{\Gamma \vdash e : *t\bullet}{\Gamma \vdash *e : t@}\ (\textsc{Deref}) \qquad \frac{\Gamma \vdash e : t@}{\Gamma \vdash \&e : *t\bullet}\ (\textsc{Ref})$$

$$\frac{\Gamma \vdash e : \mathtt{struct\{...\}}\square}{\Gamma \vdash e.\mathtt{f}_i : t_i\square}\ (\textsc{Field}) \qquad \frac{\Gamma \vdash e_1 : \mathtt{[n]}t\square \quad \Gamma \vdash e_2 : \mathtt{int}\bullet}{\Gamma \vdash e_1[e_2] : t\square}\ (\textsc{Index})$$

$$\frac{\Gamma \vdash f : (t_1\bullet \times \cdots \times t_n\bullet) \to t'\bullet \quad \Gamma \vdash e_1 : t_1\bullet \quad \ldots \quad \Gamma \vdash e_n : t_n\bullet}{\Gamma \vdash f(e_1, \ldots, e_n) : t'\bullet}\ (\textsc{Call})$$

$$\frac{\Gamma \vdash e_1 : t_1\bullet \quad \Gamma \vdash e_2 : t_2\bullet \quad t_2 \sqsubseteq t_1}{\Gamma \vdash e_1 = e_2}\ (\textsc{Assign})$$

Figure 2.5.: Key rules of our augmented type system. We use $\square$ to range over the set $\{@, \bullet\}$. We use $t_2 \sqsubseteq t_1$ to denote that the Go type t_2 is a subtype of the Go type t_1. For the sake of brevity, we omit subtyping constraints for the rule Call.

Fig. 2.5 shows the key rules of our augmented type system. The rules Var, Deref, Field, and Index capture when an expression may be shared. An expression e is shared if either e is a variable that is annotated with the shared modifier @ (rule Var), e is a dereferenced pointer (rule Deref), e is the field of a shared struct (rule Field), or e is the index of a shared array (rule Index) or an exclusive slice (rule omitted). These rules formalize Go's notion of *addressibility* except that variables are only addressable if annotated with @. We use $\square$ to range over shared and exclusive modifiers. E.g., a field access is exclusive if the struct is exclusive. Variables that are not explicitly annotated with the shared modifier @ are considered to be annotated with the exclusive modifier $\bullet$ implicitly. The rule Ref restricts that only shared expressions may be referenced. We omit the rules for the two other restrictions, namely that closures may capture only shared variables and that only shared arrays may be sliced with Go's slicing operator, both of which take references implicitly. All other operations, including arithmetic operations, memory allocation, and calls (rule Call), take exclusive arguments and produce exclusive results. Note that the result of a reference and the argument of a dereference are also exclusive. The rule Load captures that shared expressions may always also be used as exclusive expressions. This rule reflects the access of heap data. Consider the expression `x + 3`, where `x` is a shared variable, whose value resides in the heap. To compute the value of `x + 3`, the value of `x` is read from the heap, which in our type system is captured by the cast to an exclusive type through the rule Load. Lastly, assignments are allowed if the Go type of the left-hand side is a subtype of the right-hand side, denoted as $\sqsubseteq$, but have no further restrictions. In particular, shared expressions may be assigned to exclusive expressions and vice versa since only values and not memory locations are copied. We omitted subtyping for the rule Call.

2.2.2. Core Encoding

In this section, we present the core of Gobra's type encoding and encoding of operations, detailing how we encode heap data and how we leverage Viper's permission-based reasoning, respectively. Furthermore,

$$
\begin{aligned}
[\![\texttt{bool}\bullet]\!] &\triangleq \texttt{Bool} & [\![s@]\!] &\triangleq \texttt{Ref} \mapsto [\![s\bullet]\!] \\
[\![\texttt{int}\bullet]\!] &\triangleq \texttt{Int} & &\text{where } s \text{ is not} \\
[\![*t\bullet]\!] &\triangleq [\![t@]\!] & &\text{a struct or array} \\
[\![\texttt{[]}t\bullet]\!] &\triangleq \texttt{Slice}[\![t@]\!] & & \\
[\![\texttt{[n]}t\bullet]\!] &\triangleq \texttt{Array}[\![t\bullet]\!] & [\![\texttt{[n]}t@]\!] &\triangleq \texttt{MemArray}[\![t@]\!] \\
[\![\texttt{struct\{}t_1,\ldots,t_n\texttt{\}}\bullet]\!] &\triangleq \texttt{NTuple}[\![t_1\bullet]\!],\ldots,[\![t_n\bullet]\!] & & \\
[\![\texttt{struct\{}t_1,\ldots,t_n\texttt{\}}@]\!] &\triangleq \texttt{NMemTuple}[\![t_1@]\!],\ldots,[\![t_n@]\!] & &
\end{aligned}
$$

Figure 2.6.: Encoding of types. For a Go type t, $[\![t\bullet]\!]$ and $[\![t@]\!]$ are the encodings of values of t and memory storing values of t, respectively. We use $\texttt{Ref} \mapsto \hat{t}$ to denote that a type is encoded as the Viper object type `Ref`, where we define a Viper field with type $\hat{t}$.

we illustrate based on our encoding of structs and arrays how we encode more complex types in Viper. Since we focus on our treatment of heap data, we only cover primitive types, pointers, structs, and arrays in this section. We cover more specific features, namely interfaces and closures, in the subsequent sections.

We first discuss our type encoding in Sec. 2.2.2.1. We then discuss the encoding of operations in Sec. 2.2.2.2.

To illustrate our encoding, we introduce the running example below. The method `link` takes a pointer `x` to a linked list node, allocates a node `n` pointing to `x`, and then returns a pointer to `n`. The method also sets `n.val` to `x.val`. The specification expresses that the method requires read permissions to the `val` field of `x`, returns write permissions to the allocated node, and that the value of the node `*p` is as expected.

```
1  type node struct{ next *node; val bool }
2
3  requires acc(&x.val, 1/2)
4  ensures  acc(&x.val, 1/2) && acc(p)
5  ensures  *p == node{x,x.val}
6  func link(x *node) (p *node) {
7    var n@ node
8    n.next = x
9    n.val = x.val
10   return &n
11 }
```

2.2.2.1. Type Encoding

Since we augment the type system with shared and exclusive modifiers, our encoding of types captures not only how we represent values, but also how we represent heap data. For a Go type t, we encode $t\bullet$ and $t@$ as the Viper type capturing the values of t and the Viper type capturing the memory locations storing values of t, respectively.

Fig. 2.6 shows our encoding of the relevant types. Exclusive booleans and integers are encoded straightforwardly to their Viper counterparts. For types not directly supported in Viper, we use Viper's domain mechanism to declare custom types using uninterpreted types, uninterpreted functions, and appropriate axioms. As such, we define mathematical tuples (`NTuple`) and mathematical sequences of values (`Array`) to encode exclusive structs and arrays, respectively. These custom types are parameterized in the types of the elements. As a technical detail, our encoding of types does not directly capture the size of arrays. Instead, we treat the

size of an array as a property of array values. Lastly, we encode pointers as the memory locations storing the values that are pointed to. Therefore, an exclusive pointer type $*t\bullet$ is encoded the same as $t@$.

The encoding of shared types is determined by how code may interact with the memory locations. At the lowest level, memory is a sequence of addressable word-sized values. However, for a type-safe language such as Go, this level of detail is not observable. In Go, for all types except structs, arrays, and slices, code cannot obtain references for only parts of a stored value, *e.g.* the last byte of an integer. For structs, arrays, and slices, only well-defined locations, namely struct fields and array and slice indices, can be accessed. Therefore, memory may be treated as a partition into indivisible fragments that store well-typed values, some of which are composed together to form structs, arrays, and slices. This partition is captured in our type encoding. For every type s that is not a struct or an array, $s@$ is encoded into Viper's object type `Ref` with a field of type $[\![s\bullet]\!]$ to represent the stored value. Shared slices are also encoded as `Ref` because we treat slices analogously to array pointers. A shared slice captures the memory location of an exclusive slice whereas an exclusive slice captures the memory locations of the contained elements. In Viper, fields are not part of the type definition. We use $\texttt{Ref} \mapsto \hat{t}$ to denote that a field with Viper type $\hat{t}$ is declared. Shared structs and shared arrays are encoded as tuples (`NMemTuple`) and sequences (`NMemTuple`) of memory locations, respectively. The custom type `Slice` is a wrapper around `MemArray` adding some additional data, namely the offset, size, and capacity of the slice.

For our running example, the struct type `node@` (and thereby also `*node•`) encodes to `2MemTuple[Ref,Ref]`. Note that `*node@` is encoded as `Ref`. Conversely, `node•` encodes to `2Tuple[2MemTuple[Ref,Ref],Bool]`. Below, we show a partially encoded of the `link` method where all types are encoded. As mentioned before, we use Viper methods to encode Gobra methods. We use comments for the parts that are not yet encoded.

```
1  method link(x: 2MemTuple[Ref,Ref]) (p: 2MemTuple[Ref,Ref])
2    requires // acc(&x.val, 1/2)
3    ensures  // acc(&x.val, 1/2) && acc(p)
4    ensures  // *p == node{x,x.val}
5  {
6    // var n@ node
7    // n.next = x
8    // n.val = x.val
9    // return &n
10 }
```

For the rest of the type encoding, we discuss the custom types introduced to encode shared and exclusive structs and arrays. In particular, we focus on the differences between encoding values and memory locations.

Encoding Values and Memory Locations. The methodology for encoding values of types is straightforward. For each operation on the type, we introduce an uninterpreted function and then express the desired properties of combinations of operations as axioms. The challenge is to define a set of axioms that is complete, sound, and has a good verification performance. Encoding memory locations is similar, however, the properties desired for memory locations are different from the properties of values. In contrast to values, the memory locations of structs and arrays

```
domain 2Tuple[T0,T1] {
  func get0(2Tuple[T1,T2]): T0
  func get1(2Tuple[T1,T2]): T1

  func tup2(T0,T1): 2Tuple[T0,T1]

  axiom {
    ∀ v0: T0, v1: T1 ::
      get0(tup2(v0,v1)) == v0 &&
      get1(tup2(v0,v1)) == v1
  }

  axiom {
    ∀ t: 2Tuple[T1,T2] ::
      tup2(get0(t),get1(t)) == t
  }
}
```

```
domain 2MemTuple[T0,T1] {
  func loc0(2MemTuple[T0,T1]): T0
  func loc1(2MemTuple[T0,T1]): T1

  func inv0(T0): 2MemTuple[T0,T1]
  func inv1(T1): 2MemTuple[T0,T1]

  axiom {
    ∀ t: 2MemTuple ::
      inv0(loc0(t)) == t &&
      inv1(loc1(t)) == t
  }


}
```

Figure 2.7.: Definition of the custom types that we use to represent mathematical tuples (left) and the memory of structs (right). In Viper, domains define uninterpreted types. These types may have generic parameters (`T0` and `T1`). In the body of a domain definition, we may define uninterpreted functions and axioms about these functions. For mathematical tuples, we use axioms to express that the getters yield the expected values and that tuples are equal if their elements are equal. For memory tuples, we express that, as for actual memory, the individual references are disjoint.

are made up of disjoint smaller memory locations. We have to capture this disjointness to fully reason about memory. Consider the code snippet below. The method `foo` takes two `node` pointers and compares whether the references of the `val` fields are equal. Because the memory locations of different values are disjoint, we know that if references of the field are equal, then the two `node` pointers must be equal, too.

```
type node struct{ next *node; val bool }

requires x != nil && y != nil
func foo(x, y *node) {
  if &x.val == &y.val {
    assert x == y // succeeds
  }
}
```

We formally capture the disjointness of memory locations by requiring that the operations yielding the disjoint memory locations are injective. For instance, for structs, the field memory location `&p.f` is injective in `p` for every struct pointer `p` and field `f`. With this disjointness, the above snippet verifies successfully.

Next, we present our custom type definitions for structs and arrays, namely the types `NTuple`, `NMemTuple`, and `MemArray` of our type encoding in Fig. 2.6.

Mathematical Tuples. Fig. 2.7 shows on the left our custom type definition for tuples of size two, used to encode exclusive structs. The definition introduces the type `2Tuple[T0,T1]`, where `T0` and `T1` are generic type parameters for the first and second element of the tuple, respectively. We define three functions to capture the operations on tuples. The functions `get0` and `get1` take a tuple and return the first and second element, respectively. The function `tup2` takes two elements and returns the tuple containing these two elements. Regarding axioms, the first axiom captures the behavior of the getters, namely that the result of a getter is the expected element. The second axiom defines when two tuples are equal.

Memory Tuples. The right snippet of Fig. 2.7 shows our type definition for structs in memory. As for `2Tuple`, the type has two generic type parameters. In contrast to `2Tuple`, these parameters capture the memory locations of the struct's elements. The functions `loc0` and `loc1` return

```
1  domain MemArray[T] {
2    func loc(MemArray[T], Int): T        func inv_array(T): MemArray[T]
3    func len(MemArray[T]): Int           func inv_idx(T): Int
4
5    axiom {
6      ∀ a: MemArray[T], i: Int :: 0 ≤ i < len(a)  ⟹
7          inv_array(loc(a, i)) == a && inv_idx(loc(a, i)) == i
8    }
9
10   axiom { ∀ a: MemArray[T] :: len(a) ≥ 0 }
11 }
```

Figure 2.8.: The type definition of our arrays. The function `loc` returns the memory location of indices. The function `len` returns the size of an array. We express the injectivity of `loc` using `inv_array` and `inv_idx`.

$$
\begin{aligned}
[\![\&e]\!] &\triangleq \texttt{not\_nil}([\![e]\!]_M) \\
[\![e\texttt{: } t@]\!] &\triangleq \texttt{load}([\![e]\!]_M) & [\![*e\texttt{: } t@]\!]_M &\triangleq [\![e]\!] \\
[\![e\texttt{.f: } t\bullet]\!] &\triangleq \texttt{getI}([\![e]\!]) & [\![e\texttt{.f: } t@]\!]_M &\triangleq \texttt{locI}([\![e]\!]_M) \\
[\![\texttt{x: } t\bullet]\!] &\triangleq \texttt{x} & [\![\texttt{x: } t@]\!]_M &\triangleq \texttt{x} \\
[\![\texttt{e}_1 == \texttt{e}_2]\!] &\triangleq [\![\texttt{e}_1]\!] == [\![\texttt{e}_2]\!]
\end{aligned}
$$

Figure 2.9.: The encoding of expressions. For an expression e, $[\![e]\!]$ and $[\![e]\!]_M$ encode the value and memory location of e respectively.

the memory location of the first and second struct element, respectively. In our example, these functions return the references of the `next` and `val` field, respectively. We capture that a function is injective by expressing that there exists an inverse for the results of the function. We introduce the functions `inv0` and `inv1` to capture these inverses.

Injectivity requires a careful treatment of axioms and uninterpreted functions. E.g., because of injectivity, we do not introduce a function analogous to `tup2` creating instances of tuples. Such a function implies the existence of tuples with arbitrary memory locations, in particular, `tup2(x,x)` for some memory location `x`, contradicting injectivity.

Memory Arrays. Fig. 2.8 shows our type definition for arrays in memory. We use a singular type definition to encode Go's arrays and the arrays that underlie slices. The function `loc` returns the memory location of a specific array index. Similar to structs, we introduce the functions `inv_array` and `inv_idx` to formalize that `loc(a,i)` is injective in the array `a` and index `i`. Lastly, the function `len` returns an array's size. Our two axioms express that the size of an array is always non-negative and the aforementioned injectivity of the `loc` function.

Nil. As for other C-like languages, Go has null pointers, denoted as `nil`. Since we encode pointers as the memory location storing the values, we also model null memory locations. For `Ref`, we encode `nil` as Viper's null value. For structs and arrays, `nil` is the instance where the memory locations of all elements are `nil`. As a technical detail, the nil array memory location has a length of 1 to not contradict the injectivity axiom of arrays.

2.2.2.2. Encoding of Expressions, Permissions, and Statements

For structs, arrays, and pointers, the encoding of operations on values mostly follows from our type encoding. Analogous to our type encoding, we introduce encoding functions $[\![\cdot]\!]$ and $[\![\cdot]\!]_M$ for values and memory

```
1  function load(x: ⟦s@⟧) returns (y: ⟦s•⟧)
2    requires ⟦acc(&x,_)⟧ // where s is not a struct or array type
3    ensures  y == x._s
4
5  function load(x: ⟦struct{t_1,...,t_n}@⟧) returns (y: ⟦struct{t_1,...,t_n}•⟧)
6    requires ⟦acc(&x,_)⟧
7    ensures  ⟦y.f_1 == x.f_1⟧ && ... && ⟦y.f_n == x.f_n⟧
8
9  function load(x: ⟦[n]t@⟧) returns (y: ⟦[n]t•⟧)
10   requires ⟦acc(&x,_)⟧
11   ensures  ∀ i: Int :: 0 ≤ i < n ⟹ ⟦y[i] == x[i]⟧
```

Figure 2.10.: Declarations of the `load` functions for each type. The functions read the values stored in a memory location to create an exclusive value.

$$
\begin{aligned}
&⟦\texttt{acc}(\&e : *s)⟧ \triangleq \texttt{acc}(⟦e⟧_M.\_s) \\
&⟦\texttt{acc}(\&e : *\texttt{struct}\{t_1,\ldots,t_n\})⟧ \triangleq \\
&\quad ⟦e⟧_M \texttt{ != } ⟦\texttt{nil}⟧ \texttt{ \&\& } ⟦\texttt{acc}(\&e.\texttt{f}_1)⟧ \texttt{ \&\& ... \&\& } ⟦\texttt{acc}(\&e.\texttt{f}_n)⟧ \\
&⟦\texttt{acc}(\&e : *\texttt{[n]}t)⟧ \triangleq \\
&\quad ⟦e⟧_M \texttt{ != } ⟦\texttt{nil}⟧ \texttt{ \&\& } \forall\ \texttt{i: Int :: 0} \leq \texttt{i} < \texttt{n} \implies ⟦\texttt{acc}(\&e\texttt{[i]})⟧ \\
&⟦\texttt{acc}(e : \texttt{[]}t)⟧ \triangleq \\
&\quad \forall\ \texttt{i: Int :: 0} \leq \texttt{i} < ⟦\texttt{len}(e)⟧ \implies ⟦\texttt{acc}(\&e\texttt{[i]})⟧
\end{aligned}
$$

Figure 2.11.: The encoding of permissions.

locations, respectively. Fig. 2.9 shows the encoding of the most important operations. Accessing a field is encoded as a getter of `NTuple` or `NMemTuple`, depending on whether we are encoding a memory location or not. Analogous to the type encoding, $\&e$ is encoded as the memory location of e, *i.e.* $⟦e⟧_M$. Since in Go, taking the reference of `nil` causes an exception, we furthermore assert that e's memory location is not `nil`, denoted as `not_nil(·)`. Conversely, the memory location of a dereference is encoded as the value of the pointer. For variables, the value of an exclusive variable and the memory location of a shared variable is encoded as the value of the variable itself. We obtain the exclusive value of a shared expression `e`, *i.e.* the value stored in the memory location, by reading the memory locations of `e`. As shown in Fig. 2.10, for each expression type, we define a pure function `load` that takes a shared value of type $t@$ and returns the stored exclusive value of type $t•$. Viper's pure functions correspond to Gobra's pure methods, *i.e.* they are side-effect free functions that may be invoked in assertions. The `load` functions require read permissions to the argument because of the memory reads in the function. The postconditions specify the relation between the shared and the exclusive value. For types that are not structs or arrays, we express that the returned exclusive value is equal to the value stored in the declared field $\_s$. For structs and arrays, the postcondition states that the values of the fields and indices are equal, respectively.

Permission Encoding. The encoding of permissions, shown in Fig. 2.11, also follows from the type encoding. For types that are neither structs nor arrays, the permissions to a pointer are encoded as the Viper permissions to the field $\_s$ storing the exclusive value. For structs and arrays, the permissions to a pointer are encoded as the conjunction of the permissions to all field and index pointers, respectively. Since in Go, structs and arrays may have zero fields or indices, we also include that the pointer is not `nil`. Lastly, the permissions to a slice are also encoded as the conjunction of the permissions to all index pointers.

$$
\begin{aligned}
[\![\texttt{x: } t\bullet = e]\!] &\triangleq \texttt{x} = [\![e]\!] \\
[\![e_1\texttt{.f: } t\bullet = e_2]\!] &\triangleq [\![e_1 = \texttt{upd\_f}(e_1, e_2)]\!] \\
[\![l\texttt{: } t@ = e]\!] &\triangleq \texttt{exhale } [\![\texttt{acc}(\&l)]\!] \\
&\quad\ \texttt{inhale } [\![\texttt{acc}(\&l)]\!] \ \&\& \ [\![l == e]\!] \\
[\![\texttt{var x } t\bullet]\!] &\triangleq \texttt{var x } [\![t\bullet]\!] \\
&\quad\ [\![\texttt{x} \bullet \texttt{ = default}[t\bullet]]\!] \\
[\![\texttt{var x } t@]\!] &\triangleq \texttt{var x } [\![t@]\!]\texttt{; inhale } [\![\texttt{acc(\&x)}]\!] \\
&\quad\ [\![\texttt{x@ = default}[t@]]\!] \\
[\![\texttt{x := new}(t)]\!] &\triangleq [\![\texttt{var x } t@]\!]
\end{aligned}
$$

Figure 2.12.: The encoding of assignments, variable declarations, and Go's `new` statement. Viper's `exhale` statement asserts the given assertion and removes all permissions expressed by the assertion from the verification state. Conversely, Viper's `inhale` statement assumes the given assertion and adds all permissions of the assertion to the verification state. The function `upd_f` is an auxiliary function that we introduce at the level of Gobra to streamline the encoding. The call `upd_f`(e_1, e_2) returns the struct that stores e_2 at field `f` and otherwise has the same fields as e_1. The expression `default`$[t]$ returns the default value of type t.

Given the encoding of expressions and assertions, we are able to encode the pre- and postconditions of our running example, which is shown below. The comments contain the encoded Gobra specifications. We also include the two declared fields that store the values of `int` and `*node` memory locations. To improve the performance, Gobra simplifies Viper expressions if possible. E.g., instead of comparing `load(p)` with an instance of `tup2`, Gobra compares all fields (not done in the snippet below). Note that our type system enforces that the field access `x.val` is encoded as `load(loc1(x))` instead of `get1(load(x))`. The latter requires permissions to all fields.

```
1 field _int: Int
2 field _*node: 2MemTuple[Ref,Ref]
3
4 method link(x: 2MemTuple[Ref,Ref]) (p: 2MemTuple[Ref,Ref])
5   requires acc(loc1(x)._int, 1/2) // acc(&x.val, 1/2)
6   ensures  acc(loc1(x)._int, 1/2) &&
7            acc(loc0(p)._*node) && acc(loc1(p)._int)  // acc(p)
8   ensures  load(p) == tup2(x, load(loc1(x))) // *p == node{x,x.val}
9 { ... }
```

Statement Encoding. Fig. 2.12 presents the encoding of statements that involve memory locations, namely assignments, variable declarations, and memory allocation. The declaration of an exclusive variable is encoded as a Viper variable declaration. Because in Go, all variables are initialized with a default value, the encoding furthermore assigns the default value, denoted as `default`$[t]$ for type t, to the declared variable. Assignments to exclusive variables are also encoded directly as Viper assignments. For more complex assignments to exclusive targets, *e.g.* assignments to fields, we introduce functions that update fields and indices. The call `upd_f`(e_1, e_2) returns the struct that stores e_2 in field `f` and otherwise has the same fields as e_1. Shared variables are also encoded as variables, but now represent the memory location storing the value of the variable. We use Viper's inhale statement to create the permissions for the memory location. An inhale statement `inhale A` adds to Viper's verification state all permissions of the assertion `A` and then assumes that the functional specification of `A` holds. Conversely, exhale statements first assert that the specified assertion holds and then remove from the verification state all permissions of the assertion. As for exclusive variables, the encoding also assigns the default value to the declared variable. Assignments to shared locations are encoded by first exhaling the permissions to the shared location, thereby checking

that write permissions are held, and then inhaling the permissions again and assuming that the location now stores the left-hand side of the assignment. Encoding assignments by exhaling and inhaling permissions is a standard approach for Viper [5, 19]. Since shared variables represent memory locations, allocating new memory, which in Go may be done via the statement `new`, is encoded the same as a variable declaration.

Below we show the full encoding of our running example. Again, we put the encoded Go code into comments. All statements are encoded as defined in Fig. 2.12.

```
1  method link(x: 2MemTuple[Ref,Ref]) (p: 2MemTuple[Ref,Ref])
2    requires acc(loc1(x)._int, 1/2)
3    ensures  acc(loc1(x)._int, 1/2) &&
4             acc(loc0(p)._*node) && acc(loc1(p)._int)
5    ensures  tup2(loc0(p)._*node, loc1(p)._int) == tup2(x, loc1(x)._int)
6  {
7    // var n@ node
8    var n: 2MemTuple[Ref,Ref]
9    inhale acc(loc0(n)._*node) && acc(loc1(n)._int)
10   // n.next = x
11   exhale acc(loc0(n)._*node)
12   inhale acc(loc0(n)._*node) && loc0(n)._*node == x
13   // n.val = x.val
14   exhale acc(loc1(n)._int)
15   inhale acc(loc1(n)._int) && loc1(n)._int == loc1(x)._int
16   // return &n
17   return n
18 }
```

2.2.3. Interface Encoding

Analogous to our core encoding, we first present the encoding of interface values and then present how operations on interfaces, in particular, interface predicates and pure methods, are encoded.

Type Encoding. Conceptually, we encode interfaces as tuples of the dynamic type and the dynamic value, *i.e.* the value stored in the interface. Fig. 2.13 shows the Viper domains used in our interface encoding. To access the dynamic value and dynamic type of an interface, the domain `Interface` defines the functions `dynamicVal` and `dynamicTyp`, respectively. To ensure that the dynamic value of an interface is an instance of the interface's dynamic type, we construct interface values using the functions `toInterface_`T. For a dynamic type T, the function `toInterface_`T takes a value x of type T and returns an interface containing x. The function's postconditions express that the dynamic value and type of the interface are the value x and its static type T, respectively. For instance, `toInterface_int(5)` returns the interface containing the integer 5. Inconsistent instances, such as `toInterface_int(true)`, which attempts to create an interface with the dynamic type `int` and the dynamic value `true`, are rejected by Viper's type system.

We encode dynamic types as an ADT representation of Go's types. For the sake of brevity, we omit the definition of the domain `Type` encoding this ADT. The encoding of dynamic values is a bit more involved. In Go, every value may be cast to an interface. Therefore, our Viper representation of dynamic values also has to be able to capture every possible value. Viper does not have polymorphic types natively. We represent a polymorphic

```
domain Interface {
  func dynamicVal(Interface): DynamicValue
  func dynamicTyp(Interface): Type
}

function toInterface_T(x: [[type[T]]]) returns (y: Interface)
  ensures dynamicVal(y) == box_T(x)
  ensures dynamicTyp(y) == [[type[T]]]
```

```
domain DynamicValue {
  ...
  func box_T([[T]]): DynamicValue
  func unbox_T(DynamicValue): [[T]]

  axiom { ∀ x: [[T]] :: unbox_T(box_T(x)) == x }
  axiom {
    ∀ i: Interface :: dynamicTyp(i) == [[type[T]]]
        ⟹ box_T(unbox_T(dynamicVal(i))) == dynamicVal(i)
  }
  ...
}
```

Figure 2.13.: Definition of the `Interface` domain that Gobra uses to encode interface values. The domains `DynamicValue` and `Type` are used to encode the values stored inside of interfaces and the dynamic types of interfaces, respectively.

type in Viper using the `DynamicValue` domain shown in Fig. 2.13. To capture values for a Go type T, the domain defines the functions `box_`T that takes a value of type $[\![T]\!]$ and returns the corresponding dynamic value. Vice versa, the function `unbox_`T takes dynamic values and returns the original value that was put into the dynamic value, which is specified by the first axiom. The second axiom specifies that for interfaces with a dynamic type T, the function `unbox_`T is injective, which is required for our encoding of interface predicates and interface pure method, which is discussed later.

Encoding of Operations. Given the type encoding, the encoding of the basic operations on interfaces is straightforward. As shown below, we encode Gobra's `typeOf` function and Go's type assertions (`e.(T)`) using the getter for dynamic types and values. We use $e_1 \triangleright e_2$ to denote a Viper expression with value e_2, but for which Viper checks that the condition e_1 holds. We use this expression to encode well-definedness checks. For instance, to cast an expression e to a non-interface type T (`e.(T)` below), we first assert that the dynamic type of e is T and then get the dynamic value. Casts from interfaces to interfaces (`e.(I)` below) are encoded by asserting that the dynamic type of e is a subtype of the interface type. We encode Go's subtype relation using a domain function `subtype`. The call `subtype(a,b)` returns true if the type `a` was verified to be a behavioral subtype of type `b`. More concretely, Gobra generates an axiom that `subtype(a,b)` is true if there exists a corresponding implementation proof. Our treatment of the subtype relation over-approximates Go's subtype relation, *i.e.* the verification of a cast may fail even though Go does not throw a panic. Lastly, in Go, interfaces may only be compared with other values if they are *comparable*. As for the subtype relation, Gobra uses a domain function `comparable(t)` to encode whether the Go type `t` is comparable according to Go's language specification.

$$
\begin{aligned}
[\![\texttt{typeOf(}e\texttt{)}]\!] &\triangleq \texttt{dynamicTyp(}[\![e]\!]\texttt{)} \\
[\![e\texttt{.(T)}]\!] &\triangleq [\![\texttt{typeOf(}e\texttt{) == T}]\!] \rhd \texttt{unbox\_T(dynamicVal(}[\![e]\!]\texttt{))} \\
[\![e\texttt{.(I)}]\!] &\triangleq \texttt{subtype(}[\![\texttt{typeOf(}e\texttt{)}]\!]\texttt{,} [\![I]\!]\texttt{)} \rhd [\![e]\!] \\
[\![e_1 \texttt{ == } e_2]\!] &\triangleq \texttt{comparable(}[\![e_1]\!]\texttt{) \&\& comparable(}[\![e_2]\!]\texttt{)} \rhd [\![e_1]\!] \texttt{ == } [\![e_2]\!]
\end{aligned}
$$

Interface Predicates and Pure Methods. As explained in Sec. 2.1.2, the key ingredients of our treatment of Go interfaces are the interface predicates, specification methods, and implementation proofs. We explain how we handle the former two here; based on this encoding, the encoding of implementation proofs is analogous to methods.

Intuitively, we encode interface predicates as a case split over all possible implementations. All implementations not present in the current scope are subsumed by an abstract default case. Consequently, adding an implementation does not invalidate existing proofs, which enables modular reasoning. The predicate for the `stream` example (Fig. 2.2) is encoded as follows:

```
predicate memory(x: ⟦interface{}⟧) {
  ⟦typeOf(x) == *counter⟧ ? ⟦acc(x.(*counter))⟧ : unknownMemory(x)
}
predicate unknownMemory(x: ⟦interface{}⟧)

function hasNext(x: ⟦interface{}⟧) returns (y: ⟦bool⟧)
  req ⟦acc(x.memory(), _)⟧
  ens ⟦typeOf(x) == *counter⟧ ⟹ y == hasNext_Proof(⟦x.(*counter)⟧)
```

The body of the predicate branches on the dynamic type of `x`, with a single case for the (only) given implementation. The abstract predicate `unknownMemory` encodes the default case. The encoding of pure methods such as `hasNext` uses an analogous case split, but uses $\texttt{hasNext}_{\texttt{Proof}}$, which is part of the implementation proof (Fig. 2.2 Line 27) and couples the interface and implementation method. Our encoding of interface predicates is an instance of an *abstract predicate family* [64]. For Go, we have crafted the variant illustrated above that is well-suited for implementation proofs, pure interface methods, and structural subtyping.

[64]: Parkinson et al. (2005), *Separation logic and abstraction*

2.2.4. Closure Encoding

As discussed in Sec. 2.1.3, Gobra leverages interfaces to provide a powerful support for closures. The encoding of closures themselves is rather straightforward. Gobra encodes the values of closures with a domain type `Closure`. The domain defines neither functions nor axioms. The `implements` assertion, which specifies that a closure satisfies some specification, is encoded with the uninterpreted function `implements` as shown below. We use `F` to range over specifications. We encode *specification values*, *i.e.* the second parameter of the `implements` relation, with an ADT that defines a constructor for every specification occurring in a program. As shown below, the specification expression `positive{e}` used in our example in Fig. 2.3 is encoded as `positive_ctor(`$[\![e]\!]$`)`, where `positive_ctor` is the ADT constructor. The partially applied arguments of the specification become the arguments of the ADT constructor. Local

specifications, such as f_{SPEC} in Fig. 2.3, implicitly take the captured variables as arguments. Below, the added parameter `xp` is the reference of the captured variable `x`. If a user writes `f implements fSpec` in Fig. 2.3, Gobra treats this assertion as `f implements fSpec{xp}`.

$$[\![e \texttt{ implements F}]\!] \triangleq \texttt{implements}([\![e]\!], [\![F]\!])$$
$$[\![\texttt{f implements positive}\{e\}]\!] \triangleq \texttt{implements(f, positive\_ctor}([\![e]\!]))$$
$$[\![\texttt{f implements fSpec}\{\texttt{xp}\}]\!] \triangleq \texttt{implements(f, fSpec\_ctor}(xp))$$

As shown below, the creation of a closure with local specification `F` (`x := ...F...` below) is encoded by first declaring a variable and then assuming that the variable satisfies `F`. Gobra generates a separate method to verify that the body of the function literal actually satisfies its local specification. Fig. 2.14 shows the Gobra method `fSpec` that is generated to verify the function literal of our example in Fig. 2.3. As for the specification expression, the variables that are captured by the function literal are added as parameters to the generated method. In Fig. 2.14, as before, the argument `xp` of `fSpec` is the reference of the variable `x` captured by the closure. Calls to closures are encoded by first asserting that the closure actually satisfies the given specification and then calling the specification. For specifications with parameters, the parameters become the corresponding call arguments, *e.g.* `r = f(a) as fSpec` is encoded as `assert f implements fSpec{xp}; r = fSpec(xp, a)`, where `xp` is again the captured variable. Lastly, implementation proofs are encoded by assuming that the closure satisfies the specification. As for the creation of closures, Gobra generates a separate method to verify implementation proofs. Fig. 2.14 shows the method `main_f_proof` that Gobra generates to verify the implementation proof of the example in Fig. 2.3. Again, the parameter `xp` represents the captured variable. Gobra infers the precondition `f implements fSpec{xp}` from the call annotations inside the implementation proof.

$$\begin{aligned}
[\![\texttt{x := ...F...}]\!] &\triangleq \texttt{var x: Closure} \\
&\quad\ \texttt{inhale } [\![\texttt{x implements F}]\!] \\
[\![\texttt{x = } e_1\texttt{(}e_2\texttt{) as F}]\!] &\triangleq \texttt{assert } [\![e_1 \texttt{ implements F}]\!] \\
&\quad\ \texttt{x = F(}[\![e_2]\!]\texttt{)} \\
[\![\texttt{proof } e \texttt{ implements F \{...\}}]\!] &\triangleq \texttt{inhale } [\![e \texttt{ implements F}]\!]
\end{aligned}$$

2.2.5. Soundness of the Memory Model

We use our classification of shared and exclusive expressions to determine whether or not we treat data as heap data. However, a program may store data on the heap differently. Go's language specification does not specify whether variables or allocated memory is stored on the heap or stack. E.g., variables not annotated as shared actually may be stored on the heap. Conversely, shared variables or even data allocated with Go's new statement actually may be stored on the stack or in registers. In

```
requires acc(xp)
ensures  acc(xp) && m == max(old(*xp),n) && *xp == m
func fSpec(xp *int, n int) (m int) {
  if n > *xp {
    *xp = n
  }
  return *xp
}

requires f implements fSpec{xp}
requires f_comp{xp}.inv()
ensures  f_comp{xp}.inv() && r >= 0
func main_f_proof(f func(int) int, xp *int) (r int) {
  unfold f_comp{xp}.inv()
  r = f(a) as fSpec{xp}
  fold f_comp{xp}.inv()
}
```

Figure 2.14.: The Gobra methods generated for our example in Fig. 2.3 to check that the closure satisfies its local specification (f_{SPEC}) and to check that the closures satisfies the specification `positive{&x}` (`main_f_proof`).

this subsection, we provide a sketch of a formal argument to show that properties verified for a program imply that these properties also hold for the actual executions with their unspecified memory allocation.

Our argument consists of three steps: (1) Instead of specifying where data is stored, the Go compiler guarantees that all data may be treated as if it was allocated on the heap. Therefore, for every actual execution, there exists a *heapified* execution where all data is stored on the heap. This is our Go trust assumption. (2) We show that our type system ensures that if all data is stored on the heap, then the exclusive data may be treated as if it was not allocated on the heap. Therefore, for every execution where all data is stored on the heap, there exists an execution where exclusive data is not stored on the heap, but shared data is still stored on the heap. (3) Therefore, for every actual execution, there exists an execution verified by Gobra where only shared data is stored on the heap and thus, properties verified by Gobra are also satisfied by the actual executions.

Go's Guarantees. To better understand the Go compiler's guarantee that all data may be treated heap-allocated data, consider the following program on the left that returns the pointer to a variable.

```
func alloc() (*int) {
  var x int
  x = 2
  return &x
}
```

```
func alloc() (*int) {
  var xp *int = new(int)
  *xp = 2
  return xp
}
```

Go ensures that the left program behaves the same as the right program, where the value of the variable `x` is instead stored on the heap, pointed to by the pointer `xp`. Analogous to the encoding discussed in Sec. 2.2.2, the variable declaration and assignment become a heap allocation and a pointer assignment, respectively. Note that in languages such as C, the left program is unsafe because the variable `x` is stored on the stack and, thereby, a reference to the variable becomes invalid after the method returns. Internally, Go uses an escape analysis and other optimizations to decide whether or not to store variables and other data on the heap.

The next definition formalizes our aforementioned trust assumption for the Go compiler. We denote an execution of a program c starting from state σ and reaching a state σ' as $\langle c; \sigma \rangle \rightarrow_* \sigma'$, where $\rightarrow$ is the transition relation of an assumed small-step semantics for Go. To describe

heapified executions where all data is stored on the heap, we introduce an overloaded function $\mathcal{M}$ that maps states to states where all data is stored on the heap and maps programs to programs operating on the data accordingly. For our two programs above, the right program represents the heapified version of the left program, *i.e.* the right program is the result of applying $\mathcal{M}$ to the left program.

Definition 2.2.1 (Go Trust Assumption) *For every execution* $\langle c; \sigma \rangle \rightarrow_* \sigma'$, *there exists the execution* $\langle \mathcal{M}(c); \mathcal{M}(\sigma) \rangle \rightarrow_* \mathcal{M}(\sigma')$, *where all variables and allocated memory are stored on the heap.*

Guarantees of the Type System. The next definition captures the second step of our soundness argument, namely how we define soundness for our type system. As discussed above, the type system must ensure that if all data is stored on the heap, then the exclusive data may be treated as if it was not allocated on the heap. More formally, there must exist an execution that does not crash and where the exclusive data is not stored on the heap. Analogous to the function $\mathcal{M}$, we introduce a function $\mathcal{P}_\Gamma$ that maps states and programs to the corresponding states and programs, respectively, where exclusive variables are not stored on the heap anymore. The context Γ captures which variables are annotated as shared. We use $\Gamma \vdash c$ to denote that a program c is well-typed given the annotations Γ. For our two programs above, the variable `x` is annotated as shared. Therefore, applying P_Γ to the right program does not change anything (since there are no exclusive variables). For exclusive variables, the changes done by the $\mathcal{M}$ function are reversed by $\mathcal{P}_\Gamma$.

Definition 2.2.2 (Type System Soundness) *For all well-typed programs* c *with addressibility annotations* Γ, *denoted as* $\Gamma \vdash c$, *and for every execution* $\langle \mathcal{M}(c); \mathcal{M}(\sigma) \rangle \rightarrow_* \sigma'$, *there exists the execution* $\langle \mathcal{P}_\Gamma(\mathcal{M}(c)); \mathcal{P}_\Gamma(\mathcal{M}(\sigma)) \rangle \rightarrow_* \mathcal{P}_\Gamma(\sigma')$, *where all exclusive variables are not stored on the heap.*

We have not formally proved soundness of our Go type system. Such a proof requires formalized language semantics for Go, which do not exist. We have formalized parts of the argument in Isabelle/HOL for a toy programming language without concurrency. This proof proceeds by induction on the step relation. We identify two key parts of the argument: First, because we only consider heapified executions and, thereby, all variables are allocated on the heap, we are guaranteed that accessing these variables, *e.g.* the dereference of `xp` in our example, do not cause null-pointer exceptions. Second, because well-typed programs do not take references of exclusive variables, references of exclusive variables do not influence the program behavior. More formally, for every expression e in a well-typed program and every state σ, the expression $\mathcal{P}_\Gamma(\mathcal{M}(e))$ evaluates in the state $\mathcal{P}_\Gamma(\mathcal{M}(\sigma))$ to the same value as the expression $\mathcal{M}(e)$ in the state $\mathcal{M}(\sigma)$. Both parts together justify that we may safely move exclusive variables that are allocated on the heap to the stack without changing the program behavior.

Entailment of Properties. Theorem 2.2.1 formalizes the soundness of our memory model. Assuming that the Go trust assumption and type system soundness holds, then if a program c satisfies the specification

$\{P\}c\{Q\}$ in our memory model, denoted as $\Gamma \vDash_G \{P\}c\{Q\}$, then the actual executions of the program satisfy the specification $\{\lfloor P \rfloor\}c\{\lfloor Q \rfloor\}$. The function $\lfloor \cdot \rfloor$ erases the permissions from an assertion. For instance, if we verify a postcondition `acc(&x) && x == 1` in Gobra, then just the functional specification `x == 1` holds for the actual program. Since we do not know where memory is allocated, permission annotations only have meaning when reasoning about memory safety. In particular, permissions do not have an interpretation for actual executions.

Theorem 2.2.1 *If the Go trust assumption and type system soundness holds, then* $\Gamma \vDash_G \{P\}c\{Q\}$ *implies* $\vDash \{\lfloor P \rfloor\}c\{\lfloor Q \rfloor\}$.

Before we discuss a proof sketch for Theorem 2.2.1, we first formally define when a specification holds according to Gobra's memory model and according to the actual memory model.

Def. 2.2.3 defines when the actual executions of a Go program c satisfy a spec $\{\hat{P}\}c\{\hat{Q}\}$. The definition reflects the standard definition of program correctness: The executions of a program satisfy a spec $\{\hat{P}\}c\{\hat{Q}\}$, if an execution of the statement c that starts in a state that satisfies the precondition $\hat{P}$ reaches a state σ', then σ' must satisfy the postcondition $\hat{Q}$. We use $\hat{P}$ and $\hat{Q}$ to range over assertions without permission annotations.

Definition 2.2.3 (Spec Semantics) *A program c satisfies a postcondition $\hat{Q}$ given a precondition $\hat{P}$, denoted as* $\vDash \{\hat{P}\}c\{\hat{Q}\}$, *if*

$$\forall \sigma.\ \sigma \vDash \hat{P} \land \langle c; \sigma \rangle \rightarrow_* \sigma' \Rightarrow \sigma' \vDash \hat{Q}$$

Next, Def. 2.2.4 defines when a program satisfies a spec according to Gobra's memory model. In contrast to Def. 2.2.3, Gobra considers just the executions where only shared data is stored on the heap. Furthermore, we require that the program and spec are well-typed.

Definition 2.2.4 (Gobra's Spec Semantics) *In Gobra, a program c with the addressibility annotations Γ satisfies a postcondition Q given a precondition P, denoted as* $\Gamma \vDash_G \{P\}c\{Q\}$, *if the program is well-typed, i.e.* $\Gamma \vdash c$, $\Gamma \vdash P$, *and* $\Gamma \vdash Q$ *hold, and*

$$\forall \sigma.\ \mathcal{P}_\Gamma(\mathcal{M}(\sigma)) \vDash_G P \land \langle \mathcal{P}_\Gamma(\mathcal{M}(c)); \mathcal{P}_\Gamma(\mathcal{M}(\sigma)) \rangle \rightarrow_* \sigma' \Rightarrow \sigma' \vDash_G Q$$

Finally, we provide a proof sketch for Theorem 2.2.1.

Proof. The Go trust assumption (Def. 2.2.1) entails that for every actual execution $\langle c; \sigma \rangle \rightarrow_* \sigma'$, there exists the execution $\langle \mathcal{M}(c); \mathcal{M}(\sigma) \rangle \rightarrow_* \mathcal{M}(\sigma')$. In combination with type system soundness (Def. 2.2.2), we get that for every actual execution $\langle c; \sigma \rangle \rightarrow_* \sigma'$, there exists the execution verified by Gobra $\langle \mathcal{P}_\Gamma(\mathcal{M}(c)); \mathcal{P}_\Gamma(\mathcal{M}(\sigma)) \rangle \rightarrow_* \mathcal{P}_\Gamma(\mathcal{M}(\sigma'))$. To connect the pre- and postconditions P, Q and $\lfloor P \rfloor$, $\lfloor Q \rfloor$, we have to show that whether an assertion is satisfied does not depend on whether or not data is stored on the heap. More formally, for all well-typed assertions A, $\sigma \vDash \lfloor A \rfloor \Leftrightarrow \mathcal{P}_\Gamma(\mathcal{M}(\sigma)) \vDash_G A$ holds for all states σ. This property holds since with the exception of permission assertions, which are erased, Gobra's assertion language does not specify where data is stored. The remainder of the proof of Theorem 2.2.1 is straightforward. $\sigma \vDash \lfloor P \rfloor$ implies $\mathcal{P}_\Gamma(\mathcal{M}(\sigma)) \vDash_G P$.

Therefore, we can use Def. 2.2.4 to get $\mathcal{P}_\Gamma(\mathcal{M}(\sigma')) \vDash_G Q$, which entails $\sigma' \vDash \lfloor Q \rfloor$, concluding the proof sketch. □

2.3. Slicing Verification Conditions

A challenge for the verification of large-scale software is that the encoded Viper programs may become too large for Viper's verification backends to handle, causing verification to not terminate. An issue for very large programs is that while methods and functions can be verified separately, the verification of each method and function is impacted by the program's *context*, *i.e.* the functions, predicates, domains, and fields defined in the same Viper program. As a consequence, a larger context increases the complexity of the verification task handled by Viper's backends. We reduce the size of contexts by applying program slicing techniques [65, 66] to the Viper programs generated by Gobra. More concretely, we introduce an algorithm to split a Viper program c into a set of smaller Viper programs $\mathscr{C}$ with smaller contexts such that if all programs in $\mathscr{C}$ are correct, then the program c is correct. By reducing the context per program in $\mathscr{C}$, the algorithm enables us to verify programs whose verification does not terminate otherwise. We make the algorithm available to users of Gobra via a command line option. The option specifies only (1) that programs should be split and (2) the maximum number of programs that programs should be split into. Gobra then applies the algorithm on the generated Viper program and invokes Viper's backends on each resulting program separately. Users of Gobra do not have to be aware of the details of the algorithm. In particular, users do not have to provide additional annotations. Our introduced algorithm is orthogonal to Gobra's encoding and can be applied in different contexts as well.

[65]: Weiser (1984), *Program Slicing*
[66]: Horwitz et al. (1988), *Interprocedural Slicing Using Dependence Graphs*

2.3.1. Slicing Algorithm

Our algorithm splits a Viper program c in two phases: In the first phase, for every method and function of c, we compute the smallest subset of c necessary to verify the method or function. In the second phase, we incrementally merge these subsets until the user-specified bound of the maximum number of programs is reached.

In the remainder of this section, we discuss the first phase. The second phase is driven by a heuristic that defines a cost function for the merge of two programs, where we greedily perform the cheapest merge until the targeted number of programs remains. We leave the exploration of alternative merging strategies to future work.

To illustrate the result of our algorithm, we use the toy example shown in Fig. 2.15. The Viper program on the left contains two methods. The method `code1` takes a reference, for which the precondition requires an instance of the `mem` predicate. The predicate is then unfolded in the body of `code1`, getting access to the permission `acc(x.val)`. The method `code2` takes a reference, calls `code1`, and returns the empty list. The example represents lists with the domain type `List`, which provides functions for the empty list (`nil`), adding an element to a list (`cons`), and the length of a list (`len`). Two axioms define the standard properties of the length

```
1  field val: Int
2
3  pred mem(x: Ref) { acc(x.val) }
4
5  domain List {
6    func nil(): List
7    func cons(Int, List): List
8    func len(): Int
9    axiom { len(nil()) == 0 }
10   axiom {
11     ∀i:Int, l:List :: {cons(i,l)}
12     len(cons(i,l)) == 1+len(i)
13   }
14 }
15
16
17 method code1(x: Ref)
18   requires mem(x)
19 { unfold mem(x) }
20
21 method code2(x: Ref) (r: List)
22   requires mem(x)
23 {  code1(x); r := nil() }
```

```
1  field val: Int
2
3  pred mem(x: Ref) { acc(x.val) }
4
5  method code1(x: Ref)
6    requires mem(x)
7  { unfold mem(x) }
```

```
1  pred mem(x: Ref)
2
3  domain List {
4    func nil(): List
5    func len(): Int
6    axiom { len(nil()) == 0 }
7  }
8
9  method code1(x: Ref)
10   requires mem(x)
11
12 method code2(x: Ref) (r: List)
13   requires mem(x)
14 { code1(x); r := nil() }
```

Figure 2.15.: A toy example of a Viper program (left), together with the program's smallest subset necessary to verify the method `code1` (top right) and the method `code2` (bottom right), respectively.

function. Line 11 shows a detail that we typically omit in this thesis for the sake of simplicity. As for SMT, the *trigger* [67] `{cons(i,l)}` specifies when the quantified expression is instantiated. For the context of this section, the only relevant behavior of triggers is that if the expression in the trigger, *e.g.* `cons(i,l)` cannot be constructed (because functions are missing), then the body of the quantifier does not influence verification. On the right, we show the subsets of the program necessary to verify the `code1` method (top right) and the `code2` method (bottom right), respectively, which our algorithm returns for the program on the left. For the `code1` method, the unused domain is not necessary and, thus, omitted. For the `code2` method, since the predicate instance is not unfolded in the method's body, the definition of the `mem` predicate and, thereby, the definition of the `val` field is not necessary. Furthermore, without a use of the `cons` function, the definition of the `cons` function together with its axiom are omitted, too. Our algorithm over-approximates the treatment of axioms. For instance, as shown in Fig. 2.15, we include the axiom `len(nil()) == 0` even though that axiom is not necessary to verify the method.

To compute the necessary program subsets, we construct a dependency graph for a program. In the graph, every node represents a part of the program. An edge from some part a to another part b represents that for verification, if a program contains a, then the program must also contain b. The smallest subset of a program necessary to verify a method or function m is then the set of all parts that are reachable from the node representing m.

Nodes. As illustrated by the example, we want a fine granularity of program parts that, for instance, distinguishes between predicates with and without body and between individual domain functions and axioms. We achieve this granularity by introducing a node for each part that we want to distinguish. Accordingly, for every method, function, and predicate, our dependency graph contains a node for the member with and without a body. Furthermore, we introduce separate nodes for each domain function and axiom.

$$\mathsf{uses}(\texttt{acc(p(}E_1\texttt{),}E_2\texttt{)}) = \{\mathsf{p}_{\mathsf{spec}}\} \cup \mathsf{uses}(E_1, E_2)$$
$$\mathsf{uses}(\texttt{unfolding acc(p(}E_1\texttt{),}E_2\texttt{) in }E_3) = \{\mathsf{p}_{\mathsf{body}}\} \cup \mathsf{uses}(E_1, E_2, E_3)$$
$$\mathsf{uses}(\texttt{f(}E\texttt{)}) = \{\mathsf{f}_{\mathsf{spec}}\} \cup \mathsf{uses}(E)$$
$$\mathsf{uses}(\texttt{reveal f(}E\texttt{)}) = \{\mathsf{f}_{\mathsf{body}}\} \cup \mathsf{uses}(E)$$

Figure 2.16.: A snippet of the definition of the function uses, which takes a Viper expression, type, or statement x and returns the nodes of the parts that are necessary to verify x.

$$\begin{aligned}
&\mathsf{deps}(\texttt{method m(V) (R) req P ens Q \{ B \}}) = \\
&\quad \mathsf{m}_{\mathsf{body}} \longrightarrow \mathsf{uses}(\texttt{V,R,P,Q,B}) \cup \mathsf{m}_{\mathsf{spec}} \longrightarrow \mathsf{uses}(\texttt{V,R,P,Q}) \\
&\mathsf{deps}(\texttt{function f(V) (R) req P ens Q \{ B \}}) = \\
&\quad \mathsf{f}_{\mathsf{body}} \longrightarrow \mathsf{uses}(\texttt{V,R,P,Q,B}) \cup \{\mathsf{f}_{\mathsf{spec}} \longrightarrow \mathsf{f}_{\mathsf{body}}\} \\
&\mathsf{deps}(\texttt{opaque function f(V) (R) req P ens Q \{ B \}}) = \\
&\quad \mathsf{f}_{\mathsf{body}} \longrightarrow \mathsf{uses}(\texttt{V,R,P,Q,B}) \cup \mathsf{f}_{\mathsf{spec}} \longrightarrow \mathsf{uses}(\texttt{V,R,P,Q}) \\
&\mathsf{deps}(\texttt{predicate p(V) \{ B \}}) = \\
&\quad \mathsf{p}_{\mathsf{body}} \longrightarrow \mathsf{uses}(\texttt{V,B}) \cup \mathsf{p}_{\mathsf{spec}} \longrightarrow \mathsf{uses}(\texttt{V}) \\
&\mathsf{deps}(\texttt{field f: T}) = \texttt{f} \longrightarrow \mathsf{uses}(\texttt{T}) \\
&\mathsf{deps}(\texttt{domain d \{ FN; AX \}}) = \mathsf{deps}(\texttt{FN}) \cup \mathsf{deps}(\texttt{AX}) \\
&\mathsf{deps}(\texttt{axiom ax \{ A \}}) = \\
&\quad \mathsf{srcs}(\texttt{A}) \longrightarrow \texttt{ax} \cup \texttt{ax} \longrightarrow \mathsf{uses}(\texttt{A}) \\
&\mathsf{srcs}(\texttt{forall V :: Triggers B}) = \mathsf{srcs}(\texttt{Triggers})
\end{aligned}$$

Figure 2.17.: Definition of the deps function that takes a part of a Viper program and returns the set of dependencies (represented as edges) induced by that part. We denote an edge from a node n_1 to a node n_2 as $n_1 \longrightarrow n_2$. We use $n \longrightarrow N$, where N is a set of nodes, as a shorthand for the set $\{n \longrightarrow m \mid m \in N\}$. Furthermore, we use $\mathsf{uses}(\texttt{X}_1, \ldots, \texttt{X}_i)$ as a shorthand for $\mathsf{uses}(\texttt{X}_1) \cup \cdots \cup \mathsf{uses}(\texttt{X}_n)$.

Edges. To define the dependencies that we extract from a Viper program, we introduce two functions, namely uses and deps. The function uses returns the nodes that are necessary for a specific type, expression, or assertion. Fig. 2.16 shows the most important cases of the function. The nodes $\mathsf{n}_{\mathsf{body}}$ and $\mathsf{n}_{\mathsf{spec}}$ represent the member named n with and without its body, respectively. A predicate instance of p requires that the predicate p, potentially without its body, is included in the Viper program. The body of a predicate is only required if the predicate is unfolded. Similarly, the body of a function is definitely required if a function call is annotated with Viper's `reveal` expression.

The function deps, defined in Fig. 2.17, returns all edges induced by a program part. The cases for methods, functions, predicates, and fields are straightforward. For instance, a method with a body has dependencies to the parts used in the method's arguments (V), results (R), preconditions (P), postconditions (Q), and body (B). Without its bodies, there are no dependencies to the body (B). For functions, unless a function is annotated with a special keyword `opaque`, Viper considers a function's body part of its specification. We capture this behavior with an edge from a function's specification to its body (the edge $\mathsf{f}_{\mathsf{spec}} \longrightarrow \mathsf{f}_{\mathsf{body}}$ in Fig. 2.17). Domain definitions introduce edges for all domain functions and axioms.

Since users do not annotate when axioms are used, axioms are treated differently than, for instance, methods or predicates. Identifying whether an axiom is required for verification is hard. For instance, an axiom may be required even though not all members referenced in the axiom are included in a program. Consider the two axioms $\forall$`x :: f(x) == g(x)` and $\forall$`x :: g(x) == 1`. Even if a program includes only the function `f`, but not the function `g`, both these axioms are required to verify that `f(x)` is equal

to 1. Our algorithm over-approximates whether an axiom is required by including axioms if at least one of the members referenced in the axiom is included. As an optimization, since quantified expressions are only instantiated based on their trigger, to determine whether a quantified expression inside an axiom has to be included, we only analyze the trigger of the quantified expression (excluding its body). Because of our over-approximation, some axioms are included even though they are not required to verify a method or function. For instance, as shown in Fig. 2.15, we include the axiom `len(nil()) == 0` because the function `nil()` is required to verify the method `code2`. In Fig. 2.17, the auxiliary function $\mathsf{srcs}(e)$ returns the nodes that have a dependency to the expression e occurring inside of an axiom. The srcs function returns the same nodes as the uses function, except that the body of quantified expressions is disregarded and that types are excluded.

2.4. Evaluation and Case Studies

The Gobra implementation consists of a parser and type checker for annotated Go programs and a translation of those programs into the Viper intermediate verification language. The resulting Viper program is verified using Viper's symbolic execution backend, which in turn uses the Z3 SMT solver [1]. Verification errors are translated back to the Go level, such that users are not exposed to the internal encodings. Users never have to inspect the encoding. Error messages contain the failing assertion and a reason describing why the assertion failed. Gobra's test suite contains 407 verification tests (with and without errors) with a total of 10'030 LOCs (Go code and annotations) that take 14.9 minutes to verify.

[1]: De Moura et al. (2008), *Z3: An Efficient SMT Solver*

2.4.1. Evaluation

We evaluated Gobra on 16 interesting verification problems, which include well-known algorithms and data structures, and cover Go's main features, such as interfaces (Examples 7,10,and 11), closures (Examples 8 and 13), and concurrency primitives (Examples 13 and 14), including goroutines, mutexes, wait groups, and channels. For each example, Gobra verifies memory safety and functional correctness properties. To assess Gobra's performance on failing verifications, we have additionally constructed two incorrect variations of each example, one with a seeded error in the specification and one in the implementation.

All experiments were executed on a warmed-up JVM on a MacBook Pro with a 2.3 GHz 8-Core Intel Core i9 CPU and 32 GB of RAM, running macOS 11.1 and OpenJDK 11. For each experiment, we measured its verification time using Viper's symbolic execution backend and averaged the duration of twelve executions, excluding the slowest and fastest outlier.

Fig. 2.18 summarizes the results, including the required annotations and verification times for the three variants of each example. The annotation overhead ranges between 0.3 and 3.1 lines of annotations per line of code, which is typical for SMT-based deductive verifiers. Verification

#	Example	LOC / Spec.	Viper LOC	T[s]	$T_{\text{spec error}}$[s]	$T_{\text{impl error}}$[s]
1	binary search tree	125 / 140	632	10.88	10.50	11.67
2	dutchflag	22 / 16	142	2.02	1.78	1.88
3	heapsort	47 / 93	271	16.72	19.30	15.23
4	dense and sparse matrix	69 / 62	326	10.46	10.55	10.06
5	binary tree	59 / 20	217	2.09	2.08	2.11
6	running ex. (Fig. 2.1)	10 / 11	164	1.71	1.70	1.70
7	running ex. (Fig. 2.2)	24 / 16	186	1.04	0.98	1.01
8	running ex. (Fig. 2.3)	15 / 19	172	1.13	1.04	1.05
9	relaxed prefix	25 / 36	158	7.08	5.36	4.19
10	list of interfaces	46 / 27	219	1.45	1.41	1.54
11	visitor pattern	76 / 30	475	4.38	4.22	5.45
12	zune	31 / 12	141	1.08	1.07	1.06
13	slice map	30 / 95	611	44.23	41.93	41.63
14	pair insertion sort	50 / 105	353	15.55	12.64	13.96
15	parallel search replace	35 / 94	565	53.18	51.97	61.54
16	parallel sum	31 / 98	527	58.39	50.25	57.69

Figure 2.18.: Experimental results. For each experiment, we list the number of lines of Go code (LOC), number of lines of specification and proof annotations (Spec), and the average verification time in seconds for correct examples (T), errors in the specification ($T_{\text{spec error}}$), and errors in the implementation ($T_{\text{impl error}}$). A line containing both, code and annotations, is counted as one line of Go code and one line of annotation.

times range between a second and a minute per example. The verification times are significantly higher when the verified code uses concurrency features; these examples require quantitatively more and more-complex specifications, which complicates reasoning. Lastly, there is hardly any difference between successful and failed verification attempts. Consistent performance is crucial when verifiers are used interactively, where users run them frequently, especially on programs that do not yet verify.

2.4.2. Case Studies

Since Gobra's creation, Gobra has been applied successfully to verify large real-world software. In this subsection, we provide an overview of the two largest bodies of code that have been verified with Gobra.

WireGuard. WireGuard is a widely-used Virtual Private Network (VPN). In the protocol, two agents first establish a secret session key in a handshake phase and then use this key to exchange messages in a transport phase. In the work by Arquint et al. [56], Gobra has been used to verify that a modified version of WireGuard's official Go implementation [68] refines a specification of the WireGuard protocol, which was formalized in the Tamarin [69] protocol verifier. Their verified WireGuard implementation consists of 608 lines of Go code. The program has 3,936 lines of specifications and proof annotations, mostly due to the large specification overhead of their verification technique. The annotation overhead remains below 3.1 lines of proof annotations per line of code, which as stated in the evaluation, is typical for SMT-based verifiers. The verification of the entire code base takes around 286 seconds.

The official Go implementation was changed in two ways: (1) To reduce verification effort, DDos protection, load balancing, and metrics were omitted. In particular, load balancing requires complex concurrency reasoning not supported by Gobra. (2) -Cryptographic operations and

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

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

[69]: Meier et al. (2013), *The TAMARIN Prover for the Symbolic Analysis of Security Protocols*

network operations were moved into trusted libraries. The individual steps processing a connection, *i.e.* parsing and constructing messages, have remained unchanged. In particular, the verified implementation is interoperable with WireGuard's official Go implementation.

[49]: Zhang et al. (2011), *SCION: Scalability, control, and isolation on next-generation networks*
[50]: Pereira et al. (2024), *Protocols to Code: Formal Verification of a Next-Generation Internet Router*

Scion. The Scion Internet architecture [49] is a new inter-networking infrastructure that focuses on security and reliability. In the work by Pereira et al. [50], similar to WireGuard, they used Gobra to verify that Scion's border router, the component that is responsible for packet forwarding, refines a formal specification of the Scion protocol. In contrast to the WireGuard case study, they directly targeted the performance-optimized open-source implementation of Scion's border router with the exception of three minor changes to the code: (1) They rewrote a specific combination of interfaces and Go's embedded fields that Gobra does not support. Before their verification work, we were not aware that this combination of features is permitted in Go. (2) To simplify the permission-based reasoning, they replaced some uses of iterators with standard for-loops. (3) To add proof annotations for intermediate computation results, they split some compound expressions into smaller parts.

In total, they verified 4,700 lines of Go code against 900 lines of specification with 2,400 lines of specifications for trusted libraries. Verification required 13,400 lines of proof annotations, resulting in an annotation overhead of around 2.8 lines of proof annotations per line of code. The verification of the border router with Gobra takes around three hours.

This case study in particular demonstrates that Gobra is capable of verifying large-scale real-world code.

2.5. Related Work and Conclusion

[59]: Chajed et al. (2019), *Verifying concurrent, crash-safe systems with Perennial*
[70]: Jung et al. (2018), *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*
[7]: consortium (n.d.), *The Coq proof assistant*

Besides Gobra, we are aware of two other verification approaches for Go. Perennial [59] reasons about concurrent, crash-safe systems. Their core techniques are an extension to the Iris framework [70] and independent of Go. They connect their theory to Go programs with Goose, a shallow embedding of Go into Coq [7], which proves that Go code complies with a given transition system. In contrast to Gobra, Perennial does not support core Go features such as channels and interfaces.

[57]: Lange et al. (2017), *Fencing off Go: liveness and safety for channel-based programming*
[58]: Lange et al. (2018), *A static verification framework for message passing in Go using behavioural types*
[71]: Gabet et al. (2020), *Static Race Detection and Mutex Safety and Liveness for Go Programs*
[72]: Hüttel et al. (2016), *Foundations of Session Types and Behavioural Contracts*
[73]: Cranen et al. (2013), *An Overview of the mCRL2 Toolset and Its Recent Advances*

Several prior works [57, 58, 71] infer behavioral types [72] to reason about Go's channel-based message passing. After they infer behavioral types for a given program, they check safety and liveness properties on the inferred types, using model checkers such as mCRL2 [73]. Some works use additional analyses to strengthen the provided guarantees. Lange et al. [58] add a termination analysis to enable one to verify unbounded properties under certain conditions. Gabet and Yoshida [71] extend this work by inferring behavioral types on shared variables and locks to additionally reason about data-race freedom, lock safety, and lock liveness. The approaches by Lange et al. [58] and Gabet and Yoshida [71] are vastly different from Gobra. They do not verify code contracts, but instead verify global properties such as deadlock and data-race freedom. Their automation is high and annotation overhead minimal, but their

analyses are not modular and do not verify functional properties of code. Furthermore, they do not verify properties about the state of the heap.

There are some prior works that can handle channel-based concurrency and heap-manipulating programs, but these do not apply directly to Go. Villard et al. [74] introduce a powerful contract mechanism to specify protocols that channels must adhere to. Their channel specification language is more expressive than the one presented in this paper. Their contracts are finite state machines and thus can have multiple phases. However, their channels are always shared between two peers whereas Go supports more advanced concurrency patterns where both channel endpoints are shared between an unbounded number of peers. Actris [75, 76] is a concurrent separation logic built on top of the Iris framework to reason about session types in an interactive theorem prover. Actris can go beyond two peers, but to do so, it requires a memory model that is incompatible with Go's memory model. Actris models the sharing of channel endpoints via Iris' ghost locks, which to our knowledge, implies sequentialization of sends, and dually receives, which is not guaranteed by Go's memory model.

[74]: Villard et al. (2009), *Proving Copyless Message Passing*

[75]: Hinrichsen et al. (2019), *Actris: Session-type based reasoning in separation logic*

[76]: Hinrichsen et al. (2020), *Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic*

Gobra's verification logic and encoding into Viper have been inspired by several other Viper-based verifiers, such as Nagini [18] for Python, Prusti [19] for Rust, and VerCors [16] for Java. None of these verifiers address the Go-specific features that Gobra supports.

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

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

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

Very recently, there has been work that introduces a type modifier similar to Gobra's exclusive variables. Lorenzen et al. [77] propose an extension of OCaml's type system that tracks a variety of different type modifiers to identify which data may be allocated on the stack and which updates to immutable data structures may be performed in-place. Their introduced `local` and `global` type modifier coincide with Gobra's exclusive and shared modifier, respectively. Analogous to Gobra, local memory locations may not escape their current scope. In contrast to Gobra, which uses the modifiers to determine whether permission-based reasoning is necessary for a memory location, they use the `local` modifier to determine that a memory location may be safely put on the stack, improving the performance of programs. Furthermore, they use the modifier in the context of a larger type system to reason about the uniqueness of references to immutable data structures, going significantly beyond Gobra's use of the exclusive modifier.

[77]: Lorenzen et al. (2024), *Oxidizing OCaml with Modal Memory Management*

Conclusion. We introduced Gobra, the first modular verifier for Go that supports reasoning about a crucial aspect of the language: the combination of channel-based concurrency and heap-manipulating constructs. Moreover, Gobra is the first verifier to support Go's version of interfaces and structural subtyping. Gobra is expressive and performant enough to verify large-scale real-world code. For instance, in the work by Pereira et al. [50] Gobra has been applied to verify the implementation of a full-fledged network router [49]. In other chapters, we expand the properties that can be verified with Gobra to certain hyper-properties.

[50]: Pereira et al. (2024), *Protocols to Code: Formal Verification of a Next-Generation Internet Router*

[49]: Zhang et al. (2011), *SCION: Scalability, control, and isolation on next-generation networks*

Verifiable Security Policies for Distributed Systems

3.

"Those are my principles, and if you don't like them... well, I have others."

— Groucho Marx

For programs handling confidential data, one crucial concern is *secure information flow*, meaning that confidential data is not leaked during the program's execution. In this context, *security policies* express (1) the *classification* of data, for instance, by designating part of the data as sensitive and others as public information, and (2) the *declassification* of data, that is, rules that describe when sensitive data can be deliberately treated as public.

Consider a simple authentication service. A security policy may classify that keys read from disk and network packets have high and low sensitivity, respectively. For declassification, a security policy may permit declassifying messages signed with the read key.

Recent works [26, 78–81] have introduced policy frameworks both to formally define security policies and to verify that code actually satisfies a defined policy. These existing approaches have two limitations: (1) Their frameworks are tightly linked to a programming language, which has two drawbacks. First, they cannot express policies in a language-agnostic way, which is for instance useful in distributed systems, where different nodes may be implemented in different languages. Second, reasoning about policies happens at the level of the programming language and, thus, involves the full complexity of the language. (2) Existing approaches enable verifying that an implementation satisfies a policy, but do not support reasoning about the policy itself, in particular, to validate that it expresses the intended security requirements.

[26]: Popescu et al. (2021), *Bounded-Deducibility Security (Invited Paper)*
[78]: Schoepe et al. (2020), *VERONICA: Expressive and Precise Concurrent Information Flow Security*
[79]: Murray et al. (2023), *Assume but Verify: Deductive Verification of Leaked Information in Concurrent Applications*
[80]: Banerjee et al. (2008), *Expressive Declassification Policies and Modular Static Enforcement*
[81]: Smith (2022), *Declassification Predicates for Controlled Information Release*

This Work. We introduce a new policy framework that addresses these limitations by expressing policies over traces of *I/O actions*, the basic building blocks of communication, such as sending or receiving a message. This language-independent representation is well-suited for distributed systems, where attackers observe the I/O behavior of a program, but *not* the content of the memory.

To specify *classification*, we associate with each I/O action pre- and postconditions that express the sensitivity of outputs and inputs, respectively. For *declassification*, we introduce a designated action $decl(x)$, which declassifies a value x. A security policy is then a tuple of a *classification spec* (the pre- and postconditions for I/O actions) and an *IOD spec*, specifying the traces of I/O actions and declassification actions that an implementation may produce.

Such policies are independent of the program to be verified, a specific programming language, and the verification logic used to prove that an implementation satisfies a policy. In particular, the same policy can be used for multiple different implementations, even with different programming languages, which addresses the first limitation of existing frameworks discussed above. Regarding the second limitation, a key advantage of our framework is that policies can be audited completely independent of code and programming language, both formally and informally. In particular, we introduce a verification technique to show

that *all* programs satisfying the policy guarantee that specific data remains confidential even in the presence of declassification.

To prove that a concrete program satisfies a given policy, one can use standard program verification techniques. We show how to use ghost state (state that is used for verification but erased during compilation) to store the trace of I/O actions and declassification actions produced by a program execution. We can then prove that a program satisfies a policy by showing that (1) the stored traces refine the policy's IOD spec and (2) the stored traces satisfy secure information flow as expressed by the policy's classification spec.

For the latter, we introduce a new formalization of secure information flow that can deal with declassification and I/O behavior, and is expressive enough for language agnostic policies. Our formalization is inspired by observational determinism [82, 83] and amenable to standard verification techniques such as self-composition [22, 23, 84, 85]. However, in contrast to observational determinism, our formalization is not restricted to deterministic I/O behavior, which is crucial for concurrent and distributed systems.

[82]: McLean (1992), *Proving Noninterference and Functional Correctness Using Traces*
[83]: Roscoe (1995), *CSP and determinism in security modelling*
[22]: Barthe et al. (2011), *Secure information flow by self-composition*
[23]: Barthe et al. (2011), *Relational Verification Using Product Programs*
[84]: Barthe et al. (2013), *Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification*
[85]: Eilers et al. (2018), *Modular Product Programs*

We formalize our technique for verifying an implementation against a security policy based on the SecCSL logic [14] and prove its soundness in Isabelle/HOL. We demonstrate the practicality of our approach on a variety of case studies, including an implementation of the WireGuard protocol. These case studies are carried out using Gobra [51], an automated code verifier for Go, which demonstrates that our approach is amenable to automation using SMT solvers. Moreover, they show that our policy framework supports established security policy concepts such as delimited release [86], state-dependent declassification, and robust declassification [87].

[14]: Ernst et al. (2019), *SecCSL: Security Concurrent Separation Logic*
[51]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*
[86]: Sabelfeld et al. (2003), *A Model for Delimited Information Release*
[87]: Zdancewic et al. (2001), *Robust Declassification*

Contributions. We make the following contributions:

- We introduce a new policy framework based on I/O behaviors that allows one to express classification and declassification requirements independently of a given program or programming language.
- We show how to reason formally about the guarantees provided by a policy, enabling formal audits of security policies.
- We present a technique to verify that an implementation satisfies a given policy. We formalize this technique based on SecCSL and prove its soundness in Isabelle/HOL.
- We illustrate the expressiveness of our policy framework on several case studies, including an implementation of the Wireguard protocol. These case studies also show that our technique is amenable to SMT-based verification and scales to real-world code of considerable size.

This chapter is based on a paper that is currently under submission. All proofs shown in this chapter have been proved in Isabelle/HOL.

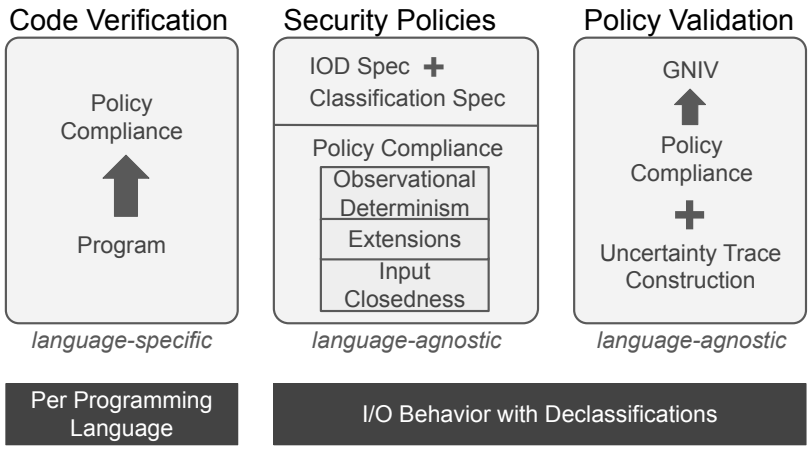

Figure 3.1.: An overview of policy framework. At the center are our security policies and a definition of policy compliance based on a variation of observational determinism. Policy validation (on the right) allows one to prove that a given policy indeed guarantees the intended security properties, in our case an adaptation of Generalized Non-Interference. Both components are based on I/O behavior with declassifications and, thereby, language agnostic. Code verification (on the left) allows one to prove that a given program satisfies a given policy.

3.1. Overview

Fig. 3.1 gives an overview of our policy framework. In this section, we give a high-level overview of its main components; details are discussed in the subsequent sections.

Language Independence. To obtain a language-independent framework for policies and policy validation, we represent programs as their *IOD behavior*, *i.e.* the traces of I/O actions and declassifications that the program may produce. This representation has two advantages: (1) As shown by previous works on protocol verification [55, 56, 88], I/O actions and, thereby, IOD behaviors provide a language-agnostic representation of program behavior. (2) Reason about traces of I/O actions allows us to abstract from implementation details such as memory representations and concurrency.

[55]: Sprenger et al. (2020), *Igloo: soundly linking compositional refinement and separation logic for distributed system verification*
[56]: Arquint et al. (2023), *Sound Verification of Security Protocols: From Design to Interoperable Implementations*
[88]: Oortwijn et al. (2019), *Practical Abstractions for Automated Verification of Message Passing Concurrency*

In contrast to policy specifications (including the definition of *policy compliance*, which defines when a policy is satisfied), verifying that a given implementation satisfies a given policy is inevitably language-specific. For this purpose, we instantiate our framework with two concrete program verification approaches: We extend an existing formalization of SecCSL to prove soundness of our approach and use the existing Gobra verifier for our case studies.

Security Policies. As we have explained in the introduction, our language-agnostic security policies consist of a *classification spec* (the pre- and postconditions for I/O actions) and an IOD spec, specifying the traces of I/O actions and declassification actions that an implementation may produce.

The main challenge of using IOD behaviors is to define policy compliance, such that it satisfies three important requirements: (1) The definition must be expressive enough to capture the behavior of realistic programs. (2) The definition must be strong enough to prove guarantees during policy validation. (3) The definition is amenable to standard program verification techniques and tools, in order to minimize the effort that is necessary to adapt code verification to other languages.

We achieve these goals through a combination of three ingredients, namely observational determinism, extensions, and input-closedness. To

facilitate code verification, we define policy compliance using observational determinism (OD), which many existing code verification techniques support. However, standard OD does not allow non-deterministic behavior, which is pervasive in concurrent and distributed systems. We solve this problem by enriching traces with information about non-deterministic choices. These *extensions* effectively externalize non-deterministic choices, such that standard observational determinism applies. Soundness is preserved by requiring *input-closedness*, a well-definedness condition for extensions.

Code Verification. Our definition of policy compliance allows us to apply standard code verification techniques and tools. Code verification proves that a program produces only the IOD behaviors permitted by the policy, which is achieved by generating appropriate proof obligations for each I/O and declassification action. Moreover, code verification needs to ensure that the implementation satisfies the classification spec, using standard OD-reasoning.

To prove soundness of our approach, we build on the formalization of SecCSL [14], an existing logic for OD-reasoning. For our case studies, we apply Gobra [51], an off-the-shelf automated verifier for Go programs. Supporting other programming languages and verification tools is straightforward and, in particular, does not require any changes to the language-agnostic parts of our framework.

[14]: Ernst et al. (2019), *SecCSL: Security Concurrent Separation Logic*

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

Policy Validation. We propose a methodology for proving—based only on a policy—that data remains confidential even in the presence of declassification. We formalize this property as *Generalized Non-Interference Modulo Views (GNIV)*. GNIV is a more flexible definition of generalized non-interference (GNI) [21] that permits programs to release whether secret inputs exist as long as the values of secret inputs remain confidential. Such a definition is better suited for distributed systems, where secret inputs may happen as a reaction to public actions. For instance, a server may query a database storing secret data as a reaction to a public query.

[21]: Clarkson et al. (2008), *Hyperproperties*

To prove GNIV, we build upon techniques for proving standard GNI [24, 25], where we use the guarantees provided by policy compliance to simplify proofs. More concretely, standard GNI is typically proved by showing how from a trace t, one can iteratively construct an uncertainty trace t' that has the same public behavior as t, but any possible secret data. Instead of the same public behavior, we require only that t' performs the same declassifications as t. Policy compliance ensures that if the declassifications are the same, then the public behavior is the same. In Fig. 3.1, this proof methodology is referred to as *uncertainty trace construction*.

[24]: Mantel (2003), *A uniform framework for the formal specification and verification of information flow security*

[25]: Goguen et al. (1984), *Unwinding and Inference Control*

Outline. We introduce our specification language for policies and define policy compliance in Sec. 3.2. In Sec. 3.3, we show how we can combine established verification techniques to verify code. We discuss the soundness of our code verification approach in Sec. 3.4. Sec. 3.5 discusses how we validate policies. Sec. 3.6 presents our Wireguard case study and illustrates how we express established policy specification patterns based on examples from previous works. The section also lists our trust assumptions. Sec. 3.7 discusses related work, and Sec. 3.8 concludes.

3.2. Security Policies

This section presents the representation of I/O behavior (Sec. 3.2.1), our policy specification language (Sec. 3.2.2), and our threat model (Sec. 3.2.3). Lastly, Sec. 3.2.4 defines when policies are satisfied.

To illustrate our policy framework and its application, we use a small running example throughout this chapter. Consider a protocol to query a person's remaining vaccine protection duration from a server: Every user has a pre-established id and private and public key $k_{\mathsf{sk}}, k_{\mathsf{pk}}$. First, a user sends their id. Next, the application sends a challenge n. The user signs the challenge with their private key. Finally, if the challenge was successful, the application sends the remaining protection duration encrypted with the public key. The server acquires the requested data by querying a database, which also returns the client's public key and a list of compatible vaccines. Below is an informal description of the protocol. For the sake of brevity, we simplify messages by omitting addresses, tags, and additional ids.

$$
\begin{aligned}
\mathsf{Client} &\longrightarrow \mathsf{Server}: && id \\
\mathsf{Server} &\longrightarrow \mathsf{Client}: && n \\
\mathsf{Client} &\longrightarrow \mathsf{Server}: && \mathsf{enc}(n, k_{\mathsf{sk}}) \\
\mathsf{Server} &\longrightarrow \mathsf{Database}: && id \\
\mathsf{Database} &\longrightarrow \mathsf{Server}: && (k_{\mathsf{pk}}, date, vaccines) \\
\mathsf{Server} &\longrightarrow \mathsf{Client}: && \mathsf{enc}(date, k_{\mathsf{pk}})
\end{aligned}
$$

3.2.1. I/O Behavior

A program's I/O behavior captures all communication with the program's environment. The I/O behavior of a program execution is represented as a sequence of *I/O actions*, which are executions of communication primitives, such as sending or receiving a message. We refer to sequences of I/O actions as *traces*.

I/O actions provide a language-independent representation of a program's I/O behavior [88–90]. To reason about a program in a specific language, I/O actions can be linked to the I/O library of that language by providing (trusted) specifications to the library methods expressing which I/O action is performed by a method.

[88]: Oortwijn et al. (2019), *Practical Abstractions for Automated Verification of Message Passing Concurrency*
[89]: Penninckx et al. (2015), *Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs*
[90]: Xia et al. (2020), *Interaction trees: representing recursive and impure programs in Coq*

I/O Behavior of Programs. I/O actions have the form $N(x, \underline{r})$ for an action name N, an output x forwarded to the environment, and an input r obtained from the environment. E.g., the actions $\mathsf{send}(1, \underline{\blacksquare})$ and $\mathsf{recv}(\blacksquare, \underline{0})$ represent sending a 1 and receiving a 0, respectively. We use a designated default value $\blacksquare$ if an action does not have an output or input. We omit $\blacksquare$ arguments when they are clear from the context.

A program's *I/O behavior* is the set of traces of all partial program executions. Considering *partial* executions allows us to represent a non-terminating program execution via the set of finite prefixes of its infinite traces. As a consequence, I/O behaviors are always prefix-closed. For brevity, we sometimes omit partial executions that follow from prefix-closedness, *e.g.* we write $\{\mathsf{recv}(\underline{0}) \cdot \mathsf{send}(1)\}$ instead of $\{\epsilon, \mathsf{recv}(\underline{0}), \mathsf{recv}(\underline{0}) \cdot$

Figure 3.2.: A deployment diagram for the running example. Network communication happens via the receive and send action. The action query communicates with the database.

send(1)}, where · and ϵ denote concatenation and the empty trace, respectively. Concurrent programs are also represented by a single set of traces. For instance, the trace recv(0) · recv(5) · send(6) · send(1) may represent an execution where two threads each increment and forward a received number, namely 0 and 5, respectively.

Example 3.2.1 Fig. 3.2 shows a deployment diagram for our running example. The application communicates with clients via the network and communicates with an external database via remote procedure calls. The action send represents sending a packet over the network, where the action's output is the sent packet. Similarly, the input of the action recv is the received packet. For brevity, a network packet is only the message payload, represented as a bytestring. Network addresses and headers are omitted. The action query represents a remote procedure call to the external database. The action's output is a user id and the action's input is the stored medical record r, consisting of the public key r_{key}, the protection date r_{date}, and the compatible vaccines r_{vacs}. A trace of a single sequential execution of the protocol may look as follows, where we use enc to denote the value obtained by encrypting a given value with a given key:

$$\mathsf{recv}(\underline{\mathtt{201}}) \cdot \mathsf{send}(\mathtt{13}) \cdot \mathsf{recv}(\underline{\mathsf{enc}(\mathtt{13}, k_{\mathsf{sk}})}) \cdot \mathsf{query}(\mathtt{201}, \underline{((k_{\mathsf{pk}}, \mathtt{02.02.22}, \mathtt{v}))}) \cdot \mathsf{send}(\mathsf{enc}(\mathtt{02.02.22}, k_{\mathsf{pk}}))$$

3.2.2. Policy Specifications

Security policies classify the sensitivity of data and define what data may be declassified, and when. A successful approach for language-based policy frameworks is to specify (1) classification by annotating library methods to express the sensitivity of inputs and outputs, and (2) when declassification is permitted as a condition on the global state of a program [78–81]. For instance in our running example, we may permit declassifying enc(*date*, *key*) if (*date*, *key*) is stored in a designated queue used by the implementation to store past queries.

To obtain language-agnostic policies, we lift this approach to I/O behaviors. For classification, we specify pre- and postconditions for I/O actions, which we refer to as *classification spec*. To reason about declassification, we extend I/O behaviors by *declassification actions* of the form decl(x), representing the declassification of value x. We call this extension *IOD behaviors*. To be language-agnostic, we specify when declassification is permitted as a condition on the trace of produced I/O actions and declassification (instead of the language and implementation-specific program state). We capture these conditions formally by specifying the set of traces of I/O actions and declassification actions that an implementation may produce, which we refer to as *IOD spec*. Our security policies are then a tuple of a classification spec and an IOD spec.

$$a ::= \mathsf{true} \mid \mathsf{Low}(e) \mid a \wedge a \mid e \Rightarrow a$$

Figure 3.3.: Assertion language for pre- and postconditions. We use a and e to range over assertions and expressions.

Definition 3.2.1 (Security Policy) *A security policy* $(\mathcal{S}, R)$ *is a tuple of a classification spec* $\mathcal{S}$ *and an IOD spec* R*.*

Expressing security policies on the level of IOD behaviors allows us to abstract from concrete computations and data representations and, thus, to express policies independent of a concrete implementation or programming language. Prior work demonstrated that traces of I/O actions [55, 56] can nevertheless express the behavior of stateful distributed systems. In particular, specifications of traces can refer to results of computations by using mathematical functions. For instance, a specification $\mathsf{recv}(\underline{x}) \cdot \mathsf{send}(\mathsf{hash}(x))$ represents all executions that receive a value x and then send the hash of x, where hash is a mathematical function describing the result of a hashing algorithm, *e.g.* SHA-256, without referring to its concrete code implementation. The formal connection between the mathematical function and the code implementation is established during code verification. This approach works for any stateful computation whose result can be described as a mathematical function of prior inputs. By abstracting from concrete states, our policies cannot express declassification based on arbitrary program state, which is not visible in the IOD behavior. However, none of our case studies required this expressiveness.

[55]: Sprenger et al. (2020), *Igloo: soundly linking compositional refinement and separation logic for distributed system verification*
[56]: Arquint et al. (2023), *Sound Verification of Security Protocols: From Design to Interoperable Implementations*

3.2.2.1. Classification Spec

A classification spec expresses sensitivity requirements and guarantees for data via pre- and postconditions for both I/O and declassification actions. E.g., the precondition of the send action may express that the sent payload must be low. We express this specification using the Hoare triple $\{\mathsf{Low}(x)\}\mathsf{send}(x)\{\mathsf{true}\}$, where $\mathsf{Low}(x)$ specifies that x has low sensitivity. Analogously, we can use a postcondition to express that received payloads are assumed to be low: $\{\mathsf{true}\}\mathsf{recv}(\underline{r})\{\mathsf{Low}(r)\}$. For simplicity, we limit sensitivity levels to $\{\mathsf{Low}, \mathsf{High}\}$, where High and Low specify that data is confidential and not confidential, respectively. An extension to arbitrary sensitivity lattices is straightforward [91].

[91]: Naumann (2006), *From Coupling Relations to Mated Invariants for Checking Information Flow*

The pre- and postconditions of a classification spec are expressed in the assertion language shown in Fig. 3.3. Assertions may combine sensitivity with logical constraints, for instance, to express sensitivity depending on the values of inputs and outputs. E.g., for the action $\mathsf{query}(id, \underline{r})$, the postcondition $\mathsf{Low}(r_{\mathsf{key}}) \wedge (id \in \mathit{Public} \Rightarrow \mathsf{Low}(r))$ specifies that the public keys are low and if the id is in some fixed set *Public*, then the entire record is low.

Every output or input that is not explicitly specified as low is, by default, considered to be potentially high. For instance, the specification triple $\{\mathsf{true}\}\mathsf{getKey}(\underline{r})\{\mathsf{Low}(\mathsf{len}(\mathsf{r}))\}$ expresses that the length of the input r is low, whereas other aspects of r, such as the actual content, are potentially high.

For declassification, the triple $\{\mathsf{Low}(p)\}\mathsf{decl}(p, x)\{\mathsf{Low}(x)\}$ expresses that after declassification, we may assume that x has low sensitivity, capturing information release. The role of the additional parameter p will be explained in Sec. 3.2.2.2.

I/O actions may leak information even if all outputs actually have low sensitivity because the occurrence of the action itself reveals information about the control flow in the program. To reason about such indirect information flow, we also classify each I/O action as low or high; the occurrence of low actions can be observed by the attacker and, thereby, must not depend on high data. For simplicity, we assume in this chapter that all I/O actions are low, *i.e.* whether an action occurs must never depend on high data. Our implementation provides an annotation to specify action sensitivity.

Example 3.2.2 In our running example, we assume that attackers have access to the network and observe sent and received payloads. Conversely, we consider the communication channel with the database to be secure. Moreover, we consider the stored public keys to be low. For the action query, we specify that the queried id and the public key of the record are low, but the protection date and the compatible vaccines are (implicitly) potentially high: $\{\mathsf{Low}(id)\}\mathsf{query}(id, \underline{r})\{\mathsf{Low}(r_{\mathsf{key}})\}$.

3.2.2.2. IOD Spec

IOD specs specify all traces permitted by a policy. Since traces include declassification actions, the permitted traces capture what data may be declassified and when. Our policy framework does not prescribe how IOD specs are expressed. In this chapter, we use *IOD-guarded transition systems*, which extend the transition systems by Sprenger et al. [55] with declassification. In our implementation and evaluation, we express IOD specs also in separation logic [55, 89, 90] to leverage existing verification tools.

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

[55]: Sprenger et al. (2020), *Igloo: soundly linking compositional refinement and separation logic for distributed system verification*
[89]: Penninckx et al. (2015), *Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs*
[90]: Xia et al. (2020), *Interaction trees: representing recursive and impure programs in Coq*

Definition 3.2.2 (IOD-guarded transition system) *An IOD-guarded transition system is a labeled transition system (S, Act, V, G, U), where S is a set of states, Act is a set of action names, V is a set of output and input values, $G : S \times Act \times V \to \{\top, \bot\}$ is a guard, and $U : S \times Act \times V \times V \to S$ is an update function.*

An IOD-guarded system induces the following transition relation:

$$\to = \{(s_0, \mathsf{N}(x, \underline{r}), s_1) \mid G(s_0, N, x) \wedge s_1 = U(s_0, N, x, r)\}$$

We lift the relation to traces, where $(s, \epsilon, s) \in \to_*$ and $(s, t{\cdot}\mathsf{N}(x, \underline{r}), s'') \in \to_*$ whenever $(s, t, s') \in \to_*$ and $(s', \mathsf{N}(x, \underline{r}), s'') \in \to$ for some s'. Given a set of initial states $S_0 \subseteq S$, the traces of an IOD-guarded system are all traces t with $(s_0, t, s') \in \to_*$ for some states $s_0 \in S_0$ and $s' \in S$.

It is essential for soundness that an IOD spec prescribes all declassifications *deterministically*. To understand why, consider a situation where an IOD spec permits the declassification of either x or y. This would allow implementations to choose which of the two to declassify. In particular, an implementation could make this choice depending on a secret and,

thereby, leak it. To prevent such situations, we require that any declassification of sensitive data in an IOD spec is determined by the previous actions on the trace.

This determinism requirement is sound, but too restrictive in practice. Continuing our hypothetical example from the previous paragraph, applications should have the freedom to declassify either x or y, as long as the choice does not depend on a secret. To allow that, we parameterize our declassification action with an additional parameter p, which can be used as a tag, to distinguish different occurrences of declassification. With this addition, we require the declassified value to be determined by the previous actions on the trace *and the value of* p. By requiring p to be low, implementations cannot choose it depending on a secret, thereby avoiding unintentional leaking. The following definition captures this intuition.

Definition 3.2.3 (Well-defined IOD spec) *An IOD spec R is well-defined if for every trace t and all values x_1, x_2, p,*

$$t \cdot \mathsf{decl}(p, x_1) \in R \land t \cdot \mathsf{decl}(p, x_2) \in R \Rightarrow x_1 = x_2.$$

Example 3.2.3 The following IOD-guarded transition system for our running example permits the declassification of the encrypted server response $\mathsf{enc}(\mathit{date}, k_{\mathsf{pk}})$, where *date* and k_{pk} are the protection date and public key returned from the database.

$$\begin{aligned} \mathsf{send}(x) &: \top \rhd s \qquad \mathsf{recv}(\underline{r}) : \top \rhd s \\ \mathsf{query}(\mathit{id}, \underline{d}) &: \top \rhd s[\mathit{id} \mapsto (d_{\mathsf{date}}, d_{\mathsf{key}})] \\ \mathsf{decl}(\mathit{id}, x) &: \mathit{id} \in s \land x = \mathsf{enc}(s[\mathit{id}]_{\mathsf{date}}, s[\mathit{id}]_{\mathsf{key}}) \rhd s \end{aligned}$$

We use the notation $\mathsf{N}(x, \underline{r}) : G(s, N, x) \rhd U(s, N, x, r)$. The state s (of the transition system, not a concrete implementation) is a map from ids to the last-queried key and date. It is changed only in the update of query. The guard for declassification expresses what data may be classified; it uses *id* as a low parameter to satisfy the well-definedness requirement from Def. 3.2.3. The guards of all other actions are true. Note how the mathematical function enc lets us describe the effect of a (possibly stateful) computation performed by the implementation, as explained above.

3.2.3. Threat Model

We consider attackers that know the executed code and can observe the low data of performed I/O. However, they do not have direct access to the machines on which code is run and cannot inspect memory. We do not consider side channels such as timing information. An extension to timing channels is interesting future work and can be tackled by moving from formal guarantees based on possibilistic secure information flow [92] to a probabilistic model [93].

The observational capabilities of attackers are defined by a security policy's classification spec. A program that complies with a given policy

[92]: Smith et al. (1998), *Secure Information Flow in a Multi-Threaded Imperative Language*

[93]: Sabelfeld et al. (2000), *Probabilistic Noninterference for Multi-Threaded Programs*

is secure against attackers that can observe (at most) the data and actions classified as low.

3.2.4. Policy Compliance

Whether a program satisfies a security policy is determined entirely over the program's IOD behavior. To satisfy the policy's IOD spec R, the IOD behavior of the program (that is, its set of traces) has to be a subset of R. To satisfy its classification spec $\mathcal{S}$, the program's IOD behavior has to satisfy secure information flow, where requirements and assumptions about low sensitivity are specified by $\mathcal{S}$. We focus on the latter property in this subsection, in particular, on defining secure information flow as a variation of observational determinism that permits non-deterministic I/O behaviors.

Note that declassification is handled by the combination of both requirements. The IOD spec expresses where a declassification may occur in a trace, and the classification spec of declassification actions ensures that the occurrence of a declassification action does not depend on a secret (see Sec. 3.2.2.2) and expresses that declassified data has low sensitivity.

3.2.4.1. Secure Information Flow

[82]: McLean (1992), *Proving Noninterference and Functional Correctness Using Traces*
[83]: Roscoe (1995), *CSP and determinism in security modelling*
[85]: Eilers et al. (2018), *Modular Product Programs*
[94]: Eilers et al. (2021), *Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security*

We define secure information flow (SIF) based on observational determinism (OD) [82, 83], a widely-used criterion that is supported by standard code verification techniques [85, 94] and, thus, enables us to use off-the-shelf program verifiers to prove that a program satisfies a policy (see Sec. 3.3).

However, OD has a critical limitation. A program satisfies OD only if its low outputs are deterministic in the low inputs. This definition ensures that low outputs do not depend on confidential data. However, real-world programs usually have non-deterministic IOD behavior, for instance, due to concurrency and memory allocation, such that standard OD is not applicable.

[14]: Ernst et al. (2019), *SecCSL: Security Concurrent Separation Logic*
[95]: O'Neill et al. (2006), *Information-Flow Security for Interactive Programs*
[96]: Ryan et al. (2011), *Applied pi calculus*
[97]: Goubault-Larrecq et al. (2007), *A Probabilistic Applied Pi-Calculus*

Existing solutions [14, 95] for language-specific frameworks solve this problem by externalizing the non-deterministic choices. That is, they make program executions artificially deterministic by parameterizing the language semantics with an *oracle* that captures the non-deterministic choices of an execution, for instance, how threads have been scheduled or how memory has been allocated. OD is then satisfied if the low outputs are deterministic in the low inputs *and* the oracle. The same technique is also used to reason about applied π-calculus and its extensions, for instance, to prove observational equivalence [96, 97].

We adapt the idea of externalizing non-deterministic choices to IOD behavior. Instead of parameterizing executions with oracles, we extend traces with the information about non-deterministic choices, which we refer to as *extensions*. While language-specific approaches know where non-deterministic choices happen, our language-agnostic framework does not have this information. Therefore, we allow extensions at any point in the trace, but introduce a soundness condition called *input-closedness* to ensure that traces are not extended incorrectly.

An IOD behavior then satisfies SIF if there exists an input-closed extension that satisfies OD. In the following, we explain extensions and input-closedness. We provide formal definitions in Sec. 3.2.4.2.

Extensions. An IOD behavior T' is an extension of T if T' is obtained by adding *auxiliary actions* into T's traces. The added auxiliary actions are not actually produced by the program, but instead are used to justify that a program satisfies SIF.

Consider the program `(send(1) || send(2))`, which sends 1 and 2 in parallel. The IOD behavior of the program is $\{\mathsf{send}(1) \cdot \mathsf{send}(2), \mathsf{send}(2) \cdot \mathsf{send}(1)\}$. The program does not satisfy OD because the sent message is not deterministic. However, the program is secure since the sent message is independent of any confidential data. Extensions allow us to make the secret-independence of non-deterministic behaviors explicit. In our example, a suitable extension introduces an auxiliary action Sched that captures whether the scheduler executes the left or right parallel branch first: $\{\mathsf{Sched}(\underline{\mathsf{L}}) \cdot \mathsf{send}(1) \cdot \mathsf{send}(2), \mathsf{Sched}(\underline{\mathsf{R}}) \cdot \mathsf{send}(2) \cdot \mathsf{send}(1)\}$. Auxiliary actions have their own classification spec. With the spec $\{\mathsf{true}\}\mathsf{Sched}(\underline{x})\{\mathsf{Low}(x)\}$, our extension satisfies OD as the sent messages are now deterministic in the scheduler choice.

To ensure that declassifications happen only if permitted by the policy, auxiliary actions must not implicitly declassify data, that is, their classification specs are not allowed to describe declassifications, *e.g.* by specifying that an output is low in the postcondition. For simplicity, we enforce that auxiliary actions always have the spec $\{\mathsf{Low}(x)\}\mathsf{N}(x, \underline{r})\{\mathsf{Low}(r)\}$, *i.e.* auxiliary actions do not take high data and, therefore, never declassify anything (for auxiliary actions that do not have an output, the precondition $\mathsf{Low}(\blacksquare)$ is equivalent to true). This spec is sufficient to verify our case studies in Gobra. Our Isabelle/HOL formalization defines a weaker criterion for the classification spec of auxiliary actions that expresses that specs do not describe declassifications.

Our formal definition of SIF below allows one to choose an extension for each program and policy. To reduce the necessary specification overhead, our implementation fixes the extension and the classification spec for auxiliary actions.

Example 3.2.4 We have formalized our approach based on SecCSL [14], a logic for a simple concurrent programming language with references. For this programming language, we introduce two auxiliary actions. For concurrency, the action $\mathsf{Sched}(\gamma, \underline{\tau})$ has as input the information which thread is scheduled next. The output is necessary for input-closedness and explained in Example 3.2.5. Because extensions contain which thread is scheduled next, the IOD behavior becomes deterministic. For heap state, the action $\mathsf{Init}(\underline{s})$ is added as the first action of a trace and has as input the initial heap memory. For both actions, input and output are classified as low. More details are discussed in Sec. 3.4.

[14]: Ernst et al. (2019), *SecCSL: Security Concurrent Separation Logic*

Input-closedness. Without restrictions, extensions can trivially invalidate SIF by laundering confidential inputs. Consider the insecure IOD behavior $\{\mathsf{getKey}(\underline{x}) \cdot \mathsf{send}(x) \mid x \in \mathbb{N}\}$, which first gets a confidential key x and then sends it on a public channel. An invalid extension can mask the origin of x by adding an auxiliary action $\mathsf{In}(\underline{x})$ that has the key as low

input, *e.g.* as $\{\mathsf{getKey}(\underline{x}) \cdot \mathsf{In}(\underline{x}) \cdot \mathsf{send}(x) \mid x \in \mathbb{N}\}$. This IOD behavior satisfies OD. However, since the auxiliary action is not present in an actual program execution, it remains insecure.

To prevent auxiliary actions from masking the origin of inputs, we require extensions to be *input-closed*: an extension must contain traces for all possible inputs of actions. The above extension violates this condition because it contains only traces where the inputs from getKey and In are *the same*. If we add traces where both actions receive *different* inputs, the insecurity becomes apparent.

Example 3.2.5 We have formally proved that our extensions chosen for SecCSL are input-closed (see Sec. 3.4). The action $\mathsf{Init}(\underline{s})$ trivially maintains input-closedness because the IOD behavior contains traces for all possible initial heap memories. For $\mathsf{Sched}(\gamma, \underline{\tau})$, we have to show that every possible input of Sched may actually be scheduled. In SecCSL's language, all unblocked threads may be scheduled; these are captured by the current program configuration γ. Defining the inputs of Sched accordingly requires us to add γ as an output (see Sec. 3.2.4.2).
By classifying Sched's output and input as low, we recover a standard OD reasoning principle for concurrent programs: If we show that control flow is low, and thus the current configuration is low then we may assume that decisions made by the scheduler are low.

3.2.4.2. Formal Definition of Secure Information Flow

We will now define the concepts introduced in the previous subsubsection.

Extensions. We introduce a projection $\mathsf{actual}(\cdot)$, removing auxiliary actions from traces, *e.g.* $\mathsf{actual}(\mathsf{In}(\underline{x}) \cdot \mathsf{recv}(\underline{r})) = \mathsf{recv}(\underline{r})$, where $\mathsf{In}(\underline{x})$ is an auxiliary action. We lift the projection also to sets of traces. Therefore, an IOD behavior T' is an extension of T if $\mathsf{actual}(T') = T$ holds.

Input-closedness. Def. 3.2.4 defines formally when a set of traces is input-closed. As discussed, if an action has some input r, then for every possible input r' of that action, there must be a trace where the action has input r'. An input is possible if it occurs in some trace (expressed with $t' \cdot N(x, \underline{r'})$ in Def. 3.2.4).

Definition 3.2.4 (Input-Closed IOD Behaviors) *A set of traces T is input-closed if*

$$t \cdot N(x, \underline{r}) \in T \land t' \cdot N(x, \underline{r'}) \in T \Rightarrow t \cdot N(x, \underline{r'}) \in T$$

Low Data Projection. To define OD, we have to express which data is required and assumed to be low. In our framework, this is the data classified as low by pre- and postconditions, respectively. For this purpose, we introduce two projections $\cdot\downarrow_S^{\mathsf{pre}}$ and $\cdot\downarrow_S^{\mathsf{post}}$ which can be applied to actions and remove any data that is *not* low according to the action's classification spec (for the pre- and postcondition, respectively). For the

classification spec of our running example, we have for instance:

$$\mathsf{send}(x)\downarrow_{\mathcal{S}}^{\mathsf{pre}} = \mathsf{send}(x) \quad \mathsf{recv}(\underline{r})\downarrow_{\mathcal{S}}^{\mathsf{post}} = \mathsf{recv}(\underline{r})$$
$$\mathsf{query}(id, \underline{d})\downarrow_{\mathcal{S}}^{\mathsf{post}} = \mathsf{query}(\underline{d_{\mathsf{key}}}) \quad \mathsf{decl}(x)\downarrow_{\mathcal{S}}^{\mathsf{post}} = \mathsf{decl}(x)$$

Recall that $\mathsf{N}(x)$ and $\mathsf{N}(\underline{r})$ are shorthands for $\mathsf{N}(x, \blacksquare)$ and $\mathsf{N}(\blacksquare, \underline{r})$, respectively. We lift the projections from single actions to traces.

Observational Determinism. We build on the definition of OD introduced by Clarkson and Schneider [21]. Their definition does not consider progress channels, *i.e.* information may be released by not producing observable actions, *e.g.* due to an infinite loop. To prevent such information leakage, we require in addition progress sensitivity [98], enforcing the absence of progress channels. In the definitions below, we refer to Clarkson and Schneider's definition as *progress-insensitive OD*.

[21]: Clarkson et al. (2008), *Hyperproperties*

[98]: Askarov et al. (2008), *Termination-Insensitive Noninterference Leaks More Than Just a Bit*

An IOD behavior satisfies progress-insensitive OD if for every action, the data expected to be low is deterministic in the data assumed to be low for previous actions. Determinism is expressed formally by considering pairs of extended traces.

Definition 3.2.5 (Progress-Insensitive $\mathcal{S}$-OD) *A set of traces T satisfies progress-insensitive $\mathcal{S}$-OD if, for all of traces t_1, t_2 and actions e_1, e_2 with $t_1 \cdot e_1 \in T$ and $t_2 \cdot e_2 \in T$,*

$$t_1\downarrow_{\mathcal{S}}^{\mathsf{post}} = t_2\downarrow_{\mathcal{S}}^{\mathsf{post}} \Rightarrow e_1\downarrow_{\mathcal{S}}^{\mathsf{pre}} = e_2\downarrow_{\mathcal{S}}^{\mathsf{pre}} .$$

The definition of progress sensitivity is analogous. An IOD behavior satisfies progress sensitivity if whether or not a trace makes progress is deterministic in the data previously assumed to be low. Formally, we specify that if one trace has more progress than another trace, *i.e.* is longer, then the shorter trace can be extended.

Definition 3.2.6 (Progress Sensitivity) *A set of traces T satisfies progress sensitivity for a classification spec $\mathcal{S}$ if, for all of traces t_1, t_2 and every action e_1 with $t_1 \cdot e_1 \in T$ and $t_2 \in T$,*

$$t_1\downarrow_{\mathcal{S}}^{\mathsf{post}} = t_2\downarrow_{\mathcal{S}}^{\mathsf{post}} \Rightarrow \exists e_2.\ t_2 \cdot e_2 \in T.$$

Definition 3.2.7 ($\mathcal{S}$-OD) *A set of traces T satisfies $\mathcal{S}$-OD if T satisfies progress-insensitive $\mathcal{S}$-OD and progress-sensitivity for $\mathcal{S}$.*

Secure Information Flow. Given all the defined ingredients, we can define SIF as discussed in Sec. 3.2.4.1. We use $(\mathcal{S} + \mathsf{NoDecl})$ to denote the classification spec, which uses $\mathcal{S}$ for actual actions and specifies the triple $\{\mathsf{Low}(x)\}\mathsf{N}(x, \underline{r})\{\mathsf{Low}(r)\}$ for all auxiliary actions.

Definition 3.2.8 ($\mathcal{S}$-SIF) *A set of traces T satisfies $\mathcal{S}$-SIF if there exists an extension T' such that (1) $\mathsf{actual}(T') = T$, (2) T' is prefix-closed, (3) T' is input-closed, (4) T' satisfies $(\mathcal{S} + \mathsf{NoDecl})$-OD.*

As motivated throughout this chapter, our definition of secure information flow does not require that IOD behavior is deterministic. In fact, we

are able prove that *every* IOD behavior satisfies secure information flow if it does not involve confidential data, *i.e.* if the arguments of all actions are classified as low by the classification spec.

Lemma 3.2.1 *Every IOD behavior T satisfies* NoDecl-*SIF*

3.2.4.3. Composition of Policy Compliance

In our running example, we consider the case that a single implementation satisfies a given policy. However, our use of IOD traces enables us to also reason about whether a set of implementations that run together in a distributed system satisfies a policy. For instance, consider that we extend our running example with the feature to also query the list of compatible vaccines (in addition to the remaining vaccine protection). To improve availability or performance, real-world implementations may want to implement the application with a set of servers, some handling only vaccine protection requests, some handling only vaccine compatibility requests, and even some handling both. Our security policies are expressive enough to handle that a single policy governing the vaccine protections *and* compatible vaccines is satisfied by such a set of servers together.

The key ingredient is that policy compliance is compositional: If two IOD behaviors T_1 and T_2 satisfy two policies $(\mathcal{S}_1, R_1)$ and $(\mathcal{S}_2, R_2)$, respectively, then the *parallel composition* of T_1 and T_2, *i.e.* the IOD behavior of both programs running together in a distributed system, satisfies a composition of both policies. Compositionality makes the following workflow possible: If a policy is satisfied by multiple programs $T_1, \ldots, T_n$, then we first decompose a policy $(\mathcal{S}, R)$ into one policy $(\mathcal{S}_i, R_i)$ per program T_i such that the parallel composition of the policies $(\mathcal{S}_i, R_i)$ is the original policy $(\mathcal{S}, R)$. Afterward, using our standard code verification technique, we verify separately that each program T_i satisfies its respective policy $(\mathcal{S}_i, R_i)$. Compositionality entails that if all programs satisfy their respective policy, then the parallel composition of all programs satisfies the original policy $(\mathcal{S}, R)$. This approach is inspired by the work by Sprenger et al. [55], where a global I/O spec is decomposed into an I/O spec per implementation, which is then verified.

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

We first define the parallel composition of IOD behaviors and then formally capture the compositionality of policy compliance.

Parallel Composition of IOD Behaviors. We define the parallel composition of IOD behaviors as the set of all interleavings of traces of both IOD behaviors. Def. 3.2.9 shows our formal definition of the set of interleavings for a pair of traces. The interleavings are all combinations of actions that maintain the relative order of actions. For instance, the interleavings of $\mathsf{recv}(\underline{a_1}) \cdot \mathsf{send}(a_2)$ and $\mathsf{send}(b_1)$ are

$$\mathsf{send}_\mathsf{R}(b_1) \cdot \mathsf{recv}_\mathsf{L}(\underline{a_1}) \cdot \mathsf{send}_\mathsf{L}(a_2)$$
$$\mathsf{recv}_\mathsf{L}(\underline{a_1}) \cdot \mathsf{send}_\mathsf{R}(b_1) \cdot \mathsf{send}_\mathsf{L}(a_2)$$
$$\mathsf{recv}_\mathsf{L}(\underline{a_1}) \cdot \mathsf{send}_\mathsf{L}(a_2) \cdot \mathsf{send}_\mathsf{R}(b_1)$$

We mark actions ($\mathsf{N_L}$ and $\mathsf{N_R}$) to identify from which program an action originates. Recall that in our model, communication between programs happens via the environment, which may interfere with all actions. As a consequence, our composition of IOD behaviors does not synchronize actions, *e.g.* the receive action of one program does not have to happen after the send action of another program since the environment may send messages by itself.

Definition 3.2.9 (Composition of Traces) *For two traces t_0 and t_1, we define their set of interleavings, denoted as $(t_0 \parallel t_1)$, as the smallest set such that,*

- $\epsilon \in (\epsilon \parallel \epsilon)$
- $t' \in (t_0 \parallel t_1) \Rightarrow \mathsf{N_L}(x, \underline{r}) \cdot t' \in (\mathsf{N}(x, \underline{r}) \cdot t_0 \parallel t_1)$
- $t' \in (t_0 \parallel t_1) \Rightarrow \mathsf{N_R}(x, \underline{r}) \cdot t' \in (t_0 \parallel \mathsf{N}(x, \underline{r}) \cdot t_1)$

The parallel composition of two IOD behaviors T_0 and T_1 is then the union of all interleavings, *i.e.* $(T_0 \parallel T_1) = \bigcup\{(t_0 \parallel t_1) \mid t_0 \in T_0 \wedge t_1 \in T_1\}$.

Composition of Policies. Lemma 3.2.2 formalizes the compositionality of policy compliance discussed above. We use $\mathcal{S}_1 + \mathcal{S}_2$ to denote the classification spec that classifies the actions of the two IOD behaviors with $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively.

To decompose a policy $(\mathcal{S}, R)$ into two policies $(\mathcal{S}_1, R_1)$ and $(\mathcal{S}_2, R_2)$ (for two programs), we have prove that $\mathcal{S} = \mathcal{S}_1 + \mathcal{S}_2$ and $R = (R_1 \parallel R_2)$ holds. The first condition $\mathcal{S} = \mathcal{S}_1 + \mathcal{S}_2$ holds trivially as long as the decomposed classification specs $\mathcal{S}_1$ and $\mathcal{S}_2$ specify the same classifications as the original classification $\mathcal{S}$. Proving the second condition $R = (R_1 \parallel R_2)$ may be more involved depending on the representation of the IOD spec. In this work, we do not provide techniques to decompose IOD specs. For instance, the work by Sprenger et al. [55, 56] proposes techniques to perform such decompositions.

As a minor technical detail, satisfying a policy with a set of programs requires that actions have an additional parameter to account for the markings $\mathsf{N_L}$ and $\mathsf{N_R}$. Such a parameter does not rule out that a policy is satisfied by a single component.

[55]: Sprenger et al. (2020), *Igloo: soundly linking compositional refinement and separation logic for distributed system verification*
[56]: Arquint et al. (2023), *Sound Verification of Security Protocols: From Design to Interoperable Implementations*

Lemma 3.2.2 (Compositionality of Policy Compliance) *For all IOD behaviors T_1 and T_2 satisfying the policies $(\mathcal{S}_1, R_1)$ and $(\mathcal{S}_2, R_2)$, respectively, the parallel composition of the IOD behaviors $(T_1 \parallel T_2)$ satisfies the policy $(\mathcal{S}_1 + \mathcal{S}_2, (R_1 \parallel R_2))$.*

3.3. Code Verification

To enable code verification in a given language, we first equip the language and its libraries to record the performed IOD behavior, introduce auxiliary actions to enable us to verify non-deterministic programs, and prove that the resulting extended IOD behaviors are input-closed. This lets us verify that a program in that language satisfies a security policy by proving the two requirements of policy compliance, namely that the IOD behavior satisfies the IOD spec and that the extended IOD behavior satisfies OD.

We illustrate code verification using Gobra (see Chapter 2), which we adapted to support OD reasoning. We discuss how we record IOD behavior in Sec. 3.3.1, and explain how we verify policy compliance for a given program in Sec. 3.3.2.

Changes to Gobra. Gobra previously did not support OD reasoning. We implemented support for standard OD reasoning using an existing product construction [85], which simulates two executions of the input program by a single execution of the constructed product program, which can then be verified by off-the-shelf verifiers such as Gobra. Since Gobra is built on top of Viper, we did not have to implement a product construction for Go, but instead used Viper's product program construction as proposed by Eilers et al. [94].

[85]: Eilers et al. (2018), *Modular Product Programs*

[94]: Eilers et al. (2021), *Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security*

Our extended version of Gobra supports the *relational low assertion* `low(·)` [23, 85]; `low(e)` expresses that the value of the expression `e` is low according to OD (as formalized in Sec. 3.2.4.2). As such, `low(e)` holds if `e`'s value is deterministic in the values known to be low.

[23]: Barthe et al. (2011), *Relational Verification Using Product Programs*
[85]: Eilers et al. (2018), *Modular Product Programs*

We enforce the common restriction that programs must not branch on secrets [78, 93, 99, 100]. This restriction may be lifted for sequential code, but is necessary for concurrent code where timing differences between branches may affect a program's low I/O behavior. This restriction includes implicit branches on secrets. For instance, in Go, dynamic calls branch implicitly on the receiver's dynamic type. Gobra verifies for all branch conditions `e`, *e.g.* the conditions of if-statements and loops as well as the dynamic types of dynamic calls, that `low(e)` holds.

[78]: Schoepe et al. (2020), *VERONICA: Expressive and Precise Concurrent Information Flow Security*
[93]: Sabelfeld et al. (2000), *Probabilistic Noninterference for Multi-Threaded Programs*
[99]: Murray et al. (2018), *COVERN: A Logic for Compositional Verification of Information Flow Control*
[100]: Smith (2007), *Principles of Secure Information Flow Analysis*

3.3.1. Recording IOD Behavior

For every programming language, we need to define how we abstract executions of programs in that language to IOD traces. As discussed in Sec. 3.2.1, we assume that I/O actions are performed by a set of trusted library methods. For declassification, we add a *ghost* method to the I/O library that is called to declassify data. In order to reason about the IOD behavior of a program, we use an existing specification technique [79, 80]: we record the produced IOD trace of a program execution explicitly *in ghost state*. To this end, we add a global trace data structure to the program, which is accessed via a ghost pointer `Trace`. This trace is initially empty and gets extended by library methods that produce an I/O or declassification action. To describe this effect, we equip each such method with a specification that expresses which action it appends to the recorded trace. Using this technique, the abstraction of a program execution to an IOD trace is explicitly available for verification, but does not incur any run-time overhead since the trace structure is ghost code and will be erased during compilation.

[79]: Murray et al. (2023), *Assume but Verify: Deductive Verification of Leaked Information in Concurrent Applications*
[80]: Banerjee et al. (2008), *Expressive Declassification Policies and Modular Static Enforcement*

Fig. 3.4 shows the specification of some library methods, including declassification. Recall that we simplify IOD actions for the sake of brevity, so arguments such as target addresses are omitted. The preconditions, preceded by `req`, specify that permissions to the ghost pointer `Trace` and the parameter array `msg` is transferred to the method. The postconditions, preceded by `ens`, specify that the permission to the global trace is returned to the caller, such that the caller can use it for subsequent calls. Moreover, the postconditions express which action has been appended to the trace.

```
1  req acc(Trace) && acc(msg)
2  ens acc(Trace) && *Trace = old(*Trace) · send(Abs(msg))
3  func Send(msg []byte)
4
5  req acc(Trace)
6  ens acc(msg)
7  ens acc(Trace) && *Trace = old(*Trace) · recv(Abs(msg))
8  func Recv() (msg []byte)
9
10 req acc(Trace)
11 ens acc(Trace) && *Trace = old(*Trace) · decl(p,value)
12 ghost func Declassify(p, value any)
```

Figure 3.4.: Specifications of library methods producing IOD actions. Specifications about `Trace` describe an atomic effect. Data is declassified via the `Declassify` ghost method. `Abs` returns the bytes stored in an array.

The expression `old(*Trace)` denotes the value of the trace before the call. The function `Abs` returns the sequence of bytes stored in an array. For instance, `*Trace = old(*Trace) · send(Abs(msg))` expresses that at some point during the call, the trace is appended with the action `send(Abs(msg))`.

As discussed in Sec. 3.2.4, (extended) traces include, besides I/O and declassification actions, auxiliary actions that describe non-deterministic choices. These choices are typically taken implicitly by the language semantics, without invoking an explicit operation. For instance, the action Sched occurs whenever the run-time system schedules a different thread. For each programming language, we define which auxiliary actions are produced and prove, based on the language semantics, that the resulting extended IOD behavior is input-closed. In contrast to I/O and declassification actions, we do *not* record auxiliary actions on the global trace. Instead, we prove that any requirement imposed on auxiliary actions is indeed enforced by the used verification logic wherever such an auxiliary action may occur. For instance, $\mathsf{Sched}(\gamma, \underline{\tau})$ requires that the current program configuration is low, which is enforced in Gobra and SecCSL by requiring that all branch conditions are low.

In summary, our ghost trace records the I/O and declassification actions performed by a program execution, but not the auxiliary actions, which are handled differently. Since program verification proves properties of a program for all possible executions and, in particular, for all possible values of our ghost trace, verification captures the program's entire IOD behavior (except auxiliary actions). Note that the entire machinery to maintain the ghost trace and to handle auxiliary actions needs to be set up once for a programming language and can then be re-used for the verification of each program written in that language.

3.3.2. Verifying Policy Compliance

To verify that a program satisfies a policy, we have to prove that (1) the program's IOD behavior satisfies the IOD spec and (2) the program's *extended* IOD behavior satisfies $\mathcal{S}$-OD for the classification spec (together with input-closedness proved for the programming language, $\mathcal{S}$-OD implies $\mathcal{S}$-SIF). In this subsection, we explain how we specify security policies in Gobra, discuss how we verify these two properties, and illustrate verification on our running example.

Specifying Security Policies in Gobra. We express security policies in Gobra as implementations of an interface `Policy`. This interface is

```
1  interface Policy {
2    pure Classification(Action) Spec
3    pure Guard(State,Action) bool
4    pure Update(State,Action) State
5  }
6
7  pure func (Vac) Classification(action Action) Spec {
8    return (match action {
9    case query{?id,?f}: Spec{Low(id),    Low(f.key)}
10   case decl{?id,?x}:  Spec{Low(id),   Low(x)}
11   case ?a:            Spec{Low(a.out),Low(a.in)}  })}
12
13 pure func (Vac) Guard(st State, action Action) bool {
14   return (match action {
15   case decl{?id,?x}:
16       id ∈ st && x = enc(st[id].date,st[id].key)
17   case _: true })}
18
19 pure func (Vac) Update(st State, action Action) State{
20   return (match action {
21   case query{?id,?f}: st[id := {f.date,f.key}]
22   case _: st })}
```

Figure 3.5.: The `Policy` interface (top segment) prescribing the functions that need to be implemented to define a concrete security policy in Gobra. The two bottom segments specify the security policy for our running example.

defined in a re-usable library and prescribes three functions that need to be defined for each concrete security policy (see top of Fig. 3.5). This library also contains a representation of actions, the states of the IOD transition system, assertions, and specifications as terms of an algebraic datatype (ADT), together with functions that yield these terms. For instance, `True()` and `Low(e)` are function calls that yield terms for the assertions true and Low(e), respectively. Our library defines a function for each assertion of the assertion language defined in Fig. 3.3. Similarly, the call `Spec{P,Q}` yields a tuple term consisting of the precondition `P` and the postcondition `Q`.

Lines 7–22 in Fig. 3.5 show the implementation `Vac` of the `Policy` interface for our running example. The classification spec of a security policy is captured by the function `Classification`. Lines 7–11 express the classification spec for our running example as discussed in Example 3.2.2. The `Classification` function takes an action and returns its spec, consisting of a precondition and a postcondition. The function uses pattern matching to distinguish the different actions. The prefix `?` binds matched arguments. For instance, the pattern `query{?id,?f}` matches the action query and binds its output and input to the variables `id` and `f`, respectively. For instance, the case for `query{?id,?f}` expresses that the action requires `id` to be low and ensures that the key of the resulting record `f` is low. The default case at Line 11 handles receive and send actions. The ADT destructors `.out` and `.in` return an action's output and input, respectively.

The IOD-guarded transition system defining the IOD spec of a security policy is expressed via the functions `Guard` and `Update` (Lines 13–22). The guard function takes the state of the transition system and an action, and yields whether the action is enabled in that state. The update function also takes a state and an action and updates the state. Both function definitions correspond directly to the IOD transition system presented in Example 3.2.3.

Verifying the IOD Spec. As we discussed in Sec. 3.3.1, our global trace records all actions performed by a program. Therefore, we can prove that

a program's IOD behavior satisfies the IOD spec by showing that the recorded trace is one of the traces induced by the IOD-guarded transition system (see Sec. 3.2.2.2). In other words, we need to prove that there exists a state in the transition system that is reachable by performing the actions in the recorded trace. This property holds trivially at the program start, when the recorded trace is empty. We impose a proof obligation that this property is preserved whenever the trace is extended (that is, when an I/O or declassification action is performed).

To encode this approach, our library defines a function `Reaches(`p`,`t`,`s`)` to express that t is a trace of the IOD spec defined by the policy p and reaches state s from the initial state. Conceptually, we impose a proof obligation $\exists\, st :: $ `Reaches(p, *Trace,` st`)` for each operation that performs an action to check that the trace extended by the performed action is still permitted by the IOD-guarded transition system (here, `p` is the instance of the security policy). In practice, we avoid the existential quantifier by storing the transition system state explicitly in a ghost variable and updating it using the `Update` function of the policy whenever an action is performed. This allows us to instantiate the existential quantifier directly and, thereby, avoid a well-known weakness of SMT solvers.

Imposing proof obligations whenever an action is performed (instead of checking that the trace is permitted at the end of the program) leads to simpler proof obligations and works for non-terminating programs. However, this approach cannot verify programs that branch on a secret, but perform equivalent actions in both branches, *e.g.* `if h {Send(1)} else {Send(1)}`, where `h` is confidential. This limitation is not relevant for our code verification in Gobra and SecCSL, where we disallow branching on secrets anyway.

Verifying Observational Determinism. To prove that a program satisfies OD, we have to prove progress-insensitive $\mathcal{S}$-OD (Def. 3.2.5) and progress sensitivity (Def. 3.2.6). The latter is trivial in our setting: since we do not allow branching on secrets, the termination of loops and calls (that is, progress) cannot depend on a secret.

To verify progress-insensitive $\mathcal{S}$-OD, we have to prove that the arguments of each action on the recorded trace that are classified as low by the action's preconditions are deterministic in the arguments of the previous actions on the trace that are classified as low by their postconditions. As for verifying the IOD spec, this property holds trivially for the empty trace and we check that this property is preserved whenever the trace is extended. Before performing an action, we assume `low(*Trace`$\downarrow_{\mathcal{S}}^{\text{post}}$`)` and then check after the action that `low(*Trace`$\downarrow_{\mathcal{S}}^{\text{pre}}$`)` holds. Our reusable library defines functions `Pre(`p`,`t`)` and `Post(`p`,`t`)` to express the low data projections $t\downarrow_{\mathcal{S}}^{\text{pre}}$ and $t\downarrow_{\mathcal{S}}^{\text{post}}$, respectively.

Concurrency Reasoning. As we have seen so far, our proof obligations for code verification are expressed in terms of the recorded trace, which is stored in a mutable ghost data structure. Standard separation logic ensures that mutable state is exclusively owned: only one method can hold the permission to the data structure at any point in the execution. This is problematic for concurrent implementations, where multiple threads may perform IOD actions and, thus, need mutable access to the trace.

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

In Gobra, we solve this problem using *shared invariants* [36], which express assertions that always hold. Data structures that are governed by a shared invariant may be updated concurrently under two conditions: (1) the update preserves the shared invariant and second, (2) the update is performed atomically (such that other threads cannot observe intermediate states in which the shared invariant does not hold).

For our ghost trace, we use the shared invariant shown at the top of Fig. 3.6. It provides the permission to access the ghost trace, and expresses that the current trace is compatible with the IOD transition system and its pre-projection is low. Performing an IOD action satisfies the two conditions above: (1) adding a new action to the trace preserves the shared invariant; (2) since these updates affect only ghost state, they can be treated as atomic.

Running Example. Fig. 3.6 shows our running example in Gobra. Lines 5–18 show snippets of an implementation together with some of the required proof annotations. The shown snippet can be run in parallel by multiple threads.

At Line 5, the trace is extended with a `recv` action. The permission to `Trace`, necessary for the call, is obtained from the shared invariant. The invariant holds trivially after Line 5 as `recv`'s precondition and guard are `true` and its update keeps the state unchanged. At Line 6, before the next action and after having verified `low(Pre(Vac{}))`, we assume `low(Post(Vac{}), *Trace)`, establishing that the received message is low. The other calls are verified similarly.

Line 7 checks whether the request is already processed. The IOD spec specifies that we may declassify only the most recently queried data for `id` and, thus, use it for a reply. The code handles this requirement by letting at most one thread process requests for a specific id. It stops if the request is already being handled (Line 8). Note that requests for different ids can be processed in parallel.

The precondition of the `query` action (Line 10) requires that `Abs(id)` is low, which we get from Line 6. Furthermore, the transition system's state gets updated with the queried date and key (Line 12). As discussed, we use the transition system's `Update` function to keep track of the new state.

Line 14 encrypts the date with the key. The specification of the method `Encrypt` shown at Lines 20–23 relates calls of the `Encrypt` method to the mathematical function `enc`, which illustrates how we connect concrete computations to the abstract state tracked in an IOD spec. After the call, we know from `Encrypt`'s postcondition that `Abs(reply)` is equal to `enc(Abs(info.date),Abs(info.key))`. The guard of the declassification (Line 15) holds since we declassify the most recent date encrypted with the most recent key. As a technicality, ensuring the absence of other queries since Line 10 requires some concurrency reasoning, which we omitted to focus on the essentials. The call to the `Send` method at Line 17 completes the request. We can show that the sent payload is low due to the assumption gained from the declassification at Line 16.

```
1  pred SharedInvariant() {
2    acc(Trace) && low(Pre(Vac{},*Trace)) &&
3    ∃ st :: Reaches(Vac{}, *Trace, st)
4  }
5  id := Recv()
6  // assume low(Post(Vac{}), *Trace)
7  already_processed := queue.add(id)
8  if already_processed { /* stop */ }
9  ...
10 info := Query(id)
11 // assume low(Post(Vac{}), *Trace)
12 // update transition system state
13 ...
14 reply := Encrypt(info.date, info.key)
15 ghost Declassify(Abs(id),Abs(reply))
16 // assume low(Post(Vac{}), *Trace)
17 Send(reply)
18 queue.remove(id)
19
20 req acc(data, read) && acc(key, read)
21 ens acc(data, read) && acc(key, read) && acc(ciph)
22 ens Abs(ciph) = enc(Abs(data), Abs(key))
23 func Encrypt(data, key []byte) (ciph []byte)
```

Figure 3.6.: Verification of the running example in Gobra (Lines 5–18) together with the used shared invariant (Lines 1–3). Lines 20–23 shows the specification of the encryption method, relating calls to the mathematical function enc. Proof annotations are indicated with the keyword `ghost`.

3.4. Soundness of Code Verification

In Sec. 3.3, we discuss how to verify programs using a combination of standard code verification techniques. In this section, we discuss how we apply this approach with the SecCSL logic and, in particular, we present how we proved formally for SecCSL, that the code verification is sound, *i.e.* programs that verify successfully actually satisfy the specified policy.

We follow the same structure as Sec. 3.3: We first equip the language to record the performed IOD behavior, introduce the auxiliary actions to define the extended IOD behaviors of programs, and prove that the extended IOD behaviors are input-closed. Afterward, we prove that if for a policy $(\mathcal{S}, R)$, a program is verified as discussed in Sec. 3.3.2, then the program's IOD behavior satisfies the IOD spec R and it's extended IOD behavior satisfies $\mathcal{S}$-OD (which together with input-closedness proves that the program's IOD behavior satisfies the policy $(\mathcal{S}, R)$).

We formally capture our code verification approach from Sec. 3.3 with a function $\mathsf{decorate}_{\mathcal{S},R}(\cdot)$. The function takes actual program statements and adds the proof statements necessary to verify that the policy $(\mathcal{S}, R)$ is satisfied. More concretely, the function, as discussed in Sec. 3.3.1, adds the ghost pointer Trace such that it records the produced IOD actions and, as discussed in Sec. 3.3.2, adds all checks done for program verification. To verify that a program consisting of the statements c satisfies a policy $(\mathcal{S}, R)$, we then verify $\mathsf{decorate}_{\mathcal{S},R}(\mathsf{c})$ using the SecCSL logic. We refer to c and $\mathsf{decorate}_{\mathcal{S},R}(\mathsf{c})$ as the *undecorated* and *decorated* program, respectively.

In our soundness proof, we prove that the IOD behavior of the undecorated programs satisfies the policy. To define the IOD behavior of undecorated programs, we equip SecCSL's language semantic to record the performed IOD actions. As a consequence, we record a program's IOD in two ways, namely with the ghost trace for verification and on top of the language semantic to express our soundness result.

We first provide the necessary background on SecCSL in Sec. 3.4.1. In Sec. 3.4.2, we discuss SecCSL's programming language and its language semantics and we define the IOD behavior of programs. In Sec. 3.4.3, we define the extended IOD behavior of programs and prove that they are input-closed. In Sec. 3.4.4, we show how our program verification approach (from Sec. 3.3.2) is realized with SecCSL. In particular, we define the decorate function. Lastly, in Sec. 3.4.5, we prove that if a program is verified, then the program's IOD behavior and extended IOD behavior satisfy the IOD spec and $\mathcal{S}$-OD, respectively.

All results are formalized and proved in Isabelle/HOL.

3.4.1. Background on SecCSL

SecCSL is a Floyd-Hoare style program logic to reason about the secure information flow of heap-manipulating concurrent programs. The logic is formalized in Isabelle/HOL. For our soundness proof, knowledge about SecCSL's proof rules is not necessary. In this section, we provide the necessary background on SecCSL's assertions and features.

As for Gobra, SecCSL incorporates separation logic [28] to reason about memory and concurrency and uses the relational low assertion `low(e)` to reason about observational determinism. In contrast to Gobra's permission annotation `acc(x)`, permissions are specified with the separation logic's *points-to assertion* $\mathtt{x} \mapsto \mathtt{v}$, denoting that (write) permissions to the pointer x are held *and* that the pointer stores the value v, *i.e.* $*\mathtt{x}$ is v. Otherwise, SecCSL's permission reasoning is analogous to Gobra.

SecCSL targets a simple programming language with lock-based concurrency, which we refer to as SCL (SecCSL's Language) and discuss in more detail in Sec. 3.4.2. To reason about concurrency, every lock is associated with a *lock invariant*, an assertion specifying which permissions are held by the lock and which properties must hold whenever the lock is not locked. A function Inv maps locks to their associated lock invariants. For instance, the lock invariant $\mathsf{Inv}(l_1) = \exists v.\ \mathtt{y} \mapsto v \wedge v > 0$ specifies that the lock l_1 holds permissions to the pointer y, which stores some positive value. When a lock is acquired the permissions are transferred from the lock to the method acquiring the lock and the functional properties may be assumed. Conversely, when the lock is released the permissions are transferred back to the lock and the functional properties have to hold.

SCL does not have statements to allocate memory or locks. Instead, the precondition of a program or the lock invariants are used to specify which memory locations are assumed to be allocated at the start of the program. For that reason, in SCL, a program $(\mathtt{c}, A, L)$ is a triple of a statement c, an assertion A describing the initial program state, and the initial set of locks L. For instance, the program $(\mathtt{[x]\ :=\ 2}, \mathtt{x} \mapsto \mathtt{0}, \{l_1\})$ expresses the program that starts with an allocated pointer x that initially stores 0 and a single lock l_1 and then updates that value to 2.

$$\mathtt{c} ::= \mathtt{skip} \mid \mathtt{x := e} \mid \mathtt{x := N(e)} \mid \mathtt{[e_1] := e_2} \mid \mathtt{x := [e]} \mid \mathtt{lock}\ l \mid \mathtt{unlock}\ l \mid \mathtt{c_1; c_2} \mid \mathtt{c_1 \| c_2} \mid \mathtt{if(e)\{c_1\}\{c_2\}} \mid \mathtt{while(e)\{c\}}$$

Figure 3.7.: Syntax of the SCL programming language. We use c, x, e, and l to range over statements, variables, expressions, and locks, respectively. I/O actions and declassifications are performed via the action statement `x := N(e)`, assigning the input to variable x. If an action has no input, then a unit value is assigned to x. The statement $\mathtt{c_1 \| c_2}$ denotes parallel composition.

$$\frac{v = [\![\mathtt{e}]\!]_s}{\mathsf{run}(\mathtt{x := N(e)}, L, s, h) \xrightarrow{\langle \mathsf{action}\ \mathsf{N};\ \mathsf{N}(v, \underline{r}) \rangle} \mathsf{stop}(L, s[x \mapsto r], h)}\ (\textsc{Action})$$

$$\frac{v_1 = [\![\mathtt{e_1}]\!]_s \quad v_2 = [\![\mathtt{e_2}]\!]_s \quad v_1 \in dom(h)}{\mathsf{run}(\mathtt{[e_1] := e_2}, L, s, h) \xrightarrow{\langle \mathsf{store};\ \epsilon \rangle} \mathsf{stop}(L, s, h[v_1 \mapsto v_2])}\ (\textsc{Store})$$

$$\frac{v_1 = [\![\mathtt{e_1}]\!]_s \quad v_1 \notin dom(h)}{\mathsf{run}(\mathtt{[e_1] := e_2}, L, s, h) \xrightarrow{\langle \mathsf{store};\ \epsilon \rangle} \mathsf{abort}}\ (\textsc{Store}_{\textsc{Fail}})$$

$$\frac{l \in L}{\mathsf{run}(\mathtt{lock}\ l, L, s, h) \xrightarrow{\langle \mathsf{lock};\ \epsilon \rangle} \mathsf{stop}(L - \{l\}, s, h)}\ (\textsc{Lock})$$

$$\frac{\mathsf{run}(\mathtt{c_1}, L, s, h) \xrightarrow{\langle \tau;\ t \rangle} \mathsf{run}(\mathtt{c_1'}, L', s', h')}{\mathsf{run}(\mathtt{c_1 \| c_2}, L, s, h) \xrightarrow{\langle \mathsf{left} \cdot \tau;\ t \rangle} \mathsf{run}(\mathtt{c_1' \| c_2}, L', s', h')}\ (\textsc{Par}_2)$$

$$\frac{\mathsf{run}(\mathtt{c_2}, L, s, h) \xrightarrow{\langle \tau;\ t \rangle} \mathsf{run}(\mathtt{c_2'}, L', s', h')}{\mathsf{run}(\mathtt{c_1 \| c_2}, L, s, h) \xrightarrow{\langle \mathsf{right} \cdot \tau;\ t \rangle} \mathsf{run}(\mathtt{c_1 \| c_2'}, L', s', h')}\ (\textsc{Par}_4)$$

Figure 3.8.: Subset of the semantics of the SCL programming language. The step relation records the execution schedule τ and the produced trace t. The schedule determines which statement is executed next. The expression $[\![\mathtt{e}]\!]_s$ evaluates the expression e given the store s.

3.4.2. The SCL Language and the IOD Behavior of Programs

Fig. 3.7 shows the syntax of SCL's program statements. We have extended SCL with three statements, namely *action statements* and standard assume and assert statements. The action statement `x := N(e)` performs the action named N, where e is the output for the action and the action's input is assigned to the variable x. For actions such as send, a unit value is assigned to x. For simplicity, we use the same statement to perform declassifications. The assume statement `assume A` and the assert statement `assert A` assume and assert that the assertion A holds, respectively. Assume and assert statements are ghost statements and are added by the decorate function only, *i.e.* actual code does not use them (they are omitted in Fig. 3.7 since they are not actual statements). All other statements are unchanged compared to standard SCL. The heap is modified and read with a store ($\mathtt{[e_1] := e_2}$) and load statement (`x := [e]`), respectively. For concurrency, the statement $\mathtt{c_1 \| c_2}$ denotes the parallel execution of the statements $\mathtt{c_1}$ and $\mathtt{c_2}$. As mentioned before, the language does not have statements to allocate heap memory, locks, or threads, which conceptually happens before the program execution.

SCL's Language Semantic. Fig. 3.8 shows an excerpt of the language's semantics, where we have added which IOD actions are performed in an execution step. The small-step relation $\gamma \xrightarrow{\langle \tau;\, t \rangle} \gamma'$ denotes one execution step from configuration γ to configuration γ', where τ is the schedule of the execution step and t is the produced trace of IOD actions. The schedule determines which statement is executed next. The language has three kinds of configurations: $\mathsf{run}(\mathsf{c}, L, s, h)$ denotes that the statement c remains to be executed, where s is the current store, a map from variables to values, and h is the current heap, a partial map from allocated heap locations to values. The lock set L contains all locks that are currently not locked and may be obtained by c. The terminal configuration $\mathsf{stop}(L, s, h)$ denotes that the execution ended with the given store, heap, and lock set. Lastly, the configuration abort denotes that the program aborted due to an invalid memory access or that the assertion of an assert statement did not hold.

As shown in Fig. 3.8, the action statement (Action) never aborts and may obtain any possible action input. We use $[\![\mathsf{e}]\!]_s$ to denote the evaluation of the expression e with the store s. Heap modifications (Store) cause an abort if the targeted heap location is not allocated, *i.e.* if $v_1 \in dom(h)$ does not hold. The lock statement (Lock) removes the lock from the lock set L. Recall that the schedule captures which statement is executed next. In particular, for parallel composition, the schedule determines whether the left (Par_2) or right statement (Par_4) is executed next. We lift small-step relations to relations of sequences of steps straightforwardly as shown below:

$$\frac{}{\gamma \xrightarrow{\langle \epsilon;\, \epsilon \rangle}_* \gamma} \qquad \frac{\gamma \xrightarrow{\langle \tau;\, t \rangle} \gamma'' \quad \gamma'' \xrightarrow{\langle \tau';\, t' \rangle}_* \gamma'}{\gamma \xrightarrow{\langle \tau \cdot \tau';\, t \cdot t' \rangle}_* \gamma'}$$

The IOD Behavior of Programs. Def. 3.4.1 defines the IOD behavior of programs. A trace t is an element of a program's IOD behavior if there exists an execution that produces the trace and starts from a valid store and heap. The definition of a valid store and heap is more technical. A store and heap are valid if they satisfy the assertion A and all lock invariants of the initial set of locks L, denoted as $\mathsf{Inv}(L)$. SecCSL's relational semantics of assertions $(s_1, h_1), (s_2, h_2) \vDash A'$ specifies when a pair of states satisfies an assertion A'.

Definition 3.4.1 *We define the IOD behavior of a program* (c, A, L) *as*

$$\textsc{Sem}(\mathsf{c}, A, L) = \left\{ t \;\middle|\; \begin{array}{r} \exists s, h, \tau, \gamma.\ (s, h), (s, h) \vDash A \;\&\&\; \mathsf{Inv}(L) \wedge \\ \mathsf{run}(\mathsf{c}, L, s, h) \xrightarrow{\langle \tau;\, t \rangle}_* \gamma \end{array} \right\}$$

3.4.3. SCL's Extended IOD Behavior and Input-Closedness

We define the extended IOD behavior of SCL programs as discussed in Example 3.2.4 and Example 3.2.5, which is formalized in Def. 3.4.2. The auxiliary action Init has as input the initial store and heap. Furthermore, the auxiliary actions Sched capture the schedules of every program

step. Recall that Sched takes as output everything that influences which statement may be executed next, *i.e.* the current statement and lock set. To specify the extended IOD behavior, we introduce a small-step relation for extensions $\xrightarrow[\mathsf{ex}]{}$, which we again lift to sequences of steps straightforwardly. We refer to the step relation $\xrightarrow[\mathsf{ex}]{}$ as the *extended* step relation. The extended step relation produces in every step a Sched action with the current statement, lock set, and schedule, together with the actions produced by the standard language semantics. We use $\mathsf{cl}\, X$ to denote the prefix-closure of X, *i.e.* the set $\{t \mid \exists t' \in X.\ t \leq t'\}$.

Definition 3.4.2 *We define the extensions of a program* (c, A, L) *as*

$$\mathit{Esem}(\mathsf{c}, A, L) = \mathsf{cl}\left\{ \mathsf{Init}(\underline{s, h}) \cdot t \;\middle|\; \begin{array}{r} \exists s, h, \gamma. \\ (s,h), (s,h) \vDash A \;\&\&\; \mathsf{Inv}(L) \wedge \\ \mathsf{run}(\mathsf{c}, L, s, h) \xrightarrow[\mathsf{ex}]{t}{}^{*}\ \gamma \end{array} \right\}$$

where

$$\frac{\mathsf{run}(\mathsf{c}, L, s, h) \xrightarrow{\langle \tau;\, t \rangle} \gamma}{\mathsf{run}(\mathsf{c}, L, s, h) \xrightarrow[\mathsf{ex}]{\mathsf{Sched}(\mathsf{c}, L, \underline{\tau}) \cdot t} \gamma}\ (\textsc{Extended Step})$$

Input-Closedness. As formalized in Def. 3.2.4, to prove input-closedness of all extended IOD behaviors, we show that for every program, if there exists some trace $t \cdot N(x, \underline{r}) \in \textsc{Esem}(\mathsf{c}, A, L)$ and some other possible input r', *i.e.* there exists t' with $t' \cdot N(x, \underline{r'}) \in \textsc{Esem}(\mathsf{c}, A, L)$, then $t \cdot N(x, \underline{r'}) \in \textsc{Esem}(\mathsf{c}, A, L)$ also holds. We prove this property by a case distinction on the action N. For actual actions, $t \cdot N(x, \underline{r'}) \in \textsc{Esem}(\mathsf{c}, A, L)$ is implied directly by the language semantic shown in Fig. 3.8. The case for the auxiliary Init action is trivial since it is always the first action of a trace. In particular, the trace $t' \cdot N(x, \underline{r'})$, which is in $\textsc{Esem}(\mathsf{c}, A, L)$, is the same as $t \cdot N(x, \underline{r'})$ since $t' = \epsilon = t$. The case for the auxiliary Sched action is also straightforward. If $N = \mathsf{Sched}$, then the output x is the tuple of some statement c' and lock set L' and the inputs r and r' are some schedules τ and τ', respectively. As discussed in Example 3.2.5, the case for $N = \mathsf{Sched}$ boils down to proving that whether something is the schedule of an execution step depends only on the statement and lock set of the execution step, *i.e.* the outputs of the auxiliary Sched action. More formally, we prove that if there is an execution step $\mathsf{run}(\mathsf{c}', L', s_0, h_0) \xrightarrow{\langle \tau;\, t_0 \rangle} \gamma_0$ (stemming from $t \cdot N(x, \underline{r}) \in \textsc{Esem}(\mathsf{c}, A, L)$) and another execution step $\mathsf{run}(\mathsf{c}', L', s_1, h_1) \xrightarrow{\langle \tau';\, t_1 \rangle} \gamma_1$ (stemming from $t' \cdot N(x, \underline{r'}) \in \textsc{Esem}(\mathsf{c}, A, L)$), then the first execution step is possible with the schedule τ', *i.e.* $\mathsf{run}(\mathsf{c}', L', s_0, h_0) \xrightarrow{\langle \tau';\, t_2 \rangle} \gamma_2$ holds for some t_2 and γ_2 (justifying $t \cdot N(x, \underline{r'}) \in \textsc{Esem}(\mathsf{c}, A, L)$). We prove this property by induction on the step relation, concluding the proof.

Theorem 3.4.1 $\textsc{Esem}(\mathsf{c}, A, L)$ *is input-closed for every program* (c, A, L).

$$\mathsf{decorate}_{\mathcal{S},R}(\mathsf{c}) = \begin{cases} \mathsf{t_{old}} := [\mathsf{Trace}]; & \text{if c is } \mathsf{x} := \mathsf{N(e)} \\ \mathsf{e_{old}} := \mathsf{e}; & \\ \mathsf{x} := \mathsf{N(e)}; & \\ [\mathsf{Trace}] := \mathsf{t_{old}} \cdot \mathsf{N}(\mathsf{e_{old}}, \underline{\mathsf{x}}); & \\ \texttt{assume } \mathsf{t_{old}} \cdot \mathsf{N}(\mathsf{e_{old}}, \underline{\mathsf{x}}) \leq \mathsf{Proph}; & \\ \texttt{assert low}(\mathsf{t_{old}} \cdot \mathsf{N}(\mathsf{e_{old}}, \underline{\mathsf{x}}) \downarrow_{\mathcal{S}}^{\mathsf{pre}}); & \\ \texttt{assert } \mathsf{t_{old}} \cdot \mathsf{N}(\mathsf{e_{old}}, \underline{\mathsf{x}}) \in R & \\ & \\ \mathsf{s} & \text{if c is not } \mathsf{x} := \mathsf{N(e)} \end{cases}$$

Figure 3.9.: Definition of the decorate function, which adds all necessary proof statements to SCL statements. The function adds the ghost pointer Trace that records all produced actions and the checks necessary to satisfy the policy $(\mathcal{S}, R)$.

3.4.4. Code Verification with SecCSL

As discussed at the start of this section, to verify that a program (c, A, L) satisfies a policy $(\mathcal{S}, R)$, we verify $\mathsf{decorate}_{\mathcal{S},R}(\mathsf{c})$, *i.e.* the decorated program *with* the necessary proof statements, using the SecCSL logic. More formally, we derive the triple $\{P\}\ \mathsf{decorate}_{\mathcal{S},R}(\mathsf{c})\ \{\mathsf{true}\}$ for a specific precondition P in the SecCSL logic. In this subsection, we first discuss the function decorate, and afterward, we define the precondition P.

The decorate Function. Fig. 3.9 shows the definition of the decorate function. The function adds for action statements $\mathsf{x} := \mathsf{N(e)}$, statements to (1) record the produced IOD actions (Sec. 3.3.1) and to (2) check that the IOD behavior and extended IOD behavior satisfy the IOD spec and $\mathcal{S}$-OD, respectively (Sec. 3.3.2).

To record the IOD actions, the ghost pointer is first read ($\mathsf{t_{old}} := [\mathsf{Trace}]$) and then appended with the produced action ($[\mathsf{Trace}] := \mathsf{t_{old}} \cdot \mathsf{N}(\mathsf{e_{old}}, \underline{\mathsf{x}})$). The temporary variable $\mathsf{e_{old}}$ stores the value of the output. In contrast to Gobra, where we annotate the pre- and postconditions of library methods, we assign to the ghost pointer directly because SCL's action statements do not have user-specified pre- and postconditions.

Regarding checks, as discussed in Sec. 3.3.2, after an action is produced, for the IOD spec, we check that the recorded trace is a trace of the IOD spec ($\texttt{assert } \mathsf{t_{old}} \cdot \mathsf{N}(\mathsf{e_{old}}, \underline{\mathsf{x}}) \in R$ in Fig. 3.9) and for observational determinism, we check that $\texttt{low}(*\texttt{Trace}\downarrow_{\mathcal{S}}^{\mathsf{pre}})$ holds ($\texttt{assert low}(\mathsf{t_{old}} \cdot \mathsf{N}(\mathsf{e_{old}}, \underline{\mathsf{x}})\downarrow_{\mathcal{S}}^{\mathsf{pre}})$ in Fig. 3.9).* Lastly, the statement $\texttt{assume } \mathsf{t_{old}} \cdot \mathsf{N}(\mathsf{e_{old}}, \underline{\mathsf{x}}) \leq \mathsf{Proph}$, which we explain later, enables us to assume that $\texttt{low}(*\texttt{Trace}\downarrow_{\mathcal{S}}^{\mathsf{post}})$ holds before an action is produced. Since SecCSl is formalized in Isabelle/HOL, in contrast to Gobra, we are able to use the low projections and the IOD spec directly. In particular, we do not need utility functions such as `Reaches` or `Pre`.

The Precondition. Recall that to verify that a program satisfies a policy in Gobra, we verify the program with the precondition that we have write permission to the Trace pointer and the Trace pointer stores the empty trace (see Sec. 3.3). The analogous precondition for a SCL program (c, A, L), namely $\mathsf{Trace} \mapsto \epsilon$ && A, is ill-suited to prove soundness. In a soundness proof, we have to justify why we are allowed to assume $\texttt{low}(*\texttt{Trace}\downarrow_{\mathcal{S}}^{\mathsf{post}})$ before an action is produced. To justify this assumption,

* As a technicality, SecCSL does not support assert statements with low assertions. Instead, SecCSL supports *low pointers*, whose value must always be low. We assert that a value is low by writing to such a low pointer. We have extended SecCSL's language with assume and assert statements for assertions without `low`.

we extend the precondition $\mathsf{Trace} \mapsto \epsilon$ && A with an additional assertion, which we refer to as the *justification assertion*. The justification assertion does *not* influence the verification approach, in particular, it does neither change nor add proof obligations. Because of that, we may omit the justification assertion when verifying programs, *e.g.* with Gobra. For the remainder of this subsection, we explain and specify the justification assertion.

We specify the justification assertion using a technique from literature [101, 102]: We introduce a *prophecy variable* Proph that stores, already at the beginning of a program, the trace of actions that an execution is going to produce. The justification assertion then specifies that Proph stores a trace t that satisfies the implication $\texttt{low}(t'\downarrow_S^{\mathsf{pre}}) \Rightarrow \texttt{low}(t'\downarrow_S^{\mathsf{post}})$ for all prefixes $t' \leq t$. Because we show $\texttt{low}(*\texttt{Trace}\downarrow_S^{\mathsf{pre}})$ after an action is produced and $*\texttt{Trace}$ records a prefix of the trace that is going to be produced, we achieve our goal that we are allowed to assume $\texttt{low}(*\texttt{Trace}\downarrow_S^{\mathsf{post}})$ before the next action is produced.

[101]: Tasiran et al. (2009), *Verifying Optimistic Concurrency: Prophecy Variables and Backward Reasoning*
[102]: Jung et al. (2020), *The future is ours: prophecy variables in separation logic*

We first formalize the justification assertion and then discuss how we realize the prophecy variable in SecCSL, in particular, how we capture that the prophecy variable stores the trace that is going to be produced. Lastly, we define when a program is successfully verified with SecCSL.

The Justification Assertion. Recall that we reason about observational determinism with pairs of executions. Def. 3.4.3 defines when a pair of traces satisfies the condition mentioned above, namely that all prefixes satisfy $\texttt{low}(t'\downarrow_S^{\mathsf{pre}}) \Rightarrow \texttt{low}(t'\downarrow_S^{\mathsf{post}})$, which we refer to as *assumption-compatible*. The justification assertion is then $(\mathsf{Proph} = t_1 \vee \mathsf{Proph} = t_2)$ && $t_1 \#_S t_2$, expressing that the prophecy variable Proph stores one of the assumption-compatible traces. The assertion entails that for pairs of executions, the trace t stored in the Proph variable in one execution and the trace t' stored in the other execution are always assumption-compatible. Note that traces are assumption-compatible with themselves, *i.e.* $t \#_S t$ always holds. Lemma 3.4.2 captures formally that with the justification assertion, as long as a trace $\mathtt{t}$ is a prefix of Proph, the desired implication $\texttt{low}(\mathtt{t}\downarrow_S^{\mathsf{pre}}) \Rightarrow \texttt{low}(\mathtt{t}\downarrow_S^{\mathsf{post}})$ holds. Programs never modify the prophecy variable and, thereby, the justification assertion always holds.

Definition 3.4.3 *Two traces t_1, t_2 are assumption-compatible, denoted as $t_1 \#_S t_2$, if for all prefixes $t_1' \leq t_1$ and $t_2' \leq t_2$,*

$$t_1'\downarrow_S^{\mathsf{pre}} = t_2'\downarrow_S^{\mathsf{pre}} \Rightarrow t_1'\downarrow_S^{\mathsf{post}} = t_2'\downarrow_S^{\mathsf{post}}$$

Lemma 3.4.2 *The following implication holds in SecCSL:*

$$(\mathsf{Proph} = t_1 \vee \mathsf{Proph} = t_2) \;\&\&\; t_1 \#_S t_2 \;\&\&\; \mathtt{t} \leq \mathsf{Proph} \;\&\&\; \texttt{low}(\mathtt{t}\downarrow_S^{\mathsf{pre}}) \Rightarrow \texttt{low}(\mathtt{t}\downarrow_S^{\mathsf{post}})$$

The Prophecy Variable. We realize the prophecy variable Proph with two ingredients: Firstly, to capture that Proph stores the trace that is going to be produced, we add the assumption that the recorded trace is always a prefix of Proph. More concretely, as shown in the definition of decorate in Fig. 3.9, we add the assume statement $\texttt{assume}\ \mathsf{t_{old}} \cdot \mathsf{N}(\mathsf{e_{old}}, \underline{\mathsf{x}}) \leq \mathsf{Proph}$.

Secondly, to justify this assumption, we verify a program for all values of Proph. By verifying a program for all traces, we also verify the program for the trace that correctly predicts the IOD actions that will be produced. Our soundness proof shows that this treatment is sound (see Sec. 3.4.5.3).

Verification Correctness. Def. 3.4.4 defines our notion of verification correctness: To verify that a program (c, A, L) satisfies a policy $(\mathcal{S}, R)$, we have to derive in SecCSL for all traces t_1 and t_2, the triple specified in Def. 3.4.4.

Definition 3.4.4 *A program (c, A, L) is successfully verified for a policy $(\mathcal{S}, R)$, which we denote as* $\mathsf{verified}_{\mathcal{S},R}(\mathsf{c}, A, L)$*, if for all t_1, t_2,*

$$\vdash \left\{ \begin{array}{l} (\mathsf{Proph} = t_1 \vee \mathsf{Proph} = t_2) \;\&\&\; t_1 \#_{\mathcal{S}} t_2 \\ \&\&\;\; \mathsf{Trace} \mapsto \epsilon \;\&\&\; A \end{array} \right\} \mathsf{decorate}_{\mathcal{S},R}(\mathsf{c}) \; \{\mathsf{true}\}$$

3.4.5. Soundness

In this section, we complete the soundness proof by proving that if a program (c, A, L) is successfully verified for a policy $(\mathcal{S}, R)$, then the program's IOD behavior satisfies the IOD spec, *i.e.* $\textsc{Sem}(\mathsf{c}, A, L) \subseteq R$, and the program's extended IOD behavior $\textsc{Esem}(\mathsf{c}, A, L)$ satisfies $\mathcal{S}$-OD. As discussed at the beginning of this section, together with the property that the extended IOD behaviors are input-closed for all programs, we prove the main soundness result captured in Theorem 3.4.3.

Theorem 3.4.3 *For every successfully verified program* $\mathsf{verified}_{\mathcal{S},R}(\mathsf{c}, A, L)$*, the IOD behavior $\textsc{Sem}(\mathsf{c}, A, L)$ satisfies the policy $(\mathcal{S}, R)$.*

We prove that for successfully verified programs, the IOD spec and $\mathcal{S}$-OD are satisfied in two phases: In the first phase, we prove a soundness result at the level of our extended step relation, which we refer to as *extended verification soundness*. We prove extended verification soundness using SecCSL's original soundness result about pairs of executions proved by Ernst and Murray [14]. Except for the addition of standard assume and assert statements, whose discussion we omit, we do not modify the soundness proof of the SecCSL logic. Instead, we only lift the soundness result to our extended step relation. In the second phase, we prove that extended verification soundness entails that for successfully verified programs, the IOD spec and $\mathcal{S}$-OD are satisfied by the program's IOD behavior and extended IOD behavior, respectively.[1]

1: In our Isabelle/HOL formalization, for historical reasons, we prove the second phase directly, without proving extended verification soundness separately. We present the two phases for the sake of clarity.

The remainder of this section is structured as follows: We first define extended verification soundness (Sec. 3.4.5.1). Afterward, we discuss the phases in reverse order. In Sec. 3.4.5.2, we prove that extended verification soundness entails Theorem 3.4.3. Lastly, in Sec. 3.4.5.3, we prove extended verification soundness.

3.4.5.1. Definition of Extended Verification Soundness

As defined in Lemma 3.4.4, extended verification Soundness entails that if a program is successfully verified, then for all pairs of execution that produce assumption-compatible extended traces ($t_1 \#_{(\mathcal{S}+\mathsf{NoDecl})} t_2$), we

get that (1) the reached configurations γ_1 and γ_2 are not abortions, (2) if the produced extended traces have the same number of low actions, then the reached configurations γ_1 and γ_2 are the same kind (either run or stop configurations) with the same statement and lock set, and (3) the outputs that are classified as low by the action's preconditions of both produced extended traces are equal. Point (2) entails that the control flow and, in particular, the statements and lock sets of configurations are low (for executions with an equal number of steps). The produced extended traces may have different sizes. Therefore, instead of expressing that the low data is equal, we express that the low data is equal up to the common length, which is captured by the definition of $\sim$ (also shown in Lemma 3.4.4).

Lemma 3.4.4 (Extended Verification Soundness) *For every verified program* $\mathsf{verified}_{\mathcal{S},R}(c, A, L)$ *and valid initial state* (s, h), $(s, h) \vDash A \;\&\&\; \mathsf{Inv}(L)$,

$$\begin{aligned}
&\mathsf{run}(c, L, s, h) \xrightarrow[\mathsf{ex}]{t_1}{}_* \gamma_1 \wedge \mathsf{run}(c, L, s, h) \xrightarrow[\mathsf{ex}]{t_2}{}_* \gamma_2 \wedge t_1 \,\#_{(\mathcal{S}+\mathsf{NoDecl})}\, t_2 \\
&\quad \mathsf{Ctxt}(\gamma_1) \neq \mathsf{abort} \wedge \mathsf{Ctxt}(\gamma_2) \neq \mathsf{abort} \\
\Longrightarrow \;& \wedge \; \big(|t_1{\downarrow_{(\mathcal{S}+\mathsf{NoDecl})}}| = |t_2{\downarrow_{(\mathcal{S}+\mathsf{NoDecl})}}| \Longrightarrow \mathsf{Ctxt}(\gamma_1) = \mathsf{Ctxt}(\gamma_2)\big) \\
&\quad \wedge \; t_1{\downarrow^{\mathsf{pre}}_{(\mathcal{S}+\mathsf{NoDecl})}} \sim t_2{\downarrow^{\mathsf{pre}}_{(\mathcal{S}+\mathsf{NoDecl})}}
\end{aligned}$$

where $a \sim b$ *if* $a \leq b$ *or* $a \geq b$ *and*

$$\begin{aligned}
\mathsf{Ctxt}(\mathsf{run}(c, L, s, h)) &= (\mathsf{run}, c, L) \\
\mathsf{Ctxt}(\mathsf{stop}(L, s, h)) &= (\mathsf{stop}, L) \\
\mathsf{Ctxt}(\mathsf{abort}) &= \mathsf{abort}
\end{aligned}$$

3.4.5.2. Proof of the IOD Spec and $\mathcal{S}$-OD

We first prove that the IOD behaviors of programs satisfy the IOD spec (Theorem 3.4.5). Afterward, we prove that the extended IOD behaviors of programs satisfy $\mathcal{S}$-OD by first proving progress-insensitive $\mathcal{S}$-OD (Theorem 3.4.6) and then proving progress-sensitivity (Theorem 3.4.8).

Theorem 3.4.5 *For every successfully verified program* $\mathsf{verified}_{\mathcal{S},R}(c, A, L)$, *the IOD behavior* $\textsc{Sem}(c, A, L)$ *is a subset of the IOD spec* R.

Proof. $t \in \textsc{Sem}(c, A, L)$ entails that there exists an execution that reaches some configuration γ' and produces the trace t. Since every trace is assumption-compatible with itself, *i.e.* $t \,\#_{\mathcal{S}}\, t$ always holds, we are able to use extended verification soundness (Lemma 3.4.4) to derive that γ' is not an abortion. Since γ' is not an abortion, we know that the assert statements $\texttt{assert}\ \mathsf{t_{old}} \cdot \mathsf{N}(\mathsf{e_{old}}, \underline{x}) \in R$ added by the decorate function succeeded and, thereby, $t \in R$ holds, concluding the proof. □

Theorem 3.4.6 *For every successfully verified program* $\mathsf{verified}_{\mathcal{S},R}(c, A, L)$, *the extended IOD behavior* $\textsc{Esem}(c, A, L)$ *satisfies progress-insensitive* $(\mathcal{S} + \mathsf{NoDecl})$*-OD.*

Proof. As an intermediate step, we first prove Lemma 3.4.7, which provides a more suitable way to prove progress-insensitive OD. Lemma 3.4.7 itself is proved with a straightforward induction.

Lemma 3.4.7 *The prefix-closure of an IOD behavior T satisfies progress-insensitive $\mathcal{S}'$-OD if, for all traces $t_1, t_2 \in T$,*

$$t_1 \#_{\mathcal{S}'} t_2 \Rightarrow t_1{\downarrow}^{\text{pre}}_{\mathcal{S}'} \sim t_2{\downarrow}^{\text{pre}}_{\mathcal{S}'}$$

We then prove Theorem 3.4.6 straightforwardly. If there are two extended traces t_1 and t_2 in the extended IOD behavior $\textsc{Esem}(\mathsf{c}, A, L)$, then we know that there are two executions that produce t_1 and t_2 (with the extended step relation). Together with $t_1 \#_{\mathcal{S}+\text{NoDecl}} t_2$, extended verification soundness (Lemma 3.4.4) implies that $t_1{\downarrow}^{\text{pre}}_{(\mathcal{S}+\text{NoDecl})} \sim t_2{\downarrow}^{\text{pre}}_{(\mathcal{S}+\text{NoDecl})}$ holds, concluding the proof. □

Theorem 3.4.8 *For every successfully verified program $\mathsf{verified}_{\mathcal{S},R}(\mathsf{c}, A, L)$, the extended IOD behavior $\textsc{Esem}(\mathsf{c}, A, L)$ satisfies progress-sensitivity.*

Proof. The proof boils down to proving that if there are two executions that produce the extended traces t_1 and t_2 with $t_1{\downarrow}^{\text{post}}_{\mathcal{S}+\text{NoDecl}} = t_2{\downarrow}^{\text{post}}_{\mathcal{S}+\text{NoDecl}}$ and reach configurations γ_1 and γ_2, respectively, and if there exists another execution step from γ_1, then there must also be another execution step from γ_2. $t_1{\downarrow}^{\text{post}}_{\mathcal{S}+\text{NoDecl}} = t_2{\downarrow}^{\text{post}}_{\mathcal{S}+\text{NoDecl}}$ entails $t_1 \#_{\mathcal{S}+\text{NoDecl}} t_2$. Therefore, we can use extended verification soundness (Lemma 3.4.4) to derive that the configurations γ_1 and γ_2 have the same statements and lock set. We then conclude the proof by showing that whether another execution step is possible depends only on the statement and lock set of a configuration, which is formalized with Lemma 3.4.9.

Lemma 3.4.9 *For every store s' and heap h',*

$$\mathsf{run}(\mathsf{c}, L, s, h) \xrightarrow{\langle \tau;\, t \rangle} \gamma \Rightarrow \exists \tau', t', \gamma'.\ \mathsf{run}(\mathsf{c}, L, s', h') \xrightarrow{\langle \tau';\, t' \rangle} \gamma'$$

We prove Lemma 3.4.9 with an induction on the step relation. □

3.4.5.3. Proof of Extended Verification Soundness

[14]: Ernst et al. (2019), *SecCSL: Security Concurrent Separation Logic*

As mentioned above, we prove extended verification soundness by lifting SecCSL's soundness result proved by Ernst and Murray [14] to our extended step relation. To split the proof, we first lift SecCSL's soundness result to decorated statements and then to our extended step relation. Before discussing these proofs, we first explain SecCSL's soundness result.

SecCSL's Soundness Result. Theorem 3.4.10 formalizes a simplified version of SecCSL's soundness result proved by Ernst and Murray. For verified programs, SecCSL ensures that for pairs of executions with the same schedule that reach the configurations $\hat{\gamma}_1$ and $\hat{\gamma}_2$, (1) $\hat{\gamma}_1$ and $\hat{\gamma}_2$ are not abortions, (2) $\hat{\gamma}_1$ and $\hat{\gamma}_2$ are the same kind of configuration (either run or stop) with the same statement and lock set, and (3) values stored on the heap or store that are asserted as low (with the low assertion)

are equal in both executions, denoted as $\mathsf{Low}(\hat{\gamma}_1) = \mathsf{Low}(\hat{\gamma}_2)$. Points (1) and (2) are analogous to extended verification soundness. Since traces produced by a program become the content of the Trace pointer and we assert $\mathtt{low}(*\mathsf{Trace}\downarrow_{\mathcal{S}}^{\mathsf{pre}})$ (see Fig. 3.9), point (3) entails that the outputs of produced actions that are classified as low by the action's preconditions are equal for both executions. We omit the trace parameter of the step relation when specifying the executions of undecorated statements. Since we have not modified SecCSL's soundness result, it does not include the trace parameter, which was added by us.

Theorem 3.4.10 (SecCSL's Verification Soundness) *For every SCL statement* $\hat{c}$ *verified in SecCSL with precondition* P*, i.e.* $\vdash \{P\}\hat{c}\{\mathsf{true}\}$*, and valid initial states* $(s_1, h_1), (s_2, h_2) \vDash P$ && $\mathsf{Inv}(L)$*,*

$$\mathsf{run}(c, L, s_1, h_1) \xrightarrow{\tau}_* \hat{\gamma}_1 \wedge \mathsf{run}(c, L, s_2, h_2) \xrightarrow{\tau}_* \hat{\gamma}_2$$
$$\Rightarrow \mathsf{Ctxt}(\hat{\gamma}_1) \neq \mathsf{abort} \wedge \mathsf{Ctxt}(\hat{\gamma}_1) = \mathsf{Ctxt}(\hat{\gamma}_2) \wedge \mathsf{Low}(\hat{\gamma}_1) = \mathsf{Low}(\hat{\gamma}_2)$$

Decorated Verification Soundness. Lemma 3.4.11 formalizes the soundness result lifted to the verified decorated statements. The property is similar to our extended verification soundness, except that, because the schedules of both executions are equal, we are guaranteed that the executions have the same number of steps.

Lemma 3.4.11 (Decorated Verification Soundness) *For every verified program* $\mathsf{verified}_{\mathcal{S},R}(c, A, L)$ *and valid initial state* $(s, h), (s, h) \vDash A$ && $\mathsf{Inv}(L)$*,*

$$\mathsf{run}(c, L, s, h) \xrightarrow{\langle \tau;\, t_1 \rangle}_* \gamma_1 \wedge \mathsf{run}(c, L, s, h) \xrightarrow{\langle \tau;\, t_2 \rangle}_* \gamma_2$$
$$\wedge\, t_1 \#_{\mathcal{S}}\, t_2 \Rightarrow \mathsf{Ctxt}(\gamma_1) \neq \mathsf{abort} \wedge \mathsf{Ctxt}(\gamma_1) = \mathsf{Ctxt}(\gamma_2) \wedge t_1\downarrow_{\mathcal{S}} = t_2\downarrow_{\mathcal{S}}$$

Proof. We prove Lemma 3.4.11 from SecCSL's verification soundness (Theorem 3.4.10) by proving the relation between program executions of undecorated and decorated statements. If the execution of an undecorated statement c produces a trace t, then there exists an execution of $\mathsf{decorate}_{\mathcal{S},R}(\mathsf{c})$, which either aborts (if an assertion failed) or results in a state where Trace stores the produced trace t. Lemma 3.4.12 formalizes a generalized version of the relation, where we consider that a trace t' has been produced beforehand and a trace t'' is going to be produced afterward. As before, we omit the trace parameter of the step relation when specifying executions of undecorated statements. The function $\mathsf{decod}(\tau)$ returns the schedule of the execution for $\mathsf{decorate}_{\mathcal{S},R}(\mathsf{c})$. The function accounts for the fact that the single action statements result in a sequence of statements (due to the added proof statements). We use $s[x \mapsto v]$ to denote the store where the variable x is updated with value v. Similarly, $h[l \mapsto v]$ denotes the heap where the pointer l is updated with the value v. We prove Lemma 3.4.12 by an induction on the step relation.

Lemma 3.4.12 *For every statement* c *and all traces* t, t', t'',

$$\mathsf{run}(c, L, s, h) \xrightarrow{\langle \tau;\, t \rangle}_* \gamma \Rightarrow$$
$$\mathsf{run}(\mathsf{decorate}_{\mathcal{S},R}(c), L, s[\mathsf{Proph} \mapsto t' \cdot t \cdot t''], h[\mathsf{Trace} \mapsto t']) \xrightarrow{\mathsf{decod}(\tau)}_* \hat{\gamma}$$
$$\wedge\ (\hat{\gamma} = \mathsf{abort} \vee \hat{\gamma} = \mathsf{decorate}_{\mathcal{S},R}(\gamma, t' \cdot t \cdot t'', t' \cdot t))$$

where

$$\mathsf{decorate}_{\mathcal{S},R}(\mathsf{run}(c, L, s, h), t' \cdot t \cdot t'', t' \cdot t) =$$
$$\mathsf{run}(\mathsf{decorate}_{\mathcal{S},R}(c), L, s[\mathsf{Proph} \mapsto t' \cdot t \cdot t''], h[\mathsf{Trace} \mapsto t' \cdot t])$$
$$\mathsf{decorate}_{\mathcal{S},R}(\mathsf{stop}(L, s, h), t' \cdot t \cdot t'', t' \cdot t) =$$
$$\mathsf{stop}(L, s[\mathsf{Proph} \mapsto t' \cdot t \cdot t''], h[\mathsf{Trace} \mapsto t' \cdot t])$$

Note that by assigning the trace $t' \cdot t \cdot t''$ to the prophecy variable Proph, the prophecy variable stores a trace that correctly predicts the IOD actions that are going to be produced. In particular, all assertions of the assume statements added by the decorate function are satisfied.

Given Lemma 3.4.12, Lemma 3.4.11 follows directly from SecCSL's verification soundness (Theorem 3.4.10). □

Extended Verification Soundness. Lastly, we prove extended verification soundness by lifting the verification soundness of decorated statements (Lemma 3.4.11) to the extended step relation.

Proof. Instead of just $\mathsf{run}(c, L, s, h) \xrightarrow[\mathsf{ex}]{t_i}_* \gamma_i$, we track an execution prefix leading to the execution $\mathsf{run}(c, L, s, h) \xrightarrow{\langle \tau_i;\, t_i' \rangle}_* \gamma_i'$ such that $\gamma_i' \xrightarrow[\mathsf{ex}]{t_i}_* \gamma_i$. We then prove extended verification soundness (Lemma 3.4.4) using an induction on the extended step relation. As part of the induction hypothesis, we enforce that schedules of both execution prefixes are equal, *i.e.* $\tau_1 = \tau_2$. In the induction step, we use Lemma 3.4.11 to get that the statement and lock set of γ_1' and γ_2' are equal. Since t_1 and t_2 are assumption-compatible, we get that the schedules of the next steps of γ_1' and γ_2' are equal (if such next steps exist), reestablishing that the schedules of both executions are equal. Lastly, we derive $t_1{\downarrow}_{\mathcal{S}}^{\mathsf{pre}} = t_2{\downarrow}_{\mathcal{S}}^{\mathsf{pre}}$ by applying Lemma 3.4.11 to the execution prefixes. □

3.5. Policy Validation

Like code, security policies may contain errors due to human failure. The aim of validating a security policy is to increase the confidence that the policy specifies the intended security requirements. We validate policies by proving properties that hold for all IOD behaviors satisfying the policy. In this section, we focus on validating that a policy does not permit the release of information that is intended to remain confidential.

Consider an incorrect variant of our running example's IOD spec (Example 3.2.3), where a declassification action $\mathsf{decl}(id, x)$ is permitted whenever

x is the encryption of the date and key queried for *id if* (instead of *and*) *id* has been queried. This IOD spec is bad since it permits the declassification of any value, *e.g.* the confidential list of compatible vaccines, if the id has not been queried yet.

We use two approaches to validate policies: (1) Since implementations refine a policy's IOD spec R, any trace property P satisfied by the IOD spec is also satisfied by the implementation. E.g., for our running example, we may prove that a declassification is permitted only if the id has been queried beforehand. (2) The combination of IOD spec and classification spec enables us, instead of just validating properties about when declassification is permitted, to prove directly that specific data remains confidential even in the presence of declassification. Approach (1) requires standard reasoning about transition systems; we focus on Approach (2) in this section.

We formalize the property that data remains confidential as GNIV. As mentioned in Sec. 3.1, GNIV is an adaptation of generalized noninterference [21] (GNI) that can handle distributed systems.

[21]: Clarkson et al. (2008), *Hyperproperties*

We first discuss the definition of GNIV for passive attackers (Sec. 3.5.1) and show how to prove it (Sec. 3.5.2). Afterwards, we extend GNIV to active attackers (Sec. 3.5.3). As defined in Sec. 3.2.3, passive attackers are able to observe the low data of all performed I/O actions. Active attackers are in addition able to change the low inputs of actions.

3.5.1. Definition of GNIV

Declassifications are able to release information about high data. Therefore, not all data classified as high by the classification spec remains confidential. To capture the intended confidentiality, we use a function Λ, referred to as *view*, that takes a trace and returns the data that we want to prove remains confidential in the presence of declassification. We lift Λ to sets of traces, denoted with $\cdot\downarrow_\Lambda$.

Before we discuss the definition of GNIV, we first illustrate why standard GNI is ill-suited for distributed systems. GNI is satisfied by an IOD behavior if for every pair of traces $t_1, t_2 \in T$, there exists a trace t_u with the same low data as t_1 and the same secret as t_2 (*i.e.* $\Lambda(t_u) = \Lambda(t_2)$). We refer to t_u as the *uncertainty trace*. As mentioned before, the issue is that GNI rules out IOD behaviors that let an attacker learn whether a secret input exists, regardless of whether the attacker actually is able to learn the value of the secret input itself. Consider the program below and a view $\Lambda_0 : t \mapsto t\downarrow_{\mathsf{getKey}}$, where $t\downarrow_{\mathsf{getKey}}$ denotes the sequence of inputs of getKey actions occurring in the trace t. In the code, getKey actions are produced by calls to the `GetKey` method:

```
h1 := GetKey(); if Recv() { h2 := GetKey(); }
```

The program does not satisfy GNI for Λ_0 because the low input received from `Recv` implies whether a second getKey action happens. More formally, there exists no uncertainty trace that has the same secrets as the trace $\mathsf{getKey}(\overline{h_1}) \cdot \mathsf{recv}(\mathsf{true}) \cdot \mathsf{getKey}(\overline{h_2})$ but also the same low data as the trace $\mathsf{getKey}(\overline{h_1'}) \cdot \mathsf{recv}(\mathsf{false})$ for all values h_1,h_2,h_1'. GNI fails because the number of getKey actions is different depending on a trace's low data.

However, we consider the program secure since no information about the inputs of getKey actions is released (the program does not even use these inputs).

Our definition of GNIV solves this issue by adapting GNI in two ways. First, when comparing the uncertainty trace and trace t_2, we consider secrets that are intended to remain confidential according to a view Λ. Second, we compare the two traces only up to the common number of secret inputs by allowing the uncertainty trace to have more or fewer actions, accounting for different numbers of secret inputs:

Definition 3.5.1 (Compatibility) *An uncertainty trace t_u is compatible with a secret h for a view Λ, denoted as $t_u \#_\Lambda h$, if there exists a trace t' with $\Lambda(t') = h$ and $t_u \leq t' \vee t' \leq t_u$.*

For our example, the uncertainty trace $\mathsf{getKey}(\underline{\mathsf{h1}})\cdot\mathsf{recv}(\underline{\mathsf{false}})$ is compatible with the secret of trace $\mathsf{getKey}(\underline{h_1}) \cdot \mathsf{recv}(\mathsf{true})$.

For the definition of GNIV, to express a trace's low data, we use the projection $\cdot\downarrow_{\mathcal{S}}$, combining $\cdot\downarrow_{\mathcal{S}}^{\mathsf{pre}}$ and $\cdot\downarrow_{\mathcal{S}}^{\mathsf{post}}$, *e.g.* $\mathsf{query}(id, \underline{d})\downarrow_{\mathcal{S}} = \mathsf{query}(id, \underline{d_{\mathsf{key}}})$.

Definition 3.5.2 (GNIV) *An IOD behavior T satisfies GNIV for a classification spec $\mathcal{S}$ and view Λ, if for every secret $h \in T\downarrow_\Lambda$ and every low data $l \in T\downarrow_{\mathcal{S}}$, there exists an uncertainty trace $t_u \in T$,*

$$t_u\downarrow_{\mathcal{S}} = l \wedge t_u \#_\Lambda h.$$

3.5.2. Proving GNIV

To prove that all IOD behaviors satisfying a policy also satisfy GNIV (for some view), we construct an uncertainty trace step by step. For every trace t of the IOD spec and every possible secret h (according to the view), we show that there is an uncertainty trace t_u that has the same declassifications as t but is also secret compatible with h.

We formalize the step-by-step construction as a *trace construction plan*, a function that takes the secret h, the uncertainty trace constructed so far, the low data (in particular, the declassifications) that still have to be constructed, and the next action $\mathsf{N}(x, \underline{r})$, and returns the input r' that replaces r to create the uncertainty trace. E.g., for our previous example with getKey, a suitable plan ξ satisfies $\xi(h_1 \cdot h_2, l, \epsilon, \mathsf{getKey}(\underline{h_1'})) = h_1$, *i.e.* the plan replaces the secret input h_1' of the first getKey action (the trace constructed so far is empty) with the first value h_1 of the sequence of secrets $h_1 \cdot h_2$ that the uncertainty trace has to be compatible with. The low data l that still has to be constructed is either $\mathsf{recv}(\underline{\mathsf{true}})$ or $\mathsf{recv}(\underline{\mathsf{false}})$.

To prove that a policy entails GNIV for a view, we have to show that there exists a trace construction plan that satisfies three conditions that we define below. For our examples, we have done these proofs in Isabelle/HOL; we leave the automation of such proofs to future work.

The three conditions that a trace construction plan has to satisfy are as follows: (1) The plan must be well-defined in the sense that it does not change low data and does not return impossible inputs. (2) The plan must be *secret-compatible*, meaning that created uncertainty traces are actually

compatible with secret h. (3) The plan must be *declassification-compatible*, meaning that created uncertainty traces are permitted to declassify the same values as the original trace.

Theorem 3.5.1 (Passive Attacker Security) *Given a program's IOD behavior T that satisfies a security policy $(\mathcal{S}, R)$. The IOD behavior T satisfies* GNIV *for the classification spec $\mathcal{S}$ and a view Λ, if there exists a well-defined plan ξ that is secret- and declassification-compatible.*

We next define the three conditions formally. **Well-Defined Plans.** Well-definedness of plans is straightforward. Since only inputs are modified by a plan, data classified by preconditions remains unchanged trivially. The condition $\exists t'.\ t' \cdot \mathsf{N}(x, \underline{r'}) \in R$ captures that the returned input must be an actual input of the action.

Definition 3.5.3 (Well-defined Plan) *A plan ξ is well-defined for a policy $(\mathcal{S}, R)$, if*

$$t \cdot \mathsf{N}(x, \underline{r'}) \in R \wedge r' = \xi(h, l, t, \mathsf{N}(x, \underline{r})) \\ \Rightarrow (\exists t'.\ t' \cdot \mathsf{N}(x, \underline{r'}) \in R) \wedge \mathsf{N}(x, \underline{r'})\downarrow_{\mathcal{S}}^{\mathsf{post}} = \mathsf{N}(x, \underline{r})\downarrow_{\mathcal{S}}^{\mathsf{post}}.$$

Secret-Compatible Plans. Every trace produced by a plan must be compatible with the secret h. Since the empty trace is always compatible with h, *i.e.* $\epsilon \#_\Lambda h$, we only require that appending the next modified action maintains compatibility.

Definition 3.5.4 (Secret-Compatible) *For a policy $(\mathcal{S}, R)$ and a view Λ, a plan ξ is secret-compatible if for every secret $h \in R\downarrow_\Lambda$, every trace t and action $\mathsf{N}(x, \underline{r})$ with $t \cdot \mathsf{N}(x, \underline{r}) \in R$, and every l, r',*

$$t \#_\Lambda h \wedge r' = \xi(h, l, t, \mathsf{N}(x, \underline{r})) \Rightarrow t \cdot \mathsf{N}(x, \underline{r'}) \#_\Lambda h$$

Declassification-Compatible Plans. GNIV requires that all permitted declassifications are not influenced by the view. Thus, if a declassification is permitted in the original trace, then the same declassification must be permitted in the constructed trace.

Formally, for every prefix t constructed by a plan, whenever a declassification $\mathsf{decl}(p, x)$ is the next action, then the declassification must be permitted after t, *i.e.* $t \cdot \mathsf{decl}(p, x) \in R$. To quantify over constructed prefixes, we define the image $\mathsf{Img}_{\mathcal{S},R}(\xi, h, l_p, l_c)$ of a plan ξ as the set of all prefixes that may be constructed by ξ for the secret h and the low data l_p and l_c of the prefix and continuation, respectively.

Definition 3.5.5 (Plan Image) *We define the image of a plan ξ inductively via*

$$\epsilon \in \mathsf{Img}_{\mathcal{S},R}(\xi, h, \epsilon, l_c)$$

$$t \in \mathsf{Img}_{\mathcal{S},R}(\xi, h, l_p, l_n \cdot l_c) \wedge t \cdot \mathsf{N}(x, \underline{r}) \in R \wedge \mathsf{N}(x, \underline{r})\downarrow_{\mathcal{S}} = l_n \\ \Rightarrow t \cdot \mathsf{N}(x, \underline{\xi(h, l_c, t, \mathsf{N}(x, \underline{r}))}) \in \mathsf{Img}_{\mathcal{S},R}(\xi, h, l_p \cdot l_n, l_c)$$

Definition 3.5.6 (Declassification-Compatible) *For a policy* $(\mathcal{S}, R)$ *and a view* Λ*, a plan* ξ *is declassification-compatible if for every secret* $h \in R{\downarrow_\Lambda}$ *and every low data* $l_p, \mathsf{decl}(p, \underline{x}), l_c$*,*

$$\forall t.\ t \in \mathsf{Img}_{\mathcal{S},R}(\xi, h, l_p, \mathsf{decl}(p, \underline{x}) \cdot l_c) \wedge (\exists x'.\ t \cdot \mathsf{decl}(p, x') \in R) \\ \Rightarrow t \cdot \mathsf{decl}(p, x) \in R.$$

Note that the condition $t \in \mathsf{Img}_{\mathcal{S},R}(\xi, h, l_p, \mathsf{decl}(p, \underline{x}) \cdot l_c)$ does not guarantee that a declassification may actually happen after the modified prefix, which we capture with $\exists x'.\ t \cdot \mathsf{decl}(p, x') \in R$.

Example 3.5.1 For our running example, we show GNIV for two views. Without additional assumptions, we show that the unused secret data from queries, namely the list of compatible vaccines, remains confidential. A suitable plan ξ_0 replaces (1) all queried vaccines with the secret according to the view and (2) all queried dates with the date that is encrypted in the next declassification. Otherwise, inputs remain unchanged. I.e., $\xi_0(h \cdot hs, l, t, \mathsf{query}(id, \underline{d}))$ returns $d[\mathsf{date} \mapsto \mathsf{Next}(id, \mathsf{I}), \mathsf{vac} \mapsto h']$, where $\mathsf{Next}(id, \mathsf{I})$ returns the date encrypted in the next declassification for id in l and h' is the relevant entry of h. We have to also replace the queried data to prove that subsequent declassifications are permitted.
The plan ξ_0 is trivially well-defined and secret-compatible. The plan ξ_0 is also declassification-compatible because, if a declassification may happen next, then there was a previous query action that was modified accordingly by ξ_0.
Given strong assumptions about encryption, we also show that parts of the queried dates remain confidential. The challenge is that after replacing a queried date, a subsequent declassification declassifies a different ciphertext. We resolve this issue by partitioning queried dates into confidential days and non-confidential milliseconds, where we assume that a plan can change milliseconds to obtain the desired ciphertexts. More formally, we assume that for every public key k, and dates d_0, d_1, we can change the milliseconds of d_1 such that the encryption of d_0 and the modified d_1 with k are the same. For a Dolev-Yao attacker, this assumption implies that an attacker does not know the private keys. Under this assumption, we prove that days of queried dates remain confidential. A suitable plan replaces the day of queried dates with the targeted secret and replaces the corresponding milliseconds such that subsequent declassification is correct.
Our incorrect variant of the IOD spec from the beginning of this section does not satisfy GNIV for either view. In particular, our defined plans are not declassification-compatible for the incorrect policy. If a declassification may happen next, then we are not guaranteed that there exists a previous query action whose encrypted date and key are being declassified.

3.5.2.1. Proof Sketch of Theorem 3.5.1

As for all other theorems shown in this chapter, we proved Theorem 3.5.1 in Isabelle/HOL. In this subsection, we give a sketch of our mechanized proof.

We prove Theorem 3.5.1 in two phases. In the first phase, we prove GNIV for IOD behaviors satisfying observational determinism, which is formalized in Lemma 3.5.2. In particular, we do not yet consider extensions. In the second phase, we lift this result to extensions, yielding Theorem 3.5.1.

Lemma 3.5.2 *Given an IOD behavior $\hat{T}$, an IOD spec $\hat{R}$, and a classification spec $\hat{\mathcal{S}}$ such that $\hat{R}$ is well-defined; and $\hat{T}$ is prefix-closed, input-closed, a subset of $\hat{R}$, and satisfies $\hat{\mathcal{S}}$-OD. $\hat{T}$ satisfies GNIV for a view $\hat{\Lambda}$ if there exists a well-defined trace construction plan $\hat{\xi}$ that is secret- and declassification-compatible.*

Proving Lemma 3.5.2. Let an IOD behavior $\hat{T}$, a policy $(\hat{\mathcal{S}}, \hat{R})$, and a trace construction plan $\hat{\xi}$ that satisfy all assumptions be arbitrary. Furthermore, let the secret $h \in \hat{T}{\downarrow_{\hat{\Lambda}}}$ and the low data $l \in \hat{T}{\downarrow_{\hat{\mathcal{S}}}}$ be arbitrary. We show that there exists a suitable uncertainty trace $t_u \in \hat{T}$ by induction on the prefixes l_p of the low data l. We use the following induction hypothesis $\mathsf{IH}(l_p)$:

$$\forall l_c.\ l = l_p \cdot l_c \\ \Longrightarrow \exists t_u' \in \hat{T}.\ t_u' \in \mathsf{Img}_{\hat{\mathcal{S}},\hat{R}}(\hat{\xi}, h, l_p, l_c) \wedge t_u'{\downarrow_{\hat{\mathcal{S}}}} = l_p \wedge t_u' \#_{\hat{\Lambda}} h \quad \mathsf{IH}(l_p)$$

As discussed in Sec. 3.5.2, we construct the uncertainty trace iteratively. Conceptually, t_u' captures the modified prefix and l_c captures the low data of the missing continuation. We have constructed the full required uncertainty trace t_u when the modified prefix has low data l, *i.e.* $\mathsf{IH}(l)$ holds.

The base case $\mathsf{IH}(\epsilon)$ holds trivially with $t_u' = \epsilon$. The inductive step consists of several smaller steps. The induction hypothesis entails that there exists $t_0 \in \hat{T}$ with $t_0 \in \mathsf{Img}_{\hat{\mathcal{S}},\hat{R}}(\hat{\xi}, h, l_p, w \cdot l_c)$, $t_0{\downarrow_{\hat{\mathcal{S}}}} = l_p$, and $t_0 \#_{\hat{\Lambda}} h$, where w is the low data that the next action must have. Since l is in $\hat{T}{\downarrow_{\hat{\mathcal{S}}}}$, there exists a trace $t_{\mathsf{origin}} \in \hat{T}$ with $t_{\mathsf{origin}}{\downarrow_{\hat{\mathcal{S}}}} = l = l_p \cdot w \cdot l_c$. $\hat{\mathcal{S}}$-OD entails that there exists a continuation e_0 of t_0 with $t_0 \cdot e_0 \in \hat{T}$ and $e_0{\downarrow_{\hat{\mathcal{S}}}^{\mathsf{pre}}} = \mathsf{pre}(w)$, where $\mathsf{pre}(w)$ is the data of w classified as low by the precondition.

If e_0 is a declassification, then $t_u' = t_0 \cdot e_0$ already satisfies the goal of the induction step since $e_0{\downarrow_{\hat{\mathcal{S}}}} = w$. More formally, we know $w = \mathsf{decl}(p, \underline{x})$ and $e_0 = \mathsf{decl}(p', x')$ for some p, p', x, x'. However, $e_0{\downarrow_{\hat{\mathcal{S}}}^{\mathsf{pre}}} = \mathsf{pre}(w)$ entails $p = p'$. Declassification-compatibility entails $t_0 \cdot \mathsf{decl}(p, x) \in \hat{R}$ and, thereby, well-definedness of $\hat{R}$ entails $x = x'$, concluding the case.

Otherwise, if $e = \mathsf{N}(x, \underline{r})$ is not a declassifying action, then because of that, there exists a possible output r' with $\mathsf{N}(x, \underline{r'}){\downarrow_{\hat{\mathcal{S}}}} = w$. Input-closedness entails that $t_0 \cdot \mathsf{N}(x, \underline{r'})$ is in $\hat{T}$. Lastly, we conclude the case with $t_u' = t_0 \cdot \mathsf{N}(x, \underline{\hat{\xi}(h, l_c, t_0, \mathsf{N}(x, \underline{r'}))})$. Well-definedness and secret-compatibility guarantee that t_u' has the same low data as $t_0 \cdot \mathsf{N}(x, \underline{r'})$ and stays compatible with h.

Recall that in our generalized framework, actions may have high visibility. For such actions, we do not have to show that whether an action occurs is low. In this case, the step where we get a continuation e_0 of t_0 using $\hat{\mathcal{S}}$-OD becomes more involved because there may be a sequence of high actions (actions with high visibility) between t_0 and e_0. In particular, to

conclude the induction, we require that the actions between t_0 and e_0 have also been modified by the plan $\hat{\xi}$. We deal with these high actions using another induction. We iteratively (1) apply the plan $\hat{\xi}$ to the next high action and then (2) use $\hat{\mathcal{S}}$-OD to acquire another continuation. These steps are done until all actions between t_0 and e_0 have been modified. To ensure this induction is well-founded, our generalized framework also requires a termination criteria for trace construction plans. A plan terminates if there exists a *measure*, a function from traces to natural numbers, that strictly decreases when modifying an action in a sequence of high actions. Termination holds trivially if the IOD spec does not include infinite sequences of high actions.

Proving Theorem 3.5.1. Let an IOD behavior T that satisfies the well-defined policy $(\mathcal{S}, R)$ be arbitrary. Furthermore, let ξ be a well-defined and secret- and declassification-compatible plan for the policy. We prove Theorem 3.5.1 by instantiating Lemma 3.5.2 in a specific way. Policy compliance entails that there exists an extension T' of T such that T' is prefix-closed and input-closed, and satisfies $(\mathcal{S} + \mathsf{NoDecl})$-OD. We instantiate the IOD behavior $\hat{T}$ and classification spec $\hat{\mathcal{S}}$ straightforwardly with T' and $(\mathcal{S} + \mathsf{NoDecl})$, respectively. For the IOD spec, we use the largest extension of R, namely $\hat{R} = \{t \mid \mathsf{actual}(t) \in R\}$. Importantly, every extension of a subset of R is also a subset of $\hat{R}$ and well-definedness of R trivially entails well-definedness of $\hat{R}$. Finally, we lift the plan ξ to extensions by defining that the lifted plan $\hat{\xi}$ (1) does not modify the added auxiliary actions, *i.e.* just returns the input of auxiliary actions, and (2) otherwise modifies actions in the same way as ξ. Accordingly, we use $\hat{\Lambda}(\hat{t}) = \Lambda(\mathsf{actual}(\hat{t}))$.

To use Lemma 3.5.2, we show that $\hat{\xi}$ is well-defined and secret- and declassification-compatible whenever ξ is well-defined and secret- and declassification-compatible, which holds straightforwardly. Lastly, to conclude the proof, we show that GNIV for the extension T' entails GNIV for the original IOD behavior T.

Let the secret $h \in T{\downarrow_\Lambda}$ and the low data $l \in T'{\downarrow_\mathcal{S}}$ be arbitrary. The definition of $\hat{T}$ entails $h \in T'{\downarrow_{\hat{\Lambda}}}$. Regarding the low data, we know that there exists an extended low data $l' \in T'{\downarrow_{(\mathcal{S}+\mathsf{NoDecl})}}$ with $\mathsf{actual}(l') = l$. GNIV for T' implies that there exists the uncertainty trace t'_u with $t'_u{\downarrow_{(\mathcal{S}+\mathsf{NoDecl})}} = l'$ and $t'_u \#_{\hat{\Lambda}} h$. Therefore, $\mathsf{actual}(t'_u)$ is a correct uncertainty trace for h and l since $\mathsf{actual}(t'_u){\downarrow_\mathcal{S}} = l$ and $\mathsf{actual}(t'_u) \#_\Lambda h$ hold.

3.5.3. Active Attacker

GNIV is strong enough to provide guarantees against active attackers that are also able to modify the low inputs of all actions. In this subsection, we define *Active-GNIV*, a variation of GNIV for active attackers, and show under which conditions GNIV entails Active-GNIV.

We parameterize Active-GNIV with the set of considered attackers. In our model, active attackers are able to change inputs of actions, *e.g.* by intercepting messages and modifying the payloads. We formalize an attacker as a function $A : \mathrm{Tr} \times Act \times \mathbb{U} \times \mathbb{U} \to \mathbb{U}$ that takes (1) the trace of past actions and (2) the next action $\mathsf{N}(x, \underline{r})$, and returns an attacker-chosen input r' that replaces r. To reason about attackers, we define the image of

an attacker $\mathsf{Attacks}(A, T)$ as the set of traces that are possible under the influence of an attacker A for an IOD behavior T.

Definition 3.5.7 (Attacker Image) *The image of an attacker* $A : \mathrm{Tr} \times Act \times \mathbb{U} \times \mathbb{U} \to \mathbb{U}$ *is the smallest set that satisfies:*

$$\epsilon \in \mathsf{Attacks}(A, T)$$

$$t \in \mathsf{Attacks}(A, T) \wedge t \cdot \mathsf{N}(x, \underline{r}) \in T \Rightarrow t \cdot \mathsf{N}(x, \underline{A(t, \mathsf{N}(x, \underline{r}))}) \in \mathsf{Attacks}(A, T)$$

Active-GNIV is then a variation of GNIV where for every considered attacker, the uncertainty traces must exist under the influence of the attacker. In particular, the uncertainty traces must exist regardless of whether the secret h is possible under the influence of the attacker or not.

Definition 3.5.8 (Active-GNIV) *An IOD behavior* T *satisfies Active-GNIV for a set of attackers* $\mathcal{A}$*, a classification spec* $\mathcal{S}$*, and a view* Λ*, if for every attacker* $A \in \mathcal{A}$*, secret* $h \in T{\downarrow_\Lambda}$*, and low data* $l \in \mathsf{Attacks}(A, T){\downarrow_\mathcal{S}}$*, there exists a trace* $t_u \in \mathsf{Attacks}(A, T)$*,*

$$t_u{\downarrow_\mathcal{S}} = l \wedge t_u \#_\Lambda h.$$

GNIV entails Active-GNIV if all considered attackers are *low-limited*. Intuitively, an attacker is low-limited if the attacker may only observe and modify low data. We capture this intuition formally by defining that low-limited attackers cannot distinguish traces with the same low data, *i.e.* if one trace is possible under the attacker, then every trace with the same low data is possible, too.

Definition 3.5.9 (Low-limited Attacker) *An attacker* $A : \mathrm{Tr} \times Act \times \mathbb{U} \times \mathbb{U} \to \mathbb{U}$ *is low-limited for a classification spec* $\mathcal{S}$*, if*

$$\forall T, t \in \mathsf{Attacks}(A, T), t' \in T.\ t{\downarrow_\mathcal{S}} = t'{\downarrow_\mathcal{S}} \Rightarrow t' \in \mathsf{Attacks}(A, T).$$

Corollary 3.5.3 *Given an IOD behavior* T *that satisfies GNIV for the classification spec* $\mathcal{S}$ *and a view* Λ*.* T *satisfies Active-GNIV for the attacker set* $\mathcal{A}$ *if every attacker* $A \in \mathcal{A}$ *is low-limited.*

3.5.4. Advanced Properties of GNIV

In this section, we discuss further techniques to reason about GNIV. More concretely, we discuss (1) guarantees about the parallel composition of IOD behaviors, (2) guarantees about refinements of IOD behaviors, and (3) guarantees against weaker attackers.

To be able to express these more advanced properties of GNIV, we first introduce a definition of GNIV that is parameterized in the compatibility relation $\#_\Lambda$.

Paramterized GNIV. Def. 3.5.10 shows the more generalized definition of

GNIV, which replaces the compatibility relation $\#_\Lambda$ with the relation $\mathcal{H}$ that is a parameter of the definition. This version of GNIV is more expressive than our standard definition of GNIV (Sec. 3.5.1), but also provides less guarantees about IOD behaviors. For instance, consider a view that returns some confidential bytestring. By defining that $(t_u, h) \in \mathcal{H}$ holds if the secret of the uncertainty trace $\Lambda(t_u)$ differs at most one byte from h, we permit IOD behaviors to release one byte of the secret. Similar to security policies themselves, we have to validate that the used $\mathcal{H}$ parameter does not permit more information release than intended, making this definition less suited for policy validation than our standard definition of GNIV.

Definition 3.5.10 (Parameterized GNIV) *An IOD behavior T satisfies $\mathsf{Gniv}(\mathcal{S}, \Lambda, \mathcal{H})$ for a classification spec $\mathcal{S}$, view Λ, and a relation $\mathcal{H}$, if for every secret $h \in T{\downarrow_\Lambda}$ and every low data $l \in T{\downarrow_\mathcal{S}}$, there exists an uncertainty trace $t_u \in T$,*

$$t_u{\downarrow_\mathcal{S}} = l \wedge (t_u, h) \in \mathcal{H}.$$

Parameterized GNIV is proved in the same way as standard GNIV. By adapting secret-compatibility to $\mathcal{H}$-compatibility, formalized below, we recover passive (and active) attacker security. $\mathcal{H}$-compatibility is defined analogously to secret-compatibility. Note that the second condition, namely that $(\epsilon, h) \in \mathcal{H}$ holds for all secrets, is satisfied by our definition of $\#_\Lambda$ (see Def. 3.5.1).

Theorem 3.5.4 *Given a program's IOD behavior T that satisfies a security policy $(\mathcal{S}, R)$. The IOD behavior T satisfies $\mathsf{Gniv}(\mathcal{S}, \Lambda, \mathcal{H})$ if there exists a well-defined plan ξ that is $\mathcal{H}$-compatible and declassification-compatible.*

Definition 3.5.11 ($\mathcal{H}$-Compatible) *For a policy $(\mathcal{S}, R)$, a view Λ, and a relation $\mathcal{H}$, a plan ξ is $\mathcal{H}$-compatible if*

- *for every secret $h \in R{\downarrow_\Lambda}$, every trace t and action $\mathsf{N}(x, \underline{r})$ with $t \cdot \mathsf{N}(x, \underline{r}) \in R$, and every l, r',*

$$(t, h) \in \mathcal{H} \wedge r' = \xi(h, l, t, \mathsf{N}(x, \underline{r})) \Rightarrow (t \cdot \mathsf{N}(x, \underline{r'}), h) \in \mathcal{H}$$

- *for every $h \in R{\downarrow_\Lambda}$, $(\epsilon, h) \in \mathcal{H}$ holds*

Composition of GNIV. As for policy compliance, we are able to compose the guarantees of separate IOD behaviors. More concretely, if the IOD behavior T_0 and T_1 ensure that the views Λ_0 and Λ_1 remain confidential, respectively, then the composition $(T_0 \parallel T_1)$ ensures that the composed view $\Lambda(t) = (\Lambda_0(t{\downarrow_\mathsf{L}}), \Lambda_1(t{\downarrow_\mathsf{R}}))$ remains confidential. The projections $t{\downarrow_\mathsf{L}}$ and $t{\downarrow_\mathsf{R}}$ remove all actions of T_0 and T_1, respectively.

Lemma 3.5.5 (GNIV Composition) *For every IOD behavior T_0 satisfying $\mathsf{Gniv}(\mathcal{S}_0, \Lambda_0, \mathcal{H}_0)$ and IOD behavior T_1 satisfying $\mathsf{Gniv}(\mathcal{S}_1, \Lambda_1, \mathcal{H}_1)$, the composition $(T_0 \parallel T_1)$ satisfies $\mathsf{Gniv}(\mathcal{S}_0 + \mathcal{S}_1, \Lambda', \mathcal{H}')$, where*

$$\Lambda'(t) = (\Lambda_0(t{\downarrow_\mathsf{L}}), \Lambda_1(t{\downarrow_\mathsf{R}}))$$
$$\mathcal{H}' = \{(t, (h_0, h_1)) \mid (t{\downarrow_\mathsf{L}}, h_0) \in \mathcal{H}_0 \wedge (t{\downarrow_\mathsf{R}}, h_1) \in \mathcal{H}_1\}$$

Consider Example 3.5.1, where the policy ensures that the queried dates remain confidential if private keys are confidential. Lemma 3.5.5 gives us that if applications run with clients that keep their private key confidential, then the composed system keeps the queried data *and* the private keys confidential.

Importantly, the composition of GNIV does not require that the composed IOD behaviors satisfy a policy. Since every IOD behavior keeps the trivial view $\Lambda(t) = \epsilon$ confidential (Lemma 3.5.6), we get that, if an IOD behavior keeps a view Λ confidential, then every composition also keeps the view Λ confidential.

Lemma 3.5.6 *An IOD behavior T satisfies* $\mathsf{Gniv}(\mathcal{S}, \lambda t.\ \epsilon, T \times \{\epsilon\})$.

Refinement. For real-world programs, to accurately capture I/O behavior, actions have to specify exactly which bytestrings are exchanged with other entities. However, reasoning at this level can make the specification of policies and validating properties about policies arduous.

Instead, we may specify and validate policies at a more abstract level and then refine guarantees provided by the policy to the concrete level. E.g., consider that, instead of bytestrings, we model transferred payloads as abstract data types. We then connect the abstract data types to concrete bytestrings by introducing a concretization function f, mapping abstract actions to concrete actions. In particular, for an abstract IOD behavior T, $T{\downarrow_f}$ captures the concrete IOD behavior of the program. The steps of our approach are then as follows: (1) We specify a policy $(\mathcal{S}, R)$ at the abstract level. (2) We verify that the abstract IOD behavior T of a program satisfies the policy. (3) We get that if $(\mathcal{S}, R)$ entails GNIV for a view Λ, thereby, T satisfies GNIV for Λ, then $T{\downarrow_f}$ satisfies GNIV for a refined view Λ'.

We do not show that the concrete IOD behavior $T{\downarrow_f}$ satisfies some refined policy $(\mathcal{S}', R')$. Such a property requires much stronger restrictions on the concretization function f, reducing the level of abstraction we are able to use.

Sec. 3.6.4 discusses how we use our approach to verify an implementation of the WireGuard protocol. In particular, we discuss how to verify for concrete code that its abstract IOD behavior satisfies a policy.

Lemma 3.5.7 states under which conditions and for which refined classification specs $\mathcal{S}'$ and refined views Λ', step (3) is sound. Regarding classification specs, the classification spec for concrete actions may classify less data as low. Regarding views, we require that for every concrete secret $h' \in T{\downarrow_f}{\downarrow_{\Lambda'}}$, there exists an abstract secret $h \in T{\downarrow_\Lambda}$ that is compatible with an abstract trace t iff h' is compatible with the corresponding concrete trace $t{\downarrow_f}$. The lemma's requirements hold, for instance, if there exists a homomorphism between abstract and concrete actions, which we discuss in Sec. 3.6.4.5.

Lemma 3.5.7 (GNIV Refinement) *Given a function $f : Act \times V \times V \to Act' \times V' \times V'$ from actions to actions (with different names and argument types). For every abstract IOD behavior T that satisfies* $\mathsf{Gniv}(\mathcal{S}, \Lambda, \mathcal{H})$*, the concrete IOD behavior $T{\downarrow_f}$ satisfies* $\mathsf{Gniv}(\mathcal{S}', \Lambda', \mathcal{H}')$ *if*

- *for all $t_0, t_1 \in T$, $t_0{\downarrow_\mathcal{S}} = t_1{\downarrow_\mathcal{S}}$ implies $t_0{\downarrow_f}{\downarrow_{\mathcal{S}'}} = t_1{\downarrow_f}{\downarrow_{\mathcal{S}'}}$, and*

- *for every* $t \in T$ *and* $h' \in T\downarrow_f\downarrow_{\Lambda'}$, *there exists* $h \in T\downarrow_\Lambda$ *with* $(t, h) \in \mathcal{H} \leftrightarrow (t\downarrow_f, h') \in \mathcal{H}'$.

Instead of a concretization function f, we may use an abstraction function g mapping concrete actions to abstract actions. Lemma 3.5.8 captures the corresponding refinement. Whether we use an abstraction or concretization function depends on the relation between abstract and concrete values. A concretization function requires that every abstract value has at most one concrete representation, and vice versa for abstraction functions.

Lemma 3.5.8 *Given a function* $g : Act' \times V' \times V' \to Act \times V \times V$ *from actions to actions (with different names and argument types). For every abstract IOD behavior* $T'\downarrow_g$ *that satisfies* $\mathsf{Gniv}(\mathcal{S}, \Lambda, \mathcal{H})$, *the concrete IOD behavior* T' *satisfies* $\mathsf{Gniv}(\mathcal{S}', \Lambda', \mathcal{H}')$ *if*

- *for all* $t_0, t_1 \in T'\downarrow_g$, $t_0\downarrow_g\downarrow_{\mathcal{S}} = t_1\downarrow_g\downarrow_{\mathcal{S}}$ *implies* $t_0\downarrow_{\mathcal{S}'} = t_1\downarrow_{\mathcal{S}'}$, *and*
- *for every* $t' \in T'$ *and* $h' \in T'\downarrow_{\Lambda'}$, *there exists* $h \in T'\downarrow_g\downarrow_\Lambda$ *with* $(t'\downarrow_g, h) \in \mathcal{H} \leftrightarrow (t', h') \in \mathcal{H}'$.

Weakening of Classification Specs. As discussed in our threat model (Sec. 3.2.3), we assume that attackers may observe at most the data that is classified as low by the classification spec. Simultaneously, we do not expect that attackers are able to observe everything that is classified low. For instance, attackers cannot observe declassifications, which do not correspond to actual program operations. To bridge this gap, we show that a view also remains secret against attackers that observe less data. Since we define the observational capabilities of attackers with classification specs, we only have to show that GNIV is preserved for classification specs classifying less data as low, which we refer to as *dominated* classification specs.

Formally, a classification spec $\mathcal{S}$ is dominated by another spec $\mathcal{S}'$, if whenever $\mathcal{S}'$ is satisfied, then also $\mathcal{S}$ is satisfied.

Definition 3.5.12 (Classification Spec Domination) *A classification spec* $\mathcal{S}$ *is dominated by a classification spec* $\mathcal{S}'$, *denoted as* $\mathcal{S} \leq \mathcal{S}'$, *if for all* t_0, t_1

$$t_0\downarrow_{\mathcal{S}'} = t_1\downarrow_{\mathcal{S}'} \implies t_0\downarrow_{\mathcal{S}} = t_1\downarrow_{\mathcal{S}}$$

Lemma 3.5.9 (GNIV Weakening) *If* $\mathcal{S}$ *is dominated by* $\mathcal{S}'$, *then every IOD behavior* T *that satisfies* $\mathsf{Gniv}(\mathcal{S}', \Lambda, \mathcal{H})$ *also satisfies* $\mathsf{Gniv}(\mathcal{S}, \Lambda, \mathcal{H})$.

An analogous lemma for policy compliance does not hold since $\mathcal{S}$-SIF requires the postconditions of actions to show that data is low. E.g., policy compliance does usually not hold if declassifications do not actually give us the assumption that declassified data is low.

3.6. Case Study

To show that our policy framework is powerful and applicable to real-world programs, we verified an implementation of the WireGuard pro-

tocol against an appropriate security policy defined by us (Sec. 3.6.2). Before discussing the case study, we first list our trust assumptions (Sec. 3.6.1). Lastly, we discuss several smaller programs that illustrate how we express specification patterns from previous works (Sec. 3.6.3). All verified programs are available here [103].

3.6.1. Trust Assumptions

We have fully formalized and proved in Isabelle/HOL an instantiation of our policy framework that uses the SecCSL logic for code verification. To benefit from more automation, we verified the programs discussed in this section using Gobra as shown in Sec. 3.3. When using Gobra, we make two assumptions: (1) We assume that our annotations for trusted libraries that specify I/O behavior (Sec. 3.3.1) are satisfied and that the resulting I/O behavior is input-closed. (2) We assume that Gobra is sound, *i.e.* if Gobra reports a successful verification, then the verified code actually satisfies the provided specifications.

For a security policy to be meaningful, we additionally assume that attackers satisfy our threat model (Sec. 3.2.3).

3.6.2. The WireGuard VPN

WireGuard is a widely-used Virtual Private Network (VPN). In the protocol, two agents first establish a secret session key in a handshake phase and then use this key to exchange messages in a transport phase. For our case study, we reuse results from Arquint et al. [56]. They verify that a modified version of WireGuard's official Go implementation [68] refines an I/O spec (without declassifications) generated from a Tamarin [69] model of the protocol.

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

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

[69]: Meier et al. (2013), *The TAMARIN Prover for the Symbolic Analysis of Security Protocols*

For a security policy, we defined an IOD spec by extending the I/O spec of Arquint et al. with declassification actions. Furthermore, we defined a classification spec from scratch. We then verified that the implementation of the initiator role from Arquint et al. satisfies this policy. We were able to fully reuse the refinement proof by Arquint et al. To verify the code, we added only additional proof annotations to verify that our added declassification actions are permitted, and to verify secure information flow.

The IOD spec of our security extends Arquint et al.'s I/O spec by declassifications. However, their I/O spec is expressed using separation logic [55, 89] rather than a transition system. In this formalism, the permitted I/O behavior is expressed via a co-inductive separation logic predicate that is parameterized by the current position in and the abstract state of the protocol; these two parameters correspond to the transition system state in IOD transition systems. The body of the predicate consists of a number of cases (conjuncts) that each describe (1) the condition under which an I/O action may take place, (2) the action and its arguments (expressed as separation logic predicates), and (3) the effect of the action on the protocol position and abstract state. These ingredients correspond directly to the guard and update functions of our IOD transition systems. In fact, Sprenger et al. [55] formally proved the equivalence between I/O

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

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

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

specifications in separation logic and I/O-guarded transition systems, and Arquint et al.'s I/O spec is indeed generated from such transition systems. Due to this equivalence, and since our framework does not prescribe how IOD specs are expressed (Sec. 3.2.2), we were able to re-use Arquint et al.'s I/O spec. Since this spec is expressed in separation logic, it is compatible with Gobra's verification technique. Code verification proceeds analogously to the approach shown in Sec. 3.3.

Security Policy. For the classification spec, we classify that long-term private keys, generated ephemeral keys, and user messages encrypted during the transport phase are secret. All other inputs and outputs, such as network messages, public keys, and timestamps are low data. Furthermore, we classify that the size of private keys is low, too. To reason about keys, setting up the long-term keys and generating ephemeral keys are captured as I/O actions.

For the IOD spec, to specify which declassifications are permitted, we introduce the sets *Out* and *In* that, for the current point in the protocol, contain the messages that may be sent and are being processed, respectively. Our IOD spec definition derives these sets from the transition system state (resp. the state parameters of the I/O spec expressed in separation logic). We then permit three groups of declassifications: (1) Every message m that may be sent $m \in Out$ may also be declassified. (2) Similarly, for every encryption $\mathsf{enc}(m, k)$ occurring in *Out* and decryption $\mathsf{dec}(m, k)$ occurring *In*, we may declassify whether the encryption or decryption fails. (3) We may declassify whether WireGuard's well-definedness condition holds, namely whether the responder's public key to the power of the initiator's private key or ephemeral key is zero (where the keys are also derived from the transition system state).

Our verification approach is expressive enough to verify the implementation against the security policy. All non-deterministic effects in the program, in particular, concurrency and error handling of network sockets, are handled by our auxiliary actions.

Regarding guarantees for the policy, Arquint et al. proved that the I/O spec satisfies key agreement and forward secrecy, which are preserved by our IOD spec. These guarantees entail that if the protocol is in the transport phase according to the transition system state, then indeed a successful handshake between the actors has been established. This allows the IOD spec to express that declassifications are permitted only in the transport phase by expressing a corresponding precondition for declassification actions in terms of the transition system state, such that they are guaranteed to occur after a successful handshake.

Code Changes. We have taken the implementation from Arquint et al. as is, inheriting their changes to the official Go implementation. The official Go implementation was changed in two ways: (1) To reduce verification effort, DDos protection, load balancing, and metrics were omitted. In particular, load balancing requires complex concurrency reasoning not supported by Gobra. (2) -Cryptographic operations and network operations were moved into trusted libraries. The individual steps processing a connection, *i.e.* parsing and constructing messages, have remained unchanged.

#	Program	LOC	LOS	LOP	T [s]
1	vaccinations	91	15	150	42
2	vaccinations (quantitative)	91	18	165	75
3	database [80]	116	33	190	76
4	embargoed information [104]	13	20	46	36

Figure 3.10.: Program used to illustrate expressiveness. We list the number of lines of Go code (LOC), security policy (LOS), proof annotations (LOP), and the average verification time in seconds.

Statistics. The initiator consists of 345 lines of code (LOC) that we have verified. The security policy consists of 18 LOC for the classification spec and 219 LOC for the IOD spec. The specification mechanism for the IOD spec used by Arquint et al. is more verbose than the one shown in Sec. 3.3. Out of the 219 lines, 147 lines are generated from the verified Tamarin protocol model and only 27 lines contain relevant information for declassification. To verify that the code satisfies the policy, 714 lines of proof annotations were necessary, 123 of which were added for this work. The lines added for this work are either low assertions, annotations to use the shared invariant, or annotations to prove that the declassification conditions are satisfied. The annotation overhead of proof annotations per line of code is around 2, which is typical for SMT-based deductive verifiers. Verification takes 15 minutes on a Lenovo T480s with an Intel Core i7-8650U and 24 GB of RAM. Compared to Arquint et al., the verification time has increased by 13 minutes. This increase is due to the added secure information flow reasoning.

3.6.3. Expressiveness

To show that our approach supports common specification patterns of security policies, we specified and verified several smaller programs. Fig. 3.10 depicts statistics about these programs. For robust declassification [87] and secret data over public channels, we took policies from the literature [80, 104] and wrote code implementations in Go (Program #4 and #3 respectively). For declassification with quantitative criteria, we extended our running example (Program #2). We also list our running example (Program #1). We focus our discussion on how we express specification patterns.

[87]: Zdancewic et al. (2001), *Robust Declassification*

[80]: Banerjee et al. (2008), *Expressive Declassification Policies and Modular Static Enforcement*

[104]: Askarov et al. (2010), *A Semantic Framework for Declassification and Endorsement*

State-Dependent Declassification. In this pattern, declassifications are permitted based on the state of an execution. We use this pattern in our WireGuard case study, where, for instance, we permit declassifications involving user inputs only after a successful handshake. When using IOD-guarded transition systems, we express state-dependent declassification straightforwardly by capturing in the transition system state all relevant information, *e.g.* the protocol phase. The declassification's guard may then permit declassifications based on the captured state.

Quantitative Criteria. As an extension of state-dependent declassification, declassification with quantitative criteria permits declassification based not only on the actual execution state but also based on past declassifications. Since we treat declassification as actions themselves, we express this specification pattern analogously to state-dependent declassification. As an illustrating example, we extended our running example such that declassifications are permitted at most 10 times per id (Program #2 of Fig. 3.10). For this change, we extended the state of our IOD-guarded transition system to also store the number of past

declassifications per id, which is then increased in the update of the declassification action.

Robust Declassification. In this pattern, declassification is permitted only for values with high integrity, *i.e.* trusted data [87]. The aim is to ensure that attackers are not able to influence when and what is declassified in unintended ways. In our framework, we reason about integrity by defining an additional classification spec, specifying which data has high integrity. We then specify in the precondition for declassification that the declassified data has high integrity. For endorsement, *i.e.* the act of elevating the integrity of data, we add an action endorse analogously to declassification, which is then also governed by the IOD spec. We verify the code for both classification specs. Because integrity is the dual of confidentiality [105], we are able to use our low assertions to also express high integrity. In particular, we do not have to adapt our formalism. As an example, we verified a program by Askarov and Myers [104] where data may be declassified if a received timestamp is older than a specified embargo time (Program #4 in Fig. 3.10). The received timestamp is endorsed only if it is in the past.

[87]: Zdancewic et al. (2001), *Robust Declassification*

[105]: Biba (1977), *Integrity Considerations for Secure Computer Systems*

[104]: Askarov et al. (2010), *A Semantic Framework for Declassification and Endorsement*

Secret Data from Public Inputs. In our running example, secret data originates from a remote database. However, for many applications, secret data arrives encrypted over the public network. In our framework, we express encrypted sources of secret data by classifying the decryption key as confidential. Classifying the key is sufficient, as all data derived using the confidential key is considered confidential itself. In particular, we are guaranteed that programs do not unintentionally release decrypted payloads. As an example, we verified a program inspired by Banerjee at al. (Program #3). A medical database receives encrypted medical records, parts of which are declassified and forwarded to an auditing company.

3.6.4. Verification with Abstract Messages

In our WireGuard case study, we use the refinement approach introduced in Sec. 3.5.4. Instead of bytestrings, the policy uses abstract messages to represent arguments of actions. This representation enabled Arquint et al. to use Tamarin [69, 106], a state-of-the-art tool to analyze protocols, to prove key agreement and forward secrecy for the IOD spec. However, using abstract messages instead of the concrete bytestrings that are actually manipulated by the code implementation poses two challenges: (1) For code verification, we have to bridge the abstraction gap between the abstract messages in the policy and the bytestrings manipulated by the code. (2) For policy validation, we want to derive guarantees about the concrete IOD behavior based on the guarantees validated for the policy (which uses abstract messages).

[69]: Meier et al. (2013), *The TAMARIN Prover for the Symbolic Analysis of Security Protocols*

[106]: Schmidt et al. (2012), *Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties*

In this subsection, we explain how we relate bytestrings to abstract messages (Sec. 3.6.4.1) and we show how we verify that the abstract IOD behavior of code manipulating concrete bytestrings satisfies a policy (Sec. 3.6.4.2). Lastly, we discuss how to derive guarantees about the concrete IOD behavior based on the policy. The discussed techniques for code verification have originally been published in [56].

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

Our proposed method enables us to verify that pre-existing real-world code satisfies IOD specs produced from abstract Tamarin models.

3.6.4.1. Relating Abstract Messages and Bytestrings

We relate bytestrings and abstract messages using a concretization function $\gamma\colon \mathcal{M} \to \mathbb{B}^\star$ from abstract messages to bytestrings, which we also lift to actions. E.g., an action $\mathsf{N}(x, \underline{r})$ with abstract arguments is concretized to $\gamma(\mathsf{N}(x, \underline{r})) = \mathsf{N}(\gamma(x), \underline{\gamma(r)})$. We do not use an abstraction function from bytestrings to abstract messages because such a mapping assumes that each bytestring corresponds to exactly one term, and consequently, that *every* bytestring can be uniquely parsed as a term. To minimize our assumptions, we do not a priori want to exclude collisions between bytestrings, *i.e.* a bytestring may have several term interpretations.

As for Tamarin, abstract messages $m \in \mathcal{M}$ are elements of a *term algebra* $\mathcal{T} = \mathcal{T}_\Sigma(\mathcal{C} \cup \mathcal{V})$. Message terms are built over a signature Σ of function symbols and a set of constants $\mathcal{C}$ and variables $\mathcal{V}$. The arguments of actions are ground terms, *i.e.* terms without variables. The term algebra is equipped with an *equational theory* E, which is a set of equations, and we denote by $=_\mathsf{E}$ the equality modulo E.

We model the concrete messages and the operations on them as *bytestring algebras* defined as term algebras $\mathcal{B}$ with the set of bytestrings $\mathbb{B}^\star$ as the carrier set. To relate terms to bytestrings, we use a surjective term algebra homomorphism $\gamma\colon \mathcal{M} \to \mathcal{B}$, which maps constants $\mathcal{C}$ to bytestrings and the signature's symbols to functions on bytestrings:

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

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

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

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

on bytestrings. In what follows, we will use the bytestring algebra's functions in our cryptographic library's specification. This enables us to reason about message parsing and construction.

3.6.4.2. Verifying Abstract Policies

The code verification for policies with abstract messages follows the same approach as shown in Sec. 3.3. The only difference is that the memory location Trace is appended with abstract actions instead of the concrete actions. Fig. 3.11 shows the altered specification of the `Receive` method. We add ghost parameters (here `m`) to capture the abstract arguments of actions. The received concrete bytestring is then the concretization of the abstract argument. Recall that `Abs(b)` is the bytestring stored in the

```
ens Abs(ciph) = enc^B(Abs(key), Abs(msg))
func Encrypt(key, msg []byte) (ciph []byte)

ens ok ⟹ Abs(c) = enc^B(Abs(k), Abs(m))
func Decrypt(k, c []byte) (m []byte, ok bool)

req acc(Trace) && acc(b)
ens acc(Trace) && *Trace = old(*Trace) · recv(m)
ens Abs(b) = γ(m)
func Receive() (b []byte, ghost m term)
```

Figure 3.11.: Simplified specifications for encryption, decryption, and receive. The function `Abs` abstracts an in-memory byte array to $\mathcal{B}$. We omit conditions on the size of bytestrings.

array `b`. The specification of policies and the assertions that we verify to show that a policy is satisfied remain unchanged otherwise. However, our introduction of abstraction makes reasoning about the arguments of actions, especially inputs, more challenging. We first discuss how to show that the IOD spec is satisfied. We discuss how to show that the classification spec is satisfied afterward.

Reasoning about Outputs. Verifying that the IOD spec permits an action such as sending or declassifying an abstract message boils down to verifying that the bytestring $\gamma(m)$ for a permitted message m was constructed and then sent or declassified, respectively. Showing such constructions becomes straightforward by equipping a trusted cryptographic library with suitable specifications. Consider the simplified specification of an encryption function shown in Fig. 3.11. Due to the specification and the surjectivity of γ, the result of `encrypt(key, msg)` is equal to $\gamma(\mathsf{enc}(m_{key}, m_{msg}))$ for some messages m_{key} and m_{msg}, where $\gamma(m_{key})$ = `Abs(key)` and $\gamma(m_{msg})$ = `Abs(msg)`. To verify the construction of an entire message, we combine the information of all such calls.

Reasoning about Inputs. For action inputs, we do not have to show that the IOD spec permits them. However, the IOD spec may require us to show that an abstract input of an action matches a specific term. Consider a policy that permits declassification only after an encrypted challenge $\mathsf{enc}(k, n)$ has been received. To verify that a declassification is permitted, we have to verify that the term $\mathsf{enc}(k, n)$ has been received. We refer to such terms as *patterns*.

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

Definition 3.6.1 *The* pattern requirement *for a pattern $t \in \mathcal{T}$ is defined by*

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

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

```
1 // Abs(key) = γ(k) holds
2 ciph, c := Receive()
3 assert Abs(ciph) = γ(c)
4 msg, ok := Decrypt(key, ciph); if !ok {return}
5 assert ∃u. Abs(msg) = γ(u)
6         && Abs(ciph) = γ(enc(k, u))
7 PaR1(m, ...) // using the pattern requirement
8 assert ∃w. c =E enc(k, w) && Abs(msg) = γ(w)
```

Figure 3.12.: Reasoning about receiving and parsing a ciphertext.

```
req acc(Trace, read) && Reaches(p,*Trace,s) && s.key = k
req ∃x. γ(enc(k,x)) = γ(m)
ens acc(Trace, read) && ∃x'. m =E enc(k,x')
ghost func PaR1(m,s,k)
```

Figure 3.13.: Ghost function for the pattern requirement of Example 3.6.1. There is the single pattern $\mathsf{enc}(k, x)$, where k is a constant and x is a variable.

We need the pattern requirement for all patterns of the IOD spec. For code verification, we express the pattern requirement as a ghost method whose pre- and postcondition are the left-hand and right-hand side of the pattern requirement, for each pattern respectively. To apply the pattern requirement, the corresponding ghost method is called in the code. In Sec. 3.6.4.3 and Sec. 3.6.4.4, we will explain how to prove the pattern requirement for a given pattern t.

Example 3.6.1 Consider a simple protocol where a policy expects a message matching the pattern $\mathsf{enc}(k, x)$, where k is a pre-shared key. We assume that key k is stored in the model state of our IOD-guarded transition system. Fig. 3.12 shows part of an implementation. The variable `key` stores the pre-shared key, expressed as `Abs(key)` $= \gamma$`(k)`. After successfully receiving a bytestring `ciph` with a message `c`, `Abs(ciph)` $= \gamma$`(c)` holds due to `receive`'s specification (Fig. 3.11). Next, the code decrypts `ciph`. If successful, `ciph` equals the bytestring γ`(enc(k,u))` for some message `u` with `Abs(msg)` $= \gamma$`(u)` (lines 5–6) by `decrypt`'s postcondition (Fig. 3.11) and γ being a surjective homomorphism. Furthermore, we know that γ`(enc(k,u))` equals γ`(c)`, but not yet that the received message `c` matches the pattern $\mathsf{enc}(k, x)$ (line 8). For this, we apply the pattern requirement by calling the ghost method `PaR1` (Fig. 3.13). The constant k of the pattern $\mathsf{enc}(k, x)$ is passed as an argument to the call, and related to the state of the IOD spec via the ghost method's precondition (with `s.key = k`).

Verifying Classification Spec. Verification of the classification spec is straightforward. For preconditions, to show that an abstract message m is low, we show that all abstract messages involved in the construction of m are low. The pattern requirement helps us to derive the abstract terms of bytestrings involved in the message creation. For postconditions, we trivially get that the concretizations of low inputs are low since if an abstract term m is low, then also $\gamma(m)$ is low.

3.6.4.3. Deriving the Pattern Requirement

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

The first property is *image disjointness*, which has two parts: (1) the images of constants under γ are pairwise disjoint and (2) the image of any function $f^{\mathcal{B}}$ for $f \in \Sigma_{\mathsf{up}}$ neither collides with the image of any other function $g^{\mathcal{B}}$, for $g \in \Sigma$, nor with the image of constants under γ. We use Σ_{up} to denote the set of all *constructive functions*, *i.e.* functions that do not reduce a term such as decryptions or field projections. Image disjointness for destructive functions such as decryptions is not necessary. Due to equational theory, collisions of different functions are ok if there exist equal terms (mod E) that do not collide. This case is useful for equational theories with neutral elements, *e.g.* $g^1 = g$, which would otherwise cause trivial collisions. Lemma 3.6.1 states the case without equational theory.

Definition 3.6.2 (Image disjointness) *Image disjointness holds for a pattern t if, (1) γ is injective for all names $\mathcal{C}$ occurring in t and (2) for all $f \in \Sigma_{\mathsf{up}}$ occurring in t,*

$$f(t'_1, ..., t'_k) \sqsubseteq t \wedge \gamma(f(t'_1\sigma, ..., t'_k\sigma)) = \gamma(m) \\ \Rightarrow \exists b_1, ..., b_k.\ m =_{\mathsf{E}} f^{\mathcal{B}}(b_1, ..., b_k). \quad (\mathrm{ID}_f(t))$$

Lemma 3.6.1 *Image disjointness holds if:*

- *γ is injective on the set of names $\mathcal{C}$ and*
- *for all $f, g \in \Sigma_{\mathsf{up}}$ such that $f \neq g$,*

$$img(f^{\mathcal{B}}) \cap (img(g^{\mathcal{B}}) \cup \gamma(\mathcal{C})) = \emptyset.$$

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

Definition 3.6.3 (Pattern injectivity) *Pattern injectivity holds for a pattern t if, for all $f \in \Sigma_{\mathsf{up}}$ occurring in t,*

$$f(t'_1, ..., t'_k) \sqsubseteq t \wedge f^{\mathcal{B}}(\gamma(t'_1\sigma), ..., \gamma(t'_k\sigma)) = f^{\mathcal{B}}(b_1, ..., b_k) \\ \Longrightarrow \exists\sigma'.\ b_1 = \gamma(t'_1\sigma') \wedge ... \wedge b_k = \gamma(t'_k\sigma'). \quad (\mathrm{PaI}_f(t))$$

Proposition 3.6.2 *Given a linear pattern t (where every variable occurs only once), image disjointness and pattern injectivity for t imply the pattern requirement for t.*

Proof. We prove the proposition's statement for all $t' \sqsubseteq t$ by induction on t'. □

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

3.6.4.4. Assumptions and Proof Obligations

We discuss assumptions and proof obligations regarding image disjointness and pattern injectivity. In doing so, we distinguish cryptographic operations from *formats* [107]. Formats are user-defined function symbols, along with projections for all arguments, that behave like tuples. In the concretization, each format is mapped to a combination of tags (*i.e.* constant bytestrings), fixed-size fields, and variable-sized fields prepended with a length field.

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

Since we are working in a symbolic (Dolev-Yao) model, which assumes perfect cryptography, we maintain this assumption for cryptographic operations at the bytestring level in the following form.

Assumption 3.6.1 (Cryptographic operations) *We assume that*

- γ *is injective on the set of names* $\mathcal{C}$,
- $(\mathrm{ID}_f(t))$ *holds for cryptographic* $f \in \Sigma_{\mathsf{up}}$,
- $(\mathrm{PaI}_f(t))$ *holds for all protocol patterns* t *and all cryptographic* $f \in \Sigma_{\mathsf{up}}$ *occurring in* t.

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

The situation is different for formats. We can expect that the formats of a well-designed protocol are unambiguously parseable (*i.e.* injective and hence pattern-injective) and mutually disjoint (*i.e.* image disjoint). We therefore require that these properties are *proved* for formats, *e.g.* using the techniques proposed in [107, 108].

[107]: Mödersheim et al. (2014), *A Sound Abstraction of the Parsing Problem*
[108]: Ramananandro et al. (2019), *EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats*

Remark 3.6.1 An obvious way to achieve image disjointness and pattern injectiveness is to *tag* each construct of the bytestring algebra with a different bytestring. This approach is followed, *e.g.* in [55] but it is unrealistic for real protocols.
Alternatively, image disjointness holds if different operations result in differently sized bytestrings. For operations with varying output sizes, such as stream encryption, this may require restricting the allowed argument sizes in the implementation. In some cases, this approach may allow us to *prove* image disjointness even for some cryptographic operations. Indeed, we do this for our pre-existing implementation of the WireGuard protocol, which does not use tagging. However, this approach also has its limitations; for example, AES-256 or SHA-256 have the same output size.

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

3.6.4.5. Guarantees about the Concrete IOD Behavior

As discussed in Sec. 3.5.4, we get guarantees about the concrete IOD behavior (based on bytestrings) by refining the guarantees of the abstract policy. Our link between bytestrings and message terms enables us to use a more specialized version of Lemma 3.5.7. Lemma 3.6.3 states that if there exist concretization functions for classified data (denoted as γ_L) and secrets (denoted as γ_H) that extend the homomorphism to classification

specs and views, then for views kept confidential at the abstract level, the corresponding concrete view is kept confidential at the concrete level.

Lemma 3.6.3 *For every IOD behavior T that satisfies* $\mathsf{Gniv}(\mathcal{S}, \Lambda, \mathcal{H})$*, the IOD behavior* $T\downarrow_\gamma$ *satisfies* $\mathsf{Gniv}(\mathcal{S}^{\mathcal{B}}, \Lambda^{\mathcal{B}}, \mathcal{H}^{\mathcal{B}})$ *if for every* $t \in T$ *and* $h \in T\downarrow_\Lambda$*,*

- $\gamma_L(t\downarrow_{\mathcal{S}}) = t\downarrow_\gamma\downarrow_{\mathcal{S}^{\mathcal{B}}}$
- $\gamma_H(\Lambda(t)) = \Lambda^{\mathcal{B}}(t\downarrow_\gamma)$
- $(t, h) \in \mathcal{H} \leftrightarrow (t\downarrow_\gamma, \gamma_H(h)) \in \mathcal{H}^{\mathcal{B}}$.

Consider the abstract spec $\{\mathsf{Low}(id)\}\mathsf{query}(id, \underline{r})\{\mathsf{Low}(\mathsf{key}(r))\}$, where $\mathsf{key}(r)$ returns the key of the now abstract term r. To obtain a suitable concrete spec, γ_L may exchange all abstract functions with their concrete counterparts, *i.e.* $\{\mathsf{Low}(id)\}\mathsf{query}(id, \underline{r})\{\mathsf{Low}(\mathsf{key}^{\mathcal{B}}(r))\}$. Views are concretized analogously.

3.7. Related work

We categorize existing work about secure information flow policies based on 2 criteria.

[109]: Sabelfeld et al. (2003), *Language-based information-flow security*

[24]: Mantel (2003), *A uniform framework for the formal specification and verification of information flow security*

Firstly, we distinguish between programming-language-based frameworks [109], where security is defined with a fixed programming language, and system-based framework [24], where security is defined at a higher level, usually some form of general traces. Language-based approaches have the advantage that they are closer to code verification and amicable to powerful definitions of security, in particular, non-determinism is taken care of based on the fixed language semantics. However, this more concrete formalization comes at the cost that reasoning about policies is tied to the fixed semantics, changing languages requires adapting the policy framework, and programs written in different languages cannot be formally governed by the same policy.

[110]: Sabelfeld et al. (2005), *Dimensions and Principles of Declassification*

Secondly, as introduced by Sabelfeld and Sands [110], we use 4 dimensions to classify which kind of information release is governed: (1) **What** information is declassified. (2) **When** may information be declassified, in particular, which property has to hold so that declassification is permitted. (3) **Where** in a program may declassifications happen. (4) **Who** may declassify information. Our policy framework tackles the what and when dimensions. Our support for robust declassification, as outlined in Sec. 3.6.3, is sometimes considered to address part of the who dimension.

We focus our discussion of related work on approaches that involve reasoning about code.

[111]: Askarov et al. (2007), *Gradual Release: Unifying Declassification, Encryption and Key Release Policies*

Language-based definitions of security. A lot of language-based frameworks define security based on the epistemic definition introduced by Askarov and Sabelfeld [111], which has been extended in various ways [112–115]. A program is secure if the *attacker uncertainty*, *i.e.* the set of secrets compatible with low data, remains unchanged for every execution step of the program, except for declassification. To rule out that declassifications cause arbitrary information release, some frameworks [79, 80,

116] require further that the reduction of attacker uncertainty caused per declassification is bound based on the program and policy. Our introduction of GNIV follows the same motivation. Instead of enforcing that the reduction of attacker uncertainty is bound, GNIV requires that the attacker uncertainty remains unchanged for a given view of the secret data. Robust declassification [87] combines secure information flow with reasoning about the integrity of data to limit the information release even against an active attacker. In these approaches, declassification is permitted in a trusted context only, limiting the influence an active attacker can have.

[87]: Zdancewic et al. (2001), *Robust Declassification*

Another approach for language-based frameworks is to define security via a variation of low-bisimulations [92, 117]. The definition considers pairs of executions with equal low data. A program is secure if, for every step of one execution, the other execution is able to perform a step that again establishes equal low data. For declassifications, pairs that do not agree on the declassified value are disregarded [81, 86, 118].

[92]: Smith et al. (1998), *Secure Information Flow in a Multi-Threaded Imperative Language*
[117]: Mantel et al. (2011), *Assumptions and Guarantees for Compositional Noninterference*
[81]: Smith (2022), *Declassification Predicates for Controlled Information Release*
[86]: Sabelfeld et al. (2003), *A Model for Delimited Information Release*
[118]: Askarov et al. (2007), *Localized delimited release: combining the what and where dimensions of information release*

An advantage of these epistemic- and simulation-based definitions is that they consider timing-channels. Furthermore, epistemic definitions provide immediate guarantees against attackers. However, these definitions require a fixed language semantics, making them ill-suited for our purposes.

System-based definitions of security. For system-based frameworks, security is defined on top of some notion of event traces, describing the behavior of a system. Early work without declassification [24] classifies events as either secret or public and then defines security as an attacker's inability to deduce when secret events happen. These approaches consider whether an event occurs as confidential. Later works change this perspective to the notion that, instead of events themselves, the data produced by events is confidential. Observational determinism [82, 83] describes that the observable public behavior is deterministic in the low inputs of the system, thereby, ruling out that public behavior depends on secret data. Sutherland's non-deducibility [48] describes that for every public observation and every secret, there exists a trace incorporating both, which prevents possibilistic deductions about secrets based on public observations. Our definition of GNIV slightly adapts Sutherland's non-deducibility to consider different notions of whether a trace incorporates a secret.

[24]: Mantel (2003), *A uniform framework for the formal specification and verification of information flow security*

[82]: McLean (1992), *Proving Noninterference and Functional Correctness Using Traces*
[83]: Roscoe (1995), *CSP and determinism in security modelling*
[48]: Sutherland (1986), *A model of information*

To allow intended information release, a line of work [119, 120] combines LTL reasoning with secure-information flow reasoning. These approaches are able to express until when secrets are supposed to remain secret, where information release is permitted afterward. Bounded-deducibility [26] implements a similar idea, but without LTL reasoning. Policies specify under which condition declassification is *not* permitted and how much of the secrets must be protected.

[119]: Dimitrova et al. (2012), *Model Checking Information Flow in Reactive Systems*
[120]: Clarkson et al. (2014), *Temporal Logics for Hyperproperties*
[26]: Popescu et al. (2021), *Bounded-Deducibility Security (Invited Paper)*

Instead of policies specifying classification and declassification, an alternative approach is to express security policies as an abstract model that must be refined to satisfy the policy [121, 122]. Implementations are allowed to release as much information as the abstract model, in particular, permitting specific declassifications is not necessary. While some definitions of security such as observational determinism are preserved under standard refinements [21], security definitions involving

[121]: Cohen et al. (2009), *Abstraction in model checking multi-agent systems*
[122]: Baumann et al. (2021), *On Compositional Information Flow Aware Refinement*

intentional information release are often not [21, 122]. Existing works therefore restrict refinements such that implementations do not introduce more information releases than the abstraction. For instance, Baumann et al. [122] introduce a compositional refinement ensuring that the attacker uncertainty of the refined program and abstract model are equal.

[21]: Clarkson et al. (2008), *Hyperproperties*
[122]: Baumann et al. (2021), *On Compositional Information Flow Aware Refinement*

Language-based frameworks. Closest to our work, but not language-agnostic, Murray et al. [79] specify declassification policies as a condition on a trace of values and a relational assertion, specifying when and what may be declassified, respectively. To populate the trace, programs are also annotated with specifications capturing how this trace is extended. In contrast to our approach, where traces record IOD actions, programs can add arbitrary values to their traces using program annotations. As a consequence, their declassification policies are more flexible than ours, but policies provide weaker guarantees by themselves without further knowledge about the program annotations. For classification, trusted libraries are annotated directly with pre- and postconditions containing low assertions. While we do not consider this work to be language-agnostic due to the fixed programming language, the introduction of a trace to govern declassification and the flexibility of annotating trusted libraries make the approach more easily applicable to other languages than previous works.

[79]: Murray et al. (2023), *Assume but Verify: Deductive Verification of Leaked Information in Concurrent Applications*

Previous language-based frameworks specify policies similarly. Banerjee et al. [80] permit declassification based on a condition on the global program state. Schoepe et al. [78] and Smith [81] specify a predicate, defining whether a concrete declassification statement is permitted depending on the current and initial program state respectively. Since these approaches permit declassification for specific statements in a program, they also address the where dimension of declassification.

[80]: Banerjee et al. (2008), *Expressive Declassification Policies and Modular Static Enforcement*
[78]: Schoepe et al. (2020), *VERONICA: Expressive and Precise Concurrent Information Flow Security*
[81]: Smith (2022), *Declassification Predicates for Controlled Information Release*

Programs are verified using a relational verification logic [78, 79, 81], a type system [104, 118, 123], or a combination thereof [80].

[104]: Askarov et al. (2010), *A Semantic Framework for Declassification and Endorsement*
[118]: Askarov et al. (2007), *Localized delimited release: combining the what and where dimensions of information release*
[123]: Barthe et al. (2008), *Tractable Enforcement of Declassification Policies*

Lastly, a line of work [124–126] focuses on the where and what dimensions instead of the when dimension. This line of work defines policies by assigning *flow lock specs* to data. A flow lock spec is a set of logical locks that have to be opened to release data over a specific channel. These logical locks are opened through static annotations in the code, capturing the position of relevant places in the code. The work by Menz et al. [126] extends this approach to a higher-order language.

[124]: Broberg et al. (2006), *Flow Locks: Towards a Core Calculus for Dynamic Flow Policies*
[125]: Broberg et al. (2009), *Flow-sensitive semantics for dynamic information flow policies*
[126]: Menz et al. (2023), *Compositional Security Definitions for Higher-Order Where Declassification*

System-based frameworks. To our knowledge, there is no existing work that lifts policy frameworks similar to the ones discussed for language-based frameworks, *i.e.* frameworks with explicit policies that tackle the verification of existing code, to the system-based setting. Instead of verifying existing code, another approach is to verify code generated from a more abstract model. In the approach proposed by Popescu et al. [26], programs are specified as I/O automata producing I/O actions. For verification, the automata is first checked against a policy and then automatically translated into a functional programming language. Importantly, the translation maintains the guarantees provided by the policy. For classification, policies specify public observations and confidential data of the automata's transitions. Declassification is governed by bounded-deducibility as discussed before. Their approach

[26]: Popescu et al. (2021), *Bounded-Deducibility Security (Invited Paper)*

is used to verify a conference management system [127] and a social media platform [128, 129].

3.8. Conclusion

We have introduced a novel policy framework, where policies are specified and validated at the level of I/O behavior. This abstraction enables us to specify security policies independent of programs and programming languages, and to provide guarantees for all programs satisfying a security policy based on the policy alone. To validate policies, we introduce GNIV, entailing for passive and certain active attackers, that a selection of data remains confidential even in the presence of declassification. For code verification, we verify that programs satisfy our policies using a combination of standard code verification techniques. Our approach is powerful, compatible with different verification techniques, and applicable to real-world code.

We see multiple possible directions for future work. One direction is to automate proving guarantees provided by policies. In our framework, we prove such guarantees manually in Isabelle/HOL. Another direction is to extend our framework to other versions of secure information flow such as probabilistic non-interference. We expect that the main challenge is, when instantiating the framework for code verification, how to handle the technical assumptions that verification techniques require to satisfy probabilistic non-interference.

Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA

4.

"Everything is vague to a degree you do not realize till you have tried to make it precise."

— Bertrand Russel

Standard separation logic enables the modular verification of heap-manipulating sequential [28, 130] and data-race free concurrent programs [131, 132]. More recently, numerous separation logics have been proposed that enable the verification of fine-grained concurrency by incorporating ideas from concurrent separation logic, Owicki-Gries [29], and rely-guarantee [30]. Examples include CAP [31], iCAP [32], CaReSL [33], CoLoSL [34], FCSL [35], GPS [37], RSL [38], and TaDA [39] (see Brookes et al. [40] for an overview). These logics are very expressive, but challenging to apply because they often comprise many complex proof rules. E.g., our running example (Fig. 4.1) consists of two statements, but requires over 20 rule applications in TaDA, many of which have non-trivial instantiations and subtle side conditions. This complexity seems inevitable for challenging verification problems involving, *e.g.* fine-grained concurrency or weak memory.

The complexity of advanced separation logics makes it difficult to develop proofs in these logics. It is, thus, crucial to have tools that check the validity of proofs and automate parts of the proof search. One way to provide this tool support is through *proof checkers*, which take as input a nearly complete proof and check its validity. They typically embed program logics into the higher-order logic of an interactive theorem prover such as Coq. Proof checkers exist, *e.g.* for RSL [38] and FCSL [35]. Alternatively, *automated verifiers* take as input a program with specifications and devise the proof automatically. They typically combine existing reasoning engines such as SMT solvers with logic-specific proof search algorithms. Examples are Smallfoot [133] and Grasshopper [134] for traditional separation logics, and Caper [41] for fine-grained concurrency.

Proof checkers and automated verifiers strike different trade-offs in the design space. Proof checkers are typically very expressive, enabling the verification of complex programs and properties, and produce foundational proofs: ultimately based on a language semantics, with a minimal trusted core. However, existing proof checkers offer little automation. Automated verifiers, on the other hand, significantly reduce the proof effort, but compromise on expressiveness and require substantial development effort, especially, to devise custom proof search algorithms (which increase the trusted core).

It is in principle possible to increase the automation of proof checkers by developing proof tactics, or to increase the expressiveness of automated verifiers by developing stronger custom proof search algorithms. However, such developments are too costly for the vast majority of program logics, which serve mostly a scientific or educational purpose. As a result, adequate tool support is very rare, which makes it difficult for developers of such logics, lecturers and students, as well as engineers to apply, and gain experience with, such logics.

To remedy the situation, several tools took inspiration from the idea of *proof outlines* [135, 136] (see, *e.g.* Pierce et al. [137] for a detailed discussion): formal proof skeletons that contain the key proof steps, but omit most of

[28]: Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*
[130]: O'Hearn et al. (2001), *Local Reasoning about Programs that Alter Data Structures*
[131]: O'Hearn (2004), *Resources, Concurrency and Local Reasoning*
[132]: Brookes (2004), *A Semantics for Concurrent Separation Logic*
[29]: Owicki et al. (1976), *An Axiomatic Proof Technique for Parallel Programs I*
[30]: Jones (1983), *Specification and Design of (Parallel) Programs*
[31]: Dinsdale-Young et al. (2010), *Concurrent Abstract Predicates*
[32]: Svendsen et al. (2014), *Impredicative Concurrent Abstract Predicates*
[33]: Turon et al. (2013), *Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency*
[34]: Raad et al. (2015), *CoLoSL: Concurrent Local Subjective Logic*
[35]: Sergey et al. (2015), *Mechanized verification of fine-grained concurrent programs*
[37]: Turon et al. (2014), *GPS: navigating weak memory with ghosts, protocols, and separation*
[38]: Vafeiadis et al. (2013), *Relaxed separation logic: a program logic for C11 concurrency*
[39]: Rocha Pinto et al. (2014), *TaDA: A Logic for Time and Data Abstraction*
[40]: Brookes et al. (2016), *Concurrent separation logic*
[133]: Berdine et al. (2005), *Smallfoot: Modular Automatic Assertion Checking with Separation Logic*
[134]: Piskac et al. (2014), *GRASShopper - Complete Heap Verification with Mixed Specifications*
[41]: Dinsdale-Young et al. (2017), *Caper - Automatic Verification for Fine-Grained Concurrency*

[135]: Owicki (1975), *Axiomatic Proof Techniques for Parallel Programs*
[136]: Apt et al. (2009), *Verification of Sequential and Concurrent Programs*
[137]: Pierce et al. (2018), *Programming Language Foundations*

the details. Proof outlines are a standard notation to present program proofs in publications and teaching material. *Proof outline checkers* such as Starling [138] and VeriFast [11] take as input a proof outline and then check automatically that it represents a valid proof in the program logic. They provide automation for proof steps for which good proof search algorithms exist, and can support expressive logics by requiring annotations for complex proof steps. Due to this flexibility, proof outline checkers are especially useful for experimenting with a logic.

[138]: Windsor et al. (2017), *Starling: Lightweight Concurrency Verification with Views*

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

In this chapter, we present Voila, a proof outline checker for TaDA [39], which goes beyond existing proof outline checkers and automated verifiers by supporting a substantially more complex program logic, which handles fine-grained concurrency, linearizability, abstract atomicity, and other advanced features. We believe that our systematic development of Voila generalizes to other complex logics. Our contributions are as follows:

- The Voila *proof outline language*, which supports a large subset of TaDA and enables users to write proof outlines very similar to those used by the TaDA authors [39, 139].
- A systematic approach to automate the expansion of a proof outline into a full *proof candidate* via a normal form and heuristics. Our approach automates most proof steps (*e.g.* 20 out of 22 for the running example from Fig. 4.1).
- An encoding of the proof candidate into Viper [5], which checks its validity without requiring any TaDA-specific proof search algorithms.
- The Voila proof outline checker, the *first* tool that supports specification for linearization points, provides a high degree of automation, and achieves good performance. Our submission artifact [140] contains the executable Voila tool; the Voila source code is also available [141].

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

[140]: Wolf et al. (2021), *Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA*

[141]: Wolf et al. (n.d.), *The Voila source repository*

This chapter is based on a previous publication about Voila [53, 54].

[53]: Wolf et al. (2021), *Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA*

[54]: Wolf et al. (2022), *Concise outlines for a complex logic: a proof outline checker for TaDA*

Outline. Sec. 4.1 gives an overview of the TaDA logic and illustrates our approach. Sec. 4.2 presents the Voila proof outline language, and Sec. 4.3 summarizes how we verify proof outlines. We explain how we automatically expand a proof outline into a proof candidate in Sec. 4.4 and how we encode a proof candidate into Viper in Sec. 4.5. Sec. 4.6 provides a detailed soundness argument. In Sec. 4.9, we evaluate our technique by verifying several challenging examples. We discuss related work in Sec. 4.11 and conclude in Sec. 4.12.

Our technical report [142] contains a substantial appendix with many further details, including: the full version and Viper encoding of our running example, with TaDA levels (omitted from this chapter, but supported by Voila) and nested regions; additional inference heuristics; and the general Viper encoding scheme.

[142]: Wolf et al. (2020), *Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA (Full Paper)*

4.1. Running Example and TaDA Overview

Fig. 4.1 shows the first half of our running example, adapted from the original TaDA publication [39]: the TaDA proof outline of the `lock`

[39]: Rocha Pinto et al. (2014), *TaDA: A Logic for Time and Data Abstraction*

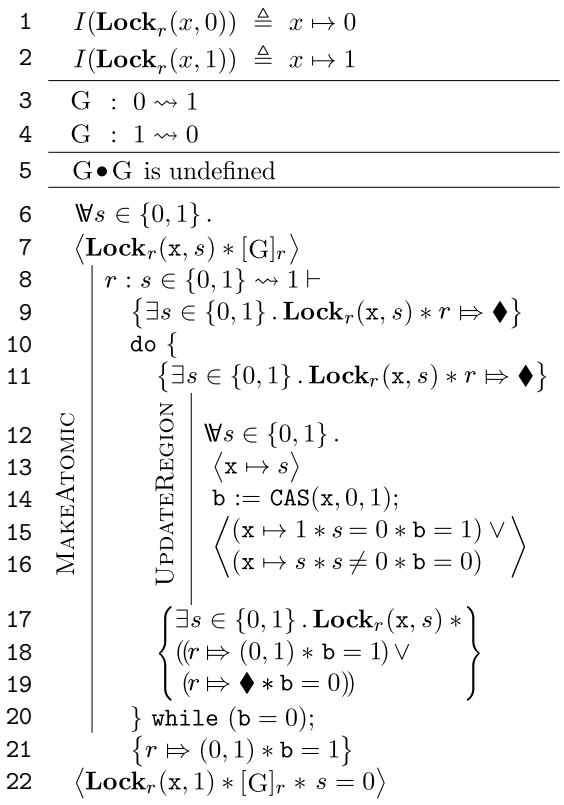

Figure 4.1.: First half of our running example: a spinlock with atomic TaDA specifications and shared region `Lock`; adapted with only minor changes from TaDA [39]. The lock region (Lines 1–2) comprises a single memory location, whose value is either 0 (available) or 1 (acquired). Guard G allows locking and unlocking (Lines 3–4), and is unique (Line 5). The proof outline (Lines 6–22) shows the implementation of a CAS-based `lock` operation with atomic specifications. Levels (denoted by λ in TaDA) are omitted from the discussion in this chapter, but supported by Voila and included in the technical report [142].

procedure of a spinlock, whose atomic specifications capture its essence as a primitive for mutual exclusion. In Sec. 4.1.4, we then discuss a non-atomic specification derived from `lock` that conceptually ties a lock to an invariant. As in the original publication [39], the outline in Fig. 4.1 shows only two out of 22 proof steps, and omits most side conditions. In a TaDA proof outline, a proof step corresponds to the application of a TaDA rule, including suitable pre- and postconditions. Our outline shows applications of the rules MakeAtomic and UpdateRegion. Deriving the shown pre- and postconditions may require additional rule applications, which are omitted.

[39]: Rocha Pinto et al. (2014), *TaDA: A Logic for Time and Data Abstraction*

We use our running example to introduce the necessary TaDA background, explain TaDA proof outlines, and illustrate the corresponding Voila proof outlines.

4.1.1. Regions and Atomicity

TaDA targets shared-memory concurrency with sequentially-consistent memory, and TaDA programs manipulate *shared regions*: data structures that are concurrently modified according to a specified *protocol* (as in rely-guarantee reasoning [30]). A shared region such as $\mathtt{Lock}_r(\mathtt{x}, s)$ (subscript r uniquely identifies a specific region instance) is an abstraction over the region's content, analogous to abstract predicates [143] in traditional separation logic. The interpretation $I(\mathtt{Lock}_r(\mathtt{x}, s))$ defines the region's content. In our example (Lines 1–2), the lock owns memory location x (denoted by separation logic's points-to predicate $x \mapsto \_$), and its *abstract state* s is 0 or 1, indicating whether it is unlocked or locked. Here, the

[30]: Jones (1983), *Specification and Design of (Parallel) Programs*

[143]: Parkinson et al. (2005), *Separation logic and abstraction*

abstract state (second region parameter) and the content of the memory location (value pointed to by x) coincide, but they may differ in general.

Unlike traditional abstract predicates, shared regions are *duplicable*, *i.e.* the equivalence $\texttt{Lock}_r(\texttt{x}, s) \Leftrightarrow \texttt{Lock}_r(\texttt{x}, s) * \texttt{Lock}_r(\texttt{x}, s)$ holds. This allows multiple threads to obtain an instance of a $\texttt{Lock}_r$ region, and to compete for the corresponding lock. However, note that duplicating a shared region indirectly also allows duplicating points-to predicates, which are unique in traditional separation logic and elsewhere in TaDA. This is nevertheless sound because TaDA's intricate proof rules ensure that a shared region is opened only for an abstractly-atomic duration, and that no two instances of the same region are opened simultaneously.

Lines 3–5 define the protocol for modifications of a lock as a labeled transition system. The labels are *guards* – abstract resources that restrict when a transition may be taken. Here, guard G allows both locking and unlocking (Lines 3-4), and is unique (Line 5). Using a unique guard in a context where multiple threads compete for acquiring a lock may seem counterintuitive, but the combination of unique guards and duplicable shared region assertions resolves this perceived conflict, as discussed in Sec. 4.1.4. Note that the transition system is defined relative to a region's abstract value, not its internal memory values, which is not directly apparent in this example, since abstract and concrete values coincide.

Lines 6–22 contain the proof outline for the `lock` procedure, which updates a lock `x` from an undetermined state – it can seesaw between locked and unlocked due to environment interference – to the locked state. Importantly, this update appears to be atomic to clients of the spinlock. These properties are expressed by the *atomic TaDA triple* (Lines 6, 7, and 22)

$$\mathbb{\forall} s \in \{0,1\} \cdot \langle \texttt{Lock}_r(\texttt{x}, s) * [\mathrm{G}]_r \rangle\ \texttt{lock(x)}\ \langle \texttt{Lock}_r(\texttt{x}, \texttt{1}) * [\mathrm{G}]_r * s = 0 \rangle$$

[144]: Herlihy et al. (1990), *Linearizability: A Correctness Condition for Concurrent Objects*

Atomic triples (angle brackets) express that their statement is linearizable [144]. The abstract state of shared regions occurring in pre- and postconditions of atomic triples is interpreted *relative to the linearization point*, *i.e.* the moment in time when the update becomes visible to other threads (here, when the CAS operation on Line 14 succeeds). In contrast, pre- and postconditions of standard Hoare triples (curly braces) are interpreted as usual: relative to the start and the end of the specified statement's execution. Intuitively, it is the combination of linearizability, shared regions with abstract state, and guarded transition systems that establishes TaDA's *abstract atomicity*: operations (*e.g.* `lock`) appear atomic when interacted with on the level of a shared region (*e.g.* $\texttt{Lock}_r(\texttt{x}, s)$), and TaDA's derivation rules ensure that the abstraction holds, even if the underlying memory is manipulated non-atomically.

The *interference context* $\mathbb{\forall} s \in \{0,1\}$ is a special binding for the abstract region state that forces callers of `lock` to guarantee that the environment keeps the lock state in the set $\{0,1\}$ until the linearization point is reached. Correspondingly, it also requires the callees to not take the region out of this abstract state (*i.e.* $\{0,1\}$) before its linearization point is reached. In this case, both restrictions are vacuous; in general, the interference context can be understood as a symmetric rely-guarantee condition.

$$\textsc{MakeAtomic}\quad \frac{r \notin \mathcal{A} \qquad \{(x,y) \mid x \in X, y \in Y\} \subseteq \mathcal{T}_{\mathrm{R}}(\mathrm{G})^* \qquad r : x \in X \rightsquigarrow Y, \mathcal{A} \vdash \left\{\exists x \in X.\, \mathbf{R}_r(\vec{z}, x) * r \Rightarrow \blacklozenge\right\} \mathbb{C} \left\{\exists x \in X, y \in Y.\, r \Rightarrow (x,y)\right\}}{\mathcal{A} \vdash \mathbb{\forall} x \in X. \left\langle \mathbf{R}_r(\vec{z}, x) * [\mathrm{G}]_r \right\rangle \mathbb{C} \left\langle \ \exists y \in Y.\, \mathbf{R}_r(\vec{z}, y) * [\mathrm{G}]_r \right\rangle}$$

$$\textsc{UpdateRegion}\quad \frac{\mathcal{A} \vdash \mathbb{\forall} x \in X. \left\langle I(\mathbf{R}_r(\vec{z}, x)) * P(x) \right\rangle \mathbb{C} \left\langle \exists y \in Y, w \in W. \begin{array}{l} I(\mathbf{R}_r(\vec{z}, y)) * Q_1(x,y,w) \\ \vee\ I(\mathbf{R}_r(\vec{z}\ x)) * Q_2(x,w) \end{array} \right\rangle}{\begin{array}{l} r : x \in X \rightsquigarrow Y, \mathcal{A} \vdash \\ \qquad \mathbb{\forall} x \in X. \left\langle \mathbf{R}_r(\vec{z}, x) * P(x) * r \Rightarrow \blacklozenge \right\rangle \\ \qquad\qquad \mathbb{C} \\ \qquad \left\langle \exists y \in Y, w \in W. \begin{array}{l} \mathbf{R}_r(\vec{z}, y) * r \Rightarrow (x,y) * Q_1(x,y,w) \\ \vee\ \mathbf{R}_r(\vec{z}, x) * r \Rightarrow \blacklozenge \qquad * Q_2(x,w) \end{array} \right\rangle \end{array}}$$

$$\textsc{UseAtomic}\quad \frac{r \notin \mathcal{A} \qquad \{(x,y) \mid x \in X, y \in Y\} \subseteq \mathcal{T}_{\mathrm{R}}(\mathrm{G})^* \qquad \mathcal{A} \vdash \mathbb{\forall} x \in X. \left\langle I(\mathbf{R}_r(\vec{z}, x)) * P(x) * [\mathrm{G}]_r \right\rangle \mathbb{C} \left\langle \exists y \in Y.\, I(\mathbf{R}_r(\vec{z}, y)) * Q(x,y) \right\rangle}{\mathcal{A} \vdash \mathbb{\forall} x \in X. \left\langle \mathbf{R}_r(\vec{z}, x) * P(x) * [\mathrm{G}]_r \right\rangle \mathbb{C} \left\langle \exists y \in Y.\, \mathbf{R}_r(\vec{z}, y) * Q(x,y) \right\rangle}$$

Figure 4.2.: Simplified versions of two key TaDA rules used in Fig. 4.1. MakeAtomic establishes an atomic triple (conclusion) for a linearizable block of code (premise), which includes checking that a state update complies with the region's transition system: $\mathcal{T}_R(G)^*$ is the reflexive, transitive closure of the transitions that G allows. $\mathbf{R}_r(\vec{z}, x)$ and $I(\mathbf{R}_r(\vec{z}, x))$ are the shared region and its content, respectively. UpdateRegion identifies a linearization point, for instance, a CAS statement. If successful, the diamond tracking-resource $r \Rightarrow \blacklozenge$ is exchanged for the witness tracking-resource $r \Rightarrow (x, y)$ to record the performed state update; otherwise, the diamond resource is kept, such that the operation can be attempted again. $P(x)$, $Q_1(x, y, w)$, and $Q_2(x, w)$ are some TaDA assertions. UseAtomic is a special combination of MakeAtomic and UpdateRegion, where the linearizable statements itself is the linearization point. Again, $P(x)$ and $Q(x, y)$ are some TaDA assertions.

The precondition of the triple states that an instance of guard G for region r, $[\mathrm{G}]_r$, is required to execute `lock(x)`. The postcondition expresses that, *at the linearization point*, the lock's abstract state was changed from unlocked ($s = 0$) to locked ($\mathtt{Lock}_r(\mathtt{x}, 1)$). Such precise specifications of state updates are enabled by the atomic triple's interpretation relative to the linearization point. In contrast, standard Hoare triples would have to account for potential environment interference *before* and *after* the linearization point – and can thus often only specify preservation of data structure invariants. In this chapter, we refer to standard Hoare triples also as *non-atomic triples*.

4.1.2. TaDA Proof Outline

Lines 6–22 of the proof outline in Fig. 4.1 show the main proof steps; Fig. 4.2 shows simplified versions of the applied key TaDA rules. The inner rule application, UpdateRegion, identifies the linearization point inside an abstractly-atomic code block. The surrounding rule application, MakeAtomic, then checks that there is exactly one such linearization point, ensuring that the block of code is indeed atomic w.r.t. a shared region abstraction, and establishes an atomic triple. This justifies the change from non-atomic premise triple (Lines 9 and 21) to an atomic conclusion triple (Lines 7 and 22), around the body of the CAS-based implementation of the `lock` procedure.

Rule MakeAtomic requires that the *atomicity context* of the premise triple, a set $\mathcal{A}$ of *pending updates* (for brevity, omitted from the previously-shown triple for `lock(x)`), includes any region updates performed by the

statement of the triple. By tracking pending updates and allowing at most one per region ($r \notin \mathcal{A}$ in Fig. 4.2), MakeAtomic intuitively prevents more than one observable change to the same shared region from happening during an abstractly-atomic operation.

In the proof outline, this requirement is reflected on Line 8, which shows the intended update of the lock's state: $r : s \in \mathit{trans} * \{0, 1\}1$ (following TaDA publications, we omitted the tail of the atomicity context from the outline). MakeAtomic checks that the update is allowed by the region's transition system with the available guards (the rule's second premise in Fig. 4.2), but following the original TaDA publication, the check is omitted from the proof outline. Then MakeAtomic temporarily exchanges the corresponding guard $[\mathrm{G}]_r$ for the *diamond tracking-resource* $r \Longmapsto \blacklozenge$ (Line 9), which serves as evidence that the intended update was not yet performed.

Inside the loop, an application of UpdateRegion identifies the CAS (Line 14) as the linearization point. The rule requires the diamond resource in its precondition (Line 11), modifies the shared region (Lines 12–16), and case-splits in its postcondition: if the update failed (Line 19) then the diamond is kept for the next attempt; otherwise (Line 18), the diamond is exchanged for the *witness tracking-resource* $r \Longmapsto (0, 1)$, which indicates that the region was updated from abstract state 0 to 1. Intuitively, the witness resource guarantees that there is exactly one linearization point where the relevant state update happened. This guarantee enables MakeAtomic to establish atomic triples from non-atomic triples. Furthermore, the witness resource is needed to carry sufficient information from the linearization point (which may not be the last statement in the procedure) to the point at which the operation's postcondition is to be established: the latter is interpreted w.r.t. the linearization point, but other threads may have changed the shared region since then. Finally, at the end of MakeAtomic (Lines 21–22), the witness resource is consumed and the desired abstractly-atomic postcondition is established, stating that the shared region was updated from 0 to 1 at the linearization point.

Note that the proof outline also illustrates how to convert between atomic and non-atomic triples in TaDA. The MakeAtomic rule is the only rule that can establish atomic triples, justified by the single linearization point. Conversely, an atomic triple can always be converted to a non-atomic triple by weakening its postcondition to account for the environment's interference. In Fig. 4.1, this happens around UpdateRegion. The need for weakening postconditions is discussed in more detail in Sec. 4.4, in the context of stable assertions.

4.1.3. Voila Proof Outline

Fig. 4.3 shows the *complete* proof outline of our example discussed so far, in the Voila proof outline language, which closely resembles the TaDA outline from Fig. 4.1. In particular, the `region` declaration defines a region's interpretation, abstract state, and transition system, just like the initial declarations in Fig. 4.1. The subsequent proof outline for procedure `lock` annotates the same two rule applications as the TaDA outline and a very similar loop invariant. The Voila proof outline verifies automatically via an encoding into Viper, but the outline is expressed completely in

```
1  struct cell { int val; }
2
3  region Lock(id r, cell x)
4    interpretation { x.val |-> ?v && (v == 0 || v == 1) }
5    state { v }
6    guards { unique G; }
7    actions { G: 0 ~> 1; G: 1 ~> 0; }
8
9  abstract_atomic procedure lock(id r, cell x)
10   interference ?s in Set(0, 1);
11   requires Lock(r, x, s) && G@r;
12   ensures  Lock(r, x, 1) && G@r && s == 0;
13 {
14   bool b;
15   make_atomic using Lock(r, x) with G@r {
16     do
17       invariant Lock(r, x);
18       invariant !b ==> r |=> <D>;
19       invariant  b ==> r |=> (0, 1);
20     {
21       update_region using Lock(r, x) {
22         b := CAS(x, 0, 1);
23       }
24     } while (!b);
25   }
26 }
```

Figure 4.3.: The Voila proof outline of our example, strongly resembling the TaDA proof outline from Fig. 4.1. `id` is the type of region identifiers; primitive types are passed by value, structs by reference. Logical variables are introduced using a question mark; *e.g.* `x.val`$\mapsto$ `?v` binds the logical variable `v` to the value of the location `x.val`. Operator && denotes separating conjunction.

terms of TaDA concepts; it does not expose any details of the underlying verification infrastructure. The successful verification shows that our tool automatically infers the additional 20 rule applications, and all omitted side conditions, thereby closing the gap between the user-provided proof outline and a corresponding full-fledged proof.

4.1.4. Locks with Resource Invariants

This subsection completes our running example by showing a TaDA outline (Fig. 4.4) and corresponding Voila code (Fig. 4.5) for a specification of `lock` that ties the spinlock to a resource invariant the lock protects. The resources in this invariant are then temporarily transferred to any threads that acquires the lock.

Following the TaDA publication, we introduce a new shared region called `CAPLock` (Lines 1–4 of Fig. 4.4), which wraps an instance of the previously declared `Lock` region, and two new guards (Lines 5–6): a vacuous *empty* guard **0** that is always available and used to acquire a lock, and a unique guard U for releasing it. Note that **0** and U both guard transitions of the abstract state of the `CAPLock` region. Transitioning the underlying `Lock` region (*i.e.* actually acquiring and releasing the lock) still requires the previously-introduced guard G. Consequently, an unlocked `CAPLock` region (Lines 1–2) contains guards G and U, and resource invariant *Inv* (left abstract for brevity). In contrast, when locked (Lines 3–4), both resources U and *Inv* are owned by the lock holder, and the shared region only contains guard G.

The body of the proof outline (Lines 8–15) uses the previously established atomic TaDA triple for procedure `lock` to derive the following, non-atomic TaDA triple:

$$\mathcal{A} \vdash \{\exists v \in \{0,1\} \cdot \texttt{CAPLock}_a(r, \texttt{x}, v)\}\ \texttt{lock(x)}\ \{\texttt{CAPLock}_a(r, \texttt{x}, 1) * [\mathrm{U}]_a * \mathit{Inv}\}$$

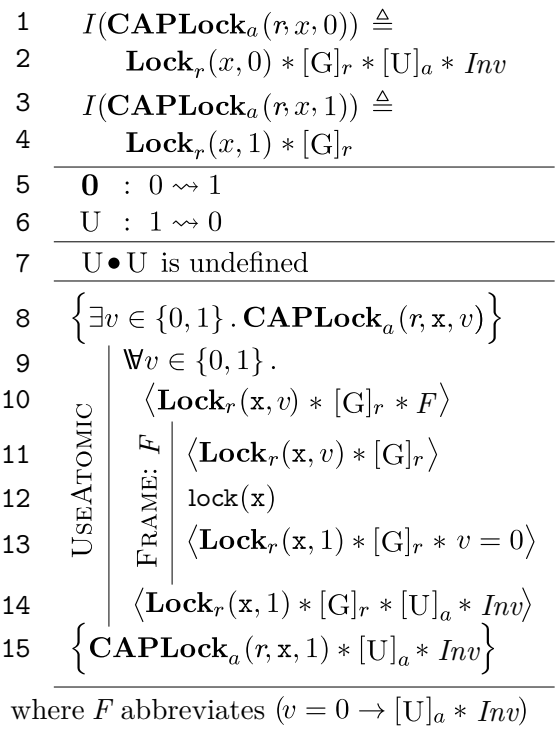

Figure 4.4.: TaDA declarations and proof outline for a shared region `CAPLock`, taken (with minor changes) from the TaDA publication [39], and building on the lock example from Fig. 4.1. The additional declarations and the outline's postcondition provide the usual semantics of a lock that protects a resource invariant: the vacuous empty guard **0** allows arbitrarily many clients to compete for the lock, but only the holder of the unique guard U can release the lock. Lock holders also temporarily gain ownership of the lock's resource invariant. Levels are again omitted, but supported by Voila and included in the technical report [142].

As before, the proof outline omits most steps, and shows only two rule applications: the frame rule and UseAtomic. The application of the frame rule enables us to preserve the assertion $v = 0 \rightarrow [\mathrm{U}]_a * Inv$ (abbreviated as F in the figure) across the call to `lock`. For the postcondition, we use an omitted rule of consequence to derive from $v = 0$ and $v = 0 \rightarrow [\mathrm{U}]_a * Inv$ the assertion $[\mathrm{U}]_a * Inv$. Intuitively, the application of UseAtomic (also shown in Fig. 4.2) applies MakeAtomic and UpdateRegion together at once.

The complete Voila proof outline for `CAPLock` is shown in Fig. 4.5, where procedure `caplock` has the desired specification. The example verifies in Voila when combined with the previously shown code from Fig. 4.3.

```
1  edicate Inv() /* Invariant, left abstract */
2
3  gion CAPLock(id a, id r, cell x)
4  interpretation {
5    Lock(r, x, ?v) && G@r && (v == 0 || v == 1) && (v == 0 ==> U@a &&
       Inv())
6  }
7  state { v }
8  guards { duplicable Z; unique U; }
9  actions { Z: 0 ~> 1; U: 1 ~> 0; }
10
11 ocedure caplock(id a, id r, cell x)
12 requires CAPLock(a, r, x) && Z@a;
13 ensures  CAPLock(a, r, x, 1) && U@a && Inv();
14
15 use_atomic using CAPLock(a, r, x) with Z@a {
16   lock(r, x);
17 }
```

Figure 4.5.: The Voila proof outline of TaDA's `CAPLock`, building on our lock example from Fig. 4.3, and strongly resembling the TaDA proof outline from Fig. 4.4. Note that Voila does not yet support TaDA's empty guard; instead, we use a duplicable guard Z.

4.2. Proof Outline Language

Proof outlines annotate programs with rule applications of a given program logic. These annotations indicate where to apply rules and how to instantiate their meta-variables. The goal of a proof outline is to convey the essential proof steps; ideally, consumers of such outlines can then construct a full proof with modest effort. Consumers may be human readers [135], or tools that automatically check the validity of a proof outline [11, 138, 145]; our focus is on the latter.

[135]: Owicki (1975), *Axiomatic Proof Techniques for Parallel Programs*

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

[138]: Windsor et al. (2017), *Starling: Lightweight Concurrency Verification with Views*

[145]: Mooij et al. (2005), *Incremental Verification of Owicki/Gries Proof Outlines Using PVS*

The key challenge of designing a proof outline language is to define annotations that accomplish this goal with low annotation overhead for proof outline authors. To approach this challenge systematically, we classify the rules of the program logic (here: TaDA) into three categories: (1) For some rules, the program prescribes where and how to apply them, *i.e.* they do not require any annotations. We call such rules *syntax-driven rules*. An example in standard Hoare logic is the assignment rule, where the assignment statement prescribes how to manipulate the adjacent assertions. (2) Some rules can be applied and instantiated in many meaningful ways. For such rules, the author of the proof outline needs to indicate where or how to apply them through suitable annotations. Since such rules often indicate essential proof steps, we call them *key rules*. In proof outlines for standard Hoare logic, the while-rule typically requires an annotation *how* to apply it, namely the loop invariant. (3) The effort of authoring a proof outline can be greatly reduced by applying some rules heuristically, based on information already present in the outline. We call such rules *bridge rules*. Heuristics reduce the annotation overhead, but may lead to incompleteness if they fail; a proof outline language may provide annotations to complement the heuristics in such situations, slightly blurring the distinction between key and bridge rules. E.g., the Dafny verifier [146] applies heuristics to guess termination measures for loops, but also offers an annotation to provide a measure manually, if necessary. Another common example is the rule of consequence: SMT-based verifiers (such as Voila) automatically discharge most entailment checks, but may require additional user annotations in cases where the underlying SMT solver is incomplete.

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

The rule classification depends on the proof search capabilities of the

verification tool that is used to check the proof outline. We use Viper [5], which provides a high degree of automation for standard separation logic and, thus, allows us to focus on the specific aspects of TaDA.

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

In the rest of this section, we give an overview of the Voila proof outline language and, in particular, discuss which TaDA rules are supported as syntax-driven, key, and bridge rules. Voila's grammar can be found in Sec. 4.10, showing that Voila strongly resembles TaDA, but requires fewer technical details.

Expressions and Statements. Voila supports all of TaDA's programming language constructs, including variables and heap locations, primitive types and operations thereon, atomic heap reads and writes, loops, and procedure calls. Consequently, Voila supports the corresponding syntax-driven TaDA rules.

Background Definitions. Voila's syntax for declaring regions and transitions closely resembles TaDA, but *e.g.* subscripts are replaced by additional parameters, such as the region identifier `r`. A region declaration defines the region's content via an `interpretation` assertion, and its value via a `state` function. The latter may refer to region parameters, as well as values bound in the interpretation, such as `v` in the example from Fig. 4.3. The region's transition system is declared by introducing the guards and the permitted *actions*, *i.e.* transitions. Voila includes several built-in guard algebras (adopted from Caper [41]); additional ones can be encoded, see the Sec. 4.8. A region declaration introduces a corresponding region predicate, which has an additional out-parameter that yields the region's abstract state (*e.g.* `s` in the precondition of procedure `lock` in Fig. 4.3), as defined by the `state` function. We omit this out-parameter when its value is irrelevant.

[41]: Dinsdale-Young et al. (2017), *Caper - Automatic Verification for Fine-Grained Concurrency*

Specifications. Voila proof outlines require specifications for procedures, and invariants for loops; we again chose a TaDA-like syntax for familiarity. Explicit loop invariants are required by Viper, but also enable us to automatically instantiate certain bridge rules (see framing in Sec. 4.4).

Recall that specifications in TaDA are written as atomic or non-atomic triples, and include an interference context and an atomicity context. Voila simplifies the notation significantly by requiring these contexts only for abstractly-atomic procedure specifications; for all statements and rule applications, they are determined automatically, despite changing regularly during a proof. For procedures with abstractly-atomic behavior (modifier `abstract_atomic`), the interference context is declared through the `interference` clause. E.g., for procedure `lock` from Fig. 4.3, it corresponds to TaDA's interference context $\mathbb{V}s \in \{0, 1\}$.

Key Rules. In addition to procedure and loop specifications, Voila requires user input only for the following fundamental TaDA rules: UpdateRegion, MakeAtomic, UseAtomic, and OpenRegion; applications of all other rules are automated. Since they capture the core ideas behind TaDA, these rules are among the most complex rules of the logic and admit a vast proof search space. Therefore, their annotation is essential, for both human readers [39, 139] and automatic checkers. As seen in Fig. 4.3, the annotations for these key rules include only the used region and, for updates, the used guard; all other information present in the corresponding TaDA rules is derived automatically.

[39]: Rocha Pinto et al. (2014), *TaDA: A Logic for Time and Data Abstraction*
[139]: Rocha Pinto (2016), *Reasoning with time and data abstractions*

Bridge Rules. All other TaDA rules are applied automatically, and thus have no Voila counterparts. This includes all structural rules for manipulating triple atomicity (*e.g.* AWeakening1, AExists), interference contexts (*e.g.* Substitution, AWeakening2), and levels (*e.g.* AWeakening3). Their applications are heuristically derived from the program, applications of key rules, and adjacent triples. TaDA's frame rule is also automatically applied by leveraging Viper's built-in support for framing, combined with additional encoding steps to satisfy TaDA's frame stability side condition. Finally, TaDA entailments are bridge rules when they can be automated by the used verification tool. For Viper, this is the case for standard separation logic entailments, which constitute the majority of entailments to perform. To support TaDA's *view shifts* [139, 147] – entailments similar to the classical rule of consequence, but involving arbitrary definitions of regions and guard algebras – Voila provides specialized annotations.

[139]: Rocha Pinto (2016), *Reasoning with time and data abstractions*
[147]: Dinsdale-Young et al. (2013), *Views: compositional reasoning for concurrent programs*

4.3. Proof Workflow

Our approach, and corresponding implementation, enables the following workflow: users provide a proof outline and possibly some annotations for complex entailments. If the outline summarizes a valid proof, verification is automatic, without a tedious process of manually applying additional rules. If the outline is invalid, our tool reports which specification (*e.g.* loop invariant) it could not prove or which key rule application it could not verify, and why (*e.g.* missing guard).

Achieving this workflow, however, is challenging: by design, proof outlines provide the important proof steps, but are not complete proofs. Consider, *e.g.* the TaDA and Voila outlines from Fig. 4.1 and Fig. 4.3, respectively. Applying UpdateRegion produces an atomic triple in its conclusion, whereas the while-rule requires a non-atomic triple for the loop body. A complete proof needs to perform the necessary adjustment through additional applications of bridge rules, which are not present in the proof outlines, and thus need to be inferred.

Our workflow is enabled by first expanding proof outlines into *proof candidates*, in two main steps: step 1 automatically inserts the applications of all syntax-driven rules; step 2 expands further by applying heuristics to insert bridge rule applications. The resulting proof candidate contains the applications of all rules of the program logic. Afterwards, we check that the proof candidate corresponds to a valid proof, by encoding it as a Viper program that checks whether all proof rules are applied correctly. Our actual implementation deviates slightly from this conceptual structure, *e.g.* because Viper does not require one to make the application of *all* syntax-driven rules, framing, and entailment checking explicit.

4.4. Expanding Proof Outlines to Proof Candidates

Automatically expanding a proof outline is ultimately a proof search problem, with a vast search space in case of complex logics such as TaDA.

Our choice of key rules (and corresponding annotations) reduces the search space, but it remains vast, due to TaDA's many structural rules that can be applied to almost all triples. To further reduce the search space, without introducing additional annotation overhead, we devised (and enforce) a *normal form* for proof candidate triples. Our normal form allows us to define *heuristics* for the application of bridge rules *locally*, based only on adjacent rule applications, without having to inspect larger proof parts. This locality reduces the search space substantially, and enables us to automatically close the gap between user-provided proof outline and finally verified proof candidate. Out of the 22 rule applications for our running example, our heuristics infer 17 applications of bridge rules. Three syntax-driven rules are also applied automatically, such that only two key rules require manual annotations. The complete TaDA proof shown in App. A details all inferred applications of bridge rules and syntax-driven rules.

It might be helpful to consider an analogy with standard Hoare logic: its rule of consequence can be applied to each Hoare triple. A suitable normal form could restrict proofs to use the rule of consequence only at the beginning of the program and for each loop (as in a weakest-precondition calculus). A heuristic can then infer the concrete applications, in particular, the entailments used in the rule application, treating the rule as a bridge rule.

Normal Form. Our normal form is established by a combination of syntactic checks and proof obligations in the final Viper encoding. Its main restrictions are as follows: (1) A triple is atomic if and only if the enclosed Voila outline statement is abstract atomic, namely a CAS operation, a call to an abstract atomic procedure, or a key rule statement. As a consequence, we can infer the triple kinds from statements and key rule applications. Due to this restriction, Voila cannot express specifications that combine atomic and non-atomic behaviors. However, such specifications do not occur frequently (see Sec. 5.2.3 in [139] for an example) and could be supported via additional annotations. (2) All triple preconditions, as well as the postconditions of non-atomic triples, are *stable*, *i.e.* cannot be invalidated by (legal) concurrent operations. In contrast, TaDA requires stability only for certain assertions. Our stronger requirement enables us to *rely* on stability at various points in the proof instead of having to *check* it – most importantly, when Viper automatically applies its frame rule. To enforce this restriction, we eagerly stabilize assertions through suitable weakening steps. (3) In atomic triples, the state of every region is bound by exactly one interference quantifier ($\forall$), which simplifies the manipulation of interference contexts, *e.g.* for procedure calls. To the best of our knowledge, this restriction does not limit the expressiveness of Voila proofs. (4) Triples must hold for a *range* of atomicity contexts $\mathcal{A}$, rather than just a single context. This stronger proof obligation rules out certain applications of MakeAtomic – which we have seen only in contrived examples – but it increases automation substantially and improves procedure modularity.

[139]: Rocha Pinto (2016), *Reasoning with time and data abstractions*

By design, our normal form prevents Voila from constructing certain TaDA proofs. However, the only practical limitation is that Voila does not support TaDA's combination of atomic and non-atomic behavior in a single triple. As far as we are aware, all other normal form restrictions do

not limit expressiveness for practical examples, or can be worked around in systematic ways.

Heuristics. We employ five main heuristics: (1) to determine when to change triple atomicity, (2) to ensure stable frames by construction, (3) to compute atomicity context ranges, (4) to compute levels, and (5) to compute interference contexts in procedure body proofs. All heuristics are based on inspecting adjacent rule applications and their proof state. We briefly discuss the first three heuristics here. We give a more detailed explanation and discuss the other two heuristics in an extended discussion of our running example in Sec. 4.7. (1) Changing triple atomicity corresponds to an application of (at least) TaDA rule AWeakening1, necessary when a non-atomic composite statement (*e.g.* the `while` statement in Fig. 4.1) has an abstract-atomic sub-statement (*e.g.* the atomic CAS in Fig. 4.1). We infer all applications of this rule. (2) A more complex heuristic is used in the context of framing: TaDA's frame rule requires the *frame*, *i.e.*, the assertion preserved across a statement, to be stable. For simple statements such as heap accesses, it is sound to rely on Viper's built-in support for framing. For composite statements with arbitrary user-provided *footprints* (assertions such as a loop invariant describing which resources the composite statement may modify), we greedily infer frame rule applications that attempt to preserve all information outside the footprint. The inferred applications are later encoded in Viper such that the resulting frame is stable, by applying suitable weakening steps. (3) Atomicity context ranges are heuristically inferred from currently-owned tracking-resources and level information. We track the range with a lower and an upper bound on atomicity contexts capturing which regions must be and are definitely not in an atomicity context, respectively. We capture the lower bound as the set of all regions that are currently being updated. The upper bound is captured as the lowest level of any region that is being updated, *i.e.* regions with a lower level are definitely not in the atomicity context. Atomicity contexts are not manipulated by a specific TaDA rule, but they need to be instantiated when applying rules. The lower bound is changed and checked by MakeAtomic and UpdateRegion, respectively. The upper bound is used for procedure calls: specifically, to ensure that there is not already a pending update for a region the callee might update as well.

In our experience, our heuristics fail *only* in two scenarios: the first are contrived examples, concerned with TaDA resources in isolation, not properties of actual code – where they fail to expand a proof outline into a valid proof. More relevant is the second scenario, where our heuristics yield a valid proof that Viper then fails to verify because it requires entailments that Viper cannot prove automatically. To work around such problems when they occur, Voila allows programmers to provide additional annotations to indicate where to apply complex entailments.

Importantly, a failure of our heuristics does *not* compromise soundness: if they infer invalid bridge rule applications, *e.g.* whose side conditions do not hold, the resulting invalid proof candidates are rejected by Viper in the final validation.

For our running example from Fig. 4.1, four of our heuristics are necessary to complete the proof candidate. The heuristic (1) is necessary around UpdateRegion to change the triple atomicity. The heuristic (2) is necessary

around the CAS operation to frame information about the arguments. The heuristic (3) is necessary so that clients can call the `lock` procedure. Lastly, the heuristic (4) is necessary around UpdateRegion to change levels.

4.5. Validating Proof Candidates in Viper

Proof candidates – *i.e.* the user-provided program with heuristically inserted bridge rule applications – do not necessarily represent valid proofs, *e.g.* when users provide incorrect loop invariants. To check whether a proof candidate actually represents a valid proof, we need to verify (1) that each rule is applied correctly, in particular, that its premises and side conditions hold, and (2) that the property shown by the proof candidate entails the intended specification. To validate proof candidates automatically, we use the existing Viper tool [5]. In this section, we give a high-level overview of how we encode proof candidates into the Viper language.

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

Viper Language. We discussed Viper in Chapter 2. We briefly reiterate the necessary background: Viper uses a variation of separation logic [148, 149] whose assertions separate access permissions from value information: separation logic's points-to assertion `x.f` $\mapsto$ `v` is expressed as `acc(x.f) && x.f == v`, and separation logic predicates [143] are similarly split into a predicate (abstracting over permissions) and a heap-dependent function (abstracting over values). Well-definedness checks ensure that the heap is accessed only under sufficient permissions. Viper provides a simple imperative language, which includes in particular two statements to manipulate the verification state: `exhale` A asserts all logical constraints in assertion A, removes the permissions in A from the current state (or fails if the permissions are not available) and assigns non-deterministic values to the corresponding memory locations (to reflect that the environment could now modify them); `inhale` A conversely assumes constraints and adds permissions.

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

[149]: Parkinson et al. (2012), *The Relationship Between Separation Logic and Implicit Dynamic Frames*

[143]: Parkinson et al. (2005), *Separation logic and abstraction*

Regions and Assertions. TaDA's regions introduce various resources such as region predicates and guards. We encode these into Viper permissions and predicates as summarized in Fig. 4.6 (left). Each region `R` gives rise to a corresponding predicate, which is defined by the region interpretation. A region's abstract state may be accessed by a Viper function `R_State`, which is defined based on the region's `state` clause, and depends on the region predicate. Moreover, we introduce an abstract Viper predicate `R_g` for each guard `g` of the region.

These declarations allow us to encode most TaDA assertions in a fairly straightforward way. E.g., the assertion $\texttt{Lock}_r(\texttt{x}, s)$ from Fig. 4.1 is encoded as a combination of a region predicate and the function yielding its abstract state: `Lock(`r`,x) && Lock_State(`r`,x) ==` s. We encode region identifiers as references in Viper, which allows us to use the permissions and values of designated fields to represent resources and information associated with a region instance. E.g., we use the permission `acc(`r`.diamond)` to encode the TaDA resource $r \Mapsto \blacklozenge$. Similarly, the permissions to the fields `R_from` and `R_to` represent TaDA's $r \Mapsto (x, y)$ resource, while the fields' values reflect the arguments x and y. Therefore, $r \Mapsto (0, 1)$

```
⟦region R(r: id, p̄: t̄)
  interpretation I
  state S
  guards G
  actions A⟧ ->
pred R(r: Ref, p:⟦t⟧) { ⟦I⟧ }

func R_State(r: Ref, p:⟦t⟧): T
  requires  R(r,p̄)
{ unfolding R(r,p̄) in ⟦S⟧ }

foreach g(p': t') ∈ G:
  pred R_g(r: Ref, p':⟦t'⟧)
end

field diamond: Bool
field R_from: T
field R_to: T

field R_X: Set[T]
field R_A: Set[T]
```

```
field val: Int

pred Lock(r: Ref, x: Ref) {
  acc(x.val) &&
  (x.val == 0 OR x.val == 1)
}

func Lock_State
              (r: Ref, x: Ref):
    Int
  requires Lock(r, x)
{ unfolding lock(r, x) in x.val }

pred Lock_G(r: Ref)

field diamond: Bool
field Lock_from: Int
field Lock_to: Int

field Lock_X: Set[Int]
field Lock_A: Set[Int]
```

Figure 4.6.: Excerpt of the Viper encoding of regions; general case (left), and for the lock region from Fig. 4.3 (right). The encoding function is denoted by double square brackets; overlines denote lists; *foreach* loops are expanded statically. Type T is the type of the state expression S, which is inferred. Actions A do not induce any global declarations. The elements of struct types and type `id` are encoded as Viper references (type `Ref`). The `unfolding` expression temporarily unfolds a predicate into its definition; it is required by Viper's backend verifiers. The struct type `cell` from Fig. 4.3 is encoded as a Viper reference with field `val` (in Viper, all objects have all fields declared in the program).

from Fig. 4.1 is encoded as `acc(`r`.Lock_from) && acc(`r`.Lock_to) &&` r`.Lock_from == 0 &&` r`.Lock_to == 1`.

Besides assertions, TaDA judgments include an interference context and an atomicity context. An interference context of the form $\forall s \in X$, associated with a region `R(`r`, ...)`, is represented by a field r`.R_X`, which stores the set of values to which the environment may set the region's abstract state. The encoding of an atomicity context $\mathcal{A}$, which tracks pending updates and prevents multiple such updates for the same region instance, is more involved. As explained in Sec. 4.4, we check the proof outline for all atomicity contexts within a range of atomicity contexts, which we express with a lower and an upper bound. The lower bound is represented by a set-typed variable `update`, local to each procedure, storing the set of all regions currently being updated. The upper bound is represented with the variable `alevel`, storing the lowest level of any region for which an update is pending. Lastly, the domain of an update $\mathcal{A}$(`r`) is encoded with a set-typed field `r.R_A`. Its value influences assertion stabilization: while an update is pending (*i.e.*, inside `make_atomic`), the environment may not take the region value out of `r.R_A`; the latter is set to `r`'s interference context (`r.R_X`) when `make_atomic` is entered.

Rule Applications. Proof candidates are tree structures, where each premise of a rule application R is established as the conclusion of another rule application, as illustrated below.

$$\dfrac{\dfrac{\vdots}{\{P_p\}s_p\{Q_p\}}}{\{P_c\}s_c\{Q_c\}}\,(R)\quad\vdots$$

In TaDA, the statement of the premise s_p is guaranteed to be a substatement of the statement of the conclusion s_c. To check the validity of a candidate, we check the validity of each rule application. For rules

```
1  INTERFERENCE_PERMITTED(Lock(r, x), from, to) ->
2       (none < perm(r.diamond) ==> Lock_State(r, x) in r.Lock_A)
3    && (   from == 0 && to == 1 && ENV_MAY_HOLD(Lock_G(r))
4        || from == 1 && to == 0 && ENV_MAY_HOLD(Lock_G(r)) )
5
6  ENV_MAY_HOLD(Lock_G(r)) -> perm(Lock_G(r)) == none
7
8  STABILIZE(Lock(r, x)) ->
9    label pre_havoc
10   havoc Lock(r, x)
11   inhale INTERFERENCE_PERMITTED(Lock(r, x),
12       old[pre_havoc](Lock_State(r, x)), Lock_State(r, x))
13
14 INFER_INTERFERENCE(Lock(r, x)) ->
15   havoc r.Lock_X
16   inhale forall s: Int :: s in r.Lock_X
17      <==> INTERFERENCE_PERMITTED(Lock(r, x), Lock_State(r, x), s)
```

Figure 4.7.: Encoding of stabilization and interference inference for the `Lock` example. Viper labels enable referring to the verification state at a particular point in the program (*i.e.*, they generalize `old` expressions, which refer to the prestate of a method). We assume that symbols introduced by macros, *e.g.* label `pre_havoc`, are always fresh and never result in name clashes. The Viper expression `perm(`ρ`)` denotes the permission currently held to a resource ρ.

that are natively supported by Viper (*e.g.* the assignment rule), Viper performs all necessary checks. Each other rule application is checked via an encoding into the following sequence of Viper instructions: (1) Exhale the precondition P_c of the conclusion to check that the required assertion holds. (2) Inhale the precondition P_p of the premise since it may be assumed when proving the premise. (3) After the encoding of the proof for the premise, exhale the postcondition Q_p of the premise to check that it was established by the proof for the premise. (4) Inhale the postcondition Q_c of the conclusion. Steps 2 and 3 are performed for each premise of the rule. Moreover, we assert the side conditions of each rule. If a proof candidate is invalid, *e.g.* composes incompatible rules, one of the checks above fails and the candidate is rejected.

Using this encoding of rule applications as building blocks, we can assemble entire procedure proofs as follows: for each procedure, we inhale its precondition, encode the rule application for its body, and then exhale its postcondition.

Example: Stability and Interference Context Inference. Recall that an assertion A is stable if and only if the environment cannot invalidate A by performing any legal region updates. In practice, this means that the environment cannot hold a guard that allows it to change the state of a region in a way that violates A. The challenge of checking stability as a side-condition is to avoid higher-order quantification over region instances and guards, which is hard to automate. We address this challenge by actively *stabilizing* assertions in the Viper encoding. That is, we remove information from Viper's verification state such that the remaining information about the state is stable. We achieve this effect by first assigning non-deterministic values to the region state, and then constraining these to be within the states permitted by the region's transition system, taking into account the guards the environment could hold.

Fig. 4.7 shows the encoding of stabilization for instances of our `Lock` region (macro `STABILIZE`). First, the region state is havocked, *i.e.*, all information about the state is thrown away. Afterwards, the new region state is assumed to be any state reachable by the environment from the old state. We encode this property of reachability by the environment in two steps: `ENV_MAY_HOLD` yields whether a guard may be held by the environment. The encoding depends on the guard kind: the environment can hold

the unique guard `G` only if it is not already present in the proof state. In contrast, duplicable guards may always be held by the environment, in which case `ENV_MAY_HOLD` would be defined as `true`. Building on `ENV_MAY_HOLD`, `INTERFERENCE_PERMITTED` encodes the actual reachability property: the environment may perform a state transition if it holds at least the guard that is required for this transition by the transition system. Furthermore, the transition has to stay within the atomicity context if an update is still pending, which is TaDA's interference rely-guarantee. To avoid computing the transitive closure, Voila requires (and checks) transition systems to be transitively closed.

The encoded reachability (macro `INTERFERENCE_PERMITTED`) is also essential for the inference of interference contexts. Intuitively, the smallest interference context, at a given program point, corresponds to the set of states that the environment could transition to, which is exactly the set we already need for stabilization. Therefore, as shown in macro `INFER_INTERFERENCE`, we can obtain a suitable interference context by constraining `r.Lock_X` to be the set of all states reachable by the environment.

4.6. Soundness

Voila is sound if the successful verification of a procedure in Voila implies that the procedure's specification can be derived in TaDA. That is, our soundness argument builds on the soundness of the TaDA logic itself, which has been proven separately [150] w.r.t. an operational semantics. Voila succeeds if the encoded proof candidate of the procedure successfully verifies in Viper. Consequently, to show soundness of Voila, we need to show that successful verification of a proof candidate in Viper implies the existence of a corresponding TaDA proof for the given procedure and its specification.

[150]: Rocha Pinto et al. (2016), *Modular Termination Verification for Non-blocking Concurrency*

Notations. Before we formalize our argument, we introduce basic terminology and notation. As discussed in Sec. 4.5, proof candidates are derivation trees in the TaDA logic. We refer to the proof candidate that is inferred for a Voila outline statement s as the proof candidate for s. We introduce a function C to model the inference of proof candidates, where $C(s)$ is the proof candidate of the Voila outline statement s. The entire proof candidate for a procedure is obtained by applying C to the procedure's body. For a proof candidate p, p's *root* is the last rule application of p, which derives the overall conclusion, and p's *children* are the proof candidates whose roots are rule applications to the premises of that last rule application in p.

As discussed in Sec. 4.5, proof candidates are encoded to Viper statements by encoding all rule applications of the proof candidate. For a proof candidate p, $[\![p]\!]$ denotes the Viper statement that p is encoded to. We use the same notation to encode TaDA assertions and expressions. *Viper verification states* model the entire knowledge of the Viper verifier at a specific point in a Viper program. Technically, each Viper verification state is a set of concrete Viper states; this set can be characterized by a Viper assertion. For a Viper assertion A, $\Upsilon(A)$ denotes the initial Viper verification state after inhaling A only. The expression $\mathsf{post}(c, v)$ denotes

the Viper verification state resulting from verifying a Viper statement c starting from the Viper verification state v. The function post is analogous to the strongest postcondition of standard Hoare logic. We refer to v and $\mathsf{post}(c, v)$ as the Viper verification pre- and poststate of c, respectively. The special error Viper verification state $\lightning$ models that an assertion failed during verification. We use s, p, v to range over Voila outline statements, proof candidates, and Viper verification states, respectively.

4.6.1. Proof Overview

We split our soundness argument into five steps: (1) We determine invariants on the Viper verification pre- and poststates of encoded rule applications. (2) We define a *judgment mapping* $(\!| v, p |\!)$, which takes a Viper verification state v satisfying our invariants from the first step, together with a proof candidate p, and returns a TaDA judgment. We refer to $(\!| v, p |\!)$ also as the TaDA judgment of v and p, sometimes omitting v.

This judgment mapping establishes our connection between verifying a Viper program and deriving a TaDA proof. (3) We show that the Viper encoding of single rule applications is sound: Consider a proof candidate p whose root is a rule application for a TaDA rule r. Let the proof candidates $p'_1, \ldots, p'_N$ be p's children and let v_0 be a Viper verification state satisfying our invariants. Since soundness considers only Voila statements, for which the Viper encoding verifies successfully, we may assume that no assertion fails when verifying the Viper encoding of r's rule application starting from v_0. As illustrated below, we then show that the TaDA rule r can be applied correctly to derive p's TaDA judgment $(\!| v_0, p |\!)$ as the conclusion. The rule application must contain the TaDA judgments $(\!| v_1, p'_1 |\!), \ldots, (\!| v_N, p'_N |\!)$ of p's children as the premises. The Viper verification states $v_1, \ldots, v_N$ are the prestates of the childrens' Viper encodings.

$$\frac{(\!| v_1, p'_1 |\!) \quad \ldots \quad (\!| v_N, p'_N |\!)}{(\!| v_0, p |\!)}\,(r)$$

(4) We show inductively that the Viper encoding of proof candidates is sound: Let s be a Voila outline statement and v be a Viper verification state satisfying our invariants. We may assume that no assertion failed when verifying the encoded proof candidate $[\![C(s)]\!]$ of s starting from v. We then show that the TaDA judgment $(\!| v, C(s) |\!)$ of the proof candidate $C(s)$ is derivable in TaDA. (5) We show soundness of the specification encoding: Consider a successfully-verified Voila procedure with precondition P, postcondition Q, and body s. We show that the procedure's specification can be derived in TaDA as a conclusion using the rule of consequence on the TaDA judgment $(\!| \Upsilon([\![P]\!]), C(s) |\!)$ of the Viper verification state $\Upsilon([\![P]\!])$ from the encoded precondition $[\![P]\!]$ and of the procedure's proof candidate $C(s)$.

Combining these steps, we formally connect verification of an encoded proof candidate to derivability of a corresponding TaDA proof, resulting in the soundness of Voila. We first illustrate this approach in more detail on a simplified version of TaDA. Afterwards, we discuss how we apply this approach to full TaDA.

4.6.2. Proof for Simplified TaDA

For a simplified version of TaDA, assume that TaDA judgments are standard Hoare judgments of the form $\vdash \{P\}\ \hat{s}\ \{Q\}$. We omit atomic triples, levels, atomicity contexts, interference contexts, and the requirement that pre- and postconditions are stable. These features are discussed in Sec. 4.6.3. Furthermore, for simplicity, we do not distinguish between Voila and TaDA assertions since they differ only in syntax. For our simplified version of TaDA, we illustrate how to instantiate our five aforementioned steps to show soundness.

Step 1: Invariants. To prove soundness, we will later (in step 2) connect Viper verification states to TaDA pre- and postconditions. However, not every Viper verification state can be connected to a TaDA assertion. E.g., TaDA does not support fractional permissions [61] for points-to predicates ($\texttt{x.f} \mapsto v$), whereas fractional permissions are generally possible in Viper verification states. We define invariants on Viper verification states that rule out Viper verification states that do not correspond to TaDA assertions. These invariants have to hold only for those Viper verification states that we have to connect to TaDA assertions, namely the pre- and poststates of encoded rule applications. In particular, intermediate Viper verification states of our Viper encoding do not have to satisfy the invariants. We use these invariants in the definition of the judgment mapping.

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

Step 2: Judgment Mapping. For our simplified version of TaDA, we define a judgment mapping for a Viper verification state v and the proof candidate of a Voila outline statement s as

$$(\!| v, C(s) |\!) = \{\phi(v)\}\ \llcorner s \lrcorner\ \{\phi(\mathsf{post}([\![C(s)]\!], v))\},$$

where ϕ maps Viper verification states to TaDA assertions and $\llcorner\cdot\lrcorner$ maps Voila outline statements to TaDA statements. We refer to ϕ as the *TaDA assertion interpretation*. The definition of ϕ is very technical and omitted for the sake of simplicity. Conversely, the definition of $\llcorner\cdot\lrcorner$ is straightforward. The function removes all rule annotations from a Voila outline statement, *e.g.* $\llcorner$`update_region{b := CAS(x,0,1)}`$\lrcorner$ is `b := CAS(x,0,1)`.

As explained for step 1, only Viper verification pre- and poststates of encoded rule applications are connected to TaDA assertions. To use the judgment mapping, we have to show that the invariants hold for the verification state-argument (v above).

To lift deductions at the level of Viper verification states to deductions at the level of TaDA assertions, we prove that the TaDA assertion interpretation ϕ satisfies two properties: (1) An entailment in Viper, *e.g.* $v_1 \vDash_{\mathsf{Viper}} v_2$ for some Viper verification states v_1 and v_2 (recall that Viper verification states correspond to Viper assertions), implies the corresponding entailment in TaDA, *i.e.* $\phi(v_1) \vDash_{\mathsf{TaDA}} \phi(v_1)$. (2) ϕ is the inverse of the encoding, *i.e.* $\phi(\Upsilon([\![P]\!])) = P$ for all TaDA assertions P. Using these two properties, we can show that if an encoded TaDA assertion $[\![P]\!]$ is successfully verified starting from a Viper verification state v, then $\phi(v)$ entails P in TaDA, *i.e.* $\phi(v) \vDash_{\mathsf{TaDA}} P$ holds. This is the basis for checking the correctness of rule applications in Viper.

Step 3: Single Rule Applications. Consider a proof candidate p whose root is an application of the syntax-driven rule for loops. Let the TaDA statement that the rule is applied to be `do invariant` I `{`$\hat{s}$`} while(`b`)` for some TaDA expression b, TaDA statement $\hat{s}$, and TaDA invariant I. Viper has while loops, but not do-while loops. A simplified Viper encoding of the loop rule application is thus $[\![p']\!]$`; while(`$[\![b]\!]$`) invariant` $[\![I]\!]$ `{`$[\![p']\!]$`}`, where the proof candidate p' is the child of p. If verification is successful, then the TaDA invariant I is preserved after the first execution of the loop body. Thus, TaDA's do-while rule can be applied to the TaDA judgment $(\!| v, p |\!)$, where v is the Viper verification state the verification started from. We show soundness of the Viper encoding of single rule applications separately for each rule, obtaining a soundness lemma per rule.

Side conditions already guaranteed by neighboring rule applications, either through checking or by construction, are not checked again. Therefore, for some rules, some of their side conditions are established by the encoding of neighboring rule applications. For the rules where neighboring rule applications guarantee some side conditions, we obtain weaker soundness lemmas, requiring that the necessary side conditions already hold. These dependencies make induction over the proof candidates difficult. Thus, instead of induction over all proof candidates, we perform induction over the Voila outline statements, as shown in the next step.

Step 4: Proof Candidates. The following lemma (SE) formalizes soundness of the Viper encoding of proof candidates, as well as that our invariants on Viper verification states, referred to as $\mathbb{I}$, are maintained by the Viper encoding.

$$v \in \mathbb{I} \land \mathsf{post}([\![C(s)]\!], v) \neq \lightning \implies (\!| v, C(s) |\!) \land \mathsf{post}([\![C(s)]\!], v) \in \mathbb{I} \quad \text{(SE)}$$

Verbally, the lemma expresses *"For all Voila outline statements s and Viper verification states satisfying our invariants $v \in \mathbb{I}$, the absence of failed assertions during the verification of the encoded proof candidate, written* $\mathsf{post}([\![C(s)]\!], v) \neq \lightning$*, implies that the proof candidate's TaDA judgment $(\!| v, C(s) |\!)$ is derivable in TaDA and that the Viper verification poststate satisfies our Viper verification state invariants"*. We apply the lemma to the body of a Voila procedure to get that the Voila proof candidate corresponds to a correct TaDA proof.

We prove the lemma (SE) by structural induction over Voila outline statements. In general, the induction proceeds as follows: Consider a compound outline statement $s\{s'\}$ (s is the compound, *e.g.* `update_region`, and s' is its body, *e.g.* `CAS(...)`) with a Viper encoding $[\![C(s\{s'\})]\!] = \mathsf{c}_1; [\![C(s')]\!]; \mathsf{c}_2$, where c_1 and c_2 are the Viper statements before and after the Viper encoding of the body's proof candidate, respectively. There are four Viper verification states of interest: the prestate v_0 of the compound statement s, the prestate $v_1 = \mathsf{post}(\mathsf{c}_1, v_0,)$ of the body s', the poststate $v_2 = \mathsf{post}([\![s']\!], v_1,)$ of s', and the poststate $v_3 = \mathsf{post}(\mathsf{c}_2, v_2,)$ of s. From the induction hypothesis, we get that the TaDA judgment $(\!| v_1, C(s') |\!) = \{\phi(v_1)\} \llcorner s' \lrcorner \{\phi(v_2)\}$ of the body's proof candidate is derivable in TaDA. We have to show that the TaDA judgment $(\!| v_0, C(s\{s'\}) |\!) = \{\phi(v_0)\} \llcorner s\{s'\} \lrcorner \{\phi(v_3)\}$ of the compound statement's proof candidate is derivable in TaDA. Showing this derivation corresponds to applying rules from TaDA to identify the missing proof steps (indicated by ?) in the TaDA proof below. The application of IH denotes

using the induction hypothesis that $\{\phi(v_1)\} \llcorner s' \lrcorner \{\phi(v_2)\}$ is derivable in TaDA. The necessary rule applications for the missing proof steps are determined by the proof candidate. For each rule application from the proof candidate, we apply the lemma obtained from the soundness of the Viper encoding of single rule applications (step 3), ultimately closing the gap.

$$\dfrac{\dfrac{\vdots}{\{\phi(v_1)\} \llcorner s' \lrcorner \{\phi(v_2)\}}\,(\text{IH})}{\{\phi(v_0)\} \llcorner s\{s'\} \lrcorner \{\phi(v_3)\}}\,(?)$$

Step 5: Specification. The previous proof steps and, in particular, Lemma (Se) show the existence of a TaDA proof for a given Voila statement, but do not yet show that this TaDA proof also establishes the pre- and postcondition of the Voila statement, which we do next. Consider a Voila procedure with a precondition P, postcondition Q, and body s. We need to prove that the procedure's TaDA specification $\{P\} \llcorner s \lrcorner \{Q\}$ is derivable as a conclusion using the rule of consequence on theTaDA judgment $(\!| \Upsilon(\llbracket P \rrbracket), C(s) |\!)$ of the procedure's proof candidate $C(s)$.

For our simplified version of TaDA discussed in this subsection, this property holds if (1) P entails $\phi(\Upsilon(\llbracket P \rrbracket))$ and (2) Q is entailed by the TaDA assertion interpretation $\phi(v_{\text{post}})$, where v_{post} is the Viper verification poststate of the encoded proof candidate $\llbracket C(s) \rrbracket$. Both entailments follow from the properties we proved for the TaDA assertion interpretation ϕ, namely that ϕ is the inverse of the encoding and that ϕ lifts Viper entailments to TaDA entailments. For (2), it is relevant that the Viper encoding asserts $\llbracket Q \rrbracket$ directly after the encoded proof candidate.

Overall Soundness of Voila. The soundness of the Viper encoding of both, specification and proof candidate, enables us to show that Voila is sound. Again, consider a Voila procedure with a precondition P, postcondition Q, and body s. Verification succeeds, if Viper successfully verifies the encoded proof candidate. More concretely, for the Viper precondition $\llbracket P \rrbracket$, the Viper tool verifies the Viper statement $\llbracket C(s) \rrbracket$ without failing an assertion and verifies the Viper postcondition $\llbracket Q \rrbracket$ afterwards. For soundness, we have to show that the corresponding TaDA specification, namely $\{P\} \llcorner s \lrcorner \{Q\}$, is derivable in TaDA.

By instantiating v with $\Upsilon(\llbracket P \rrbracket)$, Lemma (Se) gives us that the TaDA judgment $(\!| \Upsilon(\llbracket P \rrbracket), C(s) |\!)$ of the procedure's proof candidate is derivable in TaDA. From the specification encoding soundness (step 5), we get that the procedure's TaDA judgment $\vdash \{P\} \llcorner s \lrcorner \{Q\}$ can be derived from the TaDA judgment of the procedure's proof candidate using TaDA's rule of consequence. Combining both implies that the TaDA judgment $\{P\} \llcorner s \lrcorner \{Q\}$ is derivable in TaDA, completing the soundness argument.

As a technical detail, to apply lemma (Se), we need to guarantee that the initial Viper verification state $\Upsilon(\llbracket P \rrbracket)$ from the encoded precondition satisfies the invariants $\mathbb{I}$. For our simplified version of TaDA discussed in this subsection, the invariants are guaranteed by the syntactic restrictions of the Voila specification language. For full TaDA, syntactic restrictions do not suffice, since assertions also have to be stable. Voila verifies that

user-provided assertions are well-defined, *i.e.* that their corresponding Viper states are contained in $\mathbb{I}$, using additional Viper proof obligations.

4.6.3. Generalization to TaDA

The proof sketch shown in Sec. 4.6.2 does not account for TaDA's atomic triples, stability requirements, and judgment parameters, namely level, atomicity context, and interference context. To generalize the proof to full TaDA, we introduce three extensions.

First, the semantics of a TaDA assertion differs depending on its use, *i.e.* whether it is a pre- or postcondition and whether it is part of an atomic or non-atomic TaDA triple. Therefore, we need multiple mappings from Viper verification states to TaDA assertions (previously, just ϕ). Second, we extend lemma (Se) according to point (2) of our normal form from Sec. 4.4. More concretely, we add to the invariants $\mathbb{I}$ the restrictions enforced by our normal form about when TaDA pre- and postconditions have to be stable. Lastly, Voila proves TaDA judgments for a set of parameters (point (4) of our normal form from Sec. 4.4). As a consequence, the judgment mapping changes. E.g., for a non-atomic Voila outline statement s, our extended judgment mapping has the shape $(\!|\, v, C(s) \,|\!) = \forall \lambda \in \mathbb{L}(v), \mathcal{A} \in \mathbb{A}(v).\ \lambda, \mathcal{A} \vdash \{\phi(v)\} \llcorner s \lrcorner \{\phi(v')\}$ where $\mathbb{L}(v)$ and $\mathbb{A}(v)$ are the set of levels and atomicity contexts that the TaDA judgment is proved for, respectively. For atomic triples, we use a set of interference contexts as well.

4.7. Extended Discussion of our Running Example

In this section, we provide for our running example, an extended discussion of our heuristics (Sec. 4.7.1) and of our validation of rule applications (Sec. 4.7.2). Fig. 4.8 shows the Voila outline, the proof candidate, and the Viper encoding. We visualize proof candidates by adding steps for *inferred* bridge rule instantiations (*e.g.* `triple_weak`, denoting TaDA rule AWeakening1), analogous to the user-provided key rule instantiations. For simplicity, some of the inferred steps are omitted.

4.7.1. Extended Discussion of our Heuristics

Recall from Sec. 4.4 that our heuristics infer bridge rule applications locally, by inspecting only adjacent rule applications that are to be composed, and their proof state. We employ five main heuristics: to determine when to change triple atomicity, to ensure stable frames by construction, to compute atomicity context ranges, to compute levels, and to compute interference contexts in procedure body proofs. The first three heuristics have been described briefly in Sec. 4.4; here, we provide additional details and illustrate the heuristics guided by our running example.

Changing Triple Kinds. Atomicity changes of a triple are necessary when a non-atomic composite statement has an abstract-atomic sub-statement.

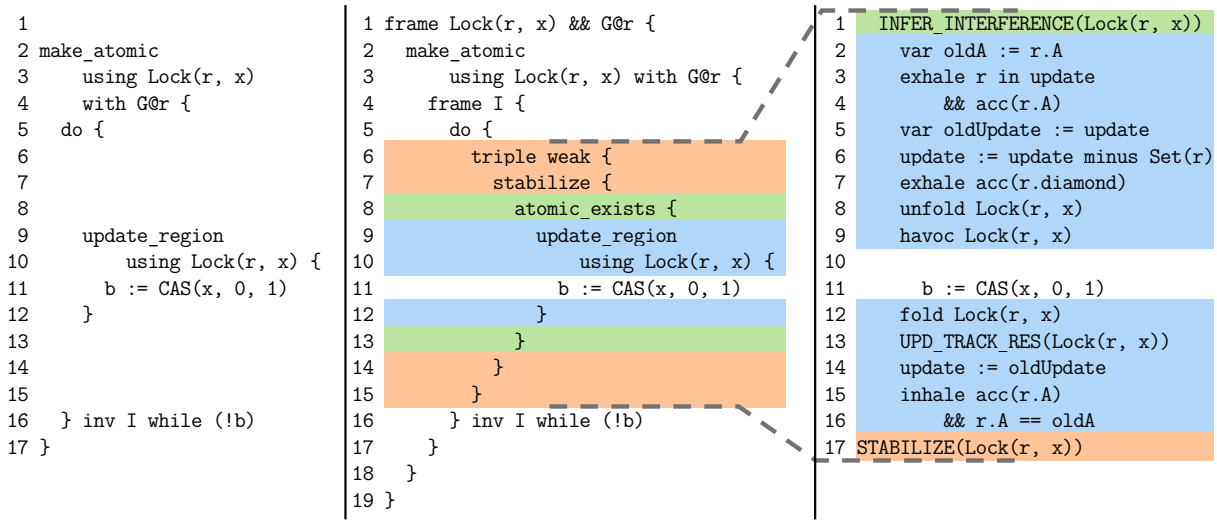

Figure 4.8.: Left to right: the core of our running example's `lock` procedure (same as Fig. 4.3), the proof candidate with inferred bridge rules, and an excerpt of its Viper encoding. Colors link operations of the proof candidate to their encoding. `I` abbreviates the loop invariant from Fig. 4.3. The encoding uses macros such as `STABILIZE` to abstract over Viper details. For the sake of brevity, in the encoding, we shorten the field `r.Lock_A`, which stores the domain of an update to `r.A`. Furthermore, changes to the `alevel` variable, which tracks the upper bound of the range of atomicity contexts, is omitted.

In such cases, we apply `triple_weak` (Line 6 in Fig. 4.8) to obtain a non-atomic triple from an atomic one. The corresponding TaDA rule AWeakening1 requires that the postcondition is stable, which we achieve via stabilization, that is, by applying a specialized TaDA entailment that weakens the postcondition to satisfy stability constructively. We denote this step with a `stabilize` annotation (Line 7) in the proof candidate.

Framing. TaDA's frame rule requires the *frame*, *i.e.*, the assertion preserved across a statement, to be stable. We infer frames greedily, that is, we (actually, Viper) frame as much information around a statement as soundly possible. For simple statements such as heap accesses, this approach automatically leads to stable frames. For composite statements with (arbitrary) user-provided *footprints* (assertions such as loop invariants describing which resources are taken into the composite statement), we need to ensure explicitly that our greedy approach does not produce an unstable frame. For this purpose, we insert explicit `frame` bridge steps (Line 4) around composite statements; all other resources are then framed across, and our encoding will ensure that these frames are stable. In our case, such composite statements are loops (invariants), calls (pre- and postconditions) and `make_atomic` (using-clauses). In each case, a step `frame` F is inserted, indicating that "everything but footprint F" will be framed across and must thus be stable.

Interference Contexts. TaDA's rules for opening a region and calling a procedure (OpenRegion and FunctionCall; both not necessary for our running example) require that the state of each involved region in the precondition is bound by exactly one interference context ($\forall$). This is not guaranteed in arbitrary TaDA proofs (where a region's state might, *e.g.* not be bound at all), but it is in Voila, due to our normal form. As a consequence, no additional step is necessary before opening a region; before calling a procedure (not used by our running example), a `substitution` step is inserted to check the compatibility of the caller's and the callee's interference contexts. However, the frequently necessary

atomicity triple changes from non-atomic to atomic triples violate the single binder restriction of our normal form since non-atomic triples have no interference contexts; similarly, opening a region may violate the restriction since the state of nested regions is typically not bound. To re-establish the normal form, we insert `atomic_exists` steps in both cases, which automatically determine suitable interference contexts for unbound region state, *e.g.* on Line 8, where the preceding `triple_weak` changes triple atomicity.

Levels. Region levels have been omitted in this chapter, but are supported by Voila. Levels are essentially an order on region instances, and are used to prevent circular reasoning when nesting TaDA's duplicable regions. When a region is opened or updated, or when a procedure is called, instantiating the corresponding rule requires a specific triple level. E.g., to open a region, the current level (conclusion) must be one higher than the level of the region to open. To meet such requirements, we infer suitable instances of AWeakening3, to change the triple level, for every rule – with specific level requirements – already present in the proof candidate. Inferring and instantiating AWeakening3 is relatively straightforward, and we believe that our heuristic never fails to infer a suitable application, if one is possible.

4.7.2. Extended Discussion of the Validation of Rule Applications

Recall from Sec. 4.5 that our proof candidates are tree structures (analogous to proof trees in standard Hoare logic), and that we check the validity of a proof candidate by checking the validity of each rule application in it. For that, we (among other things) check that the necessary triple preconditions hold, and that the executed code establishes the necessary postconditions.

We illustrate our encoding scheme on the body of the loop in our running example, see Fig. 4.8. We discuss the proof top-down in the Hoare logic proof, that is, inside-out in the proof candidate and Viper encoding, starting with the `CAS` statement. The `CAS` statement itself is encoded as a Viper method whose specification provides the semantics of the operation.

The proof candidate wraps the `CAS` statement inside an application of the UpdateRegion rule (the blue part in the middle column of Fig. 4.8; the rule itself is shown in Fig. 4.2). Lines 2-6 of the Viper encoding (right column) check and update the atomicity context according to UpdateRegion. Together with the subsequent `exhale` and `unfold`, these Viper statements encode steps 1 and 2 of the rule application: instead of exhaling the entire precondition of the conclusion (step 1) and inhaling the precondition of the premise (step 2), the encoding represents only the *net effect* of these two operations. Therefore, it exhales the diamond resource $r \Longmapsto \blacklozenge$. Going from the conclusion to the premise, UpdateRegion replaces the region predicate (here, `Lock(r,x)`) by its interpretation. Given the region encoding from Fig. 4.6, this is exactly what Viper's unfold operation does. Note that we instantiate the conjunct $P(x)$ in the UpdateRegion rule to represent all other resources and properties that hold in the prestate of the rule application. Hence, it does not show up in the encoding. The

subsequent `havoc` operation assigns a non-deterministic value to the state of all held, *still folded* `Lock(r,x)` predicates. This step is necessary because TaDA region predicates are duplicable. $P(x)$ thus could contain such predicate instances (in addition to the unfolded one), and we must prevent Viper from using those instances to frame old region state around the `CAS` statement, which would be unsound. As confirmed by the authors in personal communication, the latter problem is actually currently not addressed in TaDA.

The first two Viper statements after the `CAS` statement (right column, Lines 12-13) encode steps 3 and 4 of the rule application: the fold operation replaces the interpretation of the `Lock` predicate by the predicate itself. `UPD_TRACK_RES` is an encoding macro, which inhales, depending on the success of the `CAS` operation, one of the tracking resources $r \Longmapsto (0, 1)$ or $r \Longmapsto \blacklozenge$. Analogously to $P(x)$ in the precondition, we take Q_1 and Q_2 to represent all other resources and properties that hold in the poststate of the rule application in these two cases. Since they occur in both postconditions, there is no net effect of inhaling and then exhaling them, and we can omit them from the encoding. The final two instructions (Lines 14-16) in the blue part of the encoding maintain the atomicity context.

Besides UpdateRegion, the loop body contains three additional rule applications. `atomic_exists` (green section) establishes the interference context, which we encode via macro `INFER_INTERFERENCE`. `triple_weak` (orange) weakens an atomic triple in its premise to a non-atomic triple in its conclusion. Since our encoding does not track the triple kind explicitly, `triple_weak` is not directly reflected in the encoding. However, its conclusion – like all non-atomic triples – must be stable. This side condition is enforced in the encoding via the `STABILIZE` macro. We explain both stability and our treatment of interference contexts next.

4.8. Encoding Custom Guard Algebras

Voila provides a high degree of automation, as demonstrated by our evaluation in Sec. 4.9. For concepts not directly supported and automated, Voila provides various features, such as ghost code, to encode them manually. These features enable users of Voila to experiment, for instance, with guard algebras that are not supported by Voila. Crucially, all of these features operate on the level of Voila; programmers do not need to understand (or even be aware of) the encoding into Viper. Voila trusts that users of Voila use these features correctly. In particular, an incorrect usage results in false-positives, *i.e.* Voila may accept a proof outline that does not correspond to a valid TaDA proof. In this section, we demonstrate Voila's support for manual encodings by an example that uses a custom guard algebra.

Specifically, we chose a TaDA-adaptation [139] of Owicki-Gries' classical parallel-increment example: given multiple threads that successively increment a shared counter in parallel, prove that the final counter state equals the sum of the local increments. To achieve the latter, the TaDA proof uses the custom guard algebra defined in Fig. 4.9, which defines resources (as guards) for tracking increments, and laws that govern their use and allow relating local and total increments. The example is included

[139]: Rocha Pinto (2016), *Reasoning with time and data abstractions*

Figure 4.9.: Custom guard algebra (an instance of Iris' authoritative monoid [36]) used by the TaDA adaptation of Owicki-Gries' classical parallel-increment example. Guard Inc counts local increments, and can be split and merged, similar to fractional permissions [61], in which case the local increments are split/merged as well. Guard Total, in contrast, is exclusive and counts the overall increments. Composing the whole Inc instance with Total allows concluding that the sum of the local increments equals the total count. Lastly, both values can only be changed in lockstep.

$$\textsc{Inc}(n_1 + n_2, \pi_1 + \pi_2) = \textsc{Inc}(n_1, \pi_1) \bullet \textsc{Inc}(n_2, \pi_2) \tag{4.1}$$

$$\textsc{Total}(m) \bullet \textsc{Inc}(n, 1) \implies n = m \tag{4.2}$$

$$\textsc{Total}(m) \bullet \textsc{Inc}(n, \pi) = \textsc{Total}(m + d) \bullet \textsc{Inc}(n + d, \pi) \tag{4.3}$$

Figure 4.10.: Example declarations from the Voila encoding of TaDA's counter-client example, including the `CClient` region, and the signature of procedure `single_client` (see also Fig. 4.12), which is executed by each thread. Lemma procedure `INC_split` encodes the left-to-right direction of definition 4.1 from Fig. 4.9 by means of pre- and postconditions. The remaining algebra laws are encoded analogously, and omitted for brevity.

```
1  region CClient(id s, id r, cell x)
2    guards {
3      manual INC(int, frac);
4      manual TOTAL(int);
5    }
6    interpretation {
7      Counter(r, x, ?n) && G@r && TOTAL(n)@s
8    }
9    state { n }
10   actions {
11     ?n, ?m, ?k, ?p | 0f < p && n < m | INC(k, p): n ~> m;
12   }
13
14 lemma INC_split(id s, int k1, int k2, frac p1, frac p2)
15   requires INC(k1 + k2, p1 + p2)@s;
16   requires 0f < p1 && 0f < p2;
17   ensures  INC(k1, p1)@s && INC(k2, p2)@s;
18
19 procedure single_client(id s, id r, cell x, int m)
20   requires CClient(s, r, x, _) && INC(0, 1/2)@s;
21   ensures  CClient(s, r, x, _) && INC(m, 1/2)@s;
```

in our evaluation (`CounterCl`), and, to the best of our knowledge, cannot be encoded in any comparable tool.

Fig. 4.10 shows the Voila declaration of region `CClient`, whose manipulation is governed by aforementioned guard algebra. Guards `INC` and `TOTAL` are declared as `manual` to indicate that they are not part of a guard algebra that Voila automates (see also Sec. 4.9). In particular, this means that Voila will not make any uniqueness assumptions about these guards, *e.g.* when stabilizing region state. The laws of the guard algebra are encoded as lemma procedures such as `INC_split`, which encodes the left-to-right direction of definition 4.1. Region `CClient` abstracts over the shared `Counter(r,n,x)`, whose value `n` corresponds to the total increment count; guard `G`, declared by region `Counter` (see Fig. 4.12), is needed to increment that value. The region's `actions` clause demonstrates Voila's most general syntax for specifying region transitions, and declares that the region state can be incremented from any `n` to any larger `m`, by anybody holding a non-zero fraction of `INC` (regardless of the latter's local increments value `k`). Fig. 4.10 also shows the specification of procedure `single_client`, whose implementation (shown in Fig. 4.12) loops until it made `v` successive increments to the shared counter. Note that `single_client` could be parametric in the permission amount required for `INC` (currently fixed to `1/2`), which would allow arbitrarily many parallel instances (*e.g.* `1/t` for a statically-unknown number of `t` threads).

```
1  // Allocate memory and create region instances ...
2
3  use INC_split(s, 0, 0, 1/2, 1/2);
4
5  parallel {
6    single_client(s, r, x, 9);
7    single_client(s, r, x, 11);
8  }
9
10 use INC_merge(s, 9, 11, 1/2, 1/2);
11
12 unfold CClient(s, r, x);
13 assert Counter(r, x, ?n);
14
15 use TOTAL_INC_equality(s, n, 20);
16 assert n == 20;
17
18 // ... destroy region instances and deallocate memory
```

Figure 4.11.: The central part of the Voila encoding of TaDA's Owicki-Gries adaptation: We use a lemma method to split the guard `INC` before the parallel execution of two calls to `single_client`. After the calls, another lemma method is used to recombine `INC` and to sum up the local increments. Finally, we assert the equality between local and total increments.

Fig. 4.11 shows the central part of the verified code: first, guard `INC(0,0)` is split into two equal fractions by using lemma procedure `INC_split`; afterwards, two calls to `single_client` are run in parallel. Upon termination, lemma procedure `INC_merge` (whose straightforward declaration we omitted), corresponding to the right-to-left direction of guard algebra definition 4.1, is used to combine the `INC` guards obtained from the postconditions of `single_client` into a single instance `INC(20,1f)`. Subsequent ghost code then opens (unfolds) region `CClient` to bind the – at this point unknown – value of the counter to the logical variable `n`. Finally, lemma procedure `TOTAL_INC_equality`, corresponding to guard algebra definition 4.2, is used to learn that `n`'s value is equal to `INC`'s value, *i.e.* `20`. Note that the lemma application would (here) fail for values other than `20`, and that it is possible to work with statically unknown values, *e.g.* `m1`, `m2` and `m1 + m2` instead of constants `9`, `11` and `20`.

In addition to lemma procedures, Voila provides several ghost operations for manipulating its verification state, including: in-/exhale statements for gaining/giving up resources; unfold/fold statements for opening/-closing regions; but also region ghost fields, *e.g.* for witnessing existentials. All of these can be used to encode TaDA proof steps that are beyond what Voila automates, and to experiment with potential extensions. Ghost operations are always applied on the Voila level such that users do not need to be aware of the encoding into Viper.

```
1  struct cell {
2    int f;
3  }
4
5  region Counter(id r, cell x)
6    guards { unique G; }
7    interpretation { x.f |-> ?n }
8    state { n }
9    actions { ?n, ?m | n < m | G: n ~> m; }
10
11 abstract_atomic procedure incr(id r, cell x)
12   interference ?n in Int;
13   requires Counter(r, x, n) && G@r;
14   ensures Counter(r, x, n + 1) && G@r;
15
16 procedure single_client(id s, id r, cell x, int m)
17   requires CClient(s, r, x, _) && INC(0, 1/2)@s;
18   ensures  CClient(s, r, x, _) && INC(m, 1/2)@s;
19 {
20   int i := 0;
21
22   while (i < m)
23     invariant CClient(s, r, x, _);
24     invariant INC(i, 1/2)@s;
25   {
26     use_atomic
27       using CClient(s, r, x, ?v) with INC(i, 1/2)@s;
28     {
29       incr(r, x);
30       use TOTAL_INC_inc(s, v, i, 1/2);
31     }
32
33     i := i + 1;
34   }
35 }
```

Figure 4.12.: Further Voila code from the parallel counter example: the `Counter` region and its `incr` procedure, and the implementation of the `single_client` procedure that is executed by each thread. We slightly simplified the loop invariant by omitting obvious properties. The body of `incr`, omitted for brevity, is similar to procedure `lock` from our running example in Fig. 4.3: a loop around a `CAS` that attempts to increment the counter by one.

4.9. Evaluation

[151]: Treiber (1986), *Systems Programming: Coping with Parallelism*

We evaluated Voila on nine benchmark examples from Caper's test suite, with the Treiber's stack [151] variant `BagStack` being the most complex example, and report verification times and annotation overhead. Each example has been verified in two versions: a version with Caper's comparatively *weak* non-atomic specifications, and another version with TaDA's *strong* atomic specifications; see Sec. 4.11 for a more detailed comparison of Voila and Caper. An additional example, `CounterCl`, demonstrates the encoding of a custom guard algebra not supported in Caper (see Sec. 4.8). To evaluate the performance for both successful and failing verification attempts, we seeded four examples with errors in the loop invariant, procedure postcondition, code, and region specification, respectively. Our benchmark suite is relatively small, but each example involves nontrivial specifications. To the best of our knowledge, no other (semi-)automated tool is able to verify similarly strong specifications.

Performance. Fig. 4.13 shows the runtime for each example in seconds. All measurements were carried out on a Lenovo W540 with an Intel Core i7-4800MQ and 16GB of RAM, running Windows 10 x64 and Java HotSpot JVM 18.9 x64; Voila was compiled using Scala 2.12.7. We used a recent checkout of Viper and Z3 4.5.0 x64 (we failed to compile Caper against newer versions of Z3). Each example was verified ten times (on a continuously-running JVM); after removing the highest and lowest measurement, the remaining eight values were averaged. Caper (which

Program	LOC	Stg	Wk	Cpr
`SLock`	15	2.6	2.1	1.4
`TLock`	23	21.8	8.1	2.4
`TLockCl`	16	2.9	2.6	0.5
`CASCtr`	25	3.9	2.7	1.5
`BoundedCtr`	24	8.1	5.1	63.1
`IncDecCtr`	28	4.2	3.1	2.9
`ForkJoin`	16	2.1	1.3	1.0
`ForkJoinCl`	28	2.9	2.3	1.6
`BagStack`	29	29.9	18.0	211.6
`CounterCl`	45	-	5.8	-

Program	Err	Stg	Wk	Cpr
`CASCtr`	L	1.5	1.9	1.5
	P	2.5	1.9	11.2
	C	1.5	1.2	0.5
	R	1.2	1.1	0.3
`TLock`	L	3.9	7.2	2.0
	P	7.2	3.4	2.4
	C	15.6	1.8	0.6
	R	4.1	1.8	0.7
`TLockCl`	P	2.9	2.6	143.4
	C	2.5	2.5	115.5
	R	1.8	1.7	5.0
`BagStack`	L	26.5	17.8	> 600
	P	27.9	17.7	> 600
	C	26.3	17.8	> 600
	R	14.4	9.2	216.6

Figure 4.13.: Timings in seconds for successful (left table) and failing (right table) verification runs; lines of code (LOC) are given for Voila programs and exclude proof annotations. *Stg*/*Wk* denote strong/weak Voila specifications; *Cpr* abbreviates Caper. Programs include spin and ticket locks, counters (*Ctr*), and client programs (*Cl*) using the proven specifications. Errors (*Err*) were seeded in loop invariants (*L*), postconditions (*P*), code (*C*), and region specifications (*R*).

compiles to native code) was measured analogously.

Overall, Voila's verification times are good; most examples verify in under five seconds. Voila is slower than Caper and its logic-specific symbolic execution engine, but it exhibits stable performance for successful and failing runs, which is crucial in the common case that proof outlines are developed interactively, such that the checker is run frequently on incorrect versions. As demonstrated by the error-seeded versions of `TLockCl` and `BagStack`, Caper's performance is less stable.

Another interesting observation is that strong specifications typically do not take significantly longer to verify, although only they require the full spectrum of TaDA ingredients and make use of TaDA's most complex rules, MakeAtomic and UpdateRegion. Notable exceptions are: `BagStack`, where only the strong specification requires sequence theory reasoning; and `TLock` and `BoundedCtr`, whose complex transition systems with many disjunctions significantly increase the workload when verifying atomicity rules such as MakeAtomic.

Automation. Voila's annotation overhead, averaged over the programs with *strong* specifications from Fig. 4.13, is 0.8 lines of proof annotations (not counting declarations and procedure specifications; neither for Caper) per line of code, which demonstrates the high degree of automation Voila achieves. Caper has an average annotation overhead of 0.13 for its programs from Fig. 4.13, but significantly weaker specifications. Verifying only the latter in Voila does not reduce annotation overhead significantly since Voila was designed to support TaDA's strong specifications. The overhead reported for encodings into interactive theorem provers such as Coq [38, 152–154] is typically much higher, ranging between 10 and 20.

Supported TaDA Ingredients. Fig. 4.14 provides an overview of Voila's features, w.r.t. TaDA ingredients and Caper guard algebras [41]. The left column lists TaDA features and to which extent their incur annotation overhead. *None* means that the ingredient does not surface at all in a Voila program. *Once* means that there is a one-time annotation per Voila program, typically in the form of a background declaration such

[38]: Vafeiadis et al. (2013), *Relaxed separation logic: a program logic for C11 concurrency*
[152]: Doko et al. (2017), *Tackling Real-Life Relaxed Concurrency with FSL++*
[153]: Kaiser et al. (2017), *Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris*
[154]: Klein et al. (2009), *seL4: formal verification of an OS kernel*

[41]: Dinsdale-Young et al. (2017), *Caper - Automatic Verification for Fine-Grained Concurrency*

as a region. In contrast, *proc* means that the feature requires a one-time annotation per Voila procedure, typically as part of a procedure specification. Next, *low* means that the feature may result in more than one annotation per procedure: for regions, these are new-region statements (one per newly created region instance), in addition to region declarations. Tracking resources, on the other hand, typically appear in invariants of loops that repeat until an update succeeded. Finally, View shifts incur a *medium* annotation overhead: most standard view shifts are automated by Voila and do not require annotations, but for complex, manually encoded examples, additional annotations may be required. See also Sec. 4.8.

Most of Caper's guard algebras are supported by Voila, and as such, do not incur any additional overhead (the guards themselves must still be mentioned, *e.g.* in specifications). Only counting and sum guards are not directly supported by Voila; they can be encoded, which will require additional annotations. See also Sec. 4.8 for an example of a manually encoded guard algebra.

Ingredient	Annotations	Guard Algebra	Support
Regions	low	Trivial	built-in
Transition systems	once	All-or-nothing	built-in
Triple kinds	proc	Counting	encodable
Interference contexts	proc	Indexed	built-in
Atomicity contexts	none	Product	built-in
Levels	proc	Permissions	built-in
Tracking resources	low	Sum	encodable
Private vs. public	–		
View Shifts	medium		
Stability	none		
Framing	none		

Figure 4.14.: Supported TaDA ingredients, with a classification of the incurred annotation overhead, and Caper guard algebras [41], with a classification of their support. TaDA's combination of public and private assertions in a rule triple is currently not supported by Voila.

4.10. Voila Grammar

This section gives an overview of Voila's grammar, and shows that Voila strongly resembles TaDA, but requires fewer technical details in its annotation language.

Fig. 4.15 (top) shows Voila's core syntax for types t, expressions e, and assertions a. Types t include the type of region identifiers `id`, fractions `frac`, and struct types S. Expressions e include variables x, literals l, fields f, and the usual expression operators, *e.g.* relational ones. They also include variable binders $?x$, which are allowed in only two places: the right-hand side of points-to assertions and the last parameter of a region instance, binding the region's abstract state. Assertions a include, besides the usual separation logic assertions, region instances $R(r, \overline{e})$, where R denotes a region name, r a region identifier, and $\overline{e}$ the region's abstract state; the last argument may be omitted when the region state is unspecified. As usual, overlines denote lists. Moreover, assertions include guards $G(\overline{e})@r$, where G denotes a guard name and $\overline{e}$ the guard arguments (guards without arguments are written as $G@r$), and TaDA's two tracking resources. For brevity, we omitted levels, collection data types (*i.e.* sets, sequence, maps, and tuples), and more complex guards (but see also Sec. 4.9 and Sec. 4.8).

$$t ::= \texttt{id} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{frac} \mid S$$
$$e ::= x \mid \texttt{?}x \mid l \mid e \,\texttt{\&\&}\, e \mid e \,\texttt{||}\, e \mid \texttt{!}e \mid e \Rightarrow e \mid e \mathbin{op} e$$
$$a ::= e \mid x.f \mapsto e \mid a \,\texttt{\&\&}\, a \mid e \Rightarrow a \mid R(r, \overline{e}) \mid G(\overline{e})\texttt{@}r \mid r \Mapsto \blacklozenge \mid r \Mapsto (e, e)$$

$$\begin{array}{rcl} ns & ::= & x := e \\ & \mid & \overline{x} := P(\overline{e}) \\ & \mid & \texttt{if}\,(e)\,\{s\}\,\texttt{else}\,\{s\} \\ & \mid & \texttt{while}\,(e)\,\texttt{invariant}\,a\,\{s\} \\ & \mid & s\,;\,s \\ \\ s & ::= & t\,x \mid ns \mid as \end{array} \qquad \begin{array}{rcl} as & ::= & x.f := e \\ & \mid & y := x.f \\ & \mid & \overline{x} := P(\overline{e}) \\ & \mid & \texttt{use\_atomic using}\,R(r, \overline{e})\,\texttt{with}\,G(\overline{e})\texttt{@}r\,\{as\} \\ & \mid & \texttt{make\_atomic using}\,R(r, \overline{e})\,\texttt{with}\,G(\overline{e})\texttt{@}r\,\{s\} \\ & \mid & \texttt{open\_region using}\,R(r, \overline{e})\,\{as\} \\ & \mid & \texttt{update\_region using}\,R(r, \overline{e})\,\{as\} \end{array}$$

$$\texttt{struct}\,S\,\{\overline{t\ f}\}$$

$$\begin{array}{l} \texttt{region}\,R(\texttt{id}\,r, \overline{t\ x}) \\ \quad \texttt{interpretation}\,\{a\} \\ \quad \texttt{state}\,\{e\} \\ \quad \texttt{guards}\,\{\overline{mod\ G(\overline{t\ x})}\} \\ \quad \texttt{actions}\,\{\overline{G(\overline{e})\colon e \rightsquigarrow e}\} \end{array}$$

$$\begin{array}{l} \texttt{abstract\_atomic procedure}\,P(\overline{t\ x}) \\ \quad \texttt{returns}\,(\overline{t\ y}) \\ \quad \texttt{interference ?}x\,\texttt{in}\,e \\ \quad \texttt{requires}\,a \\ \quad \texttt{ensures}\,a \\ \{\overline{s}\} \end{array}$$

Figure 4.15.: Voila's core syntax for (top) types t, expressions e, assertions a, (middle) statements s, atomic statements as, non-atomic statements ns, (bottom) struct, region, and procedure declarations.

Next, Fig. 4.15 (middle) shows Voila's core syntax for statements s, atomic statements as, and non-atomic statements ns. Statements s comprise variable declarations as well as atomic and non-atomic statements; the categorization of the latter follows TaDA. Atomic statements as include field reads and writes, invocations of abstract-atomic procedures, and key rule statements. Following TaDA, rule statements other than `make_atomic` may only nest atomic statements. Non-atomic statements ns are local variable assignments, invocations of non-atomic procedures, and compound statements. For brevity, statements for creating struct and region instances have been omitted, as have ghost statements useful for encoding, *e.g.* complex guard algebras (see Sec. 4.8).

Lastly, Fig. 4.15 (bottom) shows Voila's core syntax for struct, region, and procedure declarations. Structs declare fields and induce homonymous types. Region declarations include name R, identifier r and further formal arguments $\overline{t\ x}$. A region's interpretation and state are an assertion and expression, respectively. Each region may declare guards $G(\overline{t\ x})$, with formal arguments $\overline{x}$ and modifier `unique` or `duplicable`, and actions that describe possible state changes. Abstract-atomic procedure declarations include an interference clause that corresponds to TaDA's interference context. More complex guard and action definitions are omitted for brevity, as are non-atomic and lemma procedures.

4.11. Related Work

We compare Voila to three groups of tools: automated verifiers, focusing on automation; proof checkers, focusing on expressiveness; and proof

outline checkers, designed to strike a balance between automation and expressiveness. Closest to our work in the kind of supported logic is the automated verifier Caper [41], from which we drew inspiration, *e.g.* for how to specify region transition systems. Caper supports an improved version of CAP [31], a predecessor logic of TaDA. Caper's symbolic execution engine achieves an impressive degree of automation, which, for more complex examples, is higher than Voila's. Caper's automation also covers slightly more guard algebras than Voila. However, the automation comes at the price of expressiveness, compared to Voila: postconditions are often significantly weaker because the logic does not support linearizability (or any other notion of abstract atomicity). E.g., Caper cannot prove that the spinlock's `unlock` procedure actually releases the lock. As was shown in Sec. 4.9, Caper is typically faster than Voila, but exhibits less stable performance when a program or its specifications are wrong.

[41]: Dinsdale-Young et al. (2017), *Caper - Automatic Verification for Fine-Grained Concurrency*

[31]: Dinsdale-Young et al. (2010), *Concurrent Abstract Predicates*

Other automated verifiers for fine-grained concurrency reasoning are SmallfootRG [42], which can prove memory safety, but not functional correctness, and CAVE [43], which can prove linearizability, but cannot reason about non-linearizable code (which TaDA and Voila can). VerCors [155] combines a concurrent separation logic with process-algebraic specifications; special program annotations are used to relate concrete program operations to terms in the abstract process algebra model. Reasoning about the resulting term sequences is automated via model checking, but is non-modular. Summers et al. [156] present an automated verifier for the RSL family of logics [38, 152, 157] for reasoning about weak-memory concurrency. Their tool also encodes into Viper and requires very few annotations because proofs in the RSL logics are more stylized than in TaDA.

[42]: Calcagno et al. (2007), *Modular Safety Checking for Fine-Grained Concurrency*

[43]: Vafeiadis (2010), *Automatically Proving Linearizability*

[155]: Oortwijn et al. (2017), *An Abstraction Technique for Describing Concurrent Program Behaviour*

[156]: Summers et al. (2018), *Automating Deductive Verification for Weak-Memory Programs*

[38]: Vafeiadis et al. (2013), *Relaxed separation logic: a program logic for C11 concurrency*

[152]: Doko et al. (2017), *Tackling Real-Life Relaxed Concurrency with FSL++*

[157]: Doko et al. (2016), *A Program Logic for C11 Memory Fences*

A variety of complex separation logics [35, 37, 70, 158–160] are supported by proof checkers, typically via Coq encodings. As discussed in the introduction, such tools strike a different trade-off than proof outline checkers: they provide foundational proofs, but typically offer little automation, which hampers experimenting with logics. Diaframe [10] introduces a custom proof search strategy for Iris [36], achieving foundational proofs and a high degree of automation. This strategy applies rules based on the syntactic shape of the verification goal. To improve completeness, users can provide hints that specify how certain goals are split into subgoals.

[35]: Sergey et al. (2015), *Mechanized verification of fine-grained concurrent programs*

[37]: Turon et al. (2014), *GPS: navigating weak memory with ghosts, protocols, and separation*

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

[158]: Nanevski et al. (2014), *Communicating State Transition Systems for Fine-Grained Concurrent Resources*

[159]: Frumin et al. (2018), *ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency*

[160]: Krebbers et al. (2018), *MoSeL: a general, extensible modal framework for interactive proofs in separation logic*

Starling [138] is a proof outline checker and closest to Voila in terms of the overall design, but it focuses on proofs that are *easy* to automate. To achieve this, it uses a simple instantiation of the Views meta-logic [147] as its logic. Starling's logic does not enable the kind of strong, linearizability-based postconditions that Voila can prove (see the discussion of Caper above). Starling generates proof obligations that can be discharged by an SMT solver, or by GRASShopper [134] if the program requires heap reasoning. The parts of an outline that involve the heap must be written in GRASShopper's input language. In contrast, Voila does not expose the underlying system, and users can work on the abstraction level of TaDA.

[10]: Mulder et al. (2022), *Diaframe: automated verification of fine-grained concurrent programs in Iris*

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

[138]: Windsor et al. (2017), *Starling: Lightweight Concurrency Verification with Views*

[147]: Dinsdale-Young et al. (2013), *Views: compositional reasoning for concurrent programs*

VeriFast [11] can be seen as an outline checker for a separation logic with impressive features such as higher-order functions and predicates. It has no dedicated support for fine-grained concurrency, but the developers manually encoded examples such as concurrent stacks and queues. VeriFast favors expressiveness over automation: proofs often require

non-trivial specification adaptations and substantial amounts of ghost code, but the results typically verify quickly.

4.12. Conclusion

We introduced Voila, a novel proof outline checker that supports most of TaDA's features, and achieves a high degree of automation and good performance. This combination enables concise proof outlines with a strong resemblance of TaDA.

Voila is the first deductive verifier that can reason automatically about a procedure's effect at its linearization point, which is essential for a wide range of concurrent programs. Earlier work either proves much weaker properties (the preservation of basic data structure invariants rather than the functional behavior of procedures) or requires substantially more user input (entire proofs rather than concise outlines).

We believe that our systematic approach to developing Voila can be generalized to other complex logics. In particular, encoding proof outlines into an existing verification framework allows one to develop proof outline checkers efficiently, without developing custom proof search algorithms. Our work also illustrates that an intermediate verification language such as Viper is suitable for encoding a highly-specialized program logic such as TaDA. During the development of Voila, we uncovered and fixed several soundness and modularity issues in TaDA, which the original authors acknowledged and had partly not been aware of. We view this as anecdotal evidence of the benefits of tool support that we described in the introduction.

Voila supports the vast majority of TaDA's features; most of the others can be supported with additional annotations. The main exception are TaDA's hybrid assertions, which combine atomic and non-atomic behavior. Adding support for those is future work. Other plans include an extension of the supported logic, *e.g.* to handle extensions of TaDA [150, 161].

[150]: Rocha Pinto et al. (2016), *Modular Termination Verification for Non-blocking Concurrency*
[161]: D'Osualdo et al. (2019), *TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs*

5. Conclusion

"Now I will have less distraction."
— Leonhard Euler

In this thesis, we have proposed techniques to automate the verification of advanced logics with complex proof state and proof rules and we have proposed techniques to specify, verify, and validate language-agnostic security policies.

We introduced Gobra, the first modular verifier for Go supporting a significant subset of the language, including channels, goroutines, heap-manipulating constructs, closures, and interfaces. Gobra is the first verifier to support Go's interfaces with structural subtyping. Our proposed specification of interfaces is expressive and flexible enough to also improve the verification of closures and security policies. Since Gobra's initial publication, Gobra has been used in other works [50, 56, 162] to verify several case studies, most notably, the implementation of the WireGuard VPN key exchange protocol and the implementation of a full-fledged network router. These case studies demonstrate that Gobra is expressive and performant enough to verify large-scale real-world code. Gobra incorporates an augmented type system to reduce permission-based reasoning and the slicing of verification conditions to improve its performance. Without these techniques, the verification of these case studies would not have been possible.

[50]: Pereira et al. (2024), *Protocols to Code: Formal Verification of a Next-Generation Internet Router*
[56]: Arquint et al. (2023), *Sound Verification of Security Protocols: From Design to Interoperable Implementations*
[162]: Arquint et al. (2023), *A Generic Methodology for the Modular Verification of Security Protocol Implementations*

We introduced a novel policy framework for the specification, verification, and validation of language-agnostic security policies. We achieve a language-agnostic specification and validation of policies by defining them on top of I/O behavior with the addition of declassification actions. Using this abstraction of program behavior, we are able to reason about the guarantees provided by policies based only on a policy itself, in particular, without facing the complexity of the programming languages with which programs are written. To demonstrate these benefits of language-agnostic policies, we introduce a reasoning technique to prove GNIV, a non-deducibility property ensuring that for passive and certain active attackers, a selection of data remains confidential even in the presence of declassification. For code verification, we present how to verify that a program satisfies a security policy using a combination of standard code verification techniques. As a result, we are able to use off-the-shelf automated program verifiers to verify code. We have applied our security policy framework to Gobra to verify real-world programs.

Lastly, we introduced Voila, a systematically developed novel proof outline checker for the TaDA logic. Voila supports reasoning about linearizability and abstract atomicity while simultaneously achieving a high degree of automation. Our work goes beyond existing proof outline checkers and automated verifiers by supporting the substantially more complex program logic TaDA. We believe that our systematic approach to developing Voila can be generalized to other complex logics. Encoding proof outlines into an existing verification language makes it possible to develop automated program verifiers without developing custom proof search algorithms. Our work further demonstrates that an intermediate verification language such as Viper is suitable for encoding highly-specialized program logics such as TaDA.

Appendix

Full TaDA Proof for the Spin Lock | A.

Fig. A.1 shows the full TaDA proof of the TaDA proof outline from Fig. 4.1. The purpose of the figure is to illustrate the complexity of full TaDA proofs. We do not expect readers to be able to understand the proof. All parts of the proof that are present in the TaDA proof outline are colored in blue. Everything else is inferred by Voila.

$$\overline{0\mathcal{A} \vdash \forall\!\!\!\forall s \cdot \left\langle \begin{array}{c} v_1 = x * v_2 = 0 * v_3 = 1 \\ * v_1 \mapsto s \end{array} \right\rangle \mathtt{c_{cas}} \left\langle \begin{array}{c} v_1 \mapsto v_3 * s = v_2 * b = 1 \\ \lor v_1 \mapsto s * s \neq v_2 * b = 0 \end{array} \right\rangle} \text{(CAS)}$$

$$\overline{0\mathcal{A} \vdash \forall\!\!\!\forall s \cdot \left\langle \begin{array}{c} v_1 = x * v_2 = 0 * v_3 = 1 \\ * v_1 = x * v_2 = 0 * v_3 = 1 \\ * v_1 \mapsto s \end{array} \right\rangle \mathtt{c_{cas}} \left\langle \begin{array}{c} v_1 = x * v_2 = 0 * v_3 = 1 \\ * (v_1 \mapsto v_3 * s = v_2 * b = 1 \\ \lor v_1 \mapsto s * s \neq v_2 * b = 0) \end{array} \right\rangle} \text{(Frame)}$$

$$\overline{0\mathcal{A} \vdash \forall\!\!\!\forall s \cdot \left\langle \begin{array}{c} \exists v_1, v_2, v_3 \cdot \\ v_1 = x * v_2 = 0 * v_3 = 1 \\ * v_1 = x * v_2 = 0 * v_3 = 1 \\ * v_1 \mapsto s \end{array} \right\rangle \mathtt{c_{cas}} \left\langle \begin{array}{c} \exists v_1, v_2, v_3 \cdot \\ v_1 = x * v_2 = 0 * v_3 = 1 \\ * (v_1 \mapsto v_3 * s = v_2 * b = 1 \\ \lor v_1 \mapsto s * s \neq v_2 * b = 0) \end{array} \right\rangle} \text{(AExists)}$$

$$\overline{0\mathcal{A} \vdash \forall\!\!\!\forall s \cdot \left\langle x \mapsto s \right\rangle \mathtt{c_{cas}} \left\langle \begin{array}{c} x \mapsto 1 * s = 0 * b = 1 \\ \lor x \mapsto s * s \neq 0 * b = 0 \end{array} \right\rangle} \text{(Consequence)}$$

$$\overline{0\mathcal{A} \vdash \forall\!\!\!\forall s \in \{0,1\} \cdot \left\langle x \mapsto s \right\rangle \mathtt{c_{cas}} \left\langle \begin{array}{c} x \mapsto 1 * s = 0 * b = 1 \\ \lor x \mapsto s * s \neq 0 * b = 0 \end{array} \right\rangle} \text{(Substitution)}$$

$$\overline{\lambda\mathcal{A} \vdash \forall\!\!\!\forall s \in \{0,1\} \cdot \left\langle x \mapsto s \right\rangle \mathtt{c_{cas}} \left\langle \begin{array}{c} x \mapsto 1 * s = 0 * b = 1 \\ \lor x \mapsto s * s \neq 0 * b = 0 \end{array} \right\rangle} \text{(AWeakening3)}$$

$$\overline{\lambda\mathcal{A} \vdash \forall\!\!\!\forall s \in \{0,1\} \cdot \left\langle x \mapsto s \right\rangle \mathtt{c_{cas}} \ \exists\!\!\!\exists t \in \{1\} \cdot \left\langle \begin{array}{c} x \mapsto 1 * s = 0 * b = 1 \\ \lor x \mapsto s * s \neq 0 * b = 0 \end{array} \right\rangle} \text{(Substitution)}$$

$$\overline{\lambda\mathcal{A} \vdash \forall\!\!\!\forall s \in \{0,1\} \cdot \left\langle x \mapsto s \right\rangle \mathtt{c_{cas}} \ \exists\!\!\!\exists t \in \{1\} \cdot \left\langle \begin{array}{c} x \mapsto t * s = 0 * b = 1 \\ \lor x \mapsto s * s \neq 0 * b = 0 \end{array} \right\rangle} \text{(Consequence)}$$

$$\overline{\lambda{+}1\mathcal{A}' \vdash \forall\!\!\!\forall s \in \{0,1\} \cdot \left\langle \begin{array}{c} \mathtt{Lock}_r^\lambda(\mathtt{x}, s) \\ * r \Rrightarrow \blacklozenge \end{array} \right\rangle \mathtt{c_{cas}} \ \exists\!\!\!\exists t \in \{1\} \cdot \left\langle \begin{array}{c} \mathtt{Lock}_r^\lambda(\mathtt{x}, t) * s = 0 \\ * b = 1 * r \Rrightarrow (s,t) \\ \lor \mathtt{Lock}_r^\lambda(\mathtt{x}, s) * s \neq 0 \\ * b = 0 * r \Rrightarrow \blacklozenge \end{array} \right\rangle} \text{(UpdateRegion)}$$

$$\overline{l\mathcal{A}' \vdash \forall\!\!\!\forall s \in \{0,1\} \cdot \left\langle \begin{array}{c} \mathtt{Lock}_r^\lambda(\mathtt{x}, s) \\ * r \Rrightarrow \blacklozenge \end{array} \right\rangle \mathtt{c_{cas}} \ \exists\!\!\!\exists t \in \{1\} \cdot \left\langle \begin{array}{c} \mathtt{Lock}_r^\lambda(\mathtt{x}, t) * s = 0 \\ * b = 1 * r \Rrightarrow (s,t) \\ \lor \mathtt{Lock}_r^\lambda(\mathtt{x}, s) * s \neq 0 \\ * b = 0 * r \Rrightarrow \blacklozenge \end{array} \right\rangle} \text{(AWeakening3)}$$

$$\overline{l\mathcal{A}' \vdash \left\langle \begin{array}{c} \exists s \in \{0,1\} \cdot \mathtt{Lock}_r^\lambda(\mathtt{x}, s) \\ * r \Rrightarrow \blacklozenge \end{array} \right\rangle \mathtt{c_{cas}} \ \exists\!\!\!\exists s \in \{0,1\}, t \in \{1\} \cdot \left\langle \begin{array}{c} \mathtt{Lock}_r^\lambda(\mathtt{x}, t) * s = 0 \\ * b = 1 * r \Rrightarrow (s,t) \\ \lor \mathtt{Lock}_r^\lambda(\mathtt{x}, s) * s \neq 0 \\ * b = 0 * r \Rrightarrow \blacklozenge \end{array} \right\rangle} \text{(AExists)}$$

$$\overline{l\mathcal{A}' \vdash \left\langle \begin{array}{c} \exists s \in \{0,1\} \cdot \mathtt{Lock}_r^\lambda(\mathtt{x}, s) \\ * r \Rrightarrow \blacklozenge \end{array} \right\rangle \mathtt{c_{cas}} \ \exists\!\!\!\exists s \in \{0,1\}, t \in \{1\} \cdot \left\langle \mathit{LoopInv} \right\rangle} \text{(Consequence)}$$

$$\overline{l\mathcal{A}' \vdash \langle \exists s \in \{0,1\} \cdot \mathtt{Lock}_r^\lambda(\mathtt{x}, s) * r \Rrightarrow \blacklozenge \rangle \ \mathtt{c_{cas}} \ \langle \mathit{LoopInv} \rangle} \text{(Substitution)}$$

$$\overline{l\mathcal{A}' \vdash \{\exists s \in \{0,1\} \cdot \mathtt{Lock}_r^\lambda(\mathtt{x}, s) * r \Rrightarrow \blacklozenge\} \ \mathtt{c_{cas}} \ \{\mathit{LoopInv}\}} \text{(AWeakening1)}$$

$$\overline{l\mathcal{A}' \vdash \{\mathit{LoopInv} * b = 0\} \ \mathtt{c_{cas}} \ \{\mathit{LoopInv}\}} \text{(Consequence)}$$

$$\overline{l\mathcal{A}' \vdash \{\mathit{LoopInv}\} \ \mathtt{c_{loop}} \ \{\mathit{LoopInv} * \neg(b = 0)\}} \text{(Loop)}$$

$$\overline{l\mathcal{A}' \vdash \{\exists s \in \{0,1\} \cdot \mathtt{Lock}_r^\lambda(\mathtt{x}, s) * r \Rrightarrow \blacklozenge\} \ \mathtt{c_{loop}} \ \{r \Rrightarrow (0,1) * b = 1\}} \text{(Consequence)}$$

$$\overline{l\mathcal{A}' \vdash \{\exists s \in \{0,1\} \cdot \mathtt{Lock}_r^\lambda(\mathtt{x}, s) * r \Rrightarrow \blacklozenge\} \ \mathtt{c_{loop}} \ \{\exists s \in \{0,1\}, t \in \{1\} \cdot s = 0 * r \Rrightarrow (s,t)\}} \text{(Consequence)}$$

$$\overline{l\mathcal{A} \vdash \forall\!\!\!\forall s \in \{0,1\} \cdot \langle \mathtt{Lock}_r^\lambda(\mathtt{x}, s) * [\mathrm{G}]_r \rangle \ \mathtt{c_{loop}} \ \exists\!\!\!\exists t \in \{1\} \cdot \langle \mathtt{Lock}_r^\lambda(\mathtt{x}, \mathtt{t}) * [\mathrm{G}]_r * s = 0 \rangle} \text{(MakeAtomic)}$$

$$\overline{l\mathcal{A} \vdash \forall\!\!\!\forall s \in \{0,1\} \cdot \langle \mathtt{Lock}_r^\lambda(\mathtt{x}, s) * [\mathrm{G}]_r \rangle \ \mathtt{c_{loop}} \ \exists\!\!\!\exists t \in \{1\} \cdot \langle \mathtt{Lock}_r^\lambda(\mathtt{x}, 1) * [\mathrm{G}]_r * s = 0 \rangle} \text{(Consequence)}$$

$$\overline{l\mathcal{A} \vdash \forall\!\!\!\forall s \in \{0,1\} \cdot \langle \mathtt{Lock}_r^\lambda(\mathtt{x}, s) * [\mathrm{G}]_r \rangle \ \mathtt{c_{loop}} \ \langle \mathtt{Lock}_r^\lambda(\mathtt{x}, 1) * [\mathrm{G}]_r * s = 0 \rangle} \text{(Substitution)}$$

$$l\mathcal{A} \vdash \forall\!\!\!\forall s \in \{0,1\} \cdot \langle \mathtt{Lock}_r^\lambda(\mathtt{x}, s) * [\mathrm{G}]_r \rangle \ \mathtt{lock(x)} \ \langle \mathtt{Lock}_r^\lambda(\mathtt{x}, 1) * [\mathrm{G}]_r * s = 0 \rangle \quad \text{(Function)}$$

Figure A.1.: Simplified version of the full TaDA proof for the TaDA proof outline from Fig. 4.1. The parts that are present in the TaDA proof outline are colored blue. The statements $\mathtt{c_{loop}}$ and $\mathtt{c_{cas}}$ are the loop and CAS statement, respectively. The loop invariant *LoopInv* is $\exists s \in \{0,1\} \cdot \mathtt{Lock}_r^\lambda(\mathtt{x}, s) * (r \Rrightarrow (0,1) * b = 1 \lor r \Rrightarrow \blacklozenge * b = 0)$. The atomicity context $\mathcal{A}'$ is $r : s \in \mathit{trans} * \{0,1\}1, \mathcal{A}$. For simplicity, redundant quantifiers are omitted and local variables are not put into the private part of atomic assertions. Furthermore, the proof uses a variation of the MakeAtomic rule that can be derived in TaDA.

Bibliography

[1] Leonardo De Moura and Nikolaj Bjørner. 'Z3: An Efficient SMT Solver'. In: *TACAS*. Springer. 2008, pp. 337–340 (cited on pages 2, 36).

[2] Haniel Barbosa et al. 'cvc5: A Versatile and Industrial-Strength SMT Solver'. In: *TACAS (1)*. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442 (cited on page 2).

[3] Mike Barnett et al. 'Boogie: A Modular Reusable Verifier for Object-oriented Programs'. In: *FMCO*. Vol. 4111. Lecture Notes in Computer Science. Springer-Verlag, 2006, pp. 364–387 (cited on page 2).

[4] Jean-Christophe Filliâtre and Andrei Paskevich. 'Why3 - Where Programs Meet Provers'. In: *ESOP*. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 125–128 (cited on page 2).

[5] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 'Viper: A Verification Infrastructure for Permission-Based Reasoning'. In: *VMCAI*. Vol. 9583. Lecture Notes in Computer Science. Springer, 2016, pp. 41–62 (cited on pages 2, 9, 26, 98, 106, 110).

[6] Petar Maksimovic et al. 'Gillian, Part II: Real-World Verification for JavaScript and C'. In: *CAV (2)*. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 827–850 (cited on page 2).

[7] The Coq consortium. *The Coq proof assistant*. URL: https://coq.inria.fr/ (visited on 01/08/2019) (cited on pages 2, 38).

[8] Michael Sammler et al. 'RefinedC: automating the foundational verification of C code with refined ownership types'. In: *PLDI*. ACM, 2021, pp. 158–174 (cited on pages 2, 3).

[9] Lennard Gäher et al. 'RefinedRust: A Type System for High-Assurance Verification of Rust Programs'. In: *Proc. ACM Program. Lang.* 8.PLDI (2024), pp. 1115–1139 (cited on pages 2, 3).

[10] Ike Mulder, Robbert Krebbers, and Herman Geuvers. 'Diaframe: automated verification of fine-grained concurrent programs in Iris'. In: *PLDI*. New York: ACM, 2022, pp. 809–824 (cited on pages 2, 128).

[11] Bart Jacobs et al. 'VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java'. In: *NASA Formal Methods*. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 41–55 (cited on pages 3, 98, 105, 128).

[12] Jean-Christophe Filliâtre and Claude Marché. 'The Why/Krakatoa/Caduceus Platform for Deductive Program Verification'. In: *CAV*. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 173–177 (cited on page 3).

[13] Florent Kirchner et al. 'Frama-C: A software analysis perspective'. In: *Formal Aspects Comput.* 27.3 (2015), pp. 573–609 (cited on page 3).

[14] Gidon Ernst and Toby Murray. 'SecCSL: Security Concurrent Separation Logic'. In: *CAV (2)*. Vol. 11562. Lecture Notes in Computer Science. Springer, 2019, pp. 208–230 (cited on pages 3, 4, 42, 44, 50, 51, 68, 70).

[15] Ernie Cohen et al. 'Local Verification of Global Invariants in Concurrent Programs'. In: *CAV*. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 480–494 (cited on page 3).

[16] Stefan Blom and Marieke Huisman. 'The VerCors Tool for Verification of Concurrent Programs'. In: *FM*. Vol. 8442. Lecture Notes in Computer Science. Springer, 2014, pp. 127–131 (cited on pages 3, 39).

[17] K. Rustan M. Leino and Peter Müller. 'Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs'. In: *LASER Summer School*. Vol. 6029. Lecture Notes in Computer Science. Springer, 2008, pp. 91–139 (cited on page 3).

[18] Marco Eilers and Peter Müller. 'Nagini: A Static Verifier for Python'. In: *CAV*. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 596–603 (cited on pages 3, 39).

[19] V. Astrauskas et al. 'Leveraging Rust Types for Modular Specification and Verification'. In: *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. Vol. 3. OOPSLA. ACM, 2019, 147:1–147:30 (cited on pages 3, 26, 39).

[20] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 'Creusot: A Foundry for the Deductive Verification of Rust Programs'. In: *ICFEM*. Vol. 13478. Lecture Notes in Computer Science. Springer, 2022, pp. 90–105 (cited on page 3).

[21] Michael R. Clarkson and Fred B. Schneider. 'Hyperproperties'. In: *CSF*. IEEE Computer Society, 2008, pp. 51–65 (cited on pages 3, 44, 53, 73, 93, 94).

[22] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 'Secure information flow by self-composition'. In: *Math. Struct. Comput. Sci.* 21.6 (2011), pp. 1207–1252 (cited on pages 3, 42).

[23] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 'Relational Verification Using Product Programs'. In: *FM*. Vol. 6664. Lecture Notes in Computer Science. Springer, 2011, pp. 200–214 (cited on pages 3, 42, 56).

[24] Heiko Mantel. 'A uniform framework for the formal specification and verification of information flow security'. PhD thesis. Saarland University, Saarbrücken, Germany, 2003 (cited on pages 3, 44, 92, 93).

[25] Joseph A. Goguen and José Meseguer. 'Unwinding and Inference Control'. In: *S&P*. IEEE Computer Society, 1984, pp. 75–87 (cited on pages 3, 44).

[26] Andrei Popescu, Thomas Bauereiss, and Peter Lammich. 'Bounded-Deducibility Security (Invited Paper)'. In: *ITP*. Vol. 193. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 3:1–3:20 (cited on pages 3, 41, 93, 94).

[27] C. A. R. Hoare. 'An Axiomatic Basis for Computer Programming'. In: *Commun. ACM* 12.10 (1969), pp. 576–580 (cited on page 3).

[28] John C. Reynolds. 'Separation Logic: A Logic for Shared Mutable Data Structures'. In: *LICS*. IEEE Computer Society, 2002, pp. 55–74 (cited on pages 3, 9, 10, 62, 97).

[29] Susan S. Owicki and David Gries. 'An Axiomatic Proof Technique for Parallel Programs I'. In: *Acta Inf.* 6 (1976), pp. 319–340 (cited on pages 3, 97).

[30] Cliff B. Jones. 'Specification and Design of (Parallel) Programs'. In: *IFIP Congress*. 1983, pp. 321–332 (cited on pages 3, 97, 99).

[31] Thomas Dinsdale-Young et al. 'Concurrent Abstract Predicates'. In: *ECOOP*. Vol. 6183. Lecture Notes in Computer Science. New York: Springer, 2010, pp. 504–528 (cited on pages 3, 4, 97, 128).

[32] Kasper Svendsen and Lars Birkedal. 'Impredicative Concurrent Abstract Predicates'. In: *European Symposium on Programming (ESOP)*. Ed. by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. New York: Springer, 2014, pp. 149–168 (cited on pages 3, 97).

[33] Aaron Turon, Derek Dreyer, and Lars Birkedal. 'Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency'. In: *International Conference on Functional Programming (ICFP)*. Ed. by Greg Morrisett and Tarmo Uustalu. New York: ACM, 2013, pp. 377–390 (cited on pages 3, 97).

[34] Azalea Raad, Jules Villard, and Philippa Gardner. 'CoLoSL: Concurrent Local Subjective Logic'. In: *ESOP*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. New York: Springer, 2015, pp. 710–735 (cited on pages 4, 97).

[35] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 'Mechanized verification of fine-grained concurrent programs'. In: *PLDI*. New York: ACM, 2015, pp. 77–87 (cited on pages 4, 97, 128).

[36] Ralf Jung et al. 'Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning'. In: *POPL*. ACM, 2015, pp. 637–650 (cited on pages 4, 60, 122, 128).

[37] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 'GPS: navigating weak memory with ghosts, protocols, and separation'. In: *OOPSLA*. New York: ACM, 2014, pp. 691–707 (cited on pages 4, 97, 128).

[38] Viktor Vafeiadis and Chinmay Narayan. 'Relaxed separation logic: a program logic for C11 concurrency'. In: *OOPSLA*. New York: ACM, 2013, pp. 867–884 (cited on pages 4, 97, 125, 128).

[39] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 'TaDA: A Logic for Time and Data Abstraction'. In: *ECOOP*. Vol. 8586. Lecture Notes in Computer Science. New York: Springer, 2014, pp. 207–231 (cited on pages 4, 6, 97–99, 104, 106).

[40] Stephen Brookes and Peter W. O'Hearn. 'Concurrent separation logic'. In: *SIGLOG News* 3.3 (2016), pp. 47–65 (cited on pages 4, 97).

[41] Thomas Dinsdale-Young et al. 'Caper - Automatic Verification for Fine-Grained Concurrency'. In: *ESOP*. Vol. 10201. Lecture Notes in Computer Science. New York: Springer, 2017, pp. 420–447 (cited on pages 4, 97, 106, 125, 126, 128).

[42] Cristiano Calcagno, Matthew J. Parkinson, and Viktor Vafeiadis. 'Modular Safety Checking for Fine-Grained Concurrency'. In: *SAS*. Vol. 4634. Lecture Notes in Computer Science. New York: Springer, 2007, pp. 233–248 (cited on pages 4, 128).

[43] Viktor Vafeiadis. 'Automatically Proving Linearizability'. In: *CAV*. Vol. 6174. Lecture Notes in Computer Science. New York: Springer, 2010, pp. 450–464 (cited on pages 4, 128).

[44] Toby C. Murray, Robert Sison, and Kai Engelhardt. 'COVERN: A Logic for Compositional Verification of Information Flow Control'. In: *EuroS&P*. IEEE, 2018, pp. 16–30 (cited on page 4).

[45] Pengbo Yan and Toby Murray. 'SecRSL: security separation logic for C11 release-acquire concurrency'. In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–26 (cited on page 4).

[46] Marco Eilers, Thibault Dardinier, and Peter Müller. 'CommCSL: Proving Information Flow Security for Concurrent Programs using Abstract Commutativity'. In: *Proc. ACM Program. Lang.* 7.PLDI (2023), pp. 1682–1707 (cited on page 4).

[47] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 'Compositional Non-Interference for Fine-Grained Concurrent Programs'. In: *SP*. IEEE, 2021, pp. 1416–1433 (cited on page 4).

[48] David Sutherland. 'A model of information'. In: *Proceedings of the 9th national computer security conference*. Vol. 247. Washington, DC. 1986, pp. 175–183 (cited on pages 5, 93).

[49] Xin Zhang et al. 'SCION: Scalability, control, and isolation on next-generation networks'. In: *IEEE Symposium on Security and Privacy*. IEEE. 2011, pp. 212–227 (cited on pages 6, 9, 38, 39).

[50] João C. Pereira et al. 'Protocols to Code: Formal Verification of a Next-Generation Internet Router'. In: *CoRR* abs/2405.06074 (2024) (cited on pages 6, 7, 9, 38, 39, 131).

[51] Felix A. Wolf et al. '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 pages 7, 10, 42, 44).

[52] F. A. Wolf and P. Müller. 'Verifiable Security Policies for Distributed Systems'. In: *Computer and Communications Security (CCS)*. Association for Computing Machinery, 2024, pp. 4–18 (cited on page 7).

[53] Felix A. Wolf, Malte Schwerhoff, and Peter Müller. 'Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA'. In: *FM*. Vol. 13047. Lecture Notes in Computer Science. Springer, 2021, pp. 407–426 (cited on pages 7, 98).

[54] Felix A. Wolf, Malte Schwerhoff, and Peter Müller. 'Concise outlines for a complex logic: a proof outline checker for TaDA'. In: *Formal Methods Syst. Des.* 61.1 (2022), pp. 110–136 (cited on pages 7, 98).

[55] Christoph Sprenger et al. '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 7, 43, 47, 48, 54, 55, 83, 91).

[56] Linard Arquint et al. 'Sound Verification of Security Protocols: From Design to Interoperable Implementations'. In: *SP*. IEEE, 2023, pp. 1077–1093 (cited on pages 7, 37, 43, 47, 55, 83, 86, 131).

[57] Julien Lange et al. 'Fencing off Go: liveness and safety for channel-based programming'. In: (2017), pp. 748–761 (cited on pages 9, 38).

[58] Julien Lange et al. 'A static verification framework for message passing in Go using behavioural types'. In: *ICSE*. ACM, 2018, pp. 1137–1148 (cited on pages 9, 38).

[59] Tej Chajed et al. 'Verifying concurrent, crash-safe systems with Perennial'. In: *SOSP*. ACM, 2019, pp. 243–258 (cited on pages 9, 38).

[60] Felix A. Wolf et al. *Gobra: Modular Specification and Verification of Go Programs*. Version v1.0.0. 2021. DOI: 10.5281/zenodo.4716664 (cited on page 9).

[61] John Boyland. 'Checking Interference with Fractional Permissions'. In: *SAS*. Ed. by Radhia Cousot. Vol. 2694. Lecture Notes in Computer Science. Springer, 2003, pp. 55–72 (cited on pages 10, 115, 122).

[62] Barbara Liskov and Jeannette M. Wing. 'A Behavioral Notion of Subtyping'. In: *ACM Trans. Program. Lang. Syst.* 16.6 (1994), pp. 1811–1841 (cited on page 11).

[63] Felix A. Wolf et al. 'Gobra: Modular Specification and Verification of Go Programs (extended version)'. In: *CoRR* abs/2105.13840 (2021) (cited on pages 16, 17).

[64] Matthew Parkinson and Gavin Bierman. 'Separation logic and abstraction'. In: *ACM SIGPLAN Notices* 40.1 (2005), pp. 247–258 (cited on page 28).

[65] Mark D. Weiser. 'Program Slicing'. In: *IEEE Trans. Software Eng.* 10.4 (1984), pp. 352–357 (cited on page 33).

[66] Susan Horwitz, Thomas W. Reps, and David W. Binkley. 'Interprocedural Slicing Using Dependence Graphs'. In: *PLDI*. ACM, 1988, pp. 35–46 (cited on page 33).

[67] Michał Moskal. 'Programming with triggers'. In: *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. SMT '09. Association for Computing Machinery, 2009, pp. 20–29 (cited on page 34).

[68] Jason A. Donenfeld. *Go Implementation of WireGuard*. `https://git.zx2c4.com/wireguard-go`. [Online; accessed 11-March-2021] (cited on pages 37, 83).

[69] Simon Meier et al. 'The TAMARIN Prover for the Symbolic Analysis of Security Protocols'. In: *CAV*. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 696–701 (cited on pages 37, 83, 86).

[70] Ralf Jung et al. 'Iris from the ground up: A modular foundation for higher-order concurrent separation logic'. In: *J. Funct. Program.* 28 (2018), e20 (cited on pages 38, 128).

[71] Julia Gabet and Nobuko Yoshida. 'Static Race Detection and Mutex Safety and Liveness for Go Programs'. In: *ECOOP*. Vol. 166. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 4:1–4:30 (cited on page 38).

[72] Hans Hüttel et al. 'Foundations of Session Types and Behavioural Contracts'. In: *ACM Comput. Surv.* 49.1 (2016), pp. 1–36 (cited on page 38).

[73] Sjoerd Cranen et al. 'An Overview of the mCRL2 Toolset and Its Recent Advances'. In: *TACAS*. Vol. 7795. Lecture Notes in Computer Science. Springer. 2013, pp. 199–213 (cited on page 38).

[74] Jules Villard, Étienne Lozes, and Cristiano Calcagno. 'Proving Copyless Message Passing'. In: *ASPLAS*. Springer. 2009, pp. 194–209 (cited on page 39).

[75] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 'Actris: Session-type based reasoning in separation logic'. In: *Proc. ACM Program. Lang.* 4.POPL (2019), pp. 1–30 (cited on page 39).

[76] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 'Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic'. In: *arXiv preprint arXiv:2010.15030* (2020) (cited on page 39).

[77] Anton Lorenzen et al. 'Oxidizing OCaml with Modal Memory Management'. In: *Proc. ACM Program. Lang.* 8.ICFP (2024), TBA (cited on page 39).

[78] Daniel Schoepe, Toby Murray, and Andrei Sabelfeld. 'VERONICA: Expressive and Precise Concurrent Information Flow Security'. In: *CSF*. IEEE, 2020, pp. 79–94 (cited on pages 41, 46, 56, 94).

[79] Toby Murray et al. 'Assume but Verify: Deductive Verification of Leaked Information in Concurrent Applications'. In: *CCS*. ACM, 2023, pp. 1746–1760 (cited on pages 41, 46, 56, 92, 94).

[80] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. 'Expressive Declassification Policies and Modular Static Enforcement'. In: *SP*. IEEE Computer Society, 2008, pp. 339–353 (cited on pages 41, 46, 56, 85, 92, 94).

[81] Graeme Smith. 'Declassification Predicates for Controlled Information Release'. In: *ICFEM*. Vol. 13478. Lecture Notes in Computer Science. Springer, 2022, pp. 298–315 (cited on pages 41, 46, 93, 94).

[82] John McLean. 'Proving Noninterference and Functional Correctness Using Traces'. In: *J. Comput. Secur.* 1.1 (1992), pp. 37–58 (cited on pages 42, 50, 93).

[83] A. W. Roscoe. 'CSP and determinism in security modelling'. In: *S&P*. IEEE Computer Society, 1995, pp. 114–127 (cited on pages 42, 50, 93).

[84] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 'Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification'. In: *LFCS*. Vol. 7734. Lecture Notes in Computer Science. Springer, 2013, pp. 29–43 (cited on page 42).

[85] Marco Eilers, Peter Müller, and Samuel Hitz. 'Modular Product Programs'. In: *ESOP*. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 502–529 (cited on pages 42, 50, 56).

[86] Andrei Sabelfeld and Andrew C. Myers. 'A Model for Delimited Information Release'. In: *ISSS*. Vol. 3233. Lecture Notes in Computer Science. Springer, 2003, pp. 174–191 (cited on pages 42, 93).

[87] Steve Zdancewic and Andrew C. Myers. 'Robust Declassification'. In: *CSFW*. IEEE Computer Society, 2001, pp. 15–23 (cited on pages 42, 85, 86, 93).

[88] Wytse Oortwijn and Marieke Huisman. 'Practical Abstractions for Automated Verification of Message Passing Concurrency'. In: *IFM*. Vol. 11918. Lecture Notes in Computer Science. Springer, 2019, pp. 399–417 (cited on pages 43, 45).

[89] Willem Penninckx, Bart Jacobs, and Frank Piessens. 'Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs'. In: *ESOP*. Vol. 9032. Lecture Notes in Computer Science. Springer, 2015, pp. 158–182 (cited on pages 45, 48, 83).

[90] Li-yao Xia et al. 'Interaction trees: representing recursive and impure programs in Coq'. In: *Proc. ACM Program. Lang.* 4.POPL (2020), 51:1–51:32 (cited on pages 45, 48).

[91] David A. Naumann. 'From Coupling Relations to Mated Invariants for Checking Information Flow'. In: *ESORICS*. Vol. 4189. Lecture Notes in Computer Science. Springer, 2006, pp. 279–296 (cited on page 47).

[92] Geoffrey Smith and Dennis M. Volpano. 'Secure Information Flow in a Multi-Threaded Imperative Language'. In: *POPL*. ACM, 1998, pp. 355–364 (cited on pages 49, 93).

[93] Andrei Sabelfeld and David Sands. 'Probabilistic Noninterference for Multi-Threaded Programs'. In: *CSFW*. IEEE Computer Society, 2000, pp. 200–214 (cited on pages 49, 56).

[94] Marco Eilers, Severin Meier, and Peter Müller. 'Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security'. In: *CAV (1)*. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 718–741 (cited on pages 50, 56).

[95] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. 'Information-Flow Security for Interactive Programs'. In: *CSFW*. IEEE Computer Society, 2006, pp. 190–201 (cited on page 50).

[96] Mark Dermot Ryan and Ben Smyth. 'Applied pi calculus'. In: *Formal Models and Techniques for Analyzing Security Protocols*. Vol. 5. Cryptology and Information Security Series. IOS Press, 2011, pp. 112–142 (cited on page 50).

[97] Jean Goubault-Larrecq, Catuscia Palamidessi, and Angelo Troina. 'A Probabilistic Applied Pi-Calculus'. In: *APLAS*. Vol. 4807. Lecture Notes in Computer Science. Springer, 2007, pp. 175–190 (cited on page 50).

[98] Aslan Askarov et al. 'Termination-Insensitive Noninterference Leaks More Than Just a Bit'. In: *ESORICS*. Vol. 5283. Lecture Notes in Computer Science. Springer, 2008, pp. 333–348 (cited on page 53).

[99] Toby C. Murray, Robert Sison, and Kai Engelhardt. 'COVERN: A Logic for Compositional Verification of Information Flow Control'. In: *EuroS&P*. IEEE, 2018, pp. 16–30 (cited on page 56).

[100] Geoffrey Smith. 'Principles of Secure Information Flow Analysis'. In: *Malware Detection*. Vol. 27. Advances in Information Security. Springer, 2007, pp. 291–307 (cited on page 56).

[101] Serdar Tasiran, Ali Sezgin, and Shaz Qadeer. 'Verifying Optimistic Concurrency: Prophecy Variables and Backward Reasoning'. In: *Design and Validation of Concurrent Systems*. Vol. 09361. Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2009 (cited on page 67).

[102] Ralf Jung et al. 'The future is ours: prophecy variables in separation logic'. In: *Proc. ACM Program. Lang.* 4.POPL (2020), 45:1–45:32 (cited on page 67).

[103] Felix A. Wolf and Peter Müller. *Evaluation Examples for Policy Framework*. URL: https://polybox.ethz.ch/index.php/s/JvsszFa2ie2ytxv (visited on 08/08/2024) (cited on page 83).

[104] Aslan Askarov and Andrew C. Myers. 'A Semantic Framework for Declassification and Endorsement'. In: *ESOP*. Vol. 6012. Lecture Notes in Computer Science. Springer, 2010, pp. 64–84 (cited on pages 85, 86, 94).

[105] Ken Biba. 'Integrity Considerations for Secure Computer Systems'. In: (Apr. 1977), p. 68 (cited on page 86).

[106] Benedikt Schmidt et al. 'Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties'. In: *CSF*. IEEE Computer Society, 2012, pp. 78–94 (cited on page 86).

[107] Sebastian Mödersheim and Georgios Katsoris. 'A Sound Abstraction of the Parsing Problem'. In: *CSF*. IEEE Computer Society, 2014, pp. 259–273 (cited on page 91).

[108] Tahina Ramananandro et al. 'EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats'. In: *USENIX Security Symposium*. USENIX Association, 2019, pp. 1465–1482 (cited on page 91).

[109] Andrei Sabelfeld and Andrew C. Myers. 'Language-based information-flow security'. In: *IEEE J. Sel. Areas Commun.* 21.1 (2003), pp. 5–19 (cited on page 92).

[110] Andrei Sabelfeld and David Sands. 'Dimensions and Principles of Declassification'. In: *CSFW*. IEEE Computer Society, 2005, pp. 255–269 (cited on page 92).

[111] Aslan Askarov and Andrei Sabelfeld. 'Gradual Release: Unifying Declassification, Encryption and Key Release Policies'. In: *S&P*. IEEE Computer Society, 2007, pp. 207–221 (cited on page 92).

[112] Andrey Chudnov and David A. Naumann. 'Assuming You Know: Epistemic Semantics of Relational Annotations for Expressive Flow Policies'. In: *CSF*. IEEE Computer Society, 2018, pp. 189–203 (cited on page 92).

[113] Musard Balliu, Mads Dam, and Gurvan Le Guernic. 'Epistemic temporal logic for information flow security'. In: *PLAS*. ACM, 2011, p. 6 (cited on page 92).

[114] Niklas Broberg, Bart van Delft, and David Sands. 'The Anatomy and Facets of Dynamic Policies'. In: *CSF*. IEEE Computer Society, 2015, pp. 122–136 (cited on page 92).

[115] Aslan Askarov and Stephen Chong. 'Learning is Change in Knowledge: Knowledge-Based Security for Dynamic Policies'. In: *CSF*. IEEE Computer Society, 2012, pp. 308–322 (cited on page 92).

[116] Aslan Askarov and Andrei Sabelfeld. 'Tight Enforcement of Information-Release Policies for Dynamic Languages'. In: *CSF*. IEEE Computer Society, 2009, pp. 43–59 (cited on page 92).

[117] Heiko Mantel, David Sands, and Henning Sudbrock. 'Assumptions and Guarantees for Compositional Noninterference'. In: *CSF*. IEEE Computer Society, 2011, pp. 218–232 (cited on page 93).

[118] Aslan Askarov and Andrei Sabelfeld. 'Localized delimited release: combining the what and where dimensions of information release'. In: *PLAS*. ACM, 2007, pp. 53–60 (cited on pages 93, 94).

[119] Rayna Dimitrova et al. 'Model Checking Information Flow in Reactive Systems'. In: *VMCAI*. Vol. 7148. Lecture Notes in Computer Science. Springer, 2012, pp. 169–185 (cited on page 93).

[120] Michael R. Clarkson et al. 'Temporal Logics for Hyperproperties'. In: *POST*. Vol. 8414. Lecture Notes in Computer Science. Springer, 2014, pp. 265–284 (cited on page 93).

[121] Mika Cohen et al. 'Abstraction in model checking multi-agent systems'. In: *AAMAS (2)*. IFAAMAS, 2009, pp. 945–952 (cited on page 93).

[122] Christoph Baumann et al. 'On Compositional Information Flow Aware Refinement'. In: *CSF*. IEEE, 2021, pp. 1–16 (cited on pages 93, 94).

[123] Gilles Barthe, Salvador Cavadini, and Tamara Rezk. 'Tractable Enforcement of Declassification Policies'. In: *CSF*. IEEE Computer Society, 2008, pp. 83–97 (cited on page 94).

[124] Niklas Broberg and David Sands. 'Flow Locks: Towards a Core Calculus for Dynamic Flow Policies'. In: *ESOP*. Vol. 3924. Lecture Notes in Computer Science. Springer, 2006, pp. 180–196 (cited on page 94).

[125] Niklas Broberg and David Sands. 'Flow-sensitive semantics for dynamic information flow policies'. In: *PLAS*. ACM, 2009, pp. 101–112 (cited on page 94).

[126] Jan Menz et al. 'Compositional Security Definitions for Higher-Order Where Declassification'. In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023), pp. 406–433 (cited on page 94).

[127] Andrei Popescu, Peter Lammich, and Thomas Bauereiss. 'CoCon: A Confidentiality-Verified Conference Management System'. In: *Arch. Formal Proofs* 2021 (2021) (cited on page 95).

[128] Thomas Bauereiss and Andrei Popescu. 'CoSMed: A confidentiality-verified social media platform'. In: *Arch. Formal Proofs* 2021 (2021) (cited on page 95).

[129] Thomas Bauereiß et al. 'CoSMeDis: A Distributed Social Media Platform with Formally Verified Confidentiality Guarantees'. In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 729–748 (cited on page 95).

[130] 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. New York: Springer, 2001, pp. 1–19 (cited on page 97).

[131] Peter W. O'Hearn. 'Resources, Concurrency and Local Reasoning'. In: *CONCUR*. Vol. 3170. Lecture Notes in Computer Science. Springer, 2004, pp. 49–67 (cited on page 97).

[132] Stephen D. Brookes. 'A Semantics for Concurrent Separation Logic'. In: *CONCUR*. Vol. 3170. Lecture Notes in Computer Science. Springer, 2004, pp. 16–34 (cited on page 97).

[133] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 'Smallfoot: Modular Automatic Assertion Checking with Separation Logic'. In: *FMCO*. Vol. 4111. Lecture Notes in Computer Science. New York: Springer, 2005, pp. 115–137 (cited on page 97).

[134] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 'GRASShopper - Complete Heap Verification with Mixed Specifications'. In: *TACAS*. Vol. 8413. Lecture Notes in Computer Science. New York: Springer, 2014, pp. 124–139 (cited on pages 97, 128).

[135] Susan S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. Outstanding Dissertations in the Computer Sciences. New York: Garland Publishing, 1975 (cited on pages 97, 105).

[136] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. New York: Springer, 2009 (cited on page 97).

[137] Benjamin C. Pierce et al. *Programming Language Foundations*. Software Foundations series, volume 2. Pennsylvania: Electronic textbook, 2018 (cited on page 97).

[138] Matt Windsor et al. 'Starling: Lightweight Concurrency Verification with Views'. In: *CAV*. Vol. 10426. Lecture Notes in Computer Science. New York: Springer, 2017, pp. 544–569 (cited on pages 98, 105, 128).

[139] Pedro da Rocha Pinto. 'Reasoning with time and data abstractions'. PhD thesis. Imperial College London, UK, 2016 (cited on pages 98, 106–108, 121).

[140] Felix A. Wolf, Malte Schwerhoff, and Peter Müller. *Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA*. Version 1.0.0. 2021. DOI: 10.5281/zenodo.5137791 (cited on page 98).

[141] Felix A. Wolf, Malte Schwerhoff, and Peter Müller. *The Voila source repository*. URL: https://github.com/viperproject/voila (visited on 07/26/2021) (cited on page 98).

[142] Felix A. Wolf, Malte Schwerhoff, and Peter Müller. 'Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA (Full Paper)'. In: *CoRR* abs/2010.07080 (2020) (cited on pages 98, 99, 104).

[143] Matthew J. Parkinson and Gavin M. Bierman. 'Separation logic and abstraction'. In: *POPL*. ACM, 2005, pp. 247–258 (cited on pages 99, 110).

[144] Maurice Herlihy and Jeannette M. Wing. 'Linearizability: A Correctness Condition for Concurrent Objects'. In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 463–492 (cited on page 100).

[145] Arjan J. Mooij and Wieger Wesselink. 'Incremental Verification of Owicki/Gries Proof Outlines Using PVS'. In: *International Conference on Formal Engineering Methods (ICFEM)*. Ed. by Kung-Kiu Lau and Richard Banach. Vol. 3785. Lecture Notes in Computer Science. New York: Springer, 2005, pp. 390–404 (cited on page 105).

[146] K. Rustan M. Leino. 'Dafny: An Automatic Program Verifier for Functional Correctness'. In: *LPAR (Dakar)*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370 (cited on page 105).

[147] Thomas Dinsdale-Young et al. 'Views: compositional reasoning for concurrent programs'. In: *POPL*. New York: ACM, 2013, pp. 287–300 (cited on pages 107, 128).

[148] 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 page 110).

[149] M. J. Parkinson and A. J. Summers. 'The Relationship Between Separation Logic and Implicit Dynamic Frames'. In: *Logical Methods in Computer Science* 8.3:01 (2012), pp. 1–54 (cited on page 110).

[150] Pedro da Rocha Pinto et al. 'Modular Termination Verification for Non-blocking Concurrency'. In: *ESOP*. Vol. 9632. Lecture Notes in Computer Science. New York: Springer, 2016, pp. 176–201 (cited on pages 113, 129).

[151] R. Kent Treiber. *Systems Programming: Coping with Parallelism*. Tech. rep. RJ 5118. IBM Almaden Research Center, 1986 (cited on page 124).

[152] Marko Doko and Viktor Vafeiadis. 'Tackling Real-Life Relaxed Concurrency with FSL++'. In: *ESOP*. Vol. 10201. Lecture Notes in Computer Science. New York: Springer, 2017, pp. 448–475 (cited on pages 125, 128).

[153] Jan-Oliver Kaiser et al. 'Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris'. In: *ECOOP*. Vol. 74. LIPIcs. Wadern: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 17:1–17:29 (cited on page 125).

[154] Gerwin Klein et al. 'seL4: formal verification of an OS kernel'. In: *SOSP*. New York: ACM, 2009, pp. 207–220 (cited on page 125).

[155] Wytse Oortwijn et al. 'An Abstraction Technique for Describing Concurrent Program Behaviour'. In: *VSTTE*. Vol. 10712. Lecture Notes in Computer Science. New York: Springer, 2017, pp. 191–209 (cited on page 128).

[156] Alexander J. Summers and Peter Müller. 'Automating Deductive Verification for Weak-Memory Programs'. In: *TACAS (1)*. Vol. 10805. Lecture Notes in Computer Science. New York: Springer, 2018, pp. 190–209 (cited on page 128).

[157] Marko Doko and Viktor Vafeiadis. 'A Program Logic for C11 Memory Fences'. In: *VMCAI*. Vol. 9583. Lecture Notes in Computer Science. New York: Springer, 2016, pp. 413–430 (cited on page 128).

[158] Aleksandar Nanevski et al. 'Communicating State Transition Systems for Fine-Grained Concurrent Resources'. In: *ESOP*. Vol. 8410. Lecture Notes in Computer Science. New York: Springer, 2014, pp. 290–310 (cited on page 128).

[159] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 'ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency'. In: *LICS*. New York: ACM, 2018, pp. 442–451 (cited on page 128).

[160] Robbert Krebbers et al. 'MoSeL: a general, extensible modal framework for interactive proofs in separation logic'. In: *PACMPL* 2.ICFP (2018), 77:1–77:30 (cited on page 128).

[161] Emanuele D'Osualdo et al. 'TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs'. In: *CoRR* abs/1901.05750 (2019) (cited on page 129).

[162] Linard Arquint et al. 'A Generic Methodology for the Modular Verification of Security Protocol Implementations'. In: *CCS*. ACM, 2023, pp. 1377–1391 (cited on page 131).

Felix **Wolf**

Doctoral Student | Scientific Staff

About me

I am at the very end of my doctorate under the supervision of Peter Müller at ETH Zürich. As part of the VerifiedScion project and Centre for Cyber Trust, my research focused on the verification of real-world security critical software. Within that area, my research interests are the verification of concurrent code and secure information flow. My most recent research is about how to specify and verify security policies for real-world software systems. I have implemented my research results on top of Gobra, a verifier for Go programs that we have developed, where I was the main developer.

Contact

f.alex.wolf@gmail.com

ORCID: 0000-0002-8573-2387

Personal Information

Born on January 2nd, 1995

Born in Frankfurt, Germany

German Nationality

Languages

German - Native Language

English - Professional Knowledge

Programming Languages

Java | Scala | Go

Verification Tools

Viper | Gobra | Verifast | Z3 | Isabelle/HOL

Hobbies

- Board Games
- Hiking

Education

2018-2024 — **Doctorate in Computer Science**
ETH Zürich
Advisor: Peter Müller, Programming Methodology Group

2017-2018 — **MSc Computer Science**
ETH Zürich
Core Subjects: Advanced Compiler Design, Design of Parallel and High-Performance Computing, Security Engineering, Network Security, Formal Methods for Information Security, Probabilistic Artificial Intelligence, Natural Language Understanding
GPA: 5.64/6

2013-2017 — **BSc Computer Science**
ETH Zürich
GPA: 5.48/6

Work Experience

2018-2024 — **Scientific Staff**
ETH Zürich
My tasks were the development of research tools, the coordination of a lecture as head assistant, and the supervision of student projects. As head assistant for introduction to programming, I managed more than 30 teaching assistants for a course with more than 600 students.

2016-2018 — **Teaching Assistant**
ETH Zürich
My tasks were the preparation and teaching of exercise sessions.

A Selection of Supervised Student Projects

- Mihail Gurzu, MSc, Adding Support for Go's Native Interface to a Verifier
- Liming Han, MSc, Extending a Go Verifier with a Linear Typesystem
- Stefano Milizia, MSc, Verification of Closures for Go Programs
- Lino Telschow, MSc, Verifying the IO Behavior of SCION's Border Router
- Fabio Aliberti, BSc, Counterexample Generation in Gobra

Key Publications

Conference Proceedings 2024 — **Verifiable Security Policies for Distributed Systems**, F. A. Wolf, P. Müller, *CCS*, doi 10.1145/3658644.3690303

Conference Proceedings 2023 — **Sound Verification of Security Protocols: From Design to Interoperable Implementations**, L. Arquint, F. A. Wolf, J. Lallemand, R. Sasse, C. Sprenger, S. Wiesner, D. Basin, P. Müller, *S&P*, doi 10.1109/sp46215.2023.10179325

Journal Article 2022 — **Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA**, F. A. Wolf, M. Schwerhoff, P. Müller, *Formal Methods in System Design*, doi 10.1007/s10703-023-00427-w

Conference Proceedings 2021 — **Gobra: Modular Specification and Verification of Go Programs**, F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, P. Müller, *CAV*, doi 10.1007/978-3-030-81685-8_17

Conference Proceedings 2020 — **Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification**, C. Sprenger, T. Klenze, M. Eilers, F. A. Wolf, P Müller, M. Clochard, D. Basin, *OOPSLA*, doi 10.1145/3428220