A Proof-Oriented Approach to Low-Level, High-Assurance Programming

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Department of Electrical and Computer Engineering

Aymeric Fromherz

B.Sc., Computer Science, Ecole Normale Superieure
M.Sc., Computer Science, Ecole Normale Superieure

Carnegie Mellon University
Pittsburgh, PA

December, 2021

Committee Members

Bryan Parno (Co-chair), Carnegie Mellon University

Corina Păsăreanu (Co-chair), Carnegie Mellon University

Nikhil Swamy, Microsoft Research

Derek Dreyer, Max Planck Institute for Software Systems

Acknowledgements

My first thanks go to my advisors, Bryan Parno and Corina Păsăreanu for all their help over the past few years. Bryan has been a steady support over the past few years, always encouraging and present when needed. Corina's help has been invaluable to venture outside my area of expertise, and explore new research directions.

I also want to thank Nikhil Swamy and Derek Dreyer, the other members of my committee. While I did not have the chance to work with Derek, his questions and comments allowed me to identify and clarify several subtleties of my work. I've been lucky to count Nik as a close collaborator throughout my PhD, and to benefit from his knowledge and experience of programming languages and verification.

The work presented in this thesis would not have been possible without the help of many coauthors, and I am indebted to members of the Everest Project for fruitful discussions and technical support. In particular, I would like to thank Jonathan Protzenko, Aseem Rastogi, Tahina Ramananandro and Chris Hawblitzel for their warm welcome during two productive internships at Microsoft Research.

My life in Pittsburgh would not have been the same without many amazing friends surrounding me. Jonathan, Adeline and Melodie welcomed me into their home when I first moved to the US. Janos, Orsi, Josh, and Aaron were some of the first people I met when I arrived in Pittsburgh, and warmly invited me into their group. Mahmood was a fierce squash partner and a great friend, but also a true mentor in my early PhD years. Sruti was one of the best persons to talk to about a variety of topics, and was always down to hang out and complain about the hardships of the PhD life and of job hunting. Camille and Stormy gave me reasons to go outside to hang out at dog parks, and provided a much needed respite when staying at home was suffocating. Matthieu, Marie, Arthur and Susan, the "French mafia", were always eager to discuss fascinating topics ranging from category theory to how children learn language, while also sharing a fair amount of French cheese. Pedro, Sam, Emily, and Klas taught me a lot about American culture and instilled in me a passion for the Pittsburgh Steelers. Ryan, Sydney and Elena were here for me whenever I needed it and patient volunteers with my culinary experiments. Abhiram, Lisa, and Jay were always happy to chat around a cup of tea, and to have a relaxing afternoon playing board games. The Pittsburgh Scottish Country Dancers was one of the nicest groups I met, and allowed me to keep pursuing one of my passions away from home. Thank you so much for these four years, I will miss you all dearly.

I'm also thankful for my friends outside of Pittsburgh who remained an important part of my life while I was abroad. Niols, Martin, thank you for being amazing roommates and for way too many games of Towerfall. I'm excited to live closer to you again. Alice, thank you for remaining such a close friend and for always welcoming me whenever I was back in Paris. Pascaline and Christophe, thank you for sharing your love of music, and for our numerous chats while playing Civilization during the pandemic. I'm glad I got a chance to visit in New Haven before leaving the US. Elizabeth, thank you for inviting me so often to DC and giving me an excuse to get out of Pittsburgh. Denis, having you as a roommate during our internships in Seattle was a lot of fun, and I'm looking forward to being colleagues at Inria. Now that I'm back in France, can I please get my bread privileges back? Amelie, thank you for pushing me outside of my comfort zone and encouraging me to apply for a PhD in the US. Kevin and Claire, thank you for hosting me whenever I was back in Lyon, and for crossing the pond to visit. Going to dance in the woods with you for a weekend was incredible. Ju, Tom, Sarah, Jimmie, Valentin, Hugo and Camille, thank you for many remote games of coinche during the pandemic, which always were a fun distraction. Louise, thank you for all your encouragement and support during these past months. I'm so happy to have met you again. Finally, I would like to thank my family for their unwavering help and love, even when I was far away. I would not have been able to pursue a PhD without it.

The work presented in this thesis was supported in part by the Department of Navy, Office of Naval Research under Grant No. N00014-18-1-2892, two internships at Microsoft Research, a Cylab Presidential Fellowship, and a grant from the Alfred P. Sloan Foundation. Any opinions, findings, and conclusions or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of these sponsors.

Abstract

From autonomous cars to online banking, software nowadays is widely used in safety- and security-critical settings. As the complexity and size of software grows, ensuring that it behaves as a programmer intended becomes increasingly difficult, raising concerns about software's reliability.

To tackle this problem, we wish to provide strong, formal guarantees about the security and correctness of real-world critical software. In this thesis, we therefore advocate for the adoption of a *proof-oriented programming paradigm* in high-assurance software development. We argue that co-developing programs and proofs yields several benefits: the program structure can simplify the proofs, while proofs can simplify programming and improve the software quality too by, for instance, eliminating unneeded checks and cases. To validate this thesis, we rely on two case studies, which we describe next.

Our first case study targets high-performance cryptography, the cornerstone of Internet security. Relying on proof-oriented programming, we develop EverCrypt, a comprehensive collection of verified, high-performance cryptographic functionalities available via a carefully designed API. We first propose a methodology to compose and verify C and assembly cryptographic implementations against shared specifications. We then demonstrate how abstraction and zero-cost generic programming can simplify verification without sacrificing performance, leading to verified cryptographic code whose performance matches or exceeds the best unverified implementations. EverCrypt has been deployed in several high-profile open-source projects such as Mozilla Firefox and the Linux Kernel.

Our second case study investigates the use of proof-oriented programming to develop concurrent, low-level systems. To this end, we present Steel, a novel verification framework based on a higher-order, impredicative concurrent separation logic shallowly embedded in the F* proof assistant. We show how designing Steel with proofs in mind enables us to automatically separate verification conditions between separation-logic predicates and first-order logic encodeable predicates, allowing us to provide practical automation through a mixture of efficient reflective tactics that focus on the former, and SMT solving for the latter. We finally demonstrate the expressiveness and programmability of Steel on a variety of examples, including sequential, self-balancing trees; standard, concurrent data structures such as the Owicki-Gries monotonic counter and Michael and Scott's 2-lock queue; various synchronization primitives such as libraries for spin locks and fork/join parallelism; and a library for message-passing concurrency on dependently typed channels.

Contents

List of Tables

List of Figures

Part I

Overview

Chapter 1

Introduction

From autonomous cars to online banking, software nowadays is widely used in safety- and security-critical settings. As the complexity and size of software grows, ensuring that it behaves as a programmer intended becomes increasingly difficult, raising concerns about software's reliability.

The problem is not new; in 1968, despite computing still being in its infancy, attendees to the first NATO Software Enginnering Conference were already discussing what they called the *software crisis* (Naur and Randell 1969), a term that Dijkstra reused a few years later in his 1972 Turing Award lecture (Dijkstra 1972): The appearence of newer, more powerful machines enabled the use of computer programs in more applications, but also widened the gap between existing software engineering methods and software ambitions, leading David and Fraser to the following statement (Naur and Randell 1969):

> *The gap is arising at a time when the consequences of software failure in all its aspects are becoming increasingly serious. Particularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death, not only for individuals, but also for vehicles carrying hundreds of people and ultimately for nations as well.*

Since this first realization, the meteoric rise of computing power and the proliferation of software and applications has only heightened this issue. First of all, the scale of software has drastically increased; compare for instance the 25,000 lines of code needed to implement the Apollo 11 command module[1] in the 1960s to modern, industrial-grade software, such as the Linux kernel, which commonly contains several millions of lines of code. But in addition to size considerations, software also evolved to leverage hardware advances, enabling for instance the development of concurrent programs operating in distributed environments to provide better performance, at the price of increased complexity.

Guaranteeing the reliability of modern software is notoriously difficult, and dire incidents over the past 50 years abound. One of the most notable examples involves

[1] `https://github.com/chrislgarry/Apollo-11/`

a radiation therapy machine, the Therac-25: Several software defects, in particular related to a tricky interaction between processes running concurrently, led the Therac-25 to deliver massive radiation doses to patients, causing several deaths from radiation poisoning between 1985 and 1987. Such accidents caused by faulty software are not isolated cases. While devoid of the loss of human lives, bugs in the Intel Pentium processor (Edelman 1997) and in Ariane 5's Inertial Reference System (Lions et al. 1996) each led to losses of several hundreds of millions of dollars; in one case, due to Intel recalling and replacing all flawed processors, in the other due to the rocket exploding on its inaugural voyage.

In spite of these high-profile incidents, software bugs were mostly considered a minor annoyance throughout the twentieth century. Outside of safety-critical systems, software bugs could at worst lead to users being temporarily inconvenienced by a program crashing or returning the wrong outputs. In an increasingly connected world, programs now operate in a hostile environment, where bugs become vulnerabilities that attackers (also referred to as hackers by the media) *actively* attempt to exploit. As the amount of data and personal information handled by computers increases and our reliance on software grows, so do the incentives for malicious actors. More than 10 years ago, Franklin et al. (2007) and Moore, Clayton, and Anderson (2009) analyzed the economics of data theft, encapsulating for instance credit card fraud and identity theft, and concluded that determined attackers could reap profits of several million dollars in a year. Since then, the situation has only worsened, as the scale of data breaches heightened. For instance, in 2013, private data (including passwords and credit card information) from more than 150 million Adobe customers was compromised[2], while more recently, in 2020, iPhones were shown vulnerable to a complete take over by hackers in Wi-Fi range[3], potentially leading to massive privacy violations. Exploiting software vulnerabilities from another angle, the past decade has also seen the mushrooming of *ransomware* attacks, where a hacker renders a system inoperational by encrypting its data to extort a ransom out of its victim. By targeting critical infrastructure such as hospitals[4], or city governments[5], ransomware have become a lucrative, billion-dollar industry at the expense of public welfare (Hernandez-Castro et al. 2020).

Despite the threat to public safety, dozens of new vulnerabilities are discovered daily in modern software, as recorded by the National Vulnerability Database (NIST). When software flaws can have dramatic consequences, they become unacceptable. This therefore calls for a shift in existing software development techniques, in order to provide strong, formal guarantees about the correctness and security of real-world critical software.

[2]https://krebsonsecurity.com/2013/10/adobe-breach-impacted-at-least-38-million-users/

[3]https://www.wired.com/story/zero-click-ios-attack-project-zero/

[4]https://www.theguardian.com/society/2017/may/12/global-cyber-attack-nhs-trusts-malware

[5]https://www.nytimes.com/2020/02/09/technology/ransomware-attacks.html

1.1 A Brief History of Formal Verification

Early on, computer scientists realized that, while useful in development processes, testing was not sufficient to prove the correctness of programs: already in 1969, Dijkstra famously stated that "testing shows the presence, not the absence of bugs" (Randell and Buxton 1970). To tackle this problem, Hoare (1969) proposed to apply deductive reasoning to establish whether a program behaves as a user intended. Inspired by ideas from Floyd (1967), Hoare defined a set of inference rules and axioms about program executions, providing a logical foundation for program verification. The core concept of this *Hoare logic* (also called Hoare-Floyd logic) is a judgment, $\{P\}\ C\ \{Q\}$, which, paraphrasing Hoare, states that "if the assertion P is true before initiation of a program C, then the assertion Q will be true on its completion".

While seminal, Hoare's original paper only applied to simple, imperative programs, precluding side-effects, and not considering termination of programs. Hoare himself acknowledged that

> *The formal material presented here has only an expository status and represents only a minute proportion of what remains to be done. It is hoped that many of the fascinating problems involved will be taken up by others.*

Over the following years, Hoare's seminal work became the basis of *formal verification*, and sparked a flurry of extensions to prove a wider range of programs, by supporting more features from existing programming languages. To cite a few examples, Foley and Hoare (1971) added support for recursion to prove the correctness of a quicksort algorithm; Clint and Hoare (1972) extended Hoare logic to support jumps and functions in order to establish the correctness of a table lookup procedure implemented in ALGOL 60; while Hoare (1976) himself extended his work to reason about concurrent programs.

While promising in theory, formal verification long struggled to be adopted outside of a restricted academic circle. A decade after Hoare's paper, De Millo, Lipton, and Perlis (1979) strongly argued against the use and utility of formal verification. Making the parallel with more standard mathematics, part of their criticism centered on the difficulty to check, and hence *trust*, the validity of a proof: proofs of programs were presented as unreadable because of an overwhelming formalism, while also being overly large and verbose; the authors gave the example of a "300-line input/output package" which would require a "20,000-line verification". Outside of strong rebukes from Lamport and Maurer, this position seemed widely shared by the computer science community at the time (ACM FORUM 1979).

Over the rest of the twentieth century, formal methods slowly gained traction for critical applications. Some notable examples include the use of the B language (Abrial et al. 1991) to formally specify and verify a signaling system for commuter trains in Paris (Guiho and Hennebert 1990), the analysis of the shutdown system of a nuclear reactor (Archinoff et al. 1990), and the application of formal methods to digital flight control systems by NASA (Rushby 1992; Rushby and Von Henke 1993). Nevertheless, disbelief in formal methods persisted, leading Hall (1990) and Bowen and Hinchey (1995) to write position papers attempting to dispel common misconceptions, or *myths*, about formal methods, while arguing for the benefits of integrating them in industrial development processes.

To smoothen adoption by everyday programmers, a large fraction of formal methods research focused on fully automated techniques, by developing static analysis tools based on either abstract interpretation (Cousot and Cousot 1977) or symbolic execution (King 1976). Such tools are able to provide a high level of automation to verify properties of real-world programs. One famous example is the Astrée static analyzer (Blanchet et al. 2003; Cousot et al. 2005), which was used to prove the absence of run-time errors in Airbus' flight control systems consisting of hundreds of thousands of lines of C code. Nowadays, static analysis tools exist for most mainstream languages including C (Clarke et al. 2004; Gurfinkel et al. 2015), Java (Păsăreanu and Rungta 2010; Sridharan et al. 2013), Python (Fromherz et al. 2018), and JavaScript (Fragoso Santos et al. 2017), and some of them, such as the Infer static analyzer (Calcagno and Distefano 2011), are fully integrated in industrial development processes at large companies such as Amazon, Facebook, and Microsoft (Calcagno et al. 2015). Unfortunately, there is a catch: to provide scalability and automation, these tools can only reason about a restricted set of properties. In particular, they commonly focus on the absence of crashes in programs, and cannot guarantee *functional correctness*, i.e., that a program satisfies a user-provided specification for all inputs.

Taking a different approach, proof assistants such as Coq (Coq Development Team 2017), Agda (Norell 2008), or Isabelle/HOL (Nipkow et al. 2002) enable a programmer to write strong specifications, encapsulating, for instance, functional correctness. By relying on rich, dependently typed languages, the possibilities offered by proof assistants seem limitless; one can embed existing programming langauges (Jung et al. 2018a; Bond et al. 2017; Kennedy et al. 2013; Krebbers and Wiedijk 2011), encode logics (Bengtson et al. 2012; Jung et al. 2015), and build modular abstractions to reason about a variety of systems, such as processors (Harrison 2000) or software for air traffic management (Dowek et al. 2005), while proving the soundness and correctness of the different logics and applications with respect to the underlying type theory.

Early detractors of program verification claimed that "no major programs [...] whether an automatic program verifier, a compiler, a database management system, or an operating system, would ever be verified by man, woman, child, beast, or machine" (ACM FORUM 1979). In the early 2000s, resounding successes of proof-assistant-based verification would prove them wrong, with the independent development of CompCert (Leroy 2006), a verified C compiler, and of seL4 (Klein et al. 2009), a verified microkernel. Despite being verified, both systems achieve industrial standards, and are at the core of real-world, safety-critical applications: flight-control (Souyris 2014) and power-plant-control software (Kästner et al. 2018) relies on CompCert, while seL4 is used in aerospace and autonomous aviation (Klein et al. 2018). Relying on proof assistants, both projects leveraged *proof-oriented programming*, integrating formal verification in the development process instead of retrofitting verification in an existing system. A *proof-oriented programming language* enables and encourages programs and proofs to be developed in parallel. To the best of our knowledge, this term was first used by (Hoffmann 1978) to describe Lucid, a language that "uses the same denotation for writing and proving properties of programs, thus is, at the same time, a formal proof system and a programming language". More recently, Zinzindohoué-Marsaudon (2018) used the same term to present F*, a general purpose programming language aimed at program verification that we use throughout this thesis. Co-developing programs and proofs yields several benefits. The most obvious one is that structuring programs with proofs in mind simplifies verification: when software development is specification-driven, an implementation can follow the structure of its corresponding specification, simplifying both programming and proving, a style described as "type-define-refine" by Brady (2016) in the context of dependently typed languages, where specifications are expressed as types. But conversely, proofs can also simplify and improve programs, eliminating unneeded checks and cases, while also occasionally leading to design enhancements: for instance, Butler et al. (2010) describe how attempting to formally verify an air traffic separation algorithm in the PVS proof assistant (Owre et al. 1992) led to improvements and simplifications to the design of the algorithm itself.

Unfortunately, echoing some early criticism from De Millo, Lipton, and Perlis (1979), developing and verifying programs using interactive proof assistants remains time-consuming, and requires substantial manual effort. While considered small by industrial standards, consisting of about 10,000 lines of executable code each, CompCert and seL4 respectively required 6 person-years and 150,000 lines of proof (Kästner et al. 2018) and about 22 person-years and 200,000 lines of proof (Klein et al. 2009). With the exception of a few heroic efforts, type-theory-based formal methods have thus been historically hard to apply to large, real-world systems; democratizing their use necessitates proof engineering and automation advances.

1.2 Thesis Overview

In this thesis, we argue that **the proof-oriented programming paradigm is a promising way to develop high-assurance software. Co-developing programs and proofs reduces the proof burden on the developer and opens avenues for domain-specific automation, increasing the scalability of verification and enabling formal guarantees about complex, real-world systems.**

To validate this thesis, we will rely on two case studies, which we describe next. All the work presented in this thesis was performed using the F* proof assistant (Swamy et al. 2016). F* is a higher-order, dependently typed, effectful ML-like language aimed at program verification. Verification in F* mostly relies on the Z3 SMT solver (Moura and Bjørner 2008), but verification conditions can also be discharged using tactics, similar to other interactive proof assistants such as Coq (Coq Development Team 2017) or Lean (Ebner et al. 2017). We give a brief overview of F* in Chapter 3. Although the techniques presented in this thesis are tailored for use in F*, the general methodology and findings are transferrable to other dependently typed proof assistants.

The work presented in this thesis was highly collaborative; at the start of each chapter, we will clarify which contributions are primarily those of the author.

1.2.1 Case Study 1: High-Performance Cryptography

Cryptography is the cornerstone of most security-critical applications, as it can guarantee confidentiality, authenticity, and integrity of data and messages exchanged between different parties. When developing secure software, developers rarely write their own cryptographic code; they instead rely on a cryptographic provider, whose *correctness* and *security* are crucial. However, for most applications (e.g., TLS, cryptocurrencies, or disk encryption), the provider is also on the critical path of the application's *performance*. Consider for instance how secure internet browsing relies on cryptography for each exchanged message. Using a slow cryptographic provider would result in higher latency, which would provide a poor user experience, and limit the development of further applications.

Unfortunately, producing cryptographic code that is fast, correct, and secure has historically been difficult: despite tremendous effort from the security community, even the widely used OpenSSL (OpenSSL Team 2005) is not immune to attacks. Vulnerabilities such as FREAK (Beurdouche et al. 2015) or Heartbleed (National Vulnerability Database 2014) can have catastrophic consequences; once cryptographic code is compromised, entire secure applications fall apart like a house of cards. This makes cryptographic software a prime target for formal verification.

Formally verifying state-of-the-art cryptographic code raises interesting technical challenges. In their quest for performance, modern cryptographic implementations often consist of hybrid programs implemented in C while occasionally calling into hand-tuned assembly routines; this enables leveraging hardware features such as specific instruction sets (Gueron 2012; Gulley et al. 2013), and applying domain-specific optimizations that a general-purpose compiler might not be aware of. To achieve high-performance in a verified cryptographic artifact, we thus need the ability to interoperate between verified C and assembly programs, while provably preserving all security and correctness guarantees in the resulting hybrid program. We show in Chapter 4 how to encode such interoperation into F*. Leveraging ideas from dependently typed generic programming, we structure our approach to provide a clean separation between core, trusted components of the interoperation and untrusted components that are helpful in reducing the proof burden (Fromherz et al. 2019; Protzenko et al. 2020).

We use this verified interoperation at scale to design EverCrypt (Protzenko et al. 2020), a verified, comprehensive, industrial-grade cryptographic provider with support for agility and multiplexing. EverCrypt combines software engineering best practices and formal verification to expose carefully crafted APIs usable by verified and unverified clients alike. Portions of EverCrypt have been deployed in verified implementations of the Signal protocol and of a Merkle Trees library, as well as in a variety of high-profile, open-source projects such as Mozilla Firefox, the Linux Kernel, and the Wireguard VPN. We present EverCrypt in detail in Chapter 5.

By co-developing programs and proofs, we were able to scale up verification to a high-assurance cryptographic provider consisting of about 50,000 lines of executable code, while also confidently and soundly implementing complex optimizations, leading to state-of-the-art performance. The end result is a verified artifact of sufficient quality to be adopted in real-world, industrial settings, thus improving the security and reliability of existing critical systems.

1.2.2 Case Study 2: Concurrent, Low-Level Systems

Beyond cryptographic implementations, modern software increasingly exploits parallelism to reach new heights of performance. Unfortunately, concurrent programming is error-prone, and developers often make incorrect assumptions about how their programs will behave. Using formal methods to provide strong correctness guarantees is appealing, but challenging; verification frameworks either lack the expressivity required to model every advanced low-level pattern found in real-world implementations, or they do not retain the level of automation needed to ensure the scalability of verification. To address this problem, we present Steel, a new verification framework based on Concurrent Separation Logic and targeting verification of low-level, concurrent programs.

We first present Steel's foundations, dubbed SteelCore (Swamy et al. 2020), in Chapter 6. SteelCore relies on a trusted model of concurrent computations, encoded in F* as a tree of computational actions that can be composed sequentially or in parallel. Building upon this model, we derive and prove sound a highly expressive and extensible program logic, based on a shallow embedding of a higher-order, impredicative concurrent separation logic with support for dynamically allocated invariants and partial commutative monoids (PCMs).

Building upon SteelCore, we then describe in Chapter 7 Steel, a verification framework embedded in F*. In particular, we show how designing Steel with proofs in mind enables us to automatically separate verification conditions between separation logic predicates and first-order logic encodable predicates (Fromherz et al. 2021). This allows us to write efficient reflective tactics that focus on the former, while the latter are encoded efficiently to SMT by F*.

To conclude, we demonstrate in Chapter 8 the expressiveness and programmability of the Steel framework by building several verified libraries. We first verify several standard, sequential data structures, including various flavors of linked lists as well as mutable, self-balancing trees. We then show how to embed spin locks and fork/join parallelism in Steel, as well as how to implement a racy, 2-lock concurrent queue first proposed by Michael and Scott (1996). We finally present a novel PCM-based encoding of 2-party dependently typed sessions, and package our encoding as a library for message-passing concurrency on dependently typed channels.

By applying a proof-oriented methodology to the design of Steel itself, we were thus able to offer a rich, expressive logic to reason about complex, concurrent programs while smoothly combining domain-specific automation for separation logic reasoning and efficient SMT-based automation.

1.3 Limitations of Formal Verification

While formal verification can provably rule out entire classes of bugs and vulnerabilities, it cannot guarantee that software is perfect, or even hacker-proof. Paraphrasing Hall

(1990), "the fact is that formal methods are fallible. It ought to be too obvious to need saying, but nothing can achieve perfection".

The main issue is that programs are verified with respect to a *trusted specification*, which is a formal, idealized model of the behavior of the program. While formal verification can ensure that a program is correct *vis-à-vis* its specification, if the specification itself is incorrect, or does not accurately capture the behavior of the underlying hardware or the capabilities of an attacker when considering software security, then formal guarantees do not apply to real-world executions of the verified programs. Several techniques exist to increase confidence in specifications. First, developers should strive to keep specifications small and simple, to make it easier for humans to carefully review and audit them. Furthermore, when specifications are executable, they should be validated by testing them against a wide range of inputs. Many verification projects follow this methodology, such as our EverCrypt cryptographic provider, or a formalization of ARM processors (Fox and Myreen 2010). Another possibility is to apply formal methods directly to specifications (Cremers et al. 2016; Woodcock 1989; Yu et al. 1999) to guarantee that they satisfy properties of interest. Nevertheless, none of these techniques fully guarantee the trustworthiness of specifications.

Additionally, formal verification relies on the soundness of the provers and frameworks employed during verification. Even when the metatheory underlying such tools has been extensively studied (Werner 1994), their implementations can be incorrect, as for any other piece of software. While rarely encountered in practice, critical bugs exist both in proof assistants such as Coq[6] and F*[7], and also in automated theorem provers such as Z3 and CVC4 (Winterer et al. 2020).

Such issues can raise questions about the validity and usefulness of formal verification. Nonetheless, several empirical results suggest that formally verified software can be significantly more reliable and secure than its unverified counterpart. Aiming to improve the quality of C compilers, Yang et al. (2011) developed a test-case generation tool called Csmith, which they used to evaluate widely used compilers such as GCC or Clang, as well as the verified CompCert C compiler previously mentioned. While Csmith discovered more than 325 previously unreported bugs in mainstream compilers, ranging from crashes to wrong results, the only bugs found in CompCert were in unverified parts of the compiler, namely its (at the time) unverified front-end code. This led Yang et al. to conclude that "the apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users".

Going beyond functional correctness, DARPA's HACMS program (Fisher et al. 2017) explored the utility of formally verified software in adversarial, security-critical settings. The HACMS program focused on high-assurance software for vehicles, previously shown to be vulnerable to a wide range of attacks (Koscher et al. 2010; Checkoway et al. 2011). This experiment was split into several phases. First, researchers were asked to retrofit formal verification into existing, commercially available software operating a quadcopter. In the second stage, a redesign of the software allowed researchers to co-develop programs

[6]https://github.com/coq/coq/blob/master/dev/doc/critical-bugs
[7]https://github.com/FStarLang/FStar/issues/1542

and proofs, thus integrating formal methods early in the development process. To evaluate the benefits of formal verification, the study relied on a team of professional penetration testing experts, which was tasked with breaking into the quadcopter. While serious vulnerabilities were discovered in the initial, unverified software, after six weeks, the experts were unable to wirelessly disrupt the operation of the quadcopter with retrofitted verification. More impressively, despite much stronger attacker capabilities, including access to several hardware components, testers were unable to impact any of the flight-critical functionalities in the version of the software which had been entirely refactored, raising high hopes about the proof-oriented programming methodology that we advocate for in this thesis.

Chapter 2

Related Work

In this chapter, we first survey the existing literature most relevant to the work presented in this thesis. Starting with verified cryptography, we first give an overview of works aiming to provide formal guarantees about the correctness and safety of cryptographic implementations (Section 2.1), before delving into security considerations for cryptographic code (Section 2.2). We conclude by providing an overview of verification projects relying on separation logic (Section 2.3), and linking it to our work on concurrent, low-level verification.

2.1 Verified Cryptography

Cryptographic algorithms are designed to provide a wide range of guarantees, e.g., about the authenticity, integrity, and secrecy of messages exchanged between parties, which are crucial in security-critical applications. Unfortunately, cryptographic code does not always match the mathematical model of the algorithm it implements. For instance, an arithmetic bug in the implementation of a modular multiplication in OpenSSL (Brumley et al. 2012) enabled the full recovery of a cryptographic secret key, while a performance optimization in OpenSSL' implementation of the GHASH algorithm led to message forgeries (Gueron and Krasnov 2014). More famously, a memory safety issue in the OpenSSL implementation of the TLS Heartbeat extension led to the high-profile Heartbleed bug (National Vulnerability Database 2014; Durumeric et al. 2014), which could leak secrets stored in memory, including cryptographic private keys, user names and passwords, as well as business critical documents and communications.

To provide strong guarantees about the correctness and security of complex cryptographic code, many research projects thus turned to formal, computer-aided verification. Zinzindohoué, Bartzia, and Bhargavan (2016) develop a verified library of elliptic curves in the F* proof assistant (Swamy et al. 2016), which extracts to executable OCaml code. Unfortunately, their implementation is two orders of magnitude slower than equivalent C code, significantly hampering its adoption. For performance reasons, other projects often target implementations written in low-level languages, such as C or assembly.

To do so, several projects rely on interactive verification in proof assistants such as Coq (Coq Development Team 2017). Notably, several papers verify different cryptographic algorithms, e.g., SHA-256 (Appel 2015), HMAC instantiated with SHA-256 (Beringer et al. 2015), and HMAC-DRBG (Ye et al. 2017) using Coq and the Verified Software Toolchain (Appel 2011), while proving the cryptographic constructions secure using the Foundational Cryptography Framework (Petcher and Morrisett 2015). The resulting C code can then be compiled with the verified CompCert compiler (Leroy 2006), ensuring that the correctness guarantees still hold at the assembly level.

Instead of relying on manual verification, other works instead use semi-automated tools to develop verified cryptographic implementations. For example, Chen et al. (2014) verify the Montgomery Ladderstep of a highly performant, hand-optimized assembly implementation of Curve25519 (Bernstein et al. 2011) using the Coq proof assistant (Coq Development Team 2017) and an SMT solver, while Tsai, Wang, and Yang (2017) and Polyakov et al. (2018) do so using an SMT solver and a computer algebra system. Dockins et al. (2016) use a tool called SAW that relies on symbolic execution and SMT solvers to automatically prove that C and Java implementations of cryptographic algorithms (FFS, AES, the inner loop of SHA-384, ZUC and EDSA) are equivalent to reference implementations written in Cryptol (Lewis and Martin 2003). Lastly, Hawblitzel et al. (2014) rely on the Dafny language (Leino 2010), the Boogie verifier (Barnett et al. 2006), and the Z3 SMT solver (Moura and Bjørner 2008) to verify several cryptographic algorithms (SHA-1, SHA-256, HMAC and RSA signing) in a verifiable assembly language called BoogieX86 (Yang and Hawblitzel 2010), before extracting to executable assembly code using a small, trusted printer.

Also relying on Dafny and Z3 for automation and most relevant to this thesis, Bond et al. (2017) propose a framework called Vale to verify assembly implementations of various cryptographic primitives on several platforms. Using Vale, Bond et al. develop verified versions of SHA-256 on Intel x86 and ARM, of Poly1305 on Intel x64, and of a hardware-accelerated version of AES-CBC on Intel x86. To do so, Vale models assembly semantics using a deep embedding, providing abstract syntax trees (ASTs) representing assembly programs as Dafny values, and automatically generating verification conditions (VCs) ensuring that the program matches a user-supplied specification. Relying on a deep embedding opens the possibility to develop additional analyses by directly operating on the AST; Bond et al. leverage this to implement a verified analyzer which tracks potential timing and memory-access information leakage, which can lead to side-channel attacks. We extended this work (Fromherz et al. 2019), as described in Chapter 4, to reimplement Vale in F*. Leveraging dedicated Intel hardware instructions (Gueron 2012), we then developed a verified, optimized version of AES-GCM (NIST 2007), a cryptographic algorithm used in about 90% of secure web traffic (Mozilla 2018). Bosamiya et al. (2020) have since built upon Vale to automatically derive verified implementations of AES-GCM optimized for a multitude of processor generations using verified program transformations.

Aiming to benefit from both the expressiveness of type-theory-based proof assistants and SMT-based automation, Zinzindohoué et al. (2017), similarly to Zinzindohoué, Bartzia, and Bhargavan (2016) rely on the F* proof assistant to develop HACL*, a collection of verified cryptographic primitives including SHA2 hashes, the Curve25519 and Ed25519 elliptic curves, as well as Chacha20, Poly1305, and the Chacha-Poly construction for Authenticated Encryption with Additional Data (AEAD). But compared to Zinzindohoué, Bartzia, and Bhargavan (2016), HACL⋆ restricts itself to a low-level subset of the F⋆ language, called Low⋆ (Protzenko et al. 2017), which can be extracted to idiomatic, portable C code. A recent extension (Protzenko et al. 2019) of the Low⋆ toolchain also enables verified extraction of HACL⋆ to WebAssembly (Haas et al. 2017a). We later extended the HACL⋆ library to provide optimized, verified implementations relying on architecture-specific SIMD parallelism (Polubelova et al. 2020).

Finally, instead of directly working on low-level C or assembly implementations, other approaches rely on verified compilation to produce verified, low-level artifacts. In particular, Jasmin (Almeida et al. 2017) provides a domain-specific language (DSL) to write high-performance cryptographic code, which combines high-level features such as function calls and structured control flow with assembly-level instructions. The Jasmin language is then extracted to x64 assembly code through a succession of compilation passes which are verified in Coq. Jasmin programs can be automatically translated into Dafny and, more recently (Almeida et al. 2020), into EasyCrypt (Barthe et al. 2011), enabling verification of Jasmin source code. FiatCrypto (Erbsen et al. 2019) also relies on verified compilation passes to transform high-level, generic Coq specifications for bignum arithmetic to efficient C implementations which are specialized for several elliptic curves.

2.2 Security of Cryptographic Implementations

Ensuring that cryptographic implementations are memory safe and functionally correct is a first, crucial step to ensure their security. Unfortunately, it does not rule out all possible attacks against cryptographic code. The security of cryptographic constructions often relies on secrets, such as cryptographic keys that an attacker must not know. Resolute attackers can try to infer these secrets through *side-channel attacks*, where they derive information from the program's behavior.

Examples of such attacks abound in the literature. For instance, both Aciiçmez, Brumley, and Grabher (2010) and Percival (2005) show how to recover cryptographic private keys from observations about cache access patterns; Kocher (1996) and Brumley and Boneh (2003) demonstrate similar attacks by measuring response times to various queries, while Gandolfi, Mourtel, and Olivier (2001) and Masti et al. (2015) respectively exploit variations in electromagnetic power radiation and heat resulting from running processes.

Although *physical* side channels (Gandolfi et al. 2001; Masti et al. 2015) are an issue, *digital* side channels exploitable by a remote attacker (Aciiçmez et al. 2010; Brumley and Boneh 2003) are more concerning. To preclude such attacks, guidelines recommend that cryptographic implementations follow a *constant-time* coding style (Aumasson), where branching and accessing memory based on secrets is prohibited. To justify this recommendation, Barthe et al. (2014) model in Coq concurrent executions of programs in virtualization platforms and prove that, assuming a strong attacker model where an adversary controls the process scheduler and can observe the shape of the cache throughout executions, constant-time programs are indeed protected against cache-based side-channels.

Several of the approaches previously presented prove that the cryptographic code they produce satisfies the constant-time paradigm. Vale (Bond et al. 2017; Fromherz et al. 2019) does so by relying on a verified static taint analyzer, Jasmin (Almeida et al. 2020) embeds leakage traces inside programs and leverages EasyCrypt tactics to prove that executions of a given program in public-equivalent states yield the same traces, while HACL* (Zinzindohoué et al. 2017) models secret values using an abstract datatype which only allows constant-time operations.

To ensure that existing, possibly unverified cryptographic code is secure, several off-the-shelf tools exist to verify whether an implementation is constant-time. For instance, Blazy, Pichardie, and Trieu (2017) instrument the Verasco verified static analyzer (Jourdan et al. 2015) to verify the constant-time security of C programs. Flowtracker (Rodrigues et al. 2016) is an LLVM-based (Lattner and Adve 2004) tool that implements a flow-sensitive analysis. Almeida et al. (2016) provide a tool called `ct-verif` that compiles annotated C programs to LLVM using SMACK (Rakamaric and Emmi 2014) before analyzing optimized LLVM code in Boogie. CacheAudit (Doychev et al. 2015) is a static analyzer that operates on x86 binary programs, and quantifies the amount of information contained in timing and cache-based side channels.

Ensuring that a source program, for instance written in C, is constant-time does not guarantee that the executable binary will satisfy the same property: compiler optimizations and runtime libraries can also introduce vulnerabilities (Kaufmann et al. 2016). Ensuring that compilers preserve constant-time security of implementations is an active research area. CompCert-CT (Barthe et al. 2020) modifies the CompCert verified compiler to ensure that constant-time guarantees are preserved during compilation, extending the Coq proofs of correctness with a notion of CT-simulation (Barthe et al. 2018). Unfortunately, using CompCert currently yields slower code than using state-of-the-art commercial compilers such as GCC or CLANG with all their optimizations (-O3) (Leroy et al. 2016). This limits its adoption in areas that are both security- and performance-critical, such as cryptography.

To reason about the security of cryptographic implementations, formal approaches require models of the underlying hardware executing the program. For instance, to establish that an implementation is constant-time, basic CPU operations such as arithmetic additions or moving values between registers are assumed to execute in a constant time, while branching or memory accesses are considered as potentially leaking information to an attacker. When the reality diverges from the model, properties established through verification do not preclude concrete, real-world attacks. Ensuring that the behavior of the hardware matches assumptions at the software level has been a longstanding research area (Kern and Greenstreet 1999). More recent work investigated the application of hardware verification to constant-time guarantees. For instance, Gleissenthall et al. (2019) propose an approach to eliminate timing side-channels in hardware, while Tiwari et al. (2009) design a class of architectures which precisely track all information flows at the gate level, while being usable for cryptographic implementations. Focusing instead on language-based solutions, Zhang et al. (2015) present a hardware language, SecVerilog, which makes it possible to precisely reason about information flow at the hardware level. Zhang et al. demonstrate the practicality of their methodology by designing a secure MIPS microprocessor using SecVerilog.

In their quest for performance, modern hardware heavily relies on speculative execution, where the processing of an instruction starts before it is actually encountered. For instance, when a program branches between two paths, e.g., when using an if-then-else structure, the hardware's branch predictor will guess the path that the program might jump to, and start decoding the corresponding instructions. This behavior is rarely part of software verification models, and thus it can be exploited by adversaries, as demonstrated by recent, devastating side-channel attacks such as Spectre (Kocher et al. 2019) and Meltdown (Lipp et al. 2018). To address this issue, Cauligi et al. (2020) propose an extension of the constant-time paradigm capturing speculative execution attacks. Barthe et al. (2021) extend the Jasmin framework with a similar notion, and adapt previous verified implementations of ChaCha20 and Poly1305 to provide versions secure against both timing and speculative side-channel attacks, with a modest performance overhead. Also building upon the speculative constant-time guidelines, Vassena et al. (2021) implement a tool called BLADE, which automatically eliminates speculative leaks from existing WebAssembly cryptographic code.

2.3 Separation Logic-Based Verification

Since its introduction in the early 2000s (OHearn et al. 2001; Reynolds 2002; Ishtiaq and OHearn 2001), separation logic has become the foundation of many tools to verify heap-manipulating programs. In particular, many projects investigate the application of separation logic to low-level languages. For instance, Bedrock (Chlipala 2013) adopts a specification style inspired by separation logic to verify programs written in an assembly-like language, while Myreen and Gordon (2007) and Ni and Shao (2006) propose separation-logic based frameworks to verify assembly-level programs. Slightly climbing the abstraction ladder, Verifiable C (Appel et al. 2014) is a program logic to prove the correctness of C programs and is at the core of the Verified Software Toolchain (Appel 2011), while Appel and Blazy (2007) define a separation logic to verify Cminor programs, an intermediate language in the verified CompCert C compiler (Leroy 2006).

Nevertheless, separation logic is not only applicable to C or assembly languages; several other projects demonstrate its usefulness on higher-level languages with different language features and idioms. Notably, RustBelt (Jung et al. 2018a) relies on separation logic to model Rust's ownership-based type system and to verify Rust programs that internally use unsafe features. Similarly, CFML (Charguéraud 2011) defines characteristic formulae, inspired by separation logic, to verify higher-order OCaml programs. Finally, Charge! (Bengtson et al. 2012) targets verification of programs written in a subset of Java by embedding a higher-order separation logic in Coq.

Such tools provide a rich arsenal to reason about heap-manipulating programs written in a variety of languages. To provide verified building blocks to build more complex systems, Pottier (2017) and Lammich (2016) use them to verify imperative data structures; Pottier does so using the CFML framework (Charguéraud 2011), while Lammich develops a separation logic on top of Isabelle/HOL (Lammich and Meis 2012).

Beyond textbook data structures, to fully showcase the power of separation logic, several projects successfully use it to verify realistic applications; notably, a large number of projects target different critical components of operating systems and microkernels. For instance, using the Coq proof assistant, Marti et al. (2006) verify the heap manager of Topsy (Fankhauser et al. 2000), an existing operating system designed for teaching purposes. Similarly, Tuch, Klein, and Norrish (2007) embed a separation logic inside Isabelle/HOL and use it to verify the memory allocator of the L4 microkernel. This work was later extended (Kolanski and Klein 2009) to reason about virtual memory in the context of an operating system. More recently, Xu et al. (2016) verify key modules of a preemptive operating system, including its scheduler, message queues, and interrupt handlers, while Chen et al. (2015) verify the functional correctness of a file system in the presence of crashes. These two proof efforts were non-trivial, requiring several person years to verify 1.3k and about 3k lines of C code respectively.

Over the past twenty years, several extensions of separation logic have been proposed to go beyond verification of simple, sequential heap-manipulating programs. In particular, Bornat et al. (2005) propose a methodology based on Boyland's fractional permissions (Boyland 2003) to reason about read-only permissions on memory cells. Charguéraud and Pottier (2017) later proposed a simplified version of this extension, which removed the need for fractional accounting to model read-only permissions.

Particularly relevant to this thesis, Concurrent Separation Logic (CSL) (OHearn 2007; Brookes 2007) enables the use of separation logic to reason about *concurrent* programs, and it is at the heart of the Steel framework that we present in Chapters 6 and 7. Since O'Hearn's seminal work, several variants of CSLs have been proposed (Hobor et al. 2008; Jacobs and Piessens 2011; Dinsdale-Young et al. 2010; Dodds et al. 2009; Rocha Pinto et al. 2014; Sergey et al. 2015; Svendsen and Birkedal 2014; Svendsen et al. 2013; Turon et al. 2013; Nanevski et al. 2014; Jung et al. 2015).

Out of these works, the Iris framework (Jung et al. 2015; Jung et al. 2016; Krebbers et al. 2017a) has become especially prominent in recent years. Iris is a higher-order, impredicative, concurrent separation logic embedded in the Coq proof assistant (Coq Development Team 2017). Several projects rely on Iris to verify interesting programs. For instance, Chajed et al. (2019) use Iris to verify a concurrent, crash-safe mail server implemented in Goose, a subset of the Go language. Hinrichsen, Bengtson, and Krebbers (2019) develop a framework called Actris to reason about the correctness of message-passing programs that contain higher-order functions and references, as well as fork/join concurrency and locks. To do this, Actris builds a logic inspired by affine binary session types (Honda et al. 1998; Mostrous and Vasconcelos 2014) on top of Iris. Krogh-Jespersen et al. (2020) design a logic to reason about distributed systems, and use it to verify a multi-threading implementation of a load balancer. However, Iris itself is not a programming language: a user needs to instead instantiate the framework with the deeply embedded representation and semantics of one. Several Iris-based projects thus work with a small ML-like language embedded in Coq called HeapLang (Krebbers et al. 2017b), while others instantiate Iris with subsets of existing languages such as Rust (Jung et al. 2018a), Haskell (Timany et al. 2018), or C (Sammler et al. 2021). Our Steel framework (Swamy et al. 2020; Fromherz et al. 2021) is inspired by and shares many similarities with Iris, but pursues different goals: while Iris provides a foundation in which to investigate new logic and language features, Steel aims to extend a proof assistant's programming language to implement and reason about dependently-typed concurrent programs. To this end, the Steel framework shallowly embeds an impredicative, highly expressive Concurrent Separation Logic inside the F* proof assistant (Swamy et al. 2016), thus providing a logic to reason about higher-order, dependently typed concurrent F* programs. We provide a more detailed comparison between Steel and Iris in Chapter 6.

In order to facilitate verification, many tools and solvers aim to automate separation logic reasoning. While providing an entirely automated separation-logic framework to verify realistic programs is appealing, it is unfortunately also unrealistic: Brotherston and Kanovich (2010) demonstrate that determining the validity of a separation logic assertion in a concrete heap-like model is an undecidable problem. To circumvent this issue, a large number of tools identify fragments of separation logic that are amenable to fully-automated verification. Smallfoot (Berdine et al. 2005) was one of the first tools to pursue this direction; it targeted a toy programming language and did not support arbitrary inductive definitions of data structures, restricting itself to the basic primitives of separation logic as well as lists and trees. Several projects extended Smallfoot with a richer separation logic. For instance, SmallfootRG (Calcagno et al. 2007) added support for rely/guarantee reasoning (Jones 1983; Vafeiadis and Parkinson 2007), while Heap-Hop (Villard et al. 2010) focused on message-passing synchronization, encoding a form of session types called contracts. Iosif, Rogalewicz, and Simacek (2013) later proved that a fragment of separation logic using more general inductive predicates was decidable, while Le et al. (2017) proposed a decision procedure to reason about inductive predicates mixed with arithmetic properties. Building upon Smallfoot's approach, Caper (Dinsdale-Young et al. 2017) proposes a region-aware symbolic execution to automatically reason about interference on shared regions for fine-grained concurrent algorithms. As an alternative to Smallfoot, the Cyclist prover (Brotherston et al. 2011; Brotherston et al. 2012) relies on *cyclic proofs* instead of the more standard inductive proofs to automatically decide entailments in a fragment of separation logic.

Instead of targeting a toy programming language, others instead design automated tools to reason about existing languages. For instance, building upon the VCC framework (Cohen et al. 2009), Pek, Qiu, and Madhusudan (2014) develop VCDryad and use their tool to automatically verify many small C programs handling data structures. JaVerT (Fragoso Santos et al. 2017) enables semi-automated reasoning about heap strucutres in JavaScript and has been applied to verify functional correctness of several data structures as well as to tests from the official ECMAScript test suite. Distefano and Parkinson (2008) develop jStar, an automatic tool to verify Java programs using common object-oriented patterns such as visitors.

By forgoing functional correctness and focusing instead on verifying the safety of programs, several separation logic-based static analyzers reach a higher level of automation and scalablility. For instance, SpaceInvader (Yang et al. 2008) relies on a shape analysis and was used to verify the safety of Windows and Linux drivers of up to 10kLoC. Similarly, SLAyer (Berdine et al. 2011) identifies memory-safety issues in moderately-sized Windows drivers whose size ranged between 10kLoC and 30kLoC. Thor (Magill et al. 2008) combines a shape analysis with arithmetic reasoning to detect for instance safety violations related to lengths of lists. Also using a separation logic-based static analysis, Chang, Rival, and Necula (2007) verify the safety of the Linux `scull` driver, whose implementation relies on arrays of doubly-linked lists. To increase the scalability of such analyses, Calcagno et al. (2009) extend SpaceInvader with a compositional shape analysis based on biabduction, enabling the independent analysis of each procedure. The resulting tool, called Abductor, has been successfully used to analyze a complete Linux distribution containing about 2.5 millions lines of C code. Abductor was later commercialized as a tool called Infer (Calcagno and Distefano 2011), which is now part of Facebook's development cycle (Calcagno et al. 2015).

Instead of restricting the expressiveness of separation logic, others instead aimed for *auto-active verification* (Leino and Moskal 2010), a style which mixes automated and interactive proofs. Such approaches commonly rely on SMT solvers such as Z3 (Moura and Bjørner 2008) to discharge verification conditions, while allowing the user to provide annotations to help the solver when automated verification fails. For instance, VeriFast (Jacobs et al. 2011) is a framework to verify concurrent C and Java programs. VeriFast's automation relies on a symbolic execution which splits verification conditions between separation logic and pure predicates, before calling an SMT solver to verify the validity of the resulting predicates. Users can also write annotations and lemma functions to help with the verification of complex programs. To reduce the proof burden, several heuristics (Vogels et al. 2011) attempt to automatically generate some of the required user annotations. VeriFast has been used to verify several industrial applications, including Java Card applets implementing a clone of the Belgian electronic identity card (Philippaerts et al. 2011) and a Linux keyboard driver (Penninckx et al. 2012). Viper (Müller et al. 2016) relies on implicit dynamic frames (Parkinson and Summers 2012), a variant of separation logic, to verify the functional correctness of programs. To specify the expected behaviour of a program, a user can write two sorts of annotations in Viper: access permissions and first-order logic heap-fragment refinements. Several frontends for Viper exist, enabling the use of the toolchain to verify programs written in Go (Wolf et al. 2021), Rust (Astrauskas et al. 2019), or Python (Eilers and Müller 2018). The auto-active verification approach is not restricted to separation logic-based tools; frameworks such as Dafny (Leino 2010) or Low* (Protzenko et al. 2017) rely on SMT solving to discharge verification conditions based on *explicit dynamic frames* (Kassios 2006), while also allowing a user to provide additional lemmas and annotations.

Steel's automation (Fromherz et al. 2021) also follows this auto-active verification methodology, relying on a mixture of SMT solving and tactic-based automation. But compared to Viper and VeriFast, this automation applies to a higher-order, impredicative concurrent separation logic. We compare these approaches more in Chapter 8.

Chapter 3

A Primer on F*

All the work presented in this thesis has been performed using the F* proof assistant (Swamy et al. 2016). F* is a program verifier based on a dependent type theory with a hierarchy of predicative universes (like Coq (Coq Development Team 2017) or Agda (Norell 2008)). The F* language itself is a functional, ML-like language, with dependent types and an effect system. To prove properties about programs, F* mostly relies on the Z3 SMT solver (Moura and Bjørner 2008), while also allowing tactic-based verification through a metaprogramming system, Meta-F* (Martínez et al. 2019), inspired by Lean (Ebner et al. 2017) and Idris (Brady 2013). Once verified, F* programs are translated to OCaml or F#, and in some cases (Section 3.4), to C or WebAssembly for execution.

In this chapter, we give a quick overview of F*, and of some of the features that we will rely on in subsequent chapters.

3.1 F* Overview

F*'s syntax is similar to languages like OCaml or F#: Top-level signatures are defined using the keyword `val`, definitions are introduced with the keyword `let`, while recursive functions are defined using `let rec`, and pattern matching can be performed using the keyword `match`.

Binding occurences of variables b take the form x:t, declaring a variable x at type t; or #x:t, indicating that the binding is for an implicit argument. The syntax $\lambda(\mathsf{b}_1)\ ...\ (\mathsf{b}_n) \rightarrow$ t introduces a lambda abstraction, whereas $\mathsf{b}_1 \rightarrow ... \rightarrow \mathsf{b}_n \rightarrow \mathsf{c}$ is the shape of a curried function type. As usual, a bound variable is in scope to the right of its binding.

F* is a dependently-typed language; as such, it enables a programmer to define types indexed by other types or values. One classic example is vec a n, the type of vectors of size n containing values of type a. In F*, this type is defined as follows:

```
type vec (a:Type) : nat → Type =
  | Nil : vec a 0
  | Cons : #n:nat → hd:a → tl:vec a n → vec a (n+1)
```

This type definition is inductive and contains two cases: The Nil constructor models an empty vector, of size 0. Its type is therefore vec a 0. The non-empty case is captured by the Cons constructor; a value hd of type a can be appended to a vector tl of size n containing values of the same type to create a vector of size n + 1.

We omit the type in a binding when it can be inferred, and the variable name for non-dependent function types. For instance, the signature of a function concatenating two vectors is as follows:

```
val append: (#a:Type) → (#n:nat) → (#m:nat) → vec a n → vec a m → vec a (n + m)
```

In this thesis, we will also often omit implicit binders for presentation purposes, e.g., writing val append: vec a n → vec a m → vec a (n + m). All unbound variables will be assumed to be implicitly bound at the top.

In addition to dependent types, F⋆ also provides support for *refinement types*, written b{t}. For instance, the type of non-negative integers, nat, is defined as n:int{n ≥ 0}. Refinement and dependent types interoperate with each other, enabling a programmer, for instance, to define vector accesses (shown below) while ensuring that the access is always in bounds; i.e., the index i is smaller than the length n of the vector.

```
val access (#a:Type) (#n:nat) (i:nat{i < n}) (v:vec a n) : a
```

3.2 A User-Extensible Effect System

A distinctive feature of F⋆ is its effect system: in the function type $b_1 \rightarrow \ldots \rightarrow b_n \rightarrow c$, c is a *computation type*. An example of a computation type is Tot bool, the type of total computations returning a boolean. By default, F⋆ functions are total and we often omit the Tot annotation in the computation type. Ghost t is the type of a computationally irrelevant computation, often used for proof purposes, which will be erased during extraction. F⋆ provides an extensible mechanism to mark certain types as non-informative, including, notably, the type erased t. Eliminating a term of a non-informative type (e.g., pattern matching on it) incurs a ghost effect, ensuring that such uses never occur in computationally relevant code. F⋆ also provides a primitive effect for divergence, Dv, which can be used to implement general recursive, possibly non-terminating functions. As with other effects, the Dv effect is isolated from F⋆'s logical, total core, ensuring the soundness of the language: non-terminating functions cannot be mistakenly used as proofs. As such, the following term is well-typed in F⋆: let rec loop : unit → Dv unit = λ() → loop (). From the perspective of F⋆'s logical core, a → Dv b is an abstract, un-eliminable type.

In addition to primitive effects like Ghost or Dv, F⋆ allows a user to define their own computation types (Ahman et al. 2017) to model programming idioms such as mutable state or exceptions. A recent F⋆ addition, *indexed effects* (Rastogi et al. 2021), extends this mechanism; programmers can reason about effectful computations using an indexed monad-like structure, while abstracting over the effect's semantic representation. We heavily rely on this feature in Steel (Chapters 7 and 8) to reason about concurrent, stateful F⋆ programs.

3.3 Tactics, Metaprogramming, and Normalization

While F* heavily relies on SMT solvers (notably Z3 (Moura and Bjørner 2008)) to discharge verification conditions, it also offers other facilities to handle complex proofs. Similar to other proof assistants, F* provides a general purpose framework, called Meta-F* (Martínez et al. 2019) to write program and proof transformations (called *tactics*).

F* tactics are themselves F* programs, encapsulated in a specific effect, Tac, which can manipulate the current proof state. For instance, to prove that for any propositions a and b, the property $a \implies b \implies b \wedge a$ holds, one can write the following F* tactic:

```
let proof () : Tac unit =
  let ha = implies_intro () in
  let hb = implies_intro () in
  split ();
  hyp hb;
  hyp ha
```

This tactic is very similar to what one would write in frameworks like Coq or Lean. We first add hypotheses a and b with names ha and hb respectively to the proof context. To prove the goal $b \wedge a$, we first destruct the conjunction by calling the tactic split, resulting in two subgoals b and a, which can finally be solved by applying the two hypotheses in the context. While this example is quite simple, Meta-F* can be used to achieve much more complex tasks; in Chapter 7, we present a partial decision procedure for a higher-order, concurrent separation logic implemented as an F* tactic.

Note that, as in other systems, F* tactics are not trusted. Meta-F* guarantees that the generated terms are well-typed, thus ensuring the soundness of any proof obtained by tactic application.

Finally, in addition to tactic-based and SMT-based verification, F* also provides a normalizer to (partially) reduce F* terms, thus enabling proofs by *computation*. To give a simple example, consider the recursive function length below, computing the length of a list.

```
let rec length (l:list a) = match l with
  | [] → 0
  | _ :: tl → 1 + length tl
```

To prove that length [1;2;3] == 3, one can simply compute the result of length on the list [1;2;3], resulting in the trivial equality 3 == 3. The F* normalizer is an interpreter that can perform this computation, avoiding calls to the SMT solver.

Normalization in F* is fine-grained and user-controlled: a user must call the primitive norm function on term e to invoke the normalizer, and must provide a list of normalization steps that can be performed, for instance specifying that only certain functions must be unfolded, or that unrolling recursive function calls is disallowed. We heavily rely on this feature in our Steel automation (Chapter 7) to carefully craft SMT-friendly verification conditions, which can then be fed to the Z3 solver.

3.4 Low*, A Shallow Embedding of C in F*

Low* (Protzenko et al. 2017) is a shallow embedding of a small, well-behaved subset of C in F* which we use to implement the EverCrypt cryptographic provider (Chapter 5). Low* programs are written in a low-level subset of F* obeying certain restrictions, for instance forbidding the use of closures or partial function applications. This restriction does not apply to specifications and proofs, which are erased during extraction; for this purpose, the full F* language remains available.

Low* programs are imperative, stateful computations, which are encapsulated in two F* effects: ST a (requires pre) (ensures post) and Stack a (requires pre) (ensures post). Both effects correspond to computations which, when run in an initial state h_0 satisfying the precondition pre h_0, either do not terminate or return a value v:a in final state h_1 such that the two-state postcondition post h_0 v h_1 is satisfied. The Stack effect additionally models that all allocations are performed on the stack, while ST computations can also perform heap allocations.

Low* operates on a C-like memory model with explicit heap and stack memory management. Low* states are modeled by values of type HS.mem, referred to as hyper-stacks (Protzenko et al. 2017). Briefly, hyper-stacks provide a region-based memory model (Tofte and Talpin 1997; Grossman et al. 2002), distinguishing *heap regions* from *stack regions*. Each region in a hyper-stack maps abstract memory addresses to typed values, e.g., fixed-width integers (uint8, uint64, etc.) or mutable arrays of values.

Well-typed Low* programs are guaranteed to be memory-safe; i.e., they never access out-of-bounds or deallocated memory, or attempt to repeatedly free the same memory. Consider for instance the following signature of the Low* function index, modeling an imperative access to a mutable array x:array t. This function is specified in terms of a pure operation get on an immutable sequence in a given memory (m[x.content] : seq t).

```
let get (m:HS.mem) (x:array t) (i:uint32{i < x.length}) : Ghost t = Seq.index (m[x.content]) i

val index: x:array t → i:uint32{i < x.length} → ST t
  (requires (λ m → live x m))
  (ensures (λ m0 r m1 → m0 == m1 ∧ r = get m0 x i))
```

The refinement {i < x.length} requires the index i to be in bounds, while the precondition live x m guarantees that the array is initially valid in memory. The combination of the two conditions ensures the spatial and temporal safety of this program.

Ultimately, a compiler, called KreMLin, compiles Low* code to idiomatic, human readable C code suitable for manual review. For instance, a call to the function index presented above would be translated to the standard C array access x[i]. To ensure that verification guarantees proven at the F* level are maintained after extraction to C, Protzenko et al. (2017) relate the semantics of Low* to CompCert's CLight (Blazy and Leroy 2009): they show (on paper) that KreMLin preserves trace equivalence with respect to the original F* semantics and that the translation does not introduce side-channels based on memory access patterns. A recent extension of the KreMLin toolchain (Protzenko et al. 2019) also enables compilation of Low* programs to verified WebAssembly (Haas et al. 2017b).

Part II

High-Performance, Provably Secure Cryptography

Chapter 4

A Verified Interoperation between C and Assembly

As our society grows increasingly connected, so does our reliance on the HTTPS ecosystem, the foundation of Internet security. The linchpin of security-critical protocols such as HTTPS is cryptography. Insecure cryptographic code thus weakens the entire ecosystem, and vulnerabilities such as FREAK (Beurdouche et al. 2015) or Heartbleed (National Vulnerability Database 2014; Durumeric et al. 2014) can have catastrophic consequences.

While the correctness and security of cryptographic implementations are critical, for use in real-world systems, so is their *performance*. To this end, cryptographic implementations in libraries such as OpenSSL (OpenSSL Team 2005) often rely on a mixture of C and hand-optimized assembly code. C-like code is convenient for writing efficient low-level code, and it is the standard choice to develop secure communication protocols relying on cryptography such as TLS (Rescorla 2018). But for maximum performance, hand-tuned assembly is required and is the *de facto* standard for state-of-the-art cryptographic implementations; by manually writing assembly code, a programmer can leverage hardware-specific features, such as the SHA-EXT (Gulley et al. 2013) and AES-NI (Gueron 2012) instruction sets, while also implementing custom optimizations that general-purpose compilers such as GCC or Clang might not detect. Hence, cryptographic libraries typically are hybrid programs, providing C routines and APIs, while periodically calling in to assembly for higher performance.

Unfortunately, as the performance of cryptographic software improves, so does its complexity. Combining correctness, security, and performance is notoriously difficult; despite being widely used and heavily audited, even OpenSSL's libcrypto reported 28 vulnerabilities between January 1, 2016 and June 1, 2021 (OpenSSL). For instance, CVE-2018-0733 reported a bug in a hand-written assembly implementation of memcpy which weakened security guarantees, allowing an attacker to more easily forge messages, while CVE-2018-0734 showed that the OpenSSL DSA signature algorithm could leak private signing keys through timing side channel attacks.

For such critical, complex code, ruling out entire classes of attacks through formal verification is appealing, but challenging. We need the ability to verify programs written in different languages (e.g., C and Assembly), while also enabling a sound interoperation between languages. Verification frameworks like Coq (Coq Development Team 2017), F* (Swamy et al. 2016), and Dafny (Leino 2010) commonly support a small number of existing languages; for instance, verified Dafny programs can be extracted to verified C# code, while Coq and F* have native support for extracting verified programs to OCaml. Given the diversity of programming languages in use, it is unlikely that any single verification framework could contain built-in support for all possible languages that programmers might need.

A common alternative is to instead use the expressiveness of such frameworks to embed domain-specific languages (DSLs), and verify properties of programs written in these languages. Examples of DSLs for verification purposes abound in the literature. For instance, Kennedy et al. (2013) propose a DSL to verify x86 assembly programs using the Coq proof assistant, while Bourgeat et al. (2020) define Koika, a DSL for hardware design inspired by SystemVerilog (Nikhil 2004), which enables formal reasoning about the performance of circuits.

In this chapter, we will also rely on proof-oriented DSLs—embedded in the F* proof assistant— to reason about high-assurance, complex, hybrid C and assembly cryptographic implementations. F* already provides Low*, a DSL for verifying C programs that we previously presented in Section 3.4. To perform assembly verification, we will build upon Vale (Bond et al. 2017), a language designed to verify structured assembly programs, such as cryptographic implementations. Prior work translated Vale code into Dafny code, and relied on Dafny to generate and discharge verification conditions. To formally model an interoperation between two different DSLs, the first required step is for them to be embedded in the same framework. To this end, we first reimplemented Vale to use F* instead of Dafny; we give an overview of Vale/F* in Section 4.1.

Each DSL allows us to independently reason about C and assembly programs respectively; to reason about hybrid programs, we also need to support fine-grained interactions between both languages. We first define in F* a trusted model of interoperation relating the execution models of both DSLs. Focusing on verified cryptographic implementations allows us to keep this model small and simple: we only need the ability to verifiably call Vale assembly routines from Low*, and cryptographic code in Low* and Vale operates on simple data structures, e.g., mutable arrays of fixed-width integers.

Building upon this model, we then provide a verified interoperation layer which lifts Vale procedures to Low* while ensuring that verification guarantees are preserved across the language boundary. Concretely, we define a generic correspondence between specifications at the Vale level and their Low* counterparts, and prove this correspondence sound with respect to our trusted model of interoperation. By designing this interoperation with verification in mind, we identify common proof patterns and reason about them generically, amortizing the cost of applying the interoperation layer to many Vale procedures and enabling its use for complex, real-world systems. We present the full interoperation framework in Section 4.2.

Contributions. I developed a first version of the Vale interoperation layer, which was presented in Fromherz et al. (2019). This version defined the correspondence between the execution models of Vale and Low⋆, but relied on a trusted printer to generate calls into Vale procedures from Low⋆. Instead of relying on this external printer, Nikhil Swamy and Chris Hawblitzel proposed a prototype of the trusted interoperation model using dependently-typed generic programming. I finally extended this prototype to support parametric calling conventions and to establish the generic correspondence between specifications, leading to the final version of the interoperation layer described in this chapter, which was presented in Protzenko et al. (2020). During the reimplementation of Vale in F⋆, I also rewrote the verified taint analysis.

4.1 The Vale Framework

4.1.1 A Deep Embedding of Intel x64 Assembly

Vale is a DSL that relies on deeply embedded hardware semantics formalized within F⋆. We present in Figure 4.1 excerpts from our F⋆ formalization, which models Intel x64 assembly. The embedding uses F⋆ datatypes to represent x64 registers (`reg`) and XMM registers (`xmm`); constant, register and memory operands (`operand`); instructions (`ins`); and abstract syntax trees for structured assembly language code (`code`). The control flow of Vale programs is restricted to structured if/else blocks and while loops, which are well-suited to implementations of cryptographic primitives, like those in OpenSSL.

We represent the machine state, shown in Figure 4.2, as a record (`state`) containing different machine components: general-purpose and XMM register files are functions mapping register names to values; status flags are a single word, and the memory is a partial map from integer addresses to bytes.

The state also contains a boolean field `ok` representing the validity of the machine state. A valid state (`ok = true`) indicates that the machine safely executed until now. For instance, a valid state ensures that no segmentation fault occured. Memory accesses and updates have validity checks based on membership in the domain of the memory map. An invalid memory access or update would therefore make the state invalid (`ok = false`).

Now that we modeled the Intel x64 syntax and machine state, the last step to provide an embedding of the assembly language is to specify how programs execute, i.e., define operational semantics for Intel x64 code. To this end, we provide a function, `eval_code`, that takes an initial state and structured assembly code of type `code` and returns the modified state after execution of the code.

To handle failure propagation while preserving the readability of the semantics, we rely on a state monad that transforms states into states. We quickly present this state monad in Figure 4.3. A function of type `st a` is a stateful function returning a value of type `a`. The composition of two stateful functions `f` and `g` is defined through `bind`; the interesting part of this composition is that the resulting state is considered valid only if both `f` and `g` do not fail, and the initial state was valid. As such, the validity of the state is a monotonic predicate; if a state is marked invalid, it will remain so

```
type reg = Rax | Rbx | Rcx | Rdx ...

type xmm = Xmm0 | Xmm1 | Xmm2 ...

type operand =
  | OConst: n:int → operand
  | OReg: r:reg → operand
  | OMem: m:mem_addr → operand

type ins =
  | Mov64: dst:operand → src:operand → ins
  | Add64: dst:operand → src:operand → ins
  ...

type cond =
  | Lt: o1:operand → o2:operand → cond
  | Eq: o1:operand → o2:operand → cond
  ...

type code =
  | Ins: ins:ins → code
  | Block: block:list code → code
  | IfElse: ifCond:cond → ifBranch:code → elseBranch:code → code
  | While: whileCond:cond → whileBody:code → code
```

Figure 4.1: Example F* definitions for Intel x64 syntax

when executing the rest of a program. To help with readability, F* provides a monadic syntax, x <- f; g, which is automatically desugared to bind f g. Leveraging this monadic structure, we finally define two helpers, check and run. check marks the current state as invalid if a stateful condition valid is not respected, leaving the state unchanged in the other case. run simply executes a stateful computation f on a given state s, returning the modified state.

Building upon this monadic structure, we can now concisely define our instruction semantics as a stateful function, eval_ins. We present in Figure 4.4 an excerpt of our model, corresponding to the specification of the instruction Add64. A first check ensures that the src and dst operands are valid, that is, that they do not access invalid memory addresses. We then compute the addition, and update the dst operand. The last step updates the flags in the machine state. Precisely modeling flags is error-prone, and can easily lead to incorrect models; we instead under-specify them. In our model, any flag update first havocs the flags, i.e., loses all known information about them, before updating some specific flags that are relevant to cryptographic implementations, such as the carry flag.

The final step is to implement eval_code, defining semantics for a full Vale program. To provide a sound logic, F* requires definitions used in proof contexts to be total; to be able to reason about Vale programs, we thus need to define eval_code as a total, always

```
type state = {
  ok:bool;
  regs:reg → nat64;
  xmms:xmm → (nat32 ∗ nat32 ∗ nat32 ∗ nat32);
  flags:nat64;
  mem:map int nat8;
}
```

Figure 4.2: Vale model of the Intel x64 machine state

```
let st (a:Type) = state → a ∗ state

let return (x:a) : st a = λs → x, s

let bind (f:st a) (g:a → st b) : st b =
  λs0 →
    let x, s1 = f s0 in
    let y, s2 = g y s1 in
    y, {s2 with ok = s0.ok && s1.ok && s2.ok}

let check (valid: state → bool) : st unit =
  s <- get;
  if valid s then
    return ()
  else
    (), {s with ok = false}

let run (f:st unit) (s:state) : state = snd (f s)
```

Figure 4.3: A stateful monad for Vale semantics

terminating function. To account for non-terminating code, for instance containing unbounded loops, eval_code takes an additional argument providing fuel (defined to be a natural number) that is consumed at each loop iteration during execution. If the fuel reaches zero, eval_code returns None, which is used only for termination checking; i.e., it is distinct from reaching an invalid state. If a Vale program c terminates when run in the initial state s, this means that there exists some fuel f such that eval_code c f s == Some s', where s' is the final, modified state after executing c.

To execute Vale programs, a small, trusted F* program ultimately prints code values as standard GNU assembly and Microsoft MASM assembly language formats. Vale programs are already close to idiomatic assembly, making extraction straightforward. The only slight mismatch appears for the IfElse and While blocks, which are translated to conditional jumps to labels.

```
let valid_operand (o:operand) (s:state) : bool = match o with
  | OConst _ | OReg _ → true
  | OMem m → valid_maddr m s

let eval_operand (o:operand) (s:state) : nat64 = match o with
  | OReg r → s.regs r
  ...

let eval_ins (ins:ins) : st unit =
  s <- get;
  match ins with
  | Mov64 dst src → ...
  | Add64 dst src →
      check (valid_operand src);;
      check (valid_operand dst);;
      let sum = eval_operand dst s + eval_operand src s in
      let new_carry = sum ≥ pow2_64 in
      set_operand dst ins (sum % pow2_64);;
      set_flags (update_cf s.flags new_carry)
  ...

let rec eval_code (c:code) (fuel:nat) (s:state) : option state =
  match c with
  | Ins ins → Some (run (eval_ins ins) s)
  | While b c → eval_while b c f s
  | ...

and eval_while (b:cond) (c:code) (fuel:nat) (s:state) : option state =
  if fuel = 0 then None else
  let s0 = run (check (valid_cond b) s0) in
  if not (eval_cond s0 b) then Some s0
  else
    match eval_code c (fuel − 1) s0 with
    | None → None
    | Some s1 → eval_while b c (fuel − 1) s1
```

Figure 4.4: Example semantics for Intel x64 assembly

4.1.2 The Vale Language

Although it is possible to hand-write Vale programs as structured code values like "Block [Ins (Add 64 (OReg Rax) (OConst 10))]" and prove properties about such code values directly in terms of the eval_code semantics, it is useful to have a friendlier syntax and tool for expressing and verifying assembly language programs. The Vale language (Bond et al. 2017) provides a syntax for writing assembly language annotated with preconditions, postconditions, loop invariants, ghost variables, calls to lemmas, etc.

Vale programs consist of a series of procedures, as shown in the examples in Figure 4.5. Procedure parameters may be operands, whose types specify what argument operands are allowed (e.g., dst_opr64 for a 64-bit destination operand). A procedure representing an individual instruction, indicated with the attribute {:instruction ...}, is verified directly against the operational semantics specified by eval_code; for example, both Add and AddWrap are verified relative to the semantics for the Add64 instruction specified in Figure 4.4. Compound procedures like Triple contain procedure bodies inside {...}; the procedure bodies are verified using Hoare logic. The Add(rax, rbx) instruction in Triple, for example, must satisfy the precondition $\mathsf{rax} + \mathsf{rbx} < \mathsf{pow2\_64}$ specified by the Add procedure. Calls from one procedure to another procedure are inlined (macro expanded).

An elaborator finally transforms a program written in the Vale language and its Hoare-style specification to a code value and a lemma relying on the trusted eval_code semantics respectively. The elaborator itself is untrusted; verification conditions and the trusted Vale printer operate directly on the generated code value.

```
procedure{:instruction Ins(Mov64(dst, src))}
  Move(out dst:dst_opr64, src:opr64)
    ensures dst == old(src);

procedure{:instruction Ins(Add64(dst, src))}
  Add(inout dst:dst_opr64, src:opr64)
    modifies flags;
    requires dst + src < pow2_64;
    ensures dst == old(dst + src);

procedure{:instruction Ins(Add64(dst, src))}
  AddWrap(inout dst:dst_opr64, src:opr64)
    modifies flags;
    ensures
      dst == old(dst + src) % pow2_64;
      cf(flags) == old(dst + src ≥ pow2_64);

procedure Triple()
    modifies rax; rbx; flags;
    requires rax < 100;
    ensures rbx == 3 * old(rax);
{
    Move(rbx, rax);
    Add(rax, rbx);
    Add(rbx, rax);
}
```

Figure 4.5: Example procedure declarations in Vale

4.1.3 Proving Assembly Code Free from Information Leakage

When implementing cryptographic algorithms, ensuring that the code is correct is not sufficient: it must be proven secure as well. More precisely, since cryptographic code operates on secrets, we must prove the absence of *leakage*. To enable such proofs, we model in Vale a strong attacker capable of observing detailed digital side-channel information. The attacker can see every instruction executed, every memory address accessed, and every element of the machine state that is not explicitly declared secret. To capture this model, we augment our machine state with a trace field, and we extend our machine semantics to record adversarial observations in this trace. We then define leakage freedom, as shown in Figure 4.6, as a classic non-interference property (Goguen and Meseguer 1982). A procedure (code) is leakage free if, for all states s1 and s2 such that the traces and memory locations marked as public in s1 and s2 are initially identical, the traces are identical in the states r1 and r2 computed by successful executions of code. Stated differently, observations from an attacker do not leak any information about the program's secrets.

```
type observation =
  | BranchPredicate: pred:bool → observation
  | MemAccess: addr:nat64 → observation

type state = {...; trace : list observation}

let eval_while b c fuel s =
  ...
  if not (eval_cond s0 b) then Some ({s0 with trace = BranchPredicate(false)::s0.trace})
  else
    match eval_code c (fuel − 1) ({s0 with trace = BranchPredicate(true)::s0.trace}) with
    ...

let is_leakage_free (code:code) (isPub:loc → bool) =
  ∀(s1 s2:state) fuel.
    let r1 = eval_code code fuel s1 in
    let r2 = eval_code code fuel s2 in
    s1.trace == s2.trace ∧ (∀ x. isPub x ⟹ s1[x] == s2[x]) ⟹
    r1.trace == r2.trace
```

Figure 4.6: We extend the machine state with adversarial observations that capture digital side channels, and we define the absence of leakage in a program code via the is_leakage_free predicate.

Our strategy for establishing that a Vale program is leakage free involves a classic use of proof by reflection. We implement in F* a static *taint analyzer* that consumes our (deeply embedded) syntax of assembly language and detects whether a program might violate the constant-time guidelines. We prove, once and for all, that our taint analysis algorithm conservatively decides the is_leakage_free property, by proving it sound with respect to our trace-augmented semantics. The F* signature of our taint analysis is shown below:

```
val taint_analysis: c:code → isPub:(loc → bool) → b:bool{b ⟹ is_leakage_free c isPub}
```

To run the taint analysis, we extract taint_analysis to OCaml (using F⋆'s existing extraction capabilities) and then concretely run OCaml code on the syntax of the program and secret labeling. We integrate this analysis in the Vale extraction pipeline, ensuring that programs deemed possibly insecure by the analysis are not extracted, and hence cannot be executed.

4.2 Interoperating between Vale and Low⋆

To verify hybrid C and assembly programs, we propose in this section an interoperation between Low⋆ and Vale. Supporting this kind of DSL interoperation poses several challenges. First, Low⋆ and Vale have very different memory models. Low⋆ models memory as a well-typed, structured heap (similar to CompCert's (Leroy et al. 2016)), while the machine model in Vale maps integer addresses to bytes. Second, calls between C and assembly are mediated by a calling convention specific to the operating system and hardware used. Finally, given our interest in security-related applications, we would like measures for side-channel resistance adopted by Vale and Low⋆ to also compose well.

In contrast to prior work investigating verified interoperation between C and assembly, our focus on verified cryptography yields a simpler setting: we want verified Low⋆ programs to be able to call into verified Vale procedures, while only transferring control from Vale back to Low⋆ via returns. As such, we do not need to model callbacks from assembly such as FunTAL (Patterson et al. 2017) does, nor do we need to consider potentially malicious assembly language contexts (Ahmed et al. 2018). Instead, the goal of our interoperation framework is to safely lift the Vale semantics to a Low⋆ specification, so that Low⋆ programs containing Vale computations can be verified within Low⋆'s program logic. To this end, we will extend the semantics of Low⋆ to model Vale procedures as atomic computational steps with effects on memory and a single word-sized result (but with variable execution time as a potential side-channel). We summarize this workflow in Figure 4.7.

4.2.1 Modeling Interoperation

Our interoperation relies on a trusted semantics of a call from Low⋆ into the Vale program c with arguments $\mathsf{arg}_1, \ldots \mathsf{arg}_n$, modeled as a function, call_assembly, sketched in Figure 4.8.

Operationally, the call is modeled as follows: at line 2, we retrieve the initial Low⋆ heap h_0; at line 3, we construct the initial Vale state s_0 from h_0 and all the arguments; at line 4, we run the Vale definitional interpreter presented in Section 4.1.1 to obtain the final Vale state s_1; at line 5, we translate this s_1 back to a Low⋆ heap h_1 and return value rax; finally, we update the Low⋆ state atomically with h_1 and return rax. We present below the different components of this model in detail.

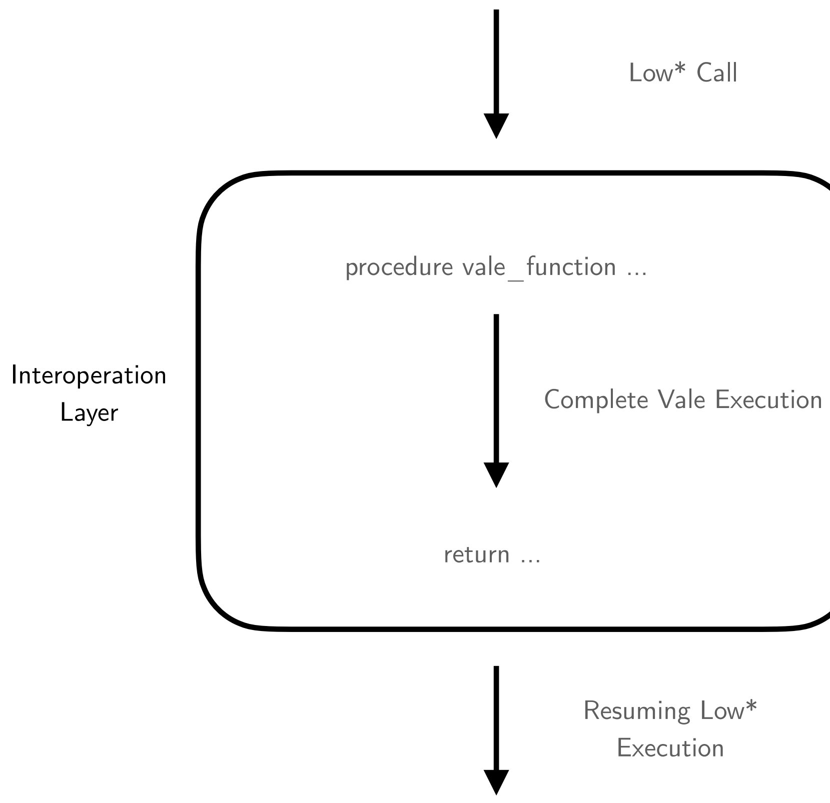


Figure 4.7: A high-level summary of the Low⋆/Vale interoperation

```
let call_assembly c $arg_1$ ... $arg_n$
  let $h_0$ = get () in
  let $s_0$ = initial_vale_state $h_0$ $arg_1$ ... $arg_n$ in
  let $s_1$ = eval_code c $s_0$ in
  let rax, $h_1$ = final_lowstar_state $h_0$ $s_1$ in
  put $h_1$; rax
```

Figure 4.8: Trusted model of interoperation between Low⋆ and Vale

Relating Memory Models. To support hybrid programs, we must enable Low⋆ and Vale programs to share selected regions of memory that correspond to the storage referred to by mutable references in Low⋆. However, aspects of a C program's memory that are not observable from Low⋆ must remain inaccessible from Vale. For instance, although a Vale program should be able to access stack data that was explicitly allocated in Low⋆, we do not wish to allow a Vale program to access the control stack of a Low⋆ program, as otherwise it could undermine the Low⋆ meta-theory that relates its semantics to C's semantics.

The memory models used by Low⋆ and Vale differ significantly. The Low⋆ memory model stores values of structured types: the types include machine integers of various widths (8-128 bits) and signedness; and arrays of structured values (as in C, pointers are just singleton arrays). In contrast, Vale treats memory as just a flat array of bytes. To unify both memory models, we thus need to, first, make explicit the layout of each shared Low⋆ structured type in Vale's flat memory model, and second, relate memory accesses at each level by ensuring that they manipulate the same underlying value.

In this work, we will restrict ourselves to shared accesses to mutable arrays of fixed-width integers; this suffices when targeting highly optimized cryptographic implementations. To model the memory layout of such structures, for each supported integer width, we build bijections to treat arrays of machine integers, e.g. array uint32 as arrays of bytes, e.g. array nat8.

```
val to_bytes32: array uint32 → array nat8

val from_bytes32: (a8:array nat8{length a8 % 4 == 0}) → array uint32

val bijection32 (a:array uint32) : Lemma (from_bytes32 (to_bytes32 a) == a)
```

The last remaining step is to define a correspondence from the fragment of Low⋆ memory (heap) containing the shared pointers to Vale's memory (mem), which we present in Figure 4.9.

```
let addr_map_pred (m:array nat8 → nat64) =
  (∀ (a1 a2:array nat8). disjoint a1 a2 ⟹
          m a1 + a1.length ≤ m a2 ∨ m a2 + a2.length ≤ m a1)

type addr_map = m:(array nat8 → nat64){addr_map_pred m}

let addrs_set addrs ptrs : set int =
  { i | ∃a:array nat8. List.mem (from_bytes a) ptrs ∧ addrs a ≤ i < addrs a + a.length }

let correct_simulation (addrs:addr_map) (a:array nat8) (heap:HS.mem) (mem:map int nat8) =
  ∀(i:nat{i < a.length}). get_heap a i heap == mem.[addrs b + i]

let correct_simulation_global addrs ptrs heap mem =
  addrs_set_global addrs ptrs == Map.domain mem ∧
  (∀ a. List.mem a ptrs ∧ live a heap ⟹ correct_simulation addrs (to_bytes a) heap mem)
```

Figure 4.9: Trusted correspondence between fragments of Low⋆'s and Vale's memory

One can see correct_simulation_global addrs ptrs : HS.mem → map int nat8 → Type as a relation between the two memories indexed by (1) a function addrs : addr_map that maps live, disjoint abstract addresses in Low⋆ to disjoint, valid address ranges in the Vale memory model; and (2) a list of array references ptrs that are to be shared between Low⋆ and Vale.

The definition states that all live arrays in ptrs have the same values in both heap (at their abstract address) and in mem (at their corresponding concrete address chosen by addr_map). It also restricts the domain of the Vale memory, ensuring that valid Vale memory accesses correspond to accesses to shared Low⋆ arrays.

The correct_simulation_global relation is at the core of the interoperation model presented in Figure 4.8. When calling from Low⋆ into Vale, initial_vale_state h0 args assumes the existence of an addr_map and creates a Vale state s_0 which satisfies the correct_simulation_global relation with h_0; similarly, at the time the call returns, the state h_1 of the Low⋆ program generated by final_lowstar_state is proven to be in correct_simulation_global relation with the final state s_1 of the Vale procedure.

Modeling the Calling Conventions. Calling conventions describe how subroutines interact with their caller. From Vale's perspective, arguments are received in specific registers and spilled on the stack if needed; in contrast, in Low⋆, as in C, arguments are just named. As we construct the initial Vale state, we need to translate between these (platform-specific) conventions, e.g., on an Intel x64 machine running Linux, the first argument of a function must be passed in the rdi register, and the second in rsi. Further, calling conventions between C and assembly often specify registers that must be *callee-saved* (e.g., r15 for Windows on x64); such registers must have the same value when entering and exiting the Vale procedure.

Calling conventions vary heavily based on the operating system, architecture, and compiler used; in some cases, such as the use of inline assembly, they can even be defined entirely by a programmer. To enable interoperation on a variety of platforms, and also providing support for inline assembly, the function creating the initial Vale state is parametric in the calling conventions.

Our framework is parameterized by two functions with types regs_modified_t and of_arg_t respectively, presented in Figure 4.10. A function of type f : regs_modified_t specifies which registers can be modified; f r = false indicates that register r was callee-saved, and thus should hold the same value in the initial and final Vale states.

The type of_arg_t is more interesting. It is parametric in an arity, n, and models functions that maps arguments to registers. It additionally imposes several restrictions ensuring the well-formedness of the calling conventions: distinct arguments must be passed in distinct registers (i.e., the mapping must be injective), and the stack register rsp cannot be used to pass arguments.

```
type of_arg_t (n:nat) =
  f:((i:nat{i < n}) → reg){injective f ∧ (∀ (i:nat{i < n}). of_arg i ≠ rsp)}

type regs_modified_t = reg → bool
```

Figure 4.10: Generic types to specify calling conventions

Using these function types, instantiating our interoperation framework with different calling conventions is straightforward. Given two functions of_arg : of_arg_t and modified : regs_modified_t, initial_vale_state will store arguments in registers according to of_arg, spilling the remaining arguments on the stack, while our framework will automatically generate an assertion that must be statically proven ensuring that all callee-saved registers according to modified are preserved by the Vale execution. We provide instantiations for common cases, e.g., x64 standard calls on Linux, as shown in Figure 4.11, and Windows. Verification conditions related to the restrictions in the type of_arg_t are automatically discharged by F⋆.

```
let x64linux_of_arg (i:nat{i < 6}) : reg = match i with
  | 0 → rdi
  | 1 → rsi
  | 2 → rdx
  | 3 → rcx
  | 4 → r8
  | 5 → r9

let x64linux_regs_modified (r:reg) : bool =
  not (r = rbx || r = rbp || r = r12 || r = r13 || r = r14 || r = r15)
```

Figure 4.11: Instantiating calling conventions for x64 standard calls on Linux. The first 6 arguments are passed in registers, extra arguments are spilled on the stack. Registers rbx, rbp, r12, r13, r14, and r15 are callee-saved.

4.2.2 Generically Lifting Specifications

Now that we have defined a semantic model of interoperation between Vale and Low⋆, we wish to use it to verify such hybrid programs. The main challenge consists of ensuring that verification guarantees proven for a Vale program are preserved when called from Low⋆. The Vale preconditions must be provable in the initial Low⋆ state and arguments in scope. Dually, the Vale postconditions must suffice to continue the proof on the Low⋆ side.

A key feature of our interoperation layer is to lift Vale specifications along the mapping between Vale and Low⋆ states, e.g., for a Vale program satisfying the Hoare triple $\{P\}\ c\ \{Q\}$, we can give call_assembly the following Low⋆ type, where lift_pre and lift_post reinterpret soundly and generically (i.e., once and for all) pre- and postconditions on Vale's flat memory model and register contents in terms of Low⋆'s structured memory and named arguments in scope.

```
val call_assembly c arg_1 ... arg_n : Stack uint64 (lift_pre P) (lift_post Q)
```

Our generic lifting of specifications between Vale and Low⋆ is proven sound against our trusted model of interoperation presented in Section 4.2.1, and is thus untrusted—thankfully so, since this is also perhaps the most complex part of our interoperation framework, for two reasons.

First, Vale and Low⋆ use subtly different core concepts (each optimized for their particular setting) including different types for integers, and different predicates for memory footprints, disjointness, and liveness of memory locations. Hence, relating their specifications involves working deep within the core of the semantic models of the two languages and proving compatibility properties among these different notions. This is only possible because both languages are embedded within the same host language, i.e., F⋆.

Second, because we model the calling conventions generically for all arities, the relations among Vale and Low⋆ core concepts must also be generic, since they state properties of the variable number and types of arguments in scope.

However, the payoff for these technical proofs is that they are done once and for all, and their development cost is easily amortized by the relative convenience of instantiating the framework at a specific arity for each call from Low⋆ to Vale.

For a sense of the scale, our full semantic model of interoperation consists of 1595 lines of F⋆ specification and modeling code; 2194 further lines of untrusted F⋆ proofs establish various generic lemmas for convenient use of the call_assembly wrapper from Low⋆ (including functions analogous to lift_pre and lift_post). Finally, we used our framework to implement 31 specific calls from Low⋆ into Vale, requiring an additional 11,558 lines of untrusted F⋆ proofs; the largest single Vale procedure that we have lifted to Low⋆ takes 17 arguments, and is specified in 181 lines of F⋆ code.

4.2.3 Side-Channel Analysis

By relating Hoare triples between Low⋆ and Vale as described in the previous sections, we have presented a framework that enables interoperation between the two DSLs while preserving formal guarantees about the correctness of Vale programs.

Unfortunately, when considering security-critical applications such as cryptography, a correct implementation is not sufficient; we must also ensure that it is robust against attacks by connecting the side-channel guarantees provided by Low⋆ and Vale.

Background: Proving Low⋆ Code Free from Information Leakage. When proving side-channel resistance, some techniques are more convenient than others, depending on the language and level of abstraction used. As described in Section 4.1.3, to demonstrate that assembly code is secure, Vale relies on proof by reflection by providing a verified taint analyzer operating on the Vale assembly syntax. Unlike Vale's deep embedding of assembly language, Low⋆ is a shallow embedding and uses type abstraction to prove side-channel resistance.

When programming with secrets, Low⋆ programs are written against an interface that provides secrets at an abstract type, as presented in Figure 4.12. The type system then ensures that the programs cannot, for example, branch on secrets or use secrets as array indices; operating on abstract secret integers can only be done through the functions provided in the interface, corresponding to basic arithmetic operations.

To specify properties about secret integers, the interface also provides a function, to_nat, which returns the underlying representation of the secret integer. Importantly, to_nat has the Ghost effect; this ensures that it can only be used for proof purposes, in computationally irrelevant contexts, thus preserving the type abstraction in executable code.

Assuming that the implementations of functions operating on secrets are secret-independent, Low⋆ then provides a secret-independence (meta-)theorem (Protzenko et al. 2017, Theorem 1). To formalize secret-independence, Low⋆ semantics are instrumented

```
val uint64 : Type

val add : uint64 → uint64 → uint64
val sub : uint64 → uint64 → uint64
val mul : uint64 → uint64 → uint64
val logxor : uint64 → uint64 → uint64
...

val to_nat: uint64 → Ghost nat
```

Figure 4.12: Interface for secret integers in Low⋆.

with traces that reflect the branching behavior (brT and brF) and the memory access patterns ($\mathsf{read}(b, n)$ and $\mathsf{write}(b, n)$):

$$\text{Trace} \quad \ell \quad ::= \quad \cdot \mid \mathsf{read}(b,n) \mid \mathsf{write}(b,n) \mid \mathsf{brT} \mid \mathsf{brF} \mid \ell_1, \ell_2$$

Protzenko et al. define an equivalence relation between Low⋆ memories and Low⋆ expressions ($H_1 \equiv_\Gamma H_2$ and $e_1 \equiv_\Gamma e_2$), which relates two memories and expressions that are equal except in subterms that have abstract types (per the type environment Γ). Their theorem then states that equivalent Low⋆ configurations (pairs of memories and expressions (H, e) that are point-wise equivalent) produce equal traces and equivalent configurations.

Proving Hybrid Programs Free from Information Leakage. To prove the security of hybrid programs, we must reconcile the notion of leakage in Vale and in Low⋆. To this end, we extend the Low⋆ secret-independence meta-theorem to account for interoperation with Vale programs.

First, we extend Low⋆'s formal syntax with an extern c expression form, that denotes the Vale code c embedded within Low⋆. We then model the call_assembly function presented in Section 4.2.1 as an opaque relation, that transforms a Low⋆ memory H, emitting an (abstract) trace z: $(H, \mathsf{extern}\ c) \longrightarrow_z H'$, along with the following extensions to the Low⋆ syntax:

$$\begin{array}{rcll} \text{Expression} & e & ::= & \cdots \mid \mathsf{let}\ v = \mathsf{extern}\ c\ \mathsf{in}\ e \\ \text{Extern trace} & z & & \\ \text{Trace} & \ell & ::= & \cdots \mid z \end{array}$$

The second and key component of the extension is lifting Vale's static taint analyzer (Section 4.1.3) to the meta-level in Low⋆– in particular the is_leakage_free property from Figure 4.6.

Proposition 4.1 (Meta-property about the Vale taint analyzer). *Let* $\Gamma \vdash \mathsf{extern}\ c$*. If* $\mathsf{taint\_analysis}(\Gamma, c) = \mathsf{true}$*, then for two well-typed heaps* H_1 *and* H_2 *s.t.* $H_1 \equiv_\Gamma H_2$*, we have* $(H_1, c) \longrightarrow_{z_1} H_1'$*,* $(H_2, c) \longrightarrow_{z_2} H_2'$*,* H_1' *and* H_2' *are well-typed in* Γ*,* $z_1 = z_2$*, and* $H_1' \equiv_\Gamma H_2'$*.*

Using the proposition above, it is straightforward to extend the secret-independence theorem from Protzenko et al. (2017) to include the new extern expression form. The detailed proof is available in Appendix A.

Theorem 4.2 (Secret-Independence for Hybrid Low*/Vale programs)**.** *Given configurations* (H_1, e_1) *and* (H_2, e_2)*, where* $\Gamma \vdash (H_1, e_1) : \tau$*,* $\Gamma \vdash (H_2, e_2) : \tau$*,* $H_1 \equiv_\Gamma H_2$ *and* $e_1 \equiv_\Gamma e_2$*, and a secret-independent implementation of the secret integer interface* P_s*, either both the configurations cannot reduce further, or* $\exists \Gamma' \supseteq \Gamma$ *s.t.* $P_s \vdash (H_1, e_1) \rightarrow^+_{\ell_1} (H_1', e_1')$*,* $P_s \vdash (H_2, e_2) \rightarrow^+_{\ell_2} (H_2', e_2')$*,* $\Gamma' \vdash (H_1', e_1') : \tau$*,* $\Gamma' \vdash (H_2', e_2') : \tau$*,* $\ell_1 = \ell_2$*,* $H_1' \equiv_{\Gamma'} H_2'$*, and* $e_1' \equiv_{\Gamma'} e_2'$*,*

The last step is to ensure that secret values are consistent in Vale and Low*, i.e., that a secret passed as argument from Low* is considered as such by the Vale taint analysis. To this end, our interoperation framework also relates secret types in Low* to the labeling function (isPub from Figure 4.6) used by the Vale static analyzer.

4.3 Summary

In the realm of unverified software, interoperation between languages is frequent; high-performance cryptography, for instance, commonly consists of hybrid C and assembly programs. In this chapter, we demonstrated how to model such interoperation in a proof assistant. We designed our interoperation to be easily extensible to new platforms, while defining suitable abstractions to simplify the verification effort required by developers when targeting hybrid programs. Building upon this work, we will show in the next chapter how our approach is usable at scale when developing a verified, industrial-grade cryptographic provider.

Chapter 5

EverCrypt: A Fast, Verified Cryptographic Provider

In this chapter, we demonstrate how the proof-oriented approach we advocate for can scale to high-asssurance, industrial-grade software. In the previous chapter, we proposed a verified interoperation layer enabling us to provide formal guarantees about hybrid C and assembly cryptographic implementations. Building upon this interoperation, we now present our proof engineering development process for a large, verified cryptographic library deployed in critical real-world settings.

When using industrial-grade cryptographic libraries such as `libsodium` (Denis 2013), OpenSSL' `libcrypto` (OpenSSL Team 2005), or the Windows Cryptography API (Microsoft 2018), developers have high expectations.

The minimal requirement of a *cryptographic provider* is to be *comprehensive*; a provider must supply all of the functionalities that security-critical applications need (asymmetric and symmetric encryption, signing, hashing, key derivation, . . .), for all the platforms that the application runs on. But modern cryptographic providers are expected to be more than a collection of cryptographic implementations, they should also follow software engineering best practices.

First, the cryptographic provider should be *agile*; that is, it should provide multiple algorithms (e.g., ChaCha-Poly (Nir and Langley 2015) and AES-GCM (NIST 2007)) for the same functionality (e.g., authenticated encryption), while providing a single, unified API, making it simple to change algorithms if one is broken (Stevens et al. 2007; Leurent and Peyrin 2020).

Additionally, a modern cryptographic provider should also support *multiplexing*, that is, the ability to choose between multiple implementations of the same algorithm. This allows the provider to employ high-performance implementations on popular hardware (OpenSSL, for example, supports dozens), while still providing a portable fallback implementation that will work on any platform. Ideally, these disparate implementations should also be exposed to the developer via a single unified API, so that the developer can easily switch to a different implementation when a new optimized version is deployed, and so that the provider can automatically choose the optimal implementation for the platform it runs on, instead of leaving this burden to the developer. Historically, this process has been error prone, with various cryptographic providers invoking illegal instructions on specific platforms (Debian Bug Tracker 2016), leading to killed processes and even crashing kernels.

As we discussed in detail in Section 2.1, several projects, including the Vale framework that we presented in Chapter 4, have produced verified implementations of different cryptographic algorithms. While all these works contributed important techniques and insights into how to best verify cryptographic code, they did not yield a verified cryptographic *provider* comparable to the unverified libraries that application developers use today.

To address this issue, we present in this chapter EverCrypt (Protzenko et al. 2020), a comprehensive, provably correct and secure cryptographic provider that supports agility, multiplexing and auto-configuration. EverCrypt builds upon two previous verification projects, HACL* (Zinzindohoué et al. 2017; Polubelova et al. 2020) which provides verified, high-performance C code for cross-platform support, and Vale (Bond et al. 2017; Fromherz et al. 2019) which produces assembly code for maximum performance on specific hardware platforms.

EverCrypt unifies these two projects under a single agile and multiplexing API; its agility ensures that multiple algorithms provably provide the same API to clients, while its multiplexing demonstrates multiple disparate implementations from Vale and HACL* verified against the same cryptographic specification. The API is carefully designed to be usable by both verified (e.g., F*) and unverified (e.g., C) clients; we describe this API in Section 5.2 and its implementation in Section 5.3.

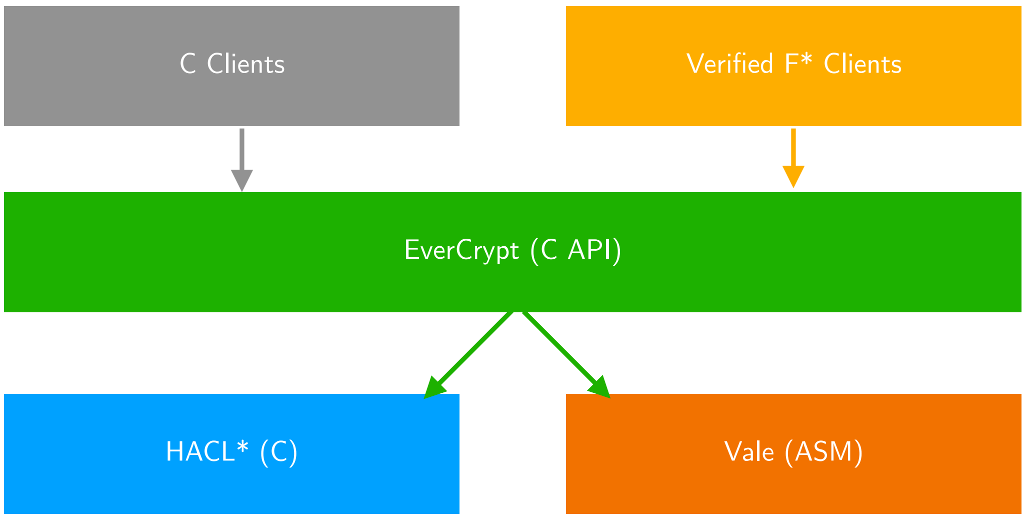


Figure 5.1: An overview of EverCrypt

Despite being verified, EverCrypt also provides state-of-the-art performance for a variety of algorithms including the elliptic curve Curve25519 (Bernstein 2006) (Section 5.4), as well as authenticated encryption with additional data (AEAD), and hash functions (Section 5.5). Leveraging the verified interoperation presented in Chapter 4, it transparently multiplexes between generic C implementations and hand-tuned x64 assembly implementations; EverCrypt's assembly code matches or exceeds the performance of the best unverified code, while the C implementations provide support across all other platforms, offering performance competitive with existing unverified C code. Since its release as an open-source library, portions of EverCrypt have been deployed in a variety of verified and unverified applications, including high-profile open-source projects such as Mozilla Firefox and the Linux kernel (Section 5.6).

Contributions. All the work presented in this chapter was first described in Protzenko et al. (2020). My primary contributions consist of using the interoperation layer presented in the previous chapter at scale, including for the fine-grained interoperation between Low⋆ and Vale in our implementation of Curve25519 (Section 5.4) and for the EverCrypt CPU-autodetection during multiplexing (Section 5.3). I was also responsible for integrating our state-of-the-art implementation of Curve25519 modular arithmetic into the Zinc cryptographic library, now in use in the Wireguard VPN and in the Linux kernel (Section 5.6). Additionally, I also implemented and verified several cryptographic primitives in EverCrypt, including the Chacha-Poly and HPKE cryptographic constructions and the Ed25519 elliptic curve, and also co-developed the EverCrypt API for authenticated encryption with additional data (AEAD).

5.1 Background: HACL⋆

HACL⋆ (Zinzindohoué et al. 2017) is a collection of cryptographic primitives: Chacha20, Poly1305, their AEAD combination, Curve25519, Ed25519 and the SHA2 family of hashes, entirely written in Low⋆ and compiling to a collection of C files. While most of these implementations provide cross-platform compatibility, a recent extension of HACL⋆, called HACLxN (Polubelova et al. 2020), exploits platform-specific single-instruction multiple data (SIMD) parallelism to provide verified C implementations whose performance is closer to hand-tuned assembly.

Like all code written in Low⋆, HACL⋆ is, by construction, devoid of memory errors such as use-after-free, or out-of-bound accesses. In addition to basic memory safety, HACL⋆ proves *functional correctness* for its algorithms; e.g., for the elliptic curve Curve25519, it shows that optimized field operations in $2^{255} - 19$ are free of mathematical errors (see Figure 5.2) Finally, using Low⋆'s constant-time model of secret integers, HACL⋆ ensures that one cannot branch on a secret or use it for array accesses, and thus guarantees that the resulting C code is free of the most egregious kinds of side-channels.

5.2 An Agile, Abstract API for EverCrypt

A key contribution of EverCrypt is the design of its API, which provides abstract specifications suitable for use both in verified cryptographic applications and in unverified code. While agility matters for security and functionality, we also find that it is an important principle to apply throughout our code: beneath the EverCrypt API, we

```
val fmul (output a b: felem): Stack unit
  (requires λh0 →
     live h0 a ∧ live h0 b ∧ live output ∧ (a == b ∨ disjoint a b))
  (ensures λh0 _ h1 →
     modifies_only output h0 h1 ∧ elem h1.[output] == mul (elem h0.[a]) (elem h0.[b]))
```

Figure 5.2: Type signature for HACL*'s implementation (in Low*) of a field multiplication for Curve25519. An felem is an array of five 64-bit integers needed to represent a field element. The input arrays a,b and the output array output are required to be live (i.e., still allocated). After fmul completes, the only change from the old heap h0 to the new heap h1 is to the output array, which matches mul, the mathematical specification of a field multiplication. The Stack effect annotation in its type guarantees that fmul is free of memory leaks, since its allocations are only on the call stack, and hence they are reclaimed as the function returns.

use agility extensively to build generic implementations with a high degree of code and proof sharing between variants of related algorithms.

Figure 5.3 outlines the overall structure of our API and implementations for hashing algorithms—similar structures are used for other classes of algorithms. At the top left, we have trusted, pure specifications of hashing algorithms. Our specifications are optimized for clarity, not efficiency; to make them trustworthy, we manually review them to ensure that they match the original informal specification (e.g., from an RFC). Additionally, our specifications are executable; we extract them to OCaml and test them using standard test vectors, enabling an early detection of basic errors such as typos or endianness issues. We discuss specifications further in Section 5.2.1

To the right of the figure, we have verified, optimized implementations. The top-level interface is EverCrypt.Hash, which multiplexes efficient, imperative implementations written in Low* and Vale. Each of these implementations is typically proven correct against a low-level specification (e.g., Derived.SHA2_256) better suited to proofs of implementation correctness than the top-level Spec.Hash. These derived specifications are then separately proven to refine the top-level specification. We discuss our top-level API in Section 5.2.2.

For reuse within our verified code, we also identify several generic idioms. For instance, we share a generic Merkle-Damgård construction (Merkle 1989; Damgard 1989) between all supported hash algorithms. Similarly, we obtain all the SHA2 variants from a generic template. The genericity saves verification effort at zero runtime cost—using F*'s support for partial evaluation, our code extracts to fully specialized C and assembly implementations, as described in Section 5.3.

5.2.1 Writing Algorithm Specifications for EverCrypt

Trusted specifications define what a given algorithm (e.g., SHA-256) should do; in cryptography, these definitions typically appear in a mixture of English, pseudocode, and math in an RFC or a peer-reviewed paper. Within EverCrypt, we port these

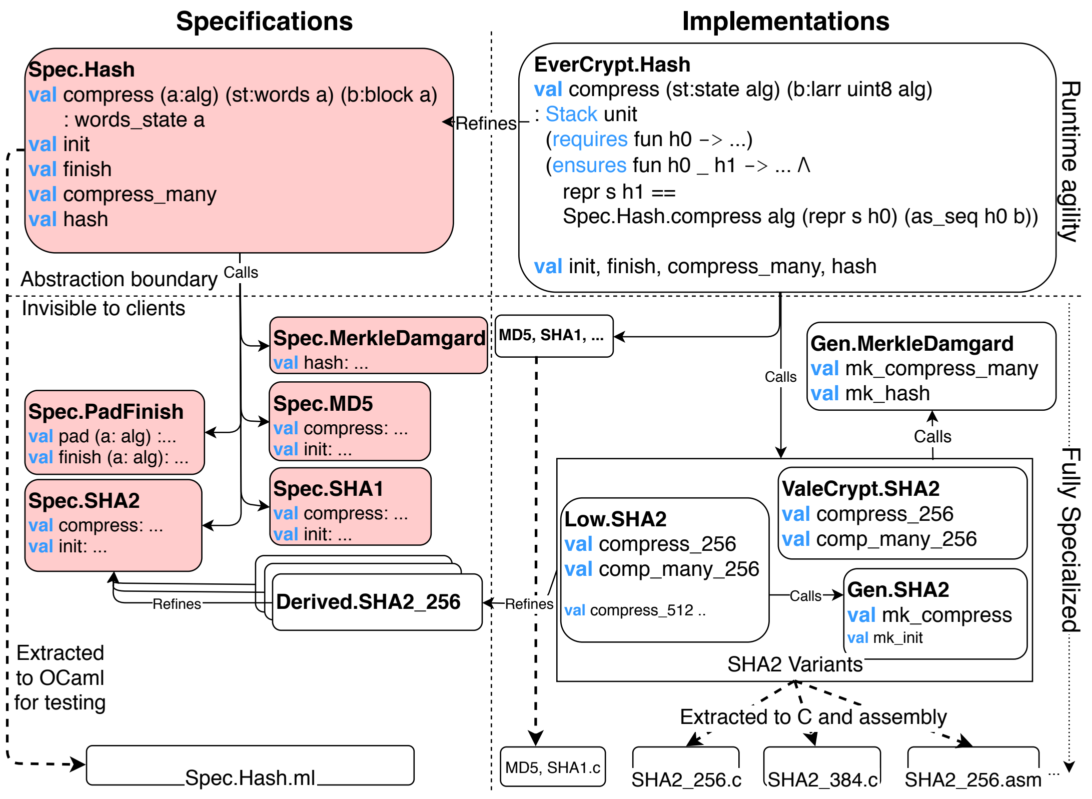


Figure 5.3: The modular structure of EverCrypt (illustrated on hashing algorithms). Components in red are part of our Trusted Computing Base (TCB).

specifications to pure mathematical functions within F*. This process is trusted, and hence we take steps to enhance its trustworthiness. We strive to keep the trusted specifications concise and declarative to facilitate human-level audits.

Taming Specification Explosion via Agility. We factor common structure shared by multiple specifications into "generic" functions parameterized by an algorithm parameter, and helper functions that branch on it to provide algorithm-specific details. This reduces the potential for errors, makes the underlying cryptographic constructions more evident, and provides a blueprint for efficient generic implementations (Section 5.3).

For example, the type below enumerates the hashing algorithms that EverCrypt supports:

```
type alg = MD5 | SHA1 | SHA2_224 | SHA2_256 | SHA2_384 | SHA2_512
```

Although MD5 and SHA1 are known to be insecure (Stevens et al. 2007; Leurent and Peyrin 2020), a practical provider must supply them for compatibility reasons. At the application level, cryptographic security theorems can be conditioned on the security of the algorithms used and, as such, would exclude MD5 or SHA1. Pragmatically, EverCrypt can also be configured to disable them, or even exclude their code at compile time.

All these algorithms use the Merkle-Damgård construction for hashing a bytestring by (1) slicing it into input blocks, with an encoding of its length in the final block; (2) calling a core, stateful compression function on each block; (3) extracting the hash from the final state. Further, the four members of the SHA2 family differ only on the lengths of their input blocks and resulting tags, and on the type and number of words in their intermediate state.

Rather than write different specifications, we define a generic state type, which is parameterized by the algorithm. Depending on the algorithm, the type word alg selects 32-bit or 64-bit unsigned integer words; words alg defines sequences of such words of the appropriate length; and block alg defines sequences of bytes of the appropriate block length.

```
let word alg = match alg with
  | MD5 | SHA1 | SHA2_224 | SHA2_256 → UInt32.t
  | SHA2_384 | SHA2_512 → UInt64.t

let words_length alg = match alg with
  | MD5 → 4
  | SHA1 → 5
  | SHA2_224 | SHA2_256 | SHA2_384 | SHA2_512 → 8

let words alg = m:seq (word alg){length m = words_length alg }

let block_length alg = match alg with
  | MD5 | SHA1 | SHA2_224 | SHA2_256 → 64
  | SHA2_384 | SHA2_512 → 128

let block alg = b:bytes{length b = block_length alg}
```

With these types, we write a generic SHA2 compression function that updates a hash state (st) by hashing an input block (b). Note, this definition illustrates the benefits of programming within a dependently typed framework—we define a single function that operates on either 32-bit or 64-bit words, promoting code and proof reuse and reducing the volume of trusted specifications.

```
module Spec.SHA2

let word_add_mod (alg:sha2_alg) = match alg with
  | SHA2_224 | SHA2_256 → UInt32.add_mod
  | SHA2_384 | SHA2_512 → UInt64.add_mod

let compress (alg:sha2_alg) (st:words alg) (b:block alg) : words_state alg
  = let block_words = words_of_bytes alg 16 b in
    let st' = shuffle alg st block_words in
    seq_map2 (word_add_mod alg) st st'
```

This function first converts its input from bytes to words, forcing us to deal with endianness—being mathematical, rather than platform dependent, our specifications fix words to be little endian. The words are then shuffled with the old state (st) to produce a new state (st') which is then combined with the old state via modular addition, all in an algorithm-parameterized manner. For instance, word_add_mod is parameterized by the SHA2 algorithm, and corresponds to an addition modulo 2^{32} for SHA2-224 and SHA2-256, and modulo 2^{64} for SHA2-384 and SHA2-512.

Going beyond the SHA2 family of algorithms, we compose multiple levels of specification sharing. For instance, we write a single agile padding function (in Spec.PadFinish) for MD5, SHA1, and SHA2; we also use a small helper function which branches on the algorithm alg to encode the input length in little-endian or big-endian, depending on whether the algorithm is MD5.

Untrusted Specification Refinements. EverCrypt's trusted algorithm specifications are designed to be concise and easily auditable, but they rarely lend themselves well to an efficient implementation. Hence, we often find it useful to develop untrusted specification refinements (e.g., Derived.SHA2_256) that provide more concrete details and are *proven* equivalent to the trusted specifications.

These refinements commonly introduce features such as more optimized representation choices, precomputations, or chunking of the operations in blocks to enable vectorized implementations. In Curve25519, for example, an algorithm refinement may introduce a Montgomery ladder (Montgomery 1987) for efficiently computing scalar multiplications.

In the case of hashes, instead of waiting for the entire message to be available and holding all the data in memory, a refined specification processes its input incrementally, and is verified against the base algorithm. This process of relating specifications to implementations through iterative refinements leads to well-structured, modular, and easier to maintain proofs.

5.2.2 EverCrypt's Top-Level API

Verified programming is a balancing act: programs must be specified precisely, but revealing too many details of an implementation breaks modularity and makes it difficult to revise or extend the code without also breaking clients.

A guiding principle for EverCrypt's API is to hide, through abstraction, as many specifics of the implementation as possible. Our choice of abstractions has been successful inasmuch as, after establishing our verified API, we have extended its implementation with new algorithms and optimized implementations of existing algorithms without any change to the API.

In EverCrypt, we use abstraction in two flavors. *Specification* abstraction hides details of an algorithm's specification from its client; e.g., although EverCrypt.Hash.compress is proven to refine Spec.Hash.compress, only the type signature of the latter, not its definition, is revealed to clients. In addition, *representation* abstraction hides details of an implementation's data structures, e.g., the type used to store the hash's internal state is hidden.

Abstract specifications have a number of benefits. They ensure that clients do not rely on the details of a particular algorithm, and that their code will work for any present or future hash function that is based on a compression function. Abstract specifications lend themselves to clean, agile specifications for cryptographic constructions (such as the Merkle-Damgård construction previously discussed). Abstraction also allows us to provide a defensive API to unverified C code, helping to minimize basic API usage flaws. Finally, abstraction also simplifies reasoning, both formally and informally, to establish the correctness of client code. In practice, abstract specifications prune the proof context presented to the proof assistant; when relying on semi-automated verification, as F⋆ does using the Z3 SMT solver, this can significantly speed up client verification.

For hashing functions, our main, low-level, imperative API is designed around an algorithm-indexed, abstract type state alg. EverCrypt clients are generally expected to observe a usage protocol. For instance, the hash API expects clients to allocate state, initialize it, then make repeated calls to compress, followed eventually by finalize. EverCrypt also provides a single-shot hash function in the API for convenience.

The interface of our low-level compress function is shown below, with some details elided. Clients of compress must pass in an abstract state s (indexed by an implicit algorithm descriptor alg), and a mutable array b holding a block of bytes to be added to the hash state. As a precondition, they must prove inv s h0, the abstract invariant of the state. This invariant is established initially by the state allocation routine, and standard framing lemmas ensure that the invariant holds for subsequent API calls as long as any intervening heap updates are to disjoint regions of the heap.

```
module EverCrypt.Hash

val compress (s:state alg) (b:block alg) : Stack unit
  (requires λh0 →
    inv s h0 ∧ live h0 b ∧ fp s h0 `disjoint` loc b)
  (ensures λh0 _ h1 →
    inv s h1 ∧ modifies_only (fp s h0) h0 h1 ∧
    repr s h1 == Spec.Hash.compress alg (repr s h0) (as_seq h0 b)
```

In addition to the invariant, clients must prove that the block b is live; and that b does not overlap with the abstract footprint of s (the memory locations of the underlying hash state). As usual in Low⋆, the Stack unit annotation states that compress is free of memory leaks and returns the uninformative unit value ().

The postcondition of compress ensures that the abstract invariant inv is preserved; it guarantees that only memory locations corresponding to the internal hash state are modified; and, most importantly, it states that the final value held in the hash state, repr s h1, corresponds *exactly* to the words computed by the pure specification Spec.Hash.compress. It is this last part of the specification that captures functional correctness, and justifies the safety of multiplexing several implementations of the same algorithm behind the API, inasmuch as they are verified to return the same results, byte for byte.

State abstraction is reflected to C clients as well, by compiling the state type as a pointer to an incomplete struct (Seacord 2018). Hence, after erasing all pre- and post-conditions, our low-level interface yields idiomatic C function declarations in the extracted evercrypt_hash.h presented in Figure 5.4.

```
// This type is shared with specifications.
#define SHA2_224 0
#define SHA2_256 1
#define SHA2_384 2
#define SHA2_512 3
#define SHA1 4
#define MD5 5
typedef uint8_t hash_alg;

// Interface for hashes, block-aligned input data
struct state_s_s;
typedef struct state_s_s state_s;

state_s *create_in(hash_alg a);
void init(state_s *s);
void compress(state_s *s, uint8_t *block1);
void compress_many(state_s *s, uint8_t *blocks, uint32_t len1);
void compress_last(state_s *s, uint8_t *last1, uint64_t total_len);
void finish(state_s *s, uint8_t *dst);
void free(state_s *s);
void copy(state_s *s_src, state_s *s_dst);

void hash(hash_alg a, uint8_t *dst, uint8_t *input, uint32_t len1);
```

Figure 5.4: A representative snippet of the EverCrypt hash API. The file was edited only to remove some module name prefixes to make the code more compact.

Given an abstract, agile, functionally correct implementation of our 6 hash algorithms, we develop and verify the rest of our API for hashes in a generic manner. We first build support for incremental hashing (similar to compress, but taking variable-sized bytestrings as inputs), then an agile standard-based implementation of keyed-hash message authentication codes (HMAC) and finally, on top of HMAC, a verified implementation of key derivation functions (HKDF) (Krawczyk and Eronen 2010).

Thanks to agility, adding a new algorithm (e.g., a hash) boils down to extending an enumeration (e.g., the hash algorithm) with a new case. This is a backward-compatible change that leaves function prototypes identical. Thanks to multiplexing, adding a new optimized implementation is purely an implementation matter that is dealt with automatically within the library, meaning once again that such a change is invisible to the client. Finally, thanks to an abstract state and framing lemmas, EverCrypt can freely optimize its representation of state, leaving verified and unverified clients equally unscathed.

5.3 Agile Implementations with Zero-Cost, Generic Programming

While agility yields clean specifications and APIs, we now show how to program *implementations* in a generic manner, and still extract them to fully specialized code with zero runtime overhead. To ground the discussion, we continue with our running example of EverCrypt's hashing API, instantiating the representation of the abstract state handle (state a) and sketching an implementation of EverCrypt.Hash.compress, which supports runtime agility and multiplexing by dispatching to implementations of specific algorithms.

Implementing EverCrypt.Hash. The abstract type state alg is defined in F⋆ as a pointer to a datatype containing an algorithm-specific state representation, as shown in Figure 5.5

```
let hash_state (a:alg) = st:array (word a){length st == words_length a}

type state_s (a:alg) = match a with
| SHA2_256: p:hash_state SHA2_256 → state_s SHA2_256
| SHA2_384: p:hash_state SHA2_384 → state_s SHA2_384
| . . .

let state alg = pointer (state_s alg)
```

Figure 5.5: F⋆ definition of the EverCrypt hash state

The state_s type is extracted to C as a tagged union, whose tag indicates the algorithm alg and whose value contains a pointer to the internal state of the corresponding algorithm, as presented in Figure 5.6.

```
struct state_s_s {
  hash_alg tag;
  union {
    uint32_t *case_SHA2_256;
    uint64_t *case_SHA2_384;
    . . .
  }
}
```

Figure 5.6: Extracted C state for EverCrypt hash functions

Compared to, say, a single void∗, the union incurs no space penalty, and prevents dangerous casts from void∗ to one of uint32_t∗ or uint64_t∗. The tag allows an agile hash implementation to dynamically dispatch based on the algorithm, as shown in Figure 5.7 for compress.

```
let compress s blocks = match !s with
  | SHA2_256 p → compress_multiplex_sha2_256 p blocks
  | SHA2_384 p → compress_sha2_384 p blocks
```

Figure 5.7: Agile F⋆ implementation of the compress hash function

In this code, since we only have one implementation of SHA2-384 provided by HACL⋆, we directly call into Low⋆. For SHA2-256, however, we have multiple implementations available in both Low⋆ and Vale. We thus dispatch to a local multiplexer that selects the best available implementation, based on other runtime configurations in scope, including CPU identity. We describe this mechanism in more detail later in this section.

Partial Evaluation for Zero-Cost Genericity. Abstract specifications and implementations, while good for encapsulation, modularity, and code reuse, can compromise the efficiency of executable code: We want to ensure that past the agile EverCrypt.Hash, nothing impedes the run-time performance of our code.

To this end, we rely on partial evaluation to derive specialized Low⋆ code suitable for calling by EverCrypt.Hash, reducing away several layers of abstraction before further compilation. The C code thus emitted is fully specialized and abstraction-free, and branching on algorithm descriptors only happens above the specialized code, where the API demands support for runtime configurability (e.g., only at the top-level of EverCrypt.Hash). We therefore retain the full generality of the agile, multiplexed API, while switching efficiently and only at a coarse granularity between fast, abstraction-free implementations.

Consider our running example: compress, the compression function for SHA-2. We managed to succintly specify all variants of this function at once, using case-generic types like word alg to cover algorithms based on both 32- and 64-bit words. Indeed, operations on word alg like word_logand below dispatch to operations on 32-bit and 64-bit integers depending on the specific variant of SHA-2 being specified.

```
let word_logand (alg:sha2_alg) (x y: word alg): word alg =
  | SHA2_224 | SHA2_256 → UInt32.logand x y
  | SHA2_384 | SHA2_512 → UInt64.logand x y
```

We wish to retain this concise style and, just like with specifications, write a *stateful* shared compress_sha2 *once*. This cannot however be done naively, as implementing bitwise-and within compress_sha2 would be a performance disaster: every bitwise-and would trigger a case analysis on the algorithm! Further, word alg would have to be compiled to a C union, also wasting space.

To combine the best of both worlds, we instead rely on inlining and F⋆'s capabilities for partial evaluation. We program most of our *stateful*, low-level code in a case-generic manner. Just like in specifications, the stateful compression function is written once (in Gen.SHA2); we trigger code specialization at the top-level by defining all the concrete instances of our agile implementation, as shown in Figure 5.8.

When extracting compress_sha2_256 for instance, F⋆ will partially evaluate the func-

```
module Low.SHA2

let compress_sha2_224 = Gen.SHA2.compress SHA2_224
let compress_sha2_256 = Gen.SHA2.compress SHA2_256
let compress_sha2_384 = Gen.SHA2.compress SHA2_384
let compress_sha2_512 = Gen.SHA2.compress SHA2_512
```

Figure 5.8: Specializing a generic compress implementation to different hash algorithms

tion Gen.SHA2.compress on SHA2_256, eventually encountering word_logand SHA2_256 and reducing it to UInt32.logand. By the time all reduction steps have been performed, no case analysis over an algorithm remains; all functions and types parameterized over alg have disappeared, leaving specialized implementations for the types, operators, and constants that are specific to SHA2-256 and can be compiled to efficient, idiomatic C code.

We take this style of partial evaluation one step further, and parameterize stateful code over algorithms *and* stateful functions. For instance, we program a generic, *higher-order* Merkle-Damgård hash construction, instantiating it with specific compression functions, including multiple implementations of the same algorithm, e.g., HACL* and Vale implementations. Specifically, the mk_compress_many function presented in Figure 5.9 is parameterized by a compression function f, and repeatedly applies f to the input. We can instantiate it with several implementations of the same algorithms as long as it satisfies the shared, agile EverCrypt hash API, relying on F* performing partial evaluation to extract distinct, idiomatic C implementations.

```
let compress_t (alg:hash_alg) = s:state alg → b:block alg → Stack unit
  (requires λh0 →
    inv s h0 ∧ live h0 b ∧ fp s h0 `disjoint` loc b)
  (ensures λh0 _ h1 →
    inv s h1 ∧ modifies_only (fp s h0) h0 h1 ∧
    repr s h1 == Spec.Hash.compress alg (repr s h0) (as_seq h0 b)

val mk_compress_many (alg:hash_alg) (f:compress_t alg) : compress_many_t alg

let compress_many_256_vale : compress_many_t SHA2_256 =
  mk_compress_many SHA2_256 compress_sha2_256_vale
let compress_many_256_lowstar : compress_many_t SHA2_256 =
  mk_compress_many SHA2_256 compress_sha2_256_lowstar
```

Figure 5.9: A generic, higher-order Merkle-Damgård construction, parameterized by an algorithm-generic compress function. The generic construction is instantiated with Vale and Low* implementations of SHA2-256.

The higher-order pattern allows for a separation of concerns: the many-block compression function does not need to be aware of *how* to multiplex between Low* and Vale, or even of *how many* choices there might be; the function type abstraction of compress is sufficient. We rely on this higher-order style in the rest of our hash API: for instance, given an implementation of compress and pad functions, mk_compress_last generates a compression function for the remainder of the input data; similarly, mk_hash requires state creation and initialization, as well as compress, pad, and finish functions to generate a complete one-shot hash function. Using these higher-order templates, we instantiate the entire set of SHA2 functions, yielding two specialized APIs with no branching: one for Low⋆ code, one for Vale code.

Safely Multiplexing Between Implementations. Hand-tuned assembly implementations provided by Vale are more performant than portable HACL⋆ implementations; when the two are available, we would prefer using the fastest one. Unfortunately, assembly implementations commonly rely on platform-specific features, e.g., an optimized Vale implementation of the authenticated encryption algorithm AES-GCM (NIST 2007) relies on the Intel x64 AES-NI instruction set (Gueron 2012). Executing this code on a platform without the AES-NI extension would result in errors due to illegal (unsupported) instructions, an issue that led in the past to killed processes and crashing kernels (Debian Bug Tracker 2016). To safely multiplex between implementations, EverCrypt must therefore ensure that Vale implementations are only called when the corresponding CPU extensions are supported.

To determine whether a given extension is available on the current platform, we rely on calls to the CPUID instruction, which is modeled in Vale. Using the verified interoperation presented in Chapter 4, we can therefore provide the following Low⋆ signature to a CPUID call detecting whether the Vale AES-GCM implementation can be safely executed. The modifies_only loc_none postcondition ensures that this call has no effect on memory. For performance reasons, instead of performing such calls at each multiplexing, EverCrypt instead caches the results during a static configuration phase.

```
val check_aesni : unit → Stack UInt64.t
  (requires λh0 → ⊤)
  (ensures λh0 ret_val h1 →
    modifies_only loc_none h0 h1 ∧ ((UInt64.v ret_val) ≠ 0 ⟹ aesni_enabled)
  )
```

A further complication arises because the CPUID instruction itself is only supported on Intel x86 and x64 platforms, but not, say, on ARM. Attempting to call check_aesni on non-Intel platforms would therefore lead, again, to errors due to illegal instructions. Hence, we add another layer of flags representing static compiler-level platform information (e.g., TargetConfig.x64). Checks for these flags are compiled as a C #ifdef. The emitted code for auto-detecting CPU features thus looks as follows:

```
#if TARGETCONFIG_X64
  if (check_aesni () != 0U) cpu_has_aesni[0U] = true;
#endif
```

This ensures that no link-time error occurs when compiling EverCrypt for platforms which do not support the CPUID instruction. The connection of the TARGETCONFIG_ macros to standard compiler definitions, e.g.,

```
#if defined(__x86_64__) || defined(_M_X64)
#define TARGETCONFIG_X64 1
#elif ...
```

are hand-written, and as such, must be carefully audited.

5.4 Achieving Best-in-Class Performance: The Case of Curve25519

Cryptographic performance is often a bottleneck for security-sensitive applications (e.g., for TLS or disk encryption). Given a choice between a complex high-performance cryptographic library, and a simple, potentially more secure one, historically much of the world has opted for better performance. In this section, we show that performance and security guarantees are not incompatible, by presenting a verified implementation of Curve25519 whose performance exceeds even the best unverified implementations.

Curve25519 (Bernstein 2006), standardized as IETF RFC7748 (Langley et al. 2016) is quickly emerging as the default elliptic curve for cryptographic applications. It is the only elliptic curve support by modern protocols like Signal (Marlinspike and Perrin 2016; Perrin and Marlinspike 2016) and Wireguard (Donenfeld 2017), and is one of the two curves commonly used with Transport Layer Security (TLS) and Secure Shell (SSH). One of the reasons for Curve25519's success is that it was designed with performance in mind, and many high-performance implementations have been published since its standardization (Bernstein 2006; Chou 2016; Düll et al. 2015; Oliveira et al. 2017).

5.4.1 Implementing Curve25519, an Overview

Curve25519 can be implemented in about 500 lines of C. About half of this code consists of a customized bignum library that implements modular arithmetic over the field of integers modulo the prime $p_{25519} = 2^{255} - 19$. The most performance-critical functions implement multiplication and squaring over this field, which internally rely on modular addition. Since each field element has up to 255 bits, it can be stored in 4 64-bit machine words, encoding a polynomial of the form:

$$e3 * 2^{192} + e2 * 2^{128} + e1 * 2^{64} + e0$$

where each coefficient is less than 2^{64}. Multiplying (or squaring) field elements amounts to textbook multiplication with a 64-bit radix: whenever a coefficient in an intermediate polymomial goes beyond 64-bits, we need to carry over the extra bits to the next higher coefficient. To avoid a timing side-channel, we must assume that every 64-bit addition may lead to a carry and propagate the (potentially zero) carry bit regardless.

Propagating these carries can be quite expensive, so a standard optimization is to delay carries by using an *unpacked* representation, with a field element stored in 5 64-bit machine words, each holding 51 bits, yielding a radix-51 polynomial:

$$e4 * 2^{204} + e3 * 2^{153} + e2 * 2^{102} + e1 * 2^{51} + e0$$

This representation means that polynomial multiplication now requires 25 64x64 multiplications (each yielding a 128-bit integer) rather than just 16 such multiplications in the radix-64 representation. However, since each product now has only 102 bits, it has lots of room to hold extra carry bits without propagating them until the final modular reduction.

Even with these delayed carries, carry propagation continues to be a performance bottleneck for modular multiplication, and high-performance implementations leverage many low-level optimizations, such as interleaving carry chains, and skipping some carry steps if the developer believes that a given coefficient is below a threshold.

Such delicate optimizations have often lead to functional correctness bugs, both in popular C implementations like Donna-64 (Langley 2014) and in high-performance assembly like amd64-64-24k (Bernstein et al. 2014). These bugs are particularly hard to find by testing or auditing, since they only occur in low-probability corner cases deep inside modular arithmetic. Nevertheless, such bugs may be exploitable by a malicious adversary once they are discovered, making elliptic curves a prime target for formal verification.

5.4.2 A Faster Curve25519 with Intel ADX

Recently, Oliveira et al. (2017) demonstrated a significantly faster implementation on Intel platforms that support Multi-Precision Add-Carry Instruction Extensions, also called Intel ADX. Unlike other fast Curve25519 implementations, Oliveira et al. use a radix-64 representation and instead optimize the carry propagation code by carefully managing Intel ADX's second carry flag. The resulting performance improvement is substantial—at least 20% faster than prior implementations on modern Intel processors.

Oliveira et al. mostly wrote their implementation in assembly, with only the Montgomery laddder and top-level functions written in C. A year after its publication, when testing and comparing this code against formally verified implementations from HACL* (Zinzindohoué et al. 2017) and Fiat-Crypto (Erbsen et al. 2019), Donenfeld and others found several critical correctness bugs (Donenfeld 2018b). These bugs were fixed (Dettman 2018) with a minor loss of performance, but they raised concerns as to whether this Curve25519 implementation, with the best published performance, is trustworthy enough for deployment in mainstream applications.

To address this issue, we develop in EverCrypt several verified implementations of Curve25519. The first is written in Low*, and generates portable C code that uses an efficient radix-51 representation. The second is inspired by Oliveira et al.'s work, and relies on verified Vale assembly for low-level field arithmetic that uses a radix-64 representation.

Notably, we carefully factor out the *generic* platform-independent Curve25519 code, including field inversion, curve operations, key encodings, etc., so that this code can be shared between our two implementations. In other words, we split our Curve25519 implementation into two logical parts:

1. The low-level field arithmetic, implemented both in Vale and in Low*, but verified against the same mathematical specification in F*

2. High-level components of Curve25519, implemented in Low*, that can use either of the two low-level field-arithmetic implementations.

A Generic Curve Implementation. To be generic in the underlying low-level field arithmetic, our high-level curve implementation relies on the methodology presented in Section 5.3; instead of parameterizing over a hash algorithm, we will parameterize over the field representation, as presented in Figure 5.10.

The type felem corresponds to the representation of a field element; the unpacked representation (radix-51) contains 5 64-bit integers, while the packed representation (radix-64) only requires 4 64-bit integers. Given such a representation, feval extracts the corresponding field element, using either the radix-51 or the radix-64 polynomial previously presented. Finally, fadd is a Low* function performing field addition, as stated by the postcondition feval s h1 out == Spec.Curve25519.fadd (feval s h0 f1) (feval s h0 f2). The additional fadd_pre s h and fadd_post s h0 h1 enable the specification of representation-specific conditions; for instance, requiring specific CPU extensions to be enabled for the Vale radix-64 implementation.

```
type field_spec = M51 | M64

let nlimb (s:field_spec) = match s with
  | M51 → 5ul
  | M64 → 6ul

let felem (s:field_spec) = e:array uint64{e.length == nlimb s}

let feval (s:field_spec) (h:mem) (e:felem s) : nat = match s with
  | M51 → (f51_as_nat h e) % Spec.Curve25519.prime
  | M64 → (f64_as_nat h e) % Spec.Curve25519.prime

val fadd (s:field_spec) (out : felem s) (f1 : felem s) (f2 : felem s)
  : Stack unit
      (requires λh → fadd_pre s h ∧
          live h out ∧ live h f1 ∧ live h f2 ∧ disjoint_or_eq [out; f1; f2])
      (ensures λh0 _ h1 →
        modifies_only out h0 h1 ∧ fadd_post s h0 h1 ∧
        feval s h1 out == Spec.Curve25519.fadd (feval s h0 f1) (feval s h0 f2))
```

Figure 5.10: A representation-parametric signature for Curve25519 field addition

Building upon this abstraction, we can then implement representation-agnostic Curve25519 functions. For instance, the scalar multiplication whose signature is presented in Figure 5.11 internally relies on a Montgomery ladder, which mostly consists of calls to representation-parametric fadd and fmul (field multiplication). As for the hash algorithms in Section 5.3, this generic implementation will ultimately be specialized at extraction-time.

```
val scalarmult (s:field_spec) (o:array uint8) (k:array uint8) (i:array uint8)
  : Stack unit
    (requires λh0 →
        o.length == 32 ∧ k.length == 32 ∧ i.length == 32 ∧
        scalarmult_pre s h0 ∧ live h0 o ∧ live h0 k ∧ live h0 i ∧ disjoint o i ∧ disjoint o k)
    (ensures λh0 _ h1 → modifies_only o h0 h1 ∧
        as_seq h1 o == Spec.Curve25519.scalarmult (as_seq h0 k) (as_seq h0 i))
```

Figure 5.11: Signature of the Curve25519 scalar multiplication

Implementing Low-Level Field Arithmetic To provide a full implementation of Curve25519, the remaining step is to provide implementations of the field arithmetic operations. The radix-51 version is implemented in Low⋆, yielding a portable C implementation, while the radix-64 variant, inspired by Oliveira et al.'s work, is written in Vale. To provide the best possible performance, calls into Vale functions rely on inline assembly, as shown in Figure 5.12; to ensure that these calls are safe and match the corresponding Low⋆ function types (e.g., fadd_t), we leverage the verified interoperation presented in Chapter 4. The resulting artifact thus is a mixed assembly-C implementation of Curve25519, with formal proofs of memory safety, functional correctness, and secret-independence for the assembly code, C code, and the glue code in between.

```
static inline void fadd (uint64_t *out, uint64_t *f1, uint64_t *f2)
{
  asm volatile(
    // Compute the raw addition of f1 + f2
    " movq 0(%0), %%r8;"
    " addq 0(%2), %%r8;"
    ...

    /////// Wrap the result back into the field //////
    ...

  : "+&r" (f2)
  : "r" (out), "r" (f1)
  : "%rax", "%rcx", "%r8", "%r9", "%r10", "%r11", "memory", "cc"
  );
}
```

Figure 5.12: Extracted inline assembly code for Vale implementation of fadd

Evaluating Curve25519's Performance We finally measure the performance of our implementations of Curve25519 against that of other implementations, including OpenSSL, Fiat-Crypto (Erbsen et al. 2019) (one of the fastest verified implementations at the time of writing), and Oliveira et al. (2017) (one of the fastest unverified implementations at the time of writing).

Experimental results are available in Figure 5.13. We compare the average number of CPU cycles needed to perform a scalar multiplication on Curve25519; a lower number corresponds to a faster implementation. All measurements are collected on an Intel Kaby Lake i7-7560 using a Linux kernel crypto benchmarking suite (Donenfeld 2018a). OpenSSL cannot be called from the kernel and was benchmarked using the same script, but in user space. All code was compiled with GCC 7.3 with flags −O3 −march=native −mtune=native.

Implementation	Radix	Language	CPU cycles
`donna64` (Langley 2008)	51	64-bit C	159634
`fiat-crypto` (Erbsen et al. 2019)	51	64-bit C	145248
`amd64-64` (Chen et al. 2014)	51	Intel x86_64 asm	143302
`sandy2x` (Chou 2016)	25.5	Intel AVX asm	135660
EverCrypt portable	51	64-bit C	135636
`openssl` (OpenSSL Team 2005)	64	Intel ADX asm	118604
Oliveira et al. (2017)	64	Intel ADX asm	115122
EverCrypt targeted	64	64-bit C + Intel ADX asm	113614

Figure 5.13: Performance comparison between Curve25519 Implementations.

Our results show that EverCrypt's combined Low* + Vale implementation narrowly exceeds that of Oliveira et al. by about 1%, which in turn exceeds that of OpenSSL by 3%. We also exceed the previous best verified implementation from Erbsen et al. by 22%. Hence, despite being verified, our hybrid C and assembly implementation was the fastest existing implementation on record at the time of writing.

5.5 Evaluation

With EverCrypt, we aimed to develop a verified, industrial-grade cryptographic provider. We showed in the previous sections how EverCrypt provides support for agility and multiplexing, two requirements of modern providers. In this section, we now compare its comprehensiveness and its performance with those of existing (unverified) cryptographic libraries.

5.5.1 EverCrypt Features

We summarize in Figure 5.14 the algorithms and systems supported by EverCrypt. As the table highlights, EverCrypt provides a variety of functionalities, including hashing, key derivation, cipher modes, message authentication, authenticated encryption with additional data (AEAD), elliptic curve operations, as well as several high-level APIs for performing public-key and secret key encryption (Box and SecretBox), and a new

Algorithm	C version	Targeted ASM version
AEAD		
AES-GCM		AES-NI + PCLMULQDQ + AVX
Chacha-Poly	yes	
High-level APIs		
Box	yes	
SecretBox	yes	
HPKE	yes	
Hashes		
MD5	yes	
SHA1	yes	
SHA2	yes	SHA-EXT (for SHA2-224+SHA2-256)
SHA3	yes	
Blake2	yes	
MACS		
HMAC	yes	agile over hash
Poly1305	yes	X64
Key Derivation		
HKDF	yes	agile over hash
ECC		
Curve25519	yes	BMI2 + ADX
Ed25519	yes	
P-256	yes	
Ciphers		
ChaCha20	yes	
AES128, 256		AES NI + AVX
AES-CTR		AES NI + AVX

Figure 5.14: Algorithms and systems supported by EverCrypt

cryptographic construction for Hybrid Public Key Encryption (HPKE) (Barnes and Bhargavan 2020).

In most cases, EverCrypt provides both a generic C implementation for cross-platform support, relying on HACL*, as well as a Vale optimized implementation for specific Intel x64 targets. EverCrypt automatically detects whether to employ the latter, and it offers agile interfaces for AEAD, hashing, HMAC and HKDF. HMAC and HKDF both build on the agile hash inteface, and hence inherit targeted implementations on supported platforms.

EverCrypt does not yet support agility over elliptic curves, nor does it yet support older asymmetric algorithms like RSA. We have also, thus far, focused on optimized implementations for x64, but prior work with Vale (Bond et al. 2017) demonstrates that we could just easily target other platforms; providing additional hand-tuned implementations for, say, ARM, would thus be possible using our methodology.

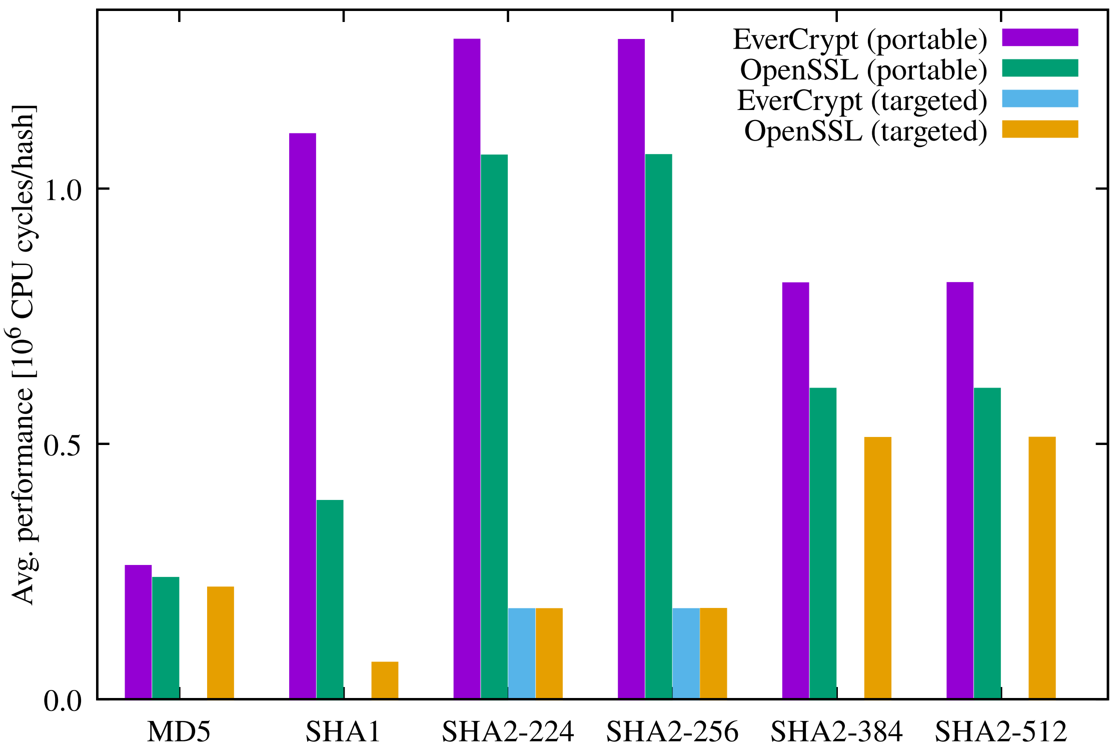


Figure 5.15: Average number of CPU cycles to compute a hash of 64 KB of random data. Lower is better.

With regards to the comprehensiveness of EverCrypt's API, the most natural (unverified) comparison is with libsodium (Denis 2013), which also aims to offer a clean API for modern cryptographic algorithms. The functionality exposed by each is quite comparable, with a few exceptions. Notably, EverCrypt current lacks a dedicated password-hashing API that uses a memory-hard hash function like Argon2 (Biryukov et al. 2016).

5.5.2 EverCrypt Run-Time Performance

EverCrypt aims to demonstrate that verification need not mean sacrificing performance. We presented in Section 5.4 an implementation of the Curve25519 elliptic curve whose performance exceeded those of the best verified and unverified implementations. We now evaluate the performance of EverCrypt's AEAD and hashing functions against OpenSSL's implementations, which prior work measured as meeting or exceeding that of other popular open-source cryptographic providers (Bond et al. 2017).

In our results, each data point represents an average of 1000 trials; error bars are omitted as the tiny variance makes them indistinguishable. All measurements are collected with hyperthreading and dynamic-processor scaling (e.g., Turbo Boost) disabled. We collect measurements on different platforms, since no single CPU supports all of our various targeted CPU features.

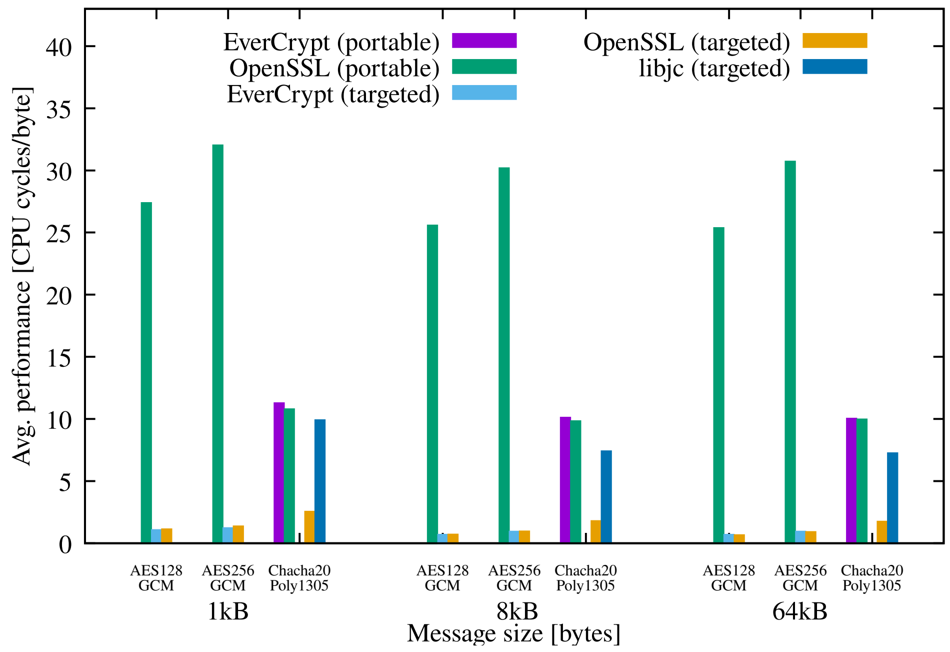


Figure 5.16: Average cycles/byte to encrypt blocks of random data using AEAD. Lower is better.

In Figure 5.15, we report on the performance of our targeted hash implementations when available (i.e., for SHA2-224 and SHA2-256), and our portable implementation otherwise, comparing with OpenSSL's corresponding implementations. We collect the measurements on a 1.5GHz Intel Celeron J3455 (which supports SHA-EXT (Gulley et al. 2013)) with 4 GB of RAM.

The results demonstrate the value of optimizing for particular platforms, as hardware support increases our performance for SHA2-224 and SHA2-256 by 7×, matching that of OpenSSL's best implementation. EverCrypt's portable performance generally tracks OpenSSL's, indicating a respectable fallback position for algorithms and platforms we have not yet targeted.

Similarly, Figure 5.16 and 5.17 report on the performance of our AEAD algorithms, with the latter omitting several implementations to make the comparison with OpenSSL's targeted version more apparent. We also compare against libjc (Almeida et al. 2020), a recently released library which contains verified, targeted implementations of Poly1305 and ChaCha20. It does not include a verified ChaCha-Poly implementation (libjc 2019), but we combined the two primitives in an unverified implementation for evaluation purposes. We collect measurements on a 3.6GHz Intel Core i9-9900K with 64 GB of RAM.

For cross-platform performance, we see that EverCrypt with ChaCha-Poly matches OpenSSL's corresponding implementation, and surpasses OpenSSL's portable AES-GCM implementation. Targeting, however, boosts both EverCrypt and OpenSSL's AES-GCM implementations beyond that of even the targeted version of ChaCha-Poly. Note that EverCrypt's targeted performance meets or exceeds that of OpenSSL, and achieves speeds of less than one cycle/byte for larger messages.

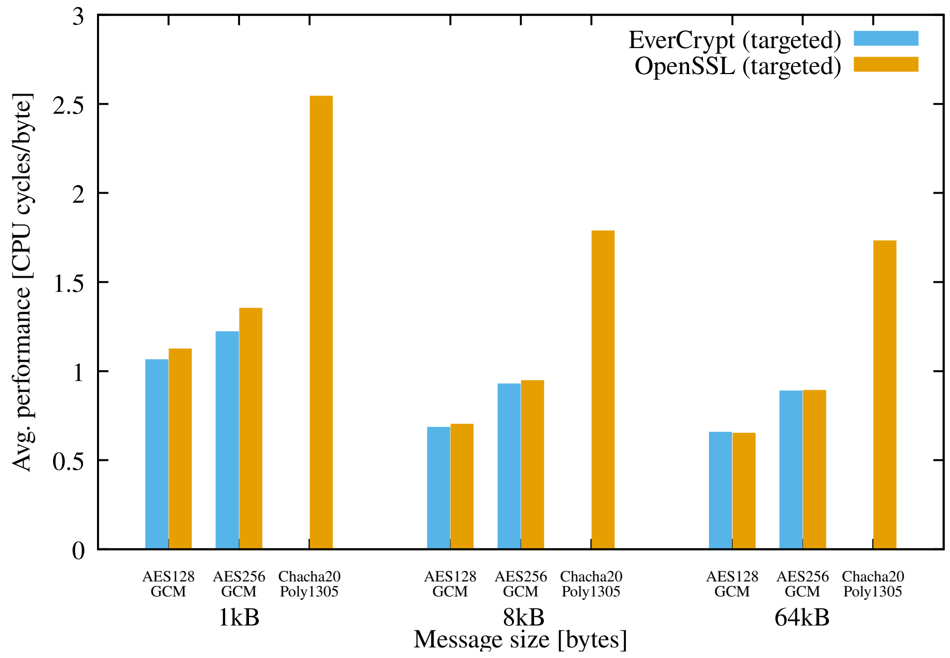


Figure 5.17: Average cycles/byte to encrypt blocks of random data with targeted AEAD. Lower is better.

Meanwhile, the performance of libjc's targeted ChaCha-Poly slightly beats that of OpenSSL and EverCrypt's portable implementations, but it is about 4× slower than OpenSSL's targeted ChaCha-Poly, and about 9-11× slower than EverCrypt's targeted AES128-GCM. We attribute this to the fact that the latter two each jointly optimize encryption and authentication together, whereas libjc optimizes the two primitives separately.

5.6 Impact and Summary

EverCrypt was a large verification project, resulting in an industrial-grade cryptographic library totaling several tens of thousands of verified C and assembly code, as summarized in Figure 5.18.

To achieve such a scale, EverCrypt required a large collaboration; designing, specifying, implementing and verifying EverCrypt took three person-years, plus approximatively one person-year spent on infrastructure, testing, benchmarking, and contributing bug fixes and other improvements to F*.

As a verified cryptographic provider, EverCrypt provides a foundation on which to build provably secure applications. For instance, EverCrypt has been used in verified implementations of modern cryptographic protocols such as Signal (Protzenko et al. 2019) and QUIC (Delignat-Lavaud et al. 2021), as well as in a verified library of Merkle trees (Protzenko et al. 2020). But more importantly, since its official release, parts of EverCrypt have been deployed in several high-impact projects: the combination of state-of-the-art performance and provable security guarantees led to the adoption

Components	**Lines of code**
Cryptographic algorithm specifications	3782
Vale interoperation specification	1595
Vale hardware specification	3269
Low* algorithms	25097
Low* support libraries	9943
Vale algorithms (written in Vale)	24574
Vale interoperation wrappers	13836
Vale proof libraries	23819
EverCrypt	5472
EverCrypt tests	4131
Vale algorithms (F* code generated from Vale files)	72039
Total (hand-written F* and Vale)	124310
Compiled code (.c files)	25052
Compiled code (.h files)	4082
Compiled code (ASM files)	14740
Total (C + assembly code)	43874

Figure 5.18: EverCrypt line counts, including whitespace and comments

of our Curve25519 implementation in the Zinc cryptographic library (Xu 2019), now part of the WireGuard VPN and the Linux kernel; Mozilla Firefox integrated verified implementations of elliptic curve cryptography and of ChaCha-Poly authenticated encryption (Jacobs and Beurdouche 2020); the Tezos blockchain also uses EverCrypt's elliptic curve cryptography, as well as its hashing functions and HMAC message authentication (Dumitrescu 2020); finally, the MirageOS unikernel relies on our implementation of Curve25519 to provide TLS 1.3 support (Mehnert 2020).

EverCrypt's industrial recognition demonstrates the tangible impact that formal verification can have on increasing the reliability and security of widely-used software. By adopting a proof-oriented methodology, we confidently and soundly implemented state-of-the-art optimizations which led to best-in-class performance for an implementation of Curve25519. Furthermore, structuring our code to ease verification led to generic implementations which significantly increased code and proof reuse, in turn easing code maintenance and simplifying extending the cryptographic provider with new implementations. The end result is a verified artifact which provides strong, formal guarantees about its correctness and security, while reaching sufficient industrial expectations to be adopted at scale in real-world, security-critical applications.

Part III

Steel: Proof-Oriented Programming in a Dependently Typed Concurrent Separation Logic

Chapter 6

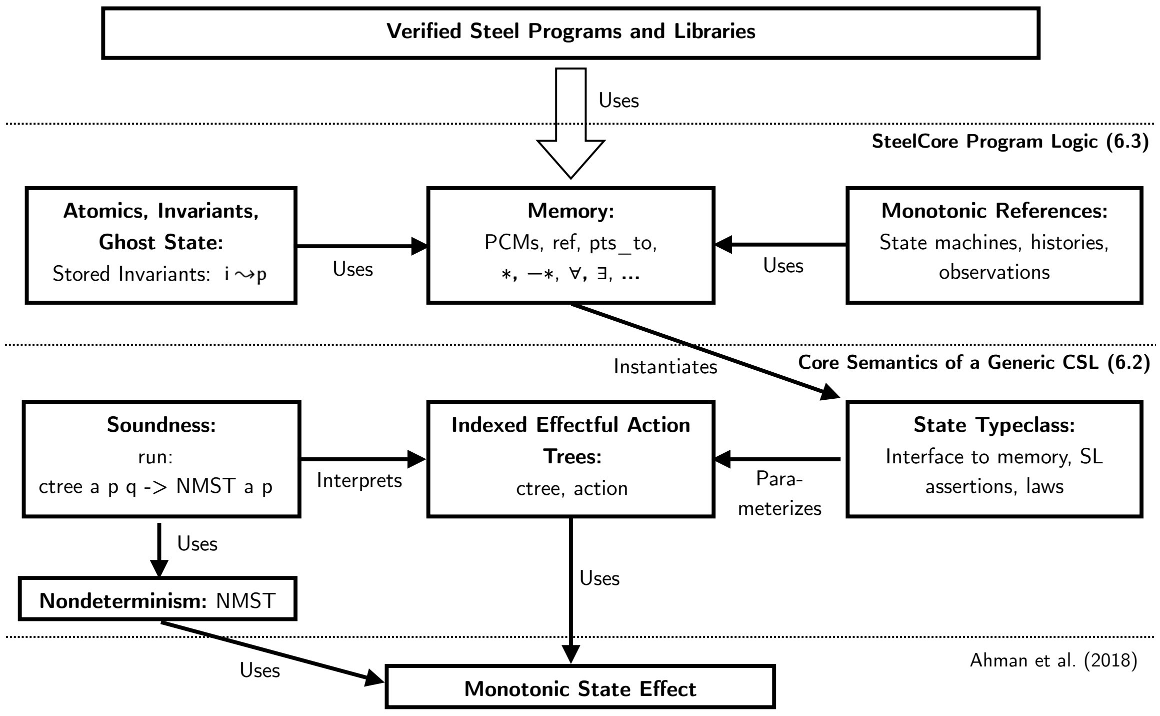
An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs

Proof assistants can be a programmer's delight, allowing one to build modular abstractions coupled with strong specifications that ensure correctness of complex programs. Their expressive power also allows one to develop new program logics to reason about a variety of programming idioms within the same framework as the programs themselves. A notable case in point is the Iris framework (Jung et al. 2018b) embedded in Coq (Coq Development Team 2017), which provides an impredicative, higher-order, concurrent separation logic (CSL) (Reynolds 2002; OHearn 2007) within which to specify and prove programs.

Iris has been used to model various languages and constructs, and to verify many interesting programs (Krogh-Jespersen et al. 2020; Chajed et al. 2019; Hinrichsen et al. 2019). However, Iris is not in itself a programming language: it must instead be instantiated with a *deeply embedded* representation and semantics of one provided by the user. For instance, several Iris-based papers work with a mini ML-like language deeply embedded in Coq (Krebbers et al. 2017b).

Taking a different approach, FCSL (Nanevski et al. 2008; Nanevski et al. 2014; Nanevski et al. 2019) embeds a predicative CSL in Coq, enabling proofs of Coq programs (rather than embedded-language programs) within a semantics that accounts for effects like state and concurrency. This allows programmers to use the full power of type theory not just for proving, but also for programming, e.g., building dependently typed programs and metaprograms over inductive datatypes, with typeclasses, a module system, and other features of a full-fledged language. However, Nanevski et al.'s program logics are inherently predicative, which makes it difficult to express constructs like dynamically allocated invariants and locks, which are natural in impredicative logics like Iris.

Aiming to provide the benefits of Nanevski et al.'s shallow embeddings, while also supporting dynamically allocated invariants and locks in the flavor of Iris, we propose a new framework called Steel. This chapter focuses on SteelCore, the core



Verified Steel Programs and Libraries
Uses
SteelCore Program Logic (6.3)
Atomics, Invariants, Ghost State:
Stored Invariants: i ⇝ p
Uses
Memory:
PCMs, ref, pts_to, ∗, −∗, ∀, ∃, ...
Uses
Monotonic References:
State machines, histories, observations
Instantiates
Core Semantics of a Generic CSL (6.2)
Soundness:
run:
ctree a p q -> NMST a p
Interprets
Indexed Effectful Action Trees:
ctree, action
Para-meterizes
State Typeclass:
Interface to memory, SL assertions, laws
Uses
Nondeterminism: NMST
Uses
Ahman et al. (2018)
Uses
Monotonic State Effect

Figure 6.1: An overview of SteelCore

semantics of Steel. We develop SteelCore in the *effectful* type theory provided by the F⋆ proof assistant (Swamy et al. 2016). Building on prior work that models the effect of monotonic state in F⋆ (Ahman et al. 2018), we develop a semantics for concurrent F⋆ programs while simultaneously deriving a CSL to reason about F⋆ programs using the effect of concurrency. The use of monotonic state enables us to account for invariants and atomic actions entirely within SteelCore. The net result is that we can program higher-order, dependently typed, generically recursive, shared-memory and message-passing concurrent F⋆ programs, and prove their partial correctness using SteelCore.

We present in Figure 6.1 the structure of SteelCore. Building on the monotonic state effect, we prove sound a generic program logic for concurrency, parametric in a memory model and a separation logic (Section 6.2). We then instantiate this semantics with a separation logic based on partial commutative monoids, store invariants, and state machines (Section 6.3). We describe several novel elements of our contributions, next.

For starters, we need to extend F⋆ with concurrency. To do this, we follow the well-known approach of encoding computational effects as definitional interpreters over free monads (Hancock and Setzer 2000; Kiselyov and Ishii 2015; Swierstra 2008; Xia et al. 2019). That is, we can represent computations as a datatype of (infinitely branching) trees of atomic actions. When providing a computational interpretation for action trees, one can pick an execution strategy (e.g., an interleaving semantics) and build an interpreter to run programs. The first main novelty of our work is that we provide an intrinsically typed definitional interpreter (Bach Poulsen et al. 2018) that both provides a semantics for concurrency while also deriving a CSL in which to reason about concurrent programs. Enabling this development is a new notion of indexed action trees, which we describe next.

Indexed Action Trees for Structured Parallelism. We represent concurrent computations as an instance of the datatype ctree st a pre post, shown below. The ctree type is a tree of atomic computational actions, composed sequentially or in parallel.

```
type ctree (st:state) : a:Type → pre:st.slprop → post:(a → st.slprop) → Type =
  | Ret : x:a → ctree st a (post x) post
  | Act : action a pre post → ctree st a pre post
  | Par : ctree st a p q → ctree st a' p' q' →
             ctree st (a & a') (p `st.star` p') (λ (x, x') → q x `st.star` q' x')
  | Bind : ctree st a p q → ((x:a) → Dv (ctree st b (q x) r)) → ctree st b p r
```

The type ctree st a pre post is parameterized by an instance st of the state typeclass, which provides a generic interface to memories, including st.slprop, the type of separation logic assertions, and st.star, the separating conjunction. The index a is the result type of the computation, while pre and post are separation logic assertions. The Act nodes hold stateful atomic actions; Par nodes combine trees in parallel; while Bind nodes sequentially compose a computation with a potentially divergent continuation, represented using F*'s primitive Dv effect presented in Chapter 3.

Interpreting Action Trees in the Effects of Nondeterminism and Monotonic State. We interpret a term (e : ctree st a pre post) as both a computation e as well as a proof of its own partial correctness Hoare triple {pre} e : a {post}. To prove this sound, we define an interpreter that non-deterministically interleaves actions run in parallel. The interpreter is itself an effectful F* function with the following (simplified) type, capturing our main soundness theorem:

```
val run (e:ctree st a p q)
  : NMST a st.evolves (λ m → st.interp p m) (λ _ x m' → st.interp (q x) m')
```

where NMST is the effect of monotonic stateful computations extended with nondeterminism. Here, we use it to represent abstract, stateful computations whose states are constrained to evolve according to the preorder st.evolves, and which when run in an initial state m satisfying the interpretation of the precondition p, produce a result x and final state m' satisfying the postcondition q x. As such, using the Hoare types of NMST, the type of run validates the Hoare rules of CSL given by the indexing structure on ctree. In doing so, we avoid the indirection of traces in Brookes (2007) original proof of CSL as well as in the work of Nanevski et al. (2014).

Atomics and Invariants: Breaking Circularities with Monotonic State. Although most widely used concurrent programming frameworks, e.g., the POSIX pthread API, support dynamically allocated locks, few existing CSL frameworks actually support them, with some notable exceptions (Buisse et al. 2011; Dodds et al. 2016; Gotsman et al. 2007; Jung et al. 2018b; Hobor et al. 2008). The main challenge is to avoid circularities that arise from storing locks that are associated with assertions about the memory in the memory itself. Iris, with its step-indexed model of impredicativity, can express this. However, other existing state of the art logics, including FCSL, cannot. In Section 6.3, we show how to leverage the underlying model of monotonic state to allocate a stored invariant, and to open and close it safely within an atomic command, without introducing step indexing.

PCMs, Ghost State, State Machines, and Implicit Dynamic Frames We base our memory model on partial commutative monoids (PCMs), allowing the user to associate a PCM of their choosing with each allocation unit. Relying on F*'s existing support for computationally irrelevant erased types, we can easily model *ghost state* by allocating values of erased types in the heap, and manipulating these values only using atomic ghost actions—all of which are erased during compilation. PCMs in SteelCore are orthogonal from ghost state: they can be used both to separate and manage access permissions to both concrete and ghost state—in practice, we use fractional permissions to control read and write access to references.

Further, SteelCore includes a notion of *monotonic* references, which when coupled with F*'s existing support for ghost values and invariants, allow programmers to code up various forms of *state machines* to control the use and evolution of shared resources. Demonstrating the flexibility of our semantics, we extend it to allow augmenting CSL assertions with frameable heap predicates, a style that combines CSL with *implicit dynamic frames* (Smans et al. 2012) within the same mechanized framework.

Contributions. The work described in this chapter was first presented in Swamy et al. (2020). I designed and implemented several earlier attempts at the SteelCore semantics and memory model. Building upon this, the final version presented here was primarily designed by Nikhil Swamy and Aseem Rastogi; in this version, I implemented atomic commands, invariant masks, and the invariant opening rule.

6.1 Basic Indexed Action Trees

As a warm-up towards the main ideas behind our indexed action trees, we start by presenting a very simple total semantics for concurrency. Relying only on the pure rather than effectful features of F*, some of the ideas in this section should also transfer to pure type theories like Agda or Coq. However, our main construction involves a partial-correctness semantics with effects like divergence, which may be harder to develop in non-effectful type theories.

A disclaimer: total correctness for realistic concurrent programs (e.g., under various scheduling policies) is a thorny issue that our work does not address at all. For this introductory example, we focus only on programs with structured parallelism, without any other synchronization constructs, and where loop bounds do not depend on effectful computations.

Action Trees for Concurrency. To model concurrency, we define an inductive type ctree_total, for trees of atomic actions, defined as action_tot a = state → Tot (a & state), indexed by a natural number (used for a termination proof). This is our first and simplest example of an indexed action tree, one that could easily be represented in another type theory. We will enrich ctree_total in Section 6.2.

```
type ctree_total : nat → Type → Type =
| Ret : x:a → ctree_total 0 a
| Act : act:action_tot a → ctree_total 1 a
| Par : ctree_total nL aL → ctree_total nR aR → ctree_total (nL+nR+1) (aL & aR)
| Bind : f:ctree_total n1 a → g:(x:a → ctree_total n2 b) → ctree_total (n1+n2+1) b

type nctree_total (a:Type) = n:nat & ctree_total n a
```

The type ctree_total induces a monad-like structure (under a suitable equivalence that quotients the use of Bind) by representing computations as trees of finite depth, with pure values (Ret) and atomic actions (Act) at the leaves; a Bind node for sequential composition of two subtrees; and a Par node for combining a left and a right subtree. The monad induced by ctree_total differs from the usual construction of a free monad for a collection of actions by including an explicit Bind node, instead of defining the monadic bind recursively. This makes ctree_total more similar to scoped operations proposed by Piróg et al. (2018), with f being in the "scope" of Bind. The nat index counts the number of Act, Par, and Bind nodes. We also define an abbreviation nctree_total a to package a tree with its index as a dependent pair.

A Definitional Interpreter for ctree_total. To give a semantics to ctree_total, we interpret its action trees in an interleaving semantics for state-passing computations, relying on a boolean tape to resolve the nondeterminism inherent in the Par nodes. To that end, we define a state and nondeterminism monad, with sample, get, and put actions:

```
type tape = nat → bool

type nst (a:Type) = tape & nat & state → a & nat & state

let return (a:Type) (x:a) : nst a = λ(_, n, s) → x, n, s

let bind (a b:Type) (f:nst a) (g:a → nst b) : nst b =
  λ(t, n, s) → let x, n1, s1 = f (t, n, s) in (g x) (t, n1, s1)

let sample () : nst bool = λ(t, n, s) → t n, n+1, s

let get () : nst state = λ(_, n, s) → s, n, s

let put (s:state) : nst unit = λ(_, n, _) → (), n, s
```

We can now interpret ctree_total trees as nst computations. It should be possible to define such an interpreter in many type theories, in a variety of styles. Here, we show one way to program it in F*, making use of its effect system to package the nst monad as a *user-defined effect.*

A user-defined effect in F* introduces a new abstract computation type backed by an existing F* definition (in our case, a computation type NST backed by the monad nst). F* automatically elaborates sequencing and application of computations using the underlying monadic combinators, without the need for do-notation, e.g., let in NST is interpreted as bind in nst. Further, F* supports sub-effects to lift between computation types, relying on a user-provided monad morphism, e.g., pure computations are silently lifted to any other effect. The following code turns the nst monad into the NST effect, with three actions, sample, get, and put.

```
total new_effect { NST : a:Type → Effect with repr=nst; return=return; bind=bind}

let sample () = NST?.reflect (sample())

let get () = NST?.reflect (get())

let put s = NST?.reflect (put s)
```

The type of sample is unit → NST bool, indicating that it has the NST effect—calling sample in a pure context is rejected by F*'s effect system. The total qualifier on the first line ensures that all the computations in the NST effect are proved terminating.

Using NST, we build an interpreter for ctree_total trees by defining run as the transitive closure of a single step. The main point of interest is the last case of step, reducing a Par a l r node by sampling a boolean and recursing to evaluate a step on either the left or the right.

```
let reduct (r:nctree_total a) = r':nctree_total a{ Ret? r' ∨ r'._1 < r._1 }

let rec step (redex:nctree_total a) : NST (reduct redex) (decreases redex._1)
  match redex._2 with
  | Ret _ → redex
  | Act act → let s0 = get () in let x, s1 = act s0 in put s1; (| _, Ret x |)
  | Bind (Ret x) g → (| _, g x |)
  | Bind f g → let (| _, f' |) = step (| _, f |) in (| _, Bind f' g |)
  | Par (Ret x) (Ret y) → (| _, Ret (x, y) |)
  | Par l (Ret y) → let (| _, l' |) = step (| _, l |) in (| _, Par l' (Ret y) |)
  | Par (Ret x) r → let (| _, r' |) = step (| _, r |) in (| _, Par (Ret x) r' |)
  | Par l r →
      if sample () then let (| _, l' |) = step (| _, l |) in (| _, Par l' r |)
      else let (| _, r' |) = step (| _, r |) in (| _, Par l r' |)

let rec run (p:nctree_total a) : NST (nctree_total a) (decreases p._1) =
  if Ret? p then p else run (step p)
```

The other interesting element is proving that these definitions are well-founded. For that, we enrich the type of step redex to return a *refinement type* reduct redex which states that the result is either a Ret node, or its index is strictly less that the index of the redex.

This, together with the decreases annotations, is sufficient for F* to automatically prove that step and run are terminating. Similar proofs could be done in other proof assistants, though the specifics would differ, e.g., in Agda, one might use sized types (Abel 2006).

6.2 A Partial Correctness Separation Logic

As stated at the start of this chapter, our goal is to define indexed action trees with the following type:

```
type ctree (st:state) (a:Type) (pre:st.slprop) (post:a → st.slprop) : Type
```

The type is indexed by st:state, a typeclass encapsulating (at least) the type of the memory st.mem and the type of separation logic assertions on the memory st.slprop. Intuitively, a ctree st a fp0 fp1 is the type of a potentially divergent, concurrent program manipulating a shared state of type st.mem. The program expects the fp0 footprint of some initial memory m0:st.mem. When run in m0, it may diverge or produce a result:a and m1:st.mem, providing the (fp1 result) fragment of m1 to the context.

The state typeclass for the semantics is shown below. First, we define a pre_state containing all the operations we need. A state is a refinement of pre_state satisfying various laws.

```
type pre_state = {
  mem: Type; (* The type of the underlying memory *)
  slprop: Type; (* The type of separation logic assertions *)
  equals: equiv slprop; (* An equivalence relation on slprops *)
  emp: slprop; (* With a unit *)
  star: slprop → slprop → slprop; (* And separating conjunction *)
  interp: slprop → mem → prop; (* Interpreting slprop as a mem predicate *)
  evolves: preorder mem; (* A preorder for MST: constrains how the state evolves *)
  inv: mem → slprop; (* A separation logic invariant on the memory *)
}

let st_laws (st:pre_state) =
  is_unit st.emp st.equals st.star ∧
  associative st.equals st.star ∧ commutative st.equals st.star ∧
  interp_extensionality st.equals st.interp ∧ star_extensionality st.equals st.star ∧ affine st

type state = s:pre_state{st_laws s}
```

We expect emp and star to form a commutative monoid over slprop and the equivalence relation equals. The relation interp interprets an slprop as a predicate on mem, and we expect the interpretation of star to be compatible with slprop-equivalence. We also expect the interpretation to be affine, in the sense that interp (p ∗ q) m ⟹ interp q m.

As we will see in Section 6.3, we can instantiate our semantics with a separation logic containing the full gamut of connectives, including conjunction, disjunction, separating implication, and universal and existential quantification. The preorder evolves and the invariant inv are opaque as far as the semantics is concerned—we will instantiate them in a way that allows us to model a source of freshness for allocating reference cells, and also to support dynamically allocated invariants. In the following, we write $*$ for st.star where st is clear from the context.

6.2.1 Frame-Preserving Actions

To define the type of action trees ctree, we first need to define the type of atomic actions at the leaves of the tree. These actions rely on F*'s support for monotonic state, captured by an effect, MST.

The MST effect encapsulates stateful computations that restrict the state to evolve according to a given preorder, i.e., a reflexive, transitive relation. Ahman et al. (2018) observe that for such computations, witnessing a property p of the state that is invariant under the preorder is sufficient to recall that p is true in the future. Ahman et al. propose the following signature for such an MST effect, and prove the partial correctness of the Hoare logic encoded in the indices of MST against an operational semantics for a dependently typed λ-calculus with primitive state.

```
effect MST (a:Type) (state:Type) (p:preorder state)
  (req:state → prop) (ens:state → a → state → prop)
```

When executing a computation (c : MST a state p req ens) in an initial state s0:state satisfying req s0, the computation either diverges, or returns a value x:a in a final state s1:state satisfying ens s0 x s1. Further, the state is transformed according to the preorder p, i.e., the initial and final states are related by p s0 s1. The MST effect provides the following actions—for readability, we tag the pre- and postcondition with requires and ensures respectively:

- *Get* the current state:

```
val get () : MST state state p (requires λs → ⊤) (ensures λs0 r s1 → s0==s1 ∧ r==s0)
```

- *Put* the state, but only when the new state s1 is related to the old one s by p:

```
val put (s1:state) : MST unit state p (requires λs → p s s1) (ensures λ_ _ s → s==s1)
```

- *Witness* stable predicates: A stable predicate is maintained across preorder-respecting state evolutions. The witness action proves an abstract proposition, witnessed q, attesting that the stable predicate q is valid.

```
let stable_sprop (p:preorder state) = q:(state → prop){∀ s0 s1. q s0 ∧ p s0 s1 ⟹ q s1}

val witnessed (q:stable_sprop p) : prop

val witness (q:stable_sprop p)
  : MST unit state p (requires λs0 → q s0) (ensures λs0 _ s1 → witnessed q ∧ s0==s1)
```

- *Recall* stable predicates: Having witnessed q, one can use recall q to re-establish it at any point.

```
val recall (q:stable_sprop p{witnessed q})
  : MST unit state p (requires λs0 → ⊤) (ensures λs0 _ s1 → s0==s1 ∧ q s1)
```

```
type ctree (st:state) : a:Type → fp0:st.slprop → fp1:(a → st.slprop) → Type =
| Act: e:action a fp0 fp1 → ctree st a fp0 fp1
| Ret: fp:(a → st.slprop) → x:a → ctree st a (fp x) fp
| Bind: f:ctree st a fp0 fp1 → g:(x:a → Dv (ctree st b (fp1 x) fp2)) → ctree st b fp0 fp2
| Par: cL:ctree st aL fp0L fp1L → cR:ctree st aR fp0R fp1R →
     ctree st (aL & aR) (fp0L ∗ fp0R) (λ (xL, xR) → fp1L xL ∗ fp1R xR)
| Frame: c:ctree st a fp0 fp1 → f:st.slprop → ctree st a (fp0 ∗ f) (λ x → fp1 x ∗ f)
| Sub: c:ctree st a fp0 fp1 { sub_ok fp0 fp1 fp0' fp1' } → ctree st a fp0' fp1'
```

with

```
let sub_ok fp0 fp1 fp0' fp1' = fp0' `stronger_than` fp0 ∧ fp1' `weaker_than` fp1

let stronger_than fp0' fp0 = ∀m f. st.interp (fp0' ∗ f) m ⟹ st.interp (fp0 ∗ f) m

let weaker_than fp1' fp1 = ∀x m f. st.interp (fp1 x ∗ f) m ⟹ st.interp (fp1' x ∗ f) m
```

Figure 6.2: SteelCore's representation of computations as indexed action trees

As such, the MST effect provides a small program logic for monotonic state computations, which we leverage to define atomic SteelCore actions. An action is an MST computation that requires its initial footprint fp0 to hold on the initial state m0. It returns an x:a and ensures its final footprint fp1 x on the final state m1. In both the pre- and postcondition, we expect st.inv to hold separately. Finally, and perhaps most importantly, the preserves_frame side conditions ensures that actions are frameable. We elaborate on that next.

```
let action a (fp0:st.slprop) (fp1:a → st.slprop) =
  unit → MST a st.mem st.evolves
  (requires λm0 → st.interp (st.inv m0 ∗ fp0) m0)
  (ensures λm0 x m1 → st.interp (st.inv m1 ∗ fp1 x) m1 ∧ preserves_frame fp0 (fp1 x) m0 m1)
```

Frame Preservation. We would like to derive a framing principle for computations as a classic frame rule (and its generalization, the rule for separating parallel composition). As observed by Dinsdale-Young et al. (2013), it is sufficient for the leaf actions to be frame-preserving for computations to be frame-preserving too. To that end, the definition of preserves_frame (that an action must provide in its postcondition) states that all frames separate from st.inv m0 ∗ pre and valid in the initial state m0 remain separate from st.inv m1 ∗ post, and are valid in m1.

```
let preserves_frame (pre post:st.slprop) (m0 m1:st.mem) = ∀(frame:st.slprop).
  st.interp (st.inv m0 ∗ pre ∗ frame) m0 ⟹ st.interp (st.inv m1 ∗ post ∗ frame) m1
```

6.2.2 CSL-Indexed Action Trees with Monotonic State

Figure 6.2 shows the way we represent computation trees in SteelCore. These trees differ from the simple action trees we used in Section 6.1. The additional indexing structure in each case of ctree posits the proof rules of a program logic for reasoning about ctree computations. In Section 6.2.3, we show that this logic is sound by denoting ctree st a fp0 fp1 trees via an interleaving, definitional interpreter into NMST computations. NMST is a user-defined extension of the MST effect, which provides an additional sample action similar to the one presented in Section 6.1 to model nondeterminism, which we need for interleaving the subtrees of Par nodes. As NMST computations are potentially divergent, we do not need to prove the termination of the definitional interpreter, removing the need for the natural number index of ctree_total.

We now describe the structure of ctree in detail, discussing each of its constructors in turn.

Atomic Actions. At the leaves of the tree, we have nodes of the form Act e, for some action e: the index of the computation inherits the indices of the action.

Returning Pure Values. Also at the leaves of the tree are Ret fp x nodes, which allow returning a pure value x in a computation. The Ret node is parametric in a footprint fp, and the indices on ctree state that in order to provide fp, we expect fp x to hold in the initial state m0.

Alternatively, we could have defined Ret : x:a → ctree st a st.emp (λ_ → st.emp), although, as we discuss in Section 6.2.3, this form is less convenient in conjunction with the frame rule.

Sequential Composition. The Bind f g node sequentially composes f and g. Its indexing structure should appear fairly canonical. The footprints of f and g are "chained", as in parameterized monads proposed by Atkey (2009), except our indices (notably fp2) are dependent. The computation type of g has the Dv effect, indicating a potentially divergent continuation.

Parallel Composition. Par cL cR composes computations in parallel. The indexing structure yields the classic CSL rule for parallel composition of computations with disjoint footprints.

Structural Rules: Framing and Subsumption. The Frame c f node preserves the frame f across the computation c. The Sub c node allows strengthening the initial footprint and weakening the final footprint of c. These nodes directly correspond to the canonical CSL frame and consequence rules.

These structural rules are essential elements of our representation. The indexing structure of ctree defines a program logic and the structural rules are manifested as a kind of re-indexing, which must be made explicit in the inductive type as additional constructors. Further, given such structural rules, the need for a separate Bind, as opposed to continuations in each node, becomes evident. Consider verifying a Hoare triple {P1 $*$ P} a1; a2; a3 {Q}, where a1, a2, a3 are actions with { P1 } a1; a2 { P1 }, and

{ P1 ∗ P } a3 { Q }. The canonical proof frames P across a1; a2 together, which is trivial in our representation, as Bind (Frame (Bind (Act a1) (λ_ → (Act a2))) P) (λ_ → (Act a3)). The frames can easily be added *outside* of a proof derivation, making the proofs modular. However, if the continuations were part of the Act (and Par) nodes, such a structural frame rule would not apply. We would have to bake-in framing in the Act nodes, and even then we would have to frame P across a1 and a2 individually. This makes the proofs less modular, since we cannot directly use the given derivation { P1 } a1; a2 { P1 }.

Although we include Frame and Sub, we lack the structural rule for disjunction. Accomodating disjunction in a shallow embedding is hard to do, since it requires giving to the same computation more than one type. One possibility may be to adopt a relational specification style, as Nanevski, Vafeiadis, and Berdine (2010) do—we leave an exploration of this possibility to future work. Meanwhile, as we instantiate the semantics with a state model in Section 6.3, we also provide several lemmas to destruct combinations of separating conjunctions and existentials (with disjunctions as a special case).

6.2.3 Soundness

To prove the soundness of the proof rules induced by the indexing structure of ctree, we follow the strategy outlined in Section 6.1, with NMST as the target denotation. Our goal is to define an interpreter with the following type, showing that it maintains the memory invariant while transforming fp0 into fp1 x.

```
val run (f:ctree st a fp0 fp1) : NMST a st.mem st.evolves
  (requires λm0 → st.interp (st.inv m0 ∗ fp0) m0)
  (ensures λm0 x m1 → st.interp (st.inv m1 ∗ fp1 x) m1)
```

As before, we proceed by first defining a single-step interpreter, and then closing it transitively to build a general recursive, multi-step interpreter. The single-step interpreter has the following type, returning (as in Section 6.1) the reduced computation tree packaged with all its indices.

```
type reduct a = | Reduct: fp0:_ → fp1:_ → ctree st a fp0 fp1 → reduct a

val step (f:ctree st a fp0 fp1) : NMST (reduct a) st.mem st.evolves
  (requires λm0 → st.interp (st.inv m0 ∗ fp0) m0)
  (ensures λm0 (Reduct fp0' fp1' _) m1 →
    st.interp (st.inv m1 ∗ fp0') m1 ∧
    preserves_frame fp0 fp0' m0 m1 ∧
    fp1' `stronger_than` fp1)
```

In addition to requiring and ensuring the invariant and footprint assertions, we have additional inductive invariants that are needed to take multiple steps. As is typical in such proofs, one needs to show that, given a term in a context E[c], reducing c by a single step produces c' that can be correctly typed within the same context, i.e., E[c'] must be well-typed. Towards this end, we need two main properties of step: (a) preserves_frame, defined in Section 6.2.1, ensures that the reduct c' can be framed with any frame used with the redex c; and (b) that the postcondition fp1' of the reduct c' is stronger than the postcondition fp1 of the redex c. Interestingly, we do not explicitly need to show that the precondition of the reduct is weaker than the precondition of the redex: that the initial footprint of the reduct holds in m1 is enough.

We show all the main cases of the single-step reduction next. In all cases, the code is typechecked as shown, with proofs semi-automated by F⋆'s SMT solving backend.

Framing. The code below shows stepping through applications of the Frame c0 f rule. When c0 is a Ret node, we remove the Frame node and restore the derivation by extending the footprint of the Ret node to include the frame f—this is one reason why it is convenient to have Ret nodes with parametric footprints, rather than just emp.

```
let rec step (c:ctree st a fp0 fp1) = match c with
  | ...
  | Frame (Ret fp0' x) f → Reduct (fp0' x ∗ f) (λ x → fp0' x ∗ f) (Ret (λ x → fp0' x ∗ f) x)
  | Frame c0 f →
      let m0 = get () in
      let Reduct fp0' fp1' c' = step c0 in
      let m1 = get () in
      preserves_frame_star fp0 fp0' m0 m1 f;
      Reduct (fp0' ∗ f) (λ x → fp1' x ∗ f) (Frame c' f)
```

When c0 is not a Ret, we recursively evaluate a step within c0, and then reconstruct a Frame around its reduct c'. This proof step makes use of a key lemma, preserves_frame_star, which states that frame preservation still holds in a larger context, i.e., preserves_frame fp0 fp0' m0 m1 ⟹ preserves_frame (fp0 ∗ f) (fp0' ∗ f) m0 m1)

Subsumption. Reduction of the other structural rule, Sub, is simpler, we just remove the Sub node, as shown below; the refinement sub_ok on the c argument of the Sub node allows F⋆ to prove the inductive invariants of step. Although we remove Sub nodes, the rule for sequential composition (next) adds them back to ensure that the reduct remains typeable in the context. An alternative may have been to treat Sub like we treat Frame, however, this form is more convenient when adding support for implicit dynamic frames, as described in Section 6.2.4.

```
  | Sub c → Reduct fp0' fp1' c
```

Sequential Composition. In case f is fully reduced to a Ret node, we simply apply the continuation g. Otherwise, we take a step in f producing a reduct f' that may have a stronger final footprint. To reconstruct the Bind node, we need to strengthen the initial footprint of g with the final footprint of f', we do so by wrapping g with a Sub. Note that the postcondition fp1' `stronger_than` fp1 of the single-step interpreter ensures that the refinement sub_ok in the Sub node holds.

```
  | Bind (Ret fp0 x) g → Reduct (fp0 x) fp2 (g x)
  | Bind f g →
      let Reduct fp0' fp1' f' = step f in
      Reduct fp0' fp2 (Bind f' (Sub #fp1 #_ #fp1' #_ g))
```

Parallel Composition. The structure of reducing Par nodes is essentially the same as in Section 6.1. When both branches are Ret nodes, we simply create a reduct with a Ret node capturing the two values.

```
| Par (Ret fp0L xL) (Ret fp0R xR) →
  Reduct (fp0L xL ∗ fp0R xR) (λ (xL, xR) → fp0L xL ∗ fp0R xR)
    (Ret (λ (xL, xR) → fp0L xL ∗ fp0R xR) (xL, xR))
| Par cL cR →
  if sample() then
    let m0 = get () in
    let Reduct fp0L' fp1L' cL' = step cL in
    let m1 = get () in
    preserves_frame_star fp0L fp0L' m0 m1 fp0R;
    Reduct (fp0L' ∗ fp0R) (λ (xL, xR) → fp1L' xL ∗ fp1R xR) (Par cL' cR)
  else ... (∗ similarly for the right branch ∗)
```

When only one of the two branches is Ret, we descend into the other one (we elide these cases from the presentation). When both the branches are candidates for reduction, we sample a boolean, and pick either the left or the right branch to descend into. Having obtained a reduct, we reconstruct the Par node, by appropriately framing the initial footprint of the unreduced branch, as shown above.

Atomic Actions. An Act e node is reduced by applying it, and returning its result in a Ret node.

```
| Act e → let x = e () in Reduct (fp1 x) fp1 (Ret fp1 x)
```

Multi-Step Interpreter. Once we defined a single-step interpreter, implementing a general recursive, multi-step interpreter is straightforward: we recursively evaluate single steps until we reach a Ret node. The type of the interpreter, shown below, is the main statement of partial correctness for our program logic.

```
let rec run (f:ctree st a fp0 fp1) : NMST a st.mem st.evolves
  (requires λm0 → st.interp (st.inv m0 ∗ fp0) m0)
  (ensures λm0 x m1 → st.interp (st.inv m1 ∗ fp1 x) m1)
= match f with
  | Ret _ x → x
  | _ → let Reduct _ _ f' = step f in run f'
```

The type states that when run in an initial state m0 satisfying the memory invariant st.inv m0 and separately the footprint assertion fp0, the code either diverges or returns x:a in a final state m1 with the invariant st.inv m1, and the footprint assertion fp1 x. The inductive stronger_than invariant about the step function providing a stronger postcondition is crucial to the proof here: the recursive call to run ensures the validity of the post-footprint of f' in the final memory, we need the inductive invariant to relate it to the post-footprint of f, as required by the postcondition of run.

6.2.4 Implicit Dynamic Frames

While we have presented the action trees, and hence the CSL semantics, using only the slprop indices, our actual implementation also contains two further indices for specifications in the style of implicit dynamic frames (Smans et al. 2012). In our representation type, ctree_idf st a fp0 fp1 req ens, the last two indices req and ens indicate the pre- and postcondition of a computation, where the precondition is a *fp0-dependent* predicate on the initial memory, and the postcondition is a two-state predicate that is fp0-dependent on the initial memory and fp1-dependent on the final one. The dependency relation, formally defined below, captures the requirement that the predicates are "self-framing" (Parkinson and Summers 2012), i.e., the slprop footprint indices fp0 and fp1 limit the parts of the memory that these predicates can depend on.

```
let fpmem (fp0:st.slprop) = m:st.mem{st.interp fp0 m}

let fp_prop (fp0:st.slprop) = (q:st.mem → prop){
  ∀(m0:fpmem fp0) (m1:st.mem{st.disjoint m0 m1}). q m0 ⟺ q (st.join m0 m1)}

let fp_prop_2 (fp0:st.slprop) (fp1:a → st.slprop) = (q:st.mem → a → st.mem → prop){
  (* We can join any disjoint memory to the pre−memory and q is still valid *)
  (∀ (x:a) (m_pre:fpmem fp0) (m_post:st.mem) (m1:st.mem{st.disjoint m_pre m1}).
    q m_pre x m_post ⟺ q (join m_pre m1) x m_post) ∧
  (* We can join any disjoint memory to the post−memory and q is still valid *)
  (∀ (x:a) (m_pre:st.mem) (m_post:fpmem (fp1 x)) (m1:st.mem{st.disjoint m_post m1}).
    q m_pre x m_post ⟺ q m_pre x (join m_post m1))
}

type ctree_idf st a (fp0:st.slprop) (fp1:st.slprop) (req:fp_prop fp0) (ens:fp_prop_2 fp0 fp1) =
  ...
```

In addition to these indices, we also add frameable memory predicates to the Frame and Par rule. For example, the following is the Frame rule in our implementation:

```
  | Frame: c:ctree_idf st a fp0 fp1 req ens → f:st.slprop → p:fp_prop f →
           ctree_idf st a (fp0 ∗ f) (λ x → fp1 x ∗ f) (frame_req req p) (frame_ens ens p)
```

with

```
let frame_req req p = λm → req m ∧ p m

let frame_ens ens p = λm0 x m1 → ens m0 x m1 ∧ p m1
```

Finally, our soundness theorem, i.e., the specification of the run function, requires the precondition (req) of the computation, and ensures its postcondition (ens). To prove this theorem, we have to enhance the inductive invariant of the step function to also mention weakening of the preconditions and strengthening of the postconditions.

Incorporating both the slprop indices and implicit dynamic frame-style requires- and ensures-indices enables more flexibility in writing program specifications. We will make good use of this feature in Chapter 8, when implementing a variety of libraries in Steel.

Discussion. It is worth noting that, although we have built a definitional interpreter with an interleaving semantics for concurrent programs, we do not intend to run programs using run, since it would be very inefficient, primarily because the interleaving semantics is encoded via sampling, but also because the representation contains a full proof tree, including the structural rules Frame and Sub. Instead, relying on F*'s support for extraction to OCaml and C, we intend to compile effectful, concurrent programs to native concurrency in the target platforms, e.g., POSIX threads. As such, the main value of run is its proof of soundness: we now have in hand a semantics for concurrent programs, and a means to reason about them deductively using a concurrent separation logic. We have built our semantics atop the effect of monotonic state, parameterizing our semantics with a preorder governing how the state evolves. So far, this preorder has not played much of a role. For the payoff, we will have to wait until we instantiate the state interface, next.

6.3 The SteelCore Program Logic

The core semantics developed in the previous section provides a soundness proof for a generic, minimalistic concurrent separation logic. In this section, we instantiate the semantics with a model of state, assertions, invariants, and actions defining the logic for Steel programs.

The logic includes the following main features:

- A core heap model addressed by typed references with explicit, manually managed lifetimes.
- Each heap cell stores a value in a user-chosen, cell-specific partial commutative monoid, supporting various forms of sharing disciplines and stateful invariants, including, e.g., a discipline of fractional permissions (Boyland 2003), for sharing among multiple threads.
- A separation logic, with all the usual connectives.
- Ghost state and ghost actions, relying on F*'s existing support for erasure.
- A model of atomic actions, including safe composition of ghost and concrete actions.
- Invariants, that can be dynamically allocated and freely shared among multiple threads and accessed and restored by atomic actions only.
- Monotonic references controlling the evolution of memory, built using preorders from the underlying monotonic state effect.

The result is a full-featured separation logic shallowly embedded in F*, with a fully mechanized soundness proof, and applicable directly to dependently typed, higher order, effectful host language programs.

6.3.1 Memory

At the heart of our state model is a representation of memory, as outlined in the type below.

```
let addr = nat

let heap = addr → option cell

type mem = { heap:heap; ctr:nat; istore:istore }

let mem_inv m : slprop = (∀ i.i ≥ m.ctr ⟹ m.heap i == None) ∧ istore_inv m

let mem_evolves m₀ m₁ =
  h_evolves m₀.heap m₁.heap ∧ i_evolves m₀.istore m₁.istore ∧ m₀.ctr ≤ m₁.ctr
```

A heap is a map from abstract addresses (represented as natural numbers) to heap cells defined below.[1] A memory augments a heap with two important fields of metadata. First, we have a counter to provide fresh addresses for allocation, with an invariant guaranteeing that all addresses above ctr are unused. Second, we have an istore for tracking dynamically allocated invariants. Actions maintain a memory invariant mem_inv and the memory is constrained to evolve according to the preorder mem_evolves. We discuss these elements in detail throughout this section.

For the definition of heap cells, we make use of partial commutative monoids (PCMs). Using PCMs to represent state is typical in the literature: starting at least with the work of Jensen and Birkedal (2012), PCMs have been used to encode a rich variety of specifications, ranging from various kinds of sharing disciplines, fictional separation, and also various forms of state machines. We represent PCMs as the typeclass pcm a shown below, where we account for partiality by restricting the domain of op by a predicate composable. We write $\preceq_p$ for the partial order induced by p:pcm a.

```
type pcm (a:Type) = {
  one:a;
  composable: a → a → prop {sym composable};
  op: x:a → y:a{composable x y} → a { comm op ∧ assoc op ∧ is_unit op one }
}

let (⪯ₚ) (x y : a) = ∃frame. p.op x frame == y

type cell = | Cell: a:Type → p:pcm a → v:a → cell
```

A cell is a triple of a type a, an instance of the typeclass of partial commutative monoids (pcm a), and a value of that type.[2]

[1] In our F⋆ sources, we define heap as the type addr ^→ option cell, the type of functions for which the functional extensionality axiom is admissible in F⋆; we gloss over this technicality in our presentation here.

[2] On universes and higher order stores: We define our memory model universe-polymorphically, so that it can store values in higher universes, e.g., values at existential types. However, the cell type resides in a universe one greater than the type it contains. By extension, heap is in the same universe

With this representation of heap, it is relatively straightforward to define two functions, disjoint and join, which we use to separate and combine disjoint memories. For an address that appears in both heaps, we require the cells at that address to agree on the type, the PCM instance, and for the values to be composable in the PCM.

```
let disjoint_addr (h h':heap) (a:addr) = match h a, h' a with
  | Some (Cell t p v), Some (Cell t' p' v') → t==t' ∧ p==p' ∧ p.composable v v'
  | _ → ⊤

let disjoint h0 h1 = ∀a. disjoint_addr h0 h1 a

let join (h0:heap) (h1:heap{disjoint h0 h1}) = λa → match h0 a, h1 a with
  | None, None → None
  | None, Some x | Some x, None → Some x
  | Some (Cell t p v0), Some (Cell _ _ v1) → Some (Cell t p (p.op v0 v1))
```

6.3.2 Separation Logic Propositions

We define the type slprop of separation logic propositions as the type of heap propositions p that are preserved under disjoint extension. We emphasize that slprops are affine *heap* propositions, rather than mem propositions—the non-heap fields in a memory are meant for internal bookkeeping and (intentionally) cannot be described by slprops. We use interp to apply an slprop to the heap within a memory. Further, being heap predicates, slprops reside in the same universe as heap. As such, slprops cannot themselves be stored in the heap, although doing so is sometimes convenient for encoding various forms of higher-order ghost state (Jung et al. 2016)—this is the main limitation of our model. However, since Steel is embedded within F*, one can sometimes work around this restriction by adopting various dependently typed programming tricks, e.g., rather than storing slprops in the heap, one might instead store codes for a suitably small sub-language of slprops instead, and work with interpretations of these codes.

Supporting higher-order ghost state in its generality is notoriously tricky and can often lead to inconsistencies (Jung et al. 2016; Krebbers et al. 2017a): circularities often arise when one can store memory predicates in the memory itself. However, the SteelCore semantics is careful to not allow storing heap predicates in the heap itself—this would be circular and run afoul of F*'s universe discipline. To be specific, from Section 6.3.1, notice that the memory contains a heap and an istore, and the istore stores heap predicates, not the heap itself. That said, SteelCore's soundness is based on the model of monotonic state propsed by Ahman et al. (2018), and this model does not include a treatment of F*'s universes. As such, to close a gap in our model, we would at least need to extend the monotonic state model to account for universes.

```
let slprop = p:(heap → prop) { ∀(h0 h1: heap). p h0 ∧ disjoint h0 h1 ⟹ p (join h0 h1) }

let interp (p:slprop) (m:mem) = p m.heap
```

as its cells. As a result, although heaps and heap-manipulating total functions cannot be stored in cells, functions in F* that include the effect of divergence are always in universe 0 and can be stored in the heap, i.e., this model is adequate for partial correctness of programs with higher order stores.

We define several basic connectives for slprop, as shown below. The existential and universal quantifiers, slex and slall support quantification over terms in arbitrary universes, including quantification over slprops themselves.

```
let slstar p1 p2 h = ∃h1 h2. h1 `disjoint` h2 ∧ h == join h1 h2 ∧ p1 h1 ∧ p2 h2

let slwand p1 p2 h = ∀h1. h `disjoint` h1 ∧ p1 h1 ⟹ p2 (join h h1)

let slemp p h = ⊤

let sland p1 p2 h = p1 h ∧ p2 h

let slor p1 p2 h = p1 h ∨ p2 h

let slex p h = ∃x. p x h

let slall p h = ∀x. p x h
```

This interpretation also induces a natural equivalence relation on slprop, i.e., p ∼ q if and only if (∀m. interp p m ⟺ interp q m), corresponding to the extensional equivalence of heap predicates. It is easy to prove that slstar and slemp form a (total) commutative monoid with respect to ∼.

We also define the atomic points-to assertion on references as follows:

```
let ref (a:Type) (p:pcm a) = addr

let pts_to (r:ref a p) (v:a) (h:heap) = match h r with
  | Some (Ref a' p' v') → a == a' ∧ p == p' ∧ v ≼_p v'
  | _ → ⊥

val pts_to_compatible (r:ref a p) (v0 v1:a) (m:mem) : Lemma
    (interp (pts_to r v0 ∗ pts_to r v1) m ⟺
         (p.composable v0 v1 ∧ interp (pts_to x (p.op v0 v1)) m))
```

A reference is represented by its address in the heap, and pts_to r v asserts partial knowledge of the contents of the reference r, i.e., that r contains some value v' compatible with v according to the PCM p associated with r. The pts_to_compatible lemma relates the separating conjunction to composition in the underlying PCM. We will see in Section 6.3.5 how to choose specific PCMs to model fractional permissions and monotonic references.

We now have most of what we need to instantiate the state interface of our semantics—two key ingredients, the memory invariant and preorder, will be presented in detail in the next subsections. Foreshadowing their presentation, our state instantiation is:

```
let st : state = {
  mem = mem;          slprop = slprop;        equals = ∼;
  emp = slemp;        star = slstar;        interp = interp;
  inv = mem_inv;                     (∗ cf. Section 6.3.3 ∗)
  evolves = mem_evolves             (∗ cf. Sections 6.3.4, 6.3.5 ∗) }
```

Actions on PCM-Indexed References. Given this instantiation, one can define several basic actions, such as the following primitives on references. Building on these generic primitives, we implement libraries for several more common use cases, including references with fractional permissions and monotonic references.

To allocate a reference, one presents both a value and a PCM to use for that reference

```
val alloc (p:pcm a) (v:a) : action (ref a p) emp (λ r → pts_to r v)
```

Reading a reference with (!) returns a value compatible with the caller's partial knowledge.

```
val (!) (r:ref a p) (v:erased a) : action (x:a{v ≼p x}) (pts_to r v) (λ _ → pts_to r v)
```

Mutating a reference r requires the new value v to be compatible with all frames compatible with the caller's partial knowledge of r, that is, the update must be frame-preserving.

```
let frame_preserving (x y:a) = ∀f. p.composable f x ⟹ p.composable f y ∧ p.op f y == y

val (:=) (r:ref a p) (v0:erased a) (v:a{frame_preserving v0 v})
  : action unit (pts_to r v0) (λ _ → pts_to r v)
```

Finally, to de-allocate a reference the caller must possess exclusive non-trivial knowledge of it.

```
let exclusive (v:erased a) = ∀frame. p.composable frame v0 ⟹ frame==p.one

val free (r:ref a p) (v0:erased a{exclusive v0}): action unit (pts_to r v) (λ _ → emp)
```

In what follows, we overload the use of F*'s existing connectives ∃, ∀, ∧ , ∨ for use with slprop. We write emp for slemp; $*$ and $-\!\!*$ for slstar and slwand. Borrowing F*'s notation for refinement types, we also write h:p{f} for sland p (λh → f) and pure p for _:emp{p}.

6.3.3 Introducing Invariants: Preorders and the istore

Beyond the traditional separation logic assertions, it is useful to also support a notion of *invariant* that allows a non-duplicable slprop to be shared among multiple threads. For some basic intuition, it is instructive to look at the design of invariants in Iris—we reproduce, below, three (slightly simplified) rules presented by Jung et al. (2018).

$$(1)\ P \Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}} \qquad\qquad (2)\ \text{persistent}\left(\boxed{P}^{\mathcal{N}}\right)$$

$$(3)\ \frac{\{\ \triangleright P * Q\ \}\ e\ \{\ \triangleright P * R\ \}_{\mathcal{E}\setminus\mathcal{N}} \qquad \text{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E}}{\left\{\ \boxed{P}^{\mathcal{N}} * Q\ \right\}\ e\ \left\{\ \boxed{P}^{\mathcal{N}} * R\ \right\}_{\mathcal{E}}}$$

The first rule states that, at any point, one can turn a resource assertion P into an *invariant* $\boxed{P}^{\mathcal{N}}$. An invariant is associated with a name, $\mathcal{N}$—we shall see its significance in Section 6.3.4.

The second rule states that an invariant is persistent, which implies it is duplicable: i.e., $\boxed{P}^{\mathcal{N}} \implies \boxed{P}^{\mathcal{N}} * \boxed{P}^{\mathcal{N}}$. Thus, by turning a resource assertion P into an invariant, one can share the invariant among multiple threads, frame it across other computations, etc.

The final rule shows how an invariant can be used. This rule is quite technical, but intuitively, it states that an atomic command e can assume the resource assertion P associated with an invariant $\boxed{P}^{\mathcal{N}}$, so long as it also restores P after executing (atomically). Some of the technicality in the rule has to do with impredicativity and step indexing. In Iris, $\boxed{P}^{\mathcal{N}}$ is a proposition in the logic like any other, and Iris allows quantification over all such propositions, including invariants themselves. This is very powerful, but it also necessitates the use of step indexing, i.e., the "later" modality $\triangleright P$ in the premise of the rule. For SteelCore, we seek to model invariants of a similar flavor, but while remaining in our predicative setting—our use of the monotonic state effect will give us a way.

Invariants in SteelCore. To allocate an invariant, we provide an action with the signature below:

```
val new_invariant (p:slprop) : action (inv p) p (λ _ → emp)
```

Recall the action type from Section 6.2.1. The type above states that given possession of p, new_invariant consumes p, providing only emp, but importantly, returning a *value* of type inv p: our representation of an invariant—new_invariant models Iris' update modality to allocate an invariant, i.e., the first of the three rules above. Being a value, inv p is freely duplicable, like any other value in F⋆—mimicking Iris' rule of persistence of invariants.

Finally, sketching (imprecisely) what we develop in detail in Section 6.3.4, we provide a combinator below that is the analog of Iris' rule for eliminating and restoring invariants in atomic commands—an atomic command that expects p ∗ q and provides p ∗ r can be turned into a command that only expects q and provides r, as long as an inv p value can be presented as evidence that p is an invariant.

```
val with_invariant (i:inv p) (e:atomic a (p ∗ q) (p ∗ r)) : atomic a q r
```

Representing Invariants. We will use the istore component of a mem to keep track of invariants allocated with new_invariant: an istore is a list of slprops and the name associated with an invariant is its position in the list. The invariant of the istore (included in inv, which, recall from Section 6.2, is expected and preserved by every step of the semantics) requires every invariant in the istore to be satisfied separately. The i_evolves preorder (part of the mem_evolves preorder shown in Section 6.3.1) states that when the memory evolves, the istore only grows. The predicate inv_for_p i p m states that the invariant name i is associated with p in the memory m—its stable form, i⇝p, makes use of the witnessed connective used with the MST monotonic state effect. i⇝p is the

SteelCore equivalent of $\boxed{p}^{i}$, i.e., the name i is always associated with invariant p. Since i⇝p is just a prop, it is naturally duplicable. It is also convenient to treat invariants as a value type, inv p—which corresponds to an invariant name i refined to be associated with p.

```
let istore = list slprop

let istore_inv (i:istore) : slprop = List.fold_right (∗) emp i

let inv_name = nat

let i_evolves is0 is1 = ∀(i:inv_name). List.nth i is0 == None ∨ List.nth i is0 == List.nth i is1

let inv_for_p (i:inv_name) (p:slprop) (m:mem) = Some p == List.nth i m.istore

let (⇝) i p = witnessed (inv_for_p i p)

let inv (p:slprop) = i:inv_name{i⇝p}
```

Now, to define the new_invariant p action, we simply extend the istore, witness that p is now an invariant, and return the address of the newly allocated invariant.

```
let new_invariant (p:slprop) : action (inv p) p (λ _ → emp) = λ() →
  let m = get () in
  put ({m with istore=m.istore@[p]});
  let i = List.length m.istore in
  witness (inv_for_p i p);
  i
```

With these definitions in place, we have all we need to instantiate the state interface of the semantics, using for each of its fields (mem, slprop, evolves, etc.) the definitions shown here.

6.3.4 Using Invariants in Atomic Commands

We have seen how to allocate duplicable invariants, i.e., the analog of the first two rules for manipulating invariants in Iris. What remains is the third rule that allows invariants to be used in atomic commands.

For starters, this requires carving out a subset of computations that are deemed to be atomic, i.e., we need a way to express something like the premise atomic(e) from the Iris rule. However, observe that our semantics from Section 6.2 already provides a notion of atomicity: individual actions in Act nodes are run to completion without any interference from other threads. Specific actions in our memory model can be marked as atomic, depending on the particular architecture being modeled. For example, one might include a primitive, atomic compare-and-swap action, while other primitive actions like reading, writing, or allocating references may or may not be atomic. Further, some actions can be marked as *ghost*, and sequences of such commands may also be considered atomic, since they are never actually executed concretely.

Next, we need a way to determine which invariants are currently "opened" by an atomic command. Recursively opening the same invariant inv p is clearly unsound since, although inv p is duplicable, p itself need not be.

Finally, to fully recover Iris' atomic actions rule, we also need to model the *later* modality ▷. As we will see, the witnessed modality provided by the monotonic state effect serves that purpose well.

The Type of Atomic Actions. The type atomic a uses is_ghost p q is a refinement of the type of actions, action a p q, presented in Section 6.2.1. The first additional index, uses, indicates the set of opened invariants—in particular, an atomic action can only assume and preserve the invariants not included in uses, as shown in the definition of istore_inv'. The second index, is_ghost, is a tag that indicates whether or not this command is a ghost action. The atomic type represents an effectful operation, and uses a total sub-effect NMSTTot of the effects of nondeterminism and monotonic state—by choosing a *total* sub-effect, we avoid pitfalls of infinitely opening invariants or introducing divergence in ghost computations. As such, due to the restriction to total computations, the type atomic a {} b p q is a subtype of the action a p q type defined in the semantics.

```
let istore_inv' uses ps = List.fold_right_i (λ p i q → if i ∈ uses then q else p ∗ q) ps emp

let inv' uses m = . . . m.ctr . . . ∧ istore_inv' uses m.istore

let atomic (a:Type) (uses:set inv_name) (is_ghost:bool) (p:slprop) (q: a → slprop) =
    unit → NMSTTot a mem mem_evolves
            (requires λm → interp (inv' uses m ∗ p) m)
            (ensures λm0 x m1 → interp (inv' uses m1 ∗ q x) m1 ∧
                                        preserves_frame p (q x) m0 m1)
```

Foreshadowing a full presentation of the Steel language in Chapter 7, we will treat the atomic type as a user-defined abstract effect in F⋆, and insist on at most one non-ghost action in a sequential composition, as shown by the signature of bind_atomic below.

```
val bind_atomic #a #b #u #p #q #r #g1 (#g2:bool{g1 || g2})
   (e1:atomic a u g1 p q) (e2: (x:a → atomic b u g2 (q x) r)) : atomic b uses (g1 && g2) p r
```

Opening and Closing an Invariant. The last piece of the puzzle is the with_invariant construct, whose signature is shown below. Given an atomic command e that uses the invariant i:inv p to gain and restore p, e can be turned into an atomic command that no longer uses i, and whose use of p is no longer revealed in its specification.

```
val with_invariant (i:inv p) (e:atomic a (i ⊎ u) g (p ∗ q) (λ x → p ∗ r x)) : atomic a u g q r
```

Finally, given a value of type e : atomic a {} g p q, we can always apply subtyping to promote it to e : action a p q, and then turn it into a computation Act e : ctree a p q.

See Ya, Later. The with_invariant rule above does not have Iris' *later* modality, yet the later modality is essential for soundness in Iris and in other logics (Dodds et al. 2016) that support stored propositions. Paraphrasing Jung et al. (2018), a logic that supports allocating persistent propositions, together with a deduction rule for the injectivity of stored propositions of the form $\mathsf{i} \rightsquigarrow \mathsf{p} * \mathsf{i} \rightsquigarrow \mathsf{q} \vdash (\mathsf{p} \iff \mathsf{q})$ is inconsistent—the conclusion of the rule must be guarded under a later, i.e., it should be $\triangleright(\mathsf{p} \iff \mathsf{q})$.

Although it may not be immediately evident, the model of monotonic state proposed by Ahman et al. (2018) also has a "later" modality in disguise. In this model, $\mathsf{witnessed} \perp \not\vdash \perp$: instead, an explicit step of computation via the recall action is necessary to extract a contradiction from $\mathsf{witnessed} \perp$. As such, $\mathsf{i} \rightsquigarrow \mathsf{p} * \mathsf{i} \rightsquigarrow \mathsf{q} \vdash (\mathsf{p} \iff \mathsf{q})$ is *not* derivable in SteelCore, although, with a step of computation, the Hoare triple $\{\mathsf{i} \rightsquigarrow \mathsf{p} * \mathsf{i} \rightsquigarrow \mathsf{q}\}$ recall i $\{\ \mathsf{p} \iff \mathsf{q}\ \}$ is. In summary, the effect of monotonic state provides a way to account for the necessary step indexing without making it explicit in the logic.

The Update Modality and Ghost Actions. As a final remark, allocating an invariant in Iris is done using its *update* modality, $\Rrightarrow_{\mathcal{E}}$. Besides allocating invariants, updates in Iris are also used to transform ghost state. In SteelCore, rather than including such a modality within the logic, we rely on F*'s existing support for erased types to model ghost state and updates within Hoare triples, rather than within the logic itself.[3] For instance, the following action represents a ghost read: it dereferences x, returning its contents only as an erased a.

```
val ghost_read (x:ref a p) : atomic (erased a) u true (∃ v. pts_to r v) (λ v → pts_to r v)
```

6.3.5 Fractional Permissions and Monotonic References

Several prior works have provided PCM-based constructions, both to capture various sharing idioms as well as to define state machines that constrain how the state is permitted to evolve. In this section, we show how to use PCMs to encode the preorder-indexed monotonic references proposed by Ahman et al. (2018). We start, however, with a simpler construction of references with fractional permissions, a construction we reuse for monotonic references.

References with Fractional Permissions. To model references to t-typed values with fractional permissions, we store at each cell a value of type frac t with a PCM pcm_frac shown below—the composable predicate allows us to use undecidable relations like propositional equality in our notion of partiality.

```
let frac t = option (t & r:real{0.0 < r})

let frac_composable (f0 f1:frac t) = match f0, f1 with
    | Some (v0, r0), Some(v1, r1) → v0==v1 ∧ r0+r1 ≤ 1.0
    | _ → ⊤
```

[3] Iris also internalizes Hoare triples, but in SteelCore, we rely on the computation types of the host language to express Hoare triples outside the logic.

```
let frac_op (f0 f1:fract t) = match f0, f1 with
    | None, f | f, None → f
    | Some (v, r0), Some(_, r1) → Some(v, r0 + r1))

let pcm_frac : pcm (frac t) = {
  one = None;
  composable = frac_composable;
  op = frac_op;
}
```

Specializing the type of references and the points-to assertion for use with pcm_frac, we recover the traditional injective points-to assertion on references, and a lemma that relates the separating conjunction in slprop to composition in pcm_frac.

```
let ref t = ref (frac t) pcm_frac

let (↦_f) r v = pts_to r (Some (v, f)) ∗ pure (f ≤ 1.0)

val share_gather (r:ref t) (f g:real) (u v:t)
  : Lemma ((r ↦_f u ∗ r ↦_g v) ∼ r ↦_{f+g} u ∗ pure (u==v))
```

Monotonic References. Whereas we have used preorders and monotonic state within our memory model to support the dynamic allocation of invariants, we now aim to expose preorders to describe state transitions on individual references, in the style of Ahman et al.'s references. Pleasently, we find that our PCM-based memory model layered above the monotonic state effect can precisely capture Ahman et al.'s construction in a generic manner.

Our goal is to provide the following interface on an abstract type mref a p of references indexed by a preorder. The main point of interest is the signature of write, which requires proving that the new value v is related to the old value by the preorder p.

```
val mref (a:Type) (p:preorder a) : Type

val (⟼_f) (x:mref a p) (v:a) : slprop

val read (r:mref a p) (v0:erased a) : action a (r ⟼_f v0) (λ v → r ⟼_f v)

val write (r:mref a p) (v0:erased a) (v:a{p v0 v}) : action unit (r ⟼_1.0 v0) (λ _ → r ⟼_1.0 v)

val observed (r:mref a p) (q:a → prop) : prop

val witness_mref (r:mref a p) (q:stable_prop p) (v:erased a{q v})
    : action unit (pts_to r f v) (λ _ → pts_to r f v ∗ pure (observed r q))

val recall_mref (r:mref a p) (q:stable_prop a p) (v:erased a)
    : action unit (pts_to r f v ∗ pure (observed r q) (λ _ → pts_to r f v ∗ pure (q v))
```

In return for preserving the preorder at each update, we provide two new operations to witness and recall properties that are invariant under the preorder. The operation witness returns a pure, abstract predicate observed r q when the current value or r satisfies a stable property q; and recall eliminates observed r q into q v, for v the current value of r. These operations are the analogue of the MST actions witness and recall exposed to SteelCore programs, but at the granularity of a single reference rather than the entire state. For instance, one could define a monotonically increasing counter as r : mref int ($\leq$), and having observed that r contains the value 17, one can recall later that r's value is at least 17.

From PCMs to Preorders. We observe that every PCM induces a preorder and, dually, every preorder can be encoded as a PCM. To interpret a PCM as a preorder, we take the infinite conjunction of all preorders refined by the frame_preserving relation; in other words, since all updates must be frame-preserving, we take the preorder of a PCM to be the strongest preorder entailed by frame preservation.

```
let induces (p:pcm a) (q:preorder a) =
  ∀(x y:a). frame_preserving p x y ⟹ (∀ (z:a). p.compatible x z ⟹ q z y)

let preorder_of_pcm (p:pcm a) : preorder a = λx y → ∀q. p `induces` q ⟹ q x y
```

With this notion in hand, we can finally define the heap evolution relation (part of the global memory preorder shown in Section 6.3.1) stating that (1) unused heap cells can change arbitrarily; (2) used heap cells remain used; and, most importantly, (3), the type and PCM associated with a ref cell does not change and its value evolves according to the preorder of the PCM. In other words, heaps evolve by the pointwise conjunction of the PCMs at each cell.

```
let h_evolves h0 h1 = ∀(a:addr). match h0 a, h1 a with
  | None, _ → ⊤
  | Some _, None → ⊥
  | Some (Ref a0 p0 v0), Some (Ref a1 p1 v1) →
      a0 == a1 ∧ p0 == p1 ∧ preorder_of_pcm p0 v0 v1
```

From Preorders to PCMs. Conversely, to interpret a preorder q : preorder a as a PCM, we define a PCM over hist q, the type of histories over a, i.e., sequences of a-values where adjacent values are related by q, with composability demanding one history to be an extension of the other; composition being history extension; and the unit being the empty history. We show the main signature below, including a round-trip property, guaranteeing that the PCM built by the construction induces the preorder corresponding to extension of q-respecting histories.

```
val pcm_of_preorder (q:preorder a) : p:pcm (hist q) {p `induces` history_extension}
```

This construction enables constructing a PCM frac_hist q to support the type mref a q, combining fractional permissions with the hist q PCM, with the property that for any predicate f : a → prop stable with respect to q, its lifting lift f : hist q → prop, which applies to the most recent value in a history, is stable with respect to the preorder

preorder_of_pcm (frac_hist q). As such, the underlying witness and recall operations of the monotonic state effect suffice to provide a model for witness_mref and recall_mref.

6.4 Summary

In this chapter, we have demonstrated how a full-fledged CSL can be embedded in an effectful dependent type theory, relying on an underlying semantics of monotonic state to model features that have otherwise required impredicative logics. While our work draws inspiration from the Iris framework (Jung et al. 2018b), several significant points of contrast stem from our differing goals. Iris is a powerful, impredicative logical framework into which other logics and programming langauges can be embedded and studied; in contrast, SteelCore instead aims to extend an existing proof assistant's programming language with an effect for concurrency, in order to reason about effectful, dependently typed programs in a CSL. This allows us to keep the embedded logic relatively simple, for instance not making use of step indexing or any of Iris' several modalities, while nevertheless recovering many lacking features in the logic using the dependently typed facilities of F*. As a shallow embedding of CSL in a dependent type theory, SteelCore also shares many similarities with FCSL (Nanevski et al. 2014; Nanevski et al. 2019; Sergey et al. 2015), which is shallowly embedded in Coq and relies on Coq's abstraction facilities for some of its expressive power. FCSL's logic, similarly to ours but unlike Iris', directly applies to Coq programs, rather than to embedded programs. However, FCSL's model is predicative; as such, it does not provide a way to dynamically allocate an invariant, making it impossible to model certain kinds of synchronization primitives. On the other hand, FCSL provides several constructs for reasoning about concurrent programs mixing styles of reasoning from CSL with rely-guarantee reasoning, something we have not explored much: our use of monotonic references may play a role in this directly, particularly in connection with other related work on rely-guarantee references (Gordon et al. 2013). With SteelCore, we thus provide a foundation to implement and reason about complex, concurrent programs. In the next chapter, we show how to use it as a building block to develop a higher-level domain-specific language embedded in F*, aiming to provide automation facilities to ease proofs of programs.

Chapter 7

Automating Separation Logic Reasoning in Steel

In the previous chapter, we presented SteelCore, an impredicative, dependently typed Concurrent Separation Logic (CSL) for partial-correctness proofs shallowly embedded into F*. SteelCore provides many useful features to reason about concurrent programs, including a user-defined partial commutative monoid (PCM)-based memory model, atomic and ghost computations, and dynamically allocated invariants. Unfortunately, it is far from being usable directly to build correct programs.

SteelCore's action trees presented in Section 6.2 are inconvenient to use to write programs. In particular, these action trees require users to manually insert Frame and Sub nodes in their programs. Consider for instance the program presented in Figure 7.1, which swaps the contents of two references, assuming a small surface syntax to make programming with action trees more palatable. Calls to frame wrapping each action combined with applications of the rule of consequence with commute_star to rearrange slprops overwhelm the program—the pain is perceptible.

```
let swap (#v1 #v2:ghost int) (r1 r2:ref int)
  : SteelCore unit
      (pts_to r1 #v1 ∗ pts_to r2 #v2)
      (λ _ → pts_to r1 #v2 ∗ pts_to r2 #v1)
  = let x1 = frame (read r1) (pts_to r2 #v2) in
    commute_star (pts_to r1 #v1) (pts_to r2 #v2);
    let x2 = frame (read r2) (pts_to r1 #v1) in
    frame (write v2 x1) (pts_to r1 #x2);
    commute_star (pts_to r2 #x1) (pts_to r1 #v1);
    frame (write r1 x2) (pts_to r2 #v2);
```

Figure 7.1: SteelCore implementation of swap

To tackle this issue, we propose in this chapter an embedded domain-specific language (DSL) within F*, based on SteelCore. By designing this language with proofs in mind, we cleanly isolate separation logic verification conditions from other verification conditions, and provide a custom decision procedure to automate the former while relying on an SMT solver for the latter. This allows us to simplify the swap function previously presented, yielding the implementation shown in Figure 7.2. The end result is a verification framework, Steel, which integrates the expressive power of the SteelCore logic within a higher-order, dependently typed programming language, with proof automation approaching what is offered by SMT-based frameworks with first-order program logics such as Dafny (Leino 2010), Chalice (Leino et al. 2009) or Viper (Müller et al. 2016), but with soundness ensured by construction upon the foundations of SteelCore.

```
let swap (r1 r2:ref int) : Steel unit
   (ptr r1 ∗ ptr r2) (λ _ → ptr r1 ∗ ptr r2)
   (requires λ_ → ⊤)
   (ensures λs _ s' → s'.[r1]=s.[r2] ∧ s'.[r2]=s.[r1])
  = let x1 = read r1 in
    let x2 = read r2 in
    write r2 x1;
    write r1 x2
```

Figure 7.2: Steel implementation of swap

Freeing this swap program from framing requires several steps, which we summarize next before presenting them in detail over the course of this chapter.

Quintuples: Selectively Separating Separation and First-Order Logic. We develop a verification condition generator and hybrid tactic- and SMT-based solver for a separation logic of quintuples whose main judgment involves a computation type Steel a p q r s, where p, q are slprops as usual, but r, s are first-order logic encodeable self-framing selector predicates that depend on the p-fragment of the initial memory and q-fragment of the final memory, i.e., they are self-framing in the terminology of Parkinson and Summers (2012). Proof obligations in our formulation are in two classes: separation logic goals, relating the slprops in a judgment, and SMT encodeable goals relating the selector predicates. This allows us to write efficient reflective tactics that focus on the former, while the latter are encoded efficiently to SMT by F*, as usual. In a style reminiscent of cooperating decision procedures proposed by Nelson and Oppen (1979), we show how tactics and SMT share information through equalities on uninterpreted symbols. Of course, proof obligations remain undecidable and what automation we do provide is partial, but being embedded in F*, additional lemmas can always be developed interactively.

A Type-and-Effect Directed Frame Rule. To control the placement of frames, we model the application of the frame rule as an effect. In particular, we formulate the system using a second related computation type that contains metavariables for an unsolved frame introduced by an application of the frame rule. This allows us to build a type-and-effect directed elaborator for Steel, inserting frames only at the leaves of a derivation, while proving that this strategy of leaf-framing is complete. Our approach enables interactions among several fragments of Steel modeled in an effect hierarchy, including atomic and ghost code.

Automating Frame Inference through AC-Matching. Automatically applying the separation logic *frame* rule in the right places only solves half of the problem; generated frames must also be inferred. We prove that our type-and-effect system yields a unitriangular[1] system of constraints on the frame metavariables that can be solved by associative-commutative (AC)-matching (Kapur and Narendran 1987). Compared to standard AC-matching algorithms, we favor an incomplete but efficient and predictable non-backtracking approach that only solves problems with unique solutions, while accounting for equalities, theory reasoning, and existentially quantified ghost variables—the interaction with theory reasoning is enabled by the quintuple formulation.

Discussion. The Steel swap program presented in Figure 7.2 is perhaps close to what one would expect in Chalice or Viper, but we emphasize that Steel is a shallow embedding in dependently typed F* and the full SteelCore logic is available within Steel. So, while Viper-style program proofs are possible and encouraged, richer, dependently typed idioms are also possible and enjoy many of the same benefits, e.g., automated framing and partial automation via SMT. Indeed, our approach seeks only to automate the most mundane aspects of proofs, focusing primarily on framing. For the rest, including introducing and eliminating quantifiers, rolling and unrolling recursive predicates, writing invariants, and manipulating ghost state, the programmer can develop lemmas in F*'s underlying type theory and invoke these lemmas at strategic points in their code—Steel provides many generic building blocks for such lemmas. The result is a style that Leino and Moskal (2010) have called *auto-active* verification, a mixture of automated and interactive proof that has been successful in other languages, including in other large F* developments, but now applied to SteelCore's expressive CSL.

Contributions. All the work presented in this chapter was primarily mine, and was first presented in Fromherz et al. (2021). The initial concept of a self-framing *selector* was suggested by Denis Merigoux.

7.1 Verification Condition Generation for Separation Logic: Overview

Like any separation logic, SteelCore has rules for framing, sequential composition, and consequence, which we presented in the previous chapter. For presentation purposes, we will consider in this section a simplified form of these rules, using combinators with the following signatures:

[1]The term is borrowed from linear algebra: a unitriangular matrix is a triangular matrix such that all values on its main diagonal are 1.

```
let stc a p q = unit → SteelCore a p q (* represents an action tree of type (m a p q) *)
val frame (_:stc a p q) : stc a (p ∗ f) (λ x → q x ∗ f)
val bind (_:stc a₁ p q’) (_: (x:a₁ → stc a₂ (q’ x) r)) : stc a₂ p r
val conseq (_:stc a p’ q’) (_:squash (p −∗ p’ ∧ q’ −∗ q)) : stc a p q
```

Our goal is to shallowly embed Steel as a DSL[2] in F⋆, whereby Steel user programs are constructed by repeated applications of combinators like frame, bind and conseq. The result is a program whose inferred type is a judgment in the SteelCore logic, subject to verification conditions (VCs) that must be discharged, e.g., the second argument of conseq, squash (p −∗ p’ ∧ q’ −∗ q), is a proof obligation, where −∗ is the separation logic implication.

For this process to work, we need to make the elaboration of a Steel program into the underlying combinator language algorithmic, resolving the inherent nondeterminism in rules like Frame and Consequence by deciding the following: first, where exactly should Frame and Consequence be applied; second, how should existentially bound variables in the rules be chosen, notably the frame f; and, finally, how should the proof obligations be discharged.

The standard approach to this problem is to define a form of weakest precondition (WP) calculus for separation logic that strategically integrates the use of frame and consequence into the other rules in the system. Starting with “backwards” rules from Ishtiaq and O’Hearn (2001), weakest precondition readings of separation logic have been customary. Hobor and Villard (2013) propose a ramified frame rule that integrates the rule of consequence with framing, while Iris’ (Jung et al. 2018b) “Texan triples” combine both ideas, integrating a form of ramified framing in the WP-Wand rule of its WP calculus. In the setting of interactive proofs, Texan triples are convenient in that every command is always specified with respect to a parametric postcondition, enabling it to be easily applied to a framed and weakened (if necessary) postcondition.

Prior attempts at encoding separation logic in F⋆ (Martínez et al. 2019) followed a similar approach, whereby a Dijkstra monad (Swamy et al. 2013) for separation logic computes weakest preconditions while automatically inserting frames around every function call or primitive action. However, Martínez et al. (2019) have not scaled their prototype to verify larger programs and we have, to date, failed to scale their WP-based approach to a mostly-automated verifier for Steel.

The main difficulty is that a WP-calculus for separation logic computes a single (often quite large) VC for a program in, naturally, separation logic. F⋆ aims to encode such VCs to an SMT solver. However, encoding a separation logic VC to an SMT solver is non-trivial. SMT solvers like Z3 (Moura and Bjørner 2008) do not handle separation logic well, in part because slprops are equivalent up to Associativity-Commutativity (AC) rewriting of ∗, and AC-rewriting is hard to automate in SMT. Besides, WP-based VCs heavily use magic wand, and computing frames involves solving for existential quantifiers over AC terms, which again is hard to automate in SMT. Viper (the underlying engine of

[2]A note on terminology: From one perspective, Steel is not domain-specific—it is a general-purpose, Turing complete language, with many kinds of computational effects. But, from the perspective of its host language F⋆, Steel is a domain-specific language for proof-oriented stateful and concurrent programming.

Chalice) does provide an SMT-encoding for a permission system with implicit dynamic frames that is equivalent to a fragment of separation logic (Parkinson and Summers 2012); however, we do not have such an encoding for SteelCore's more expressive logic. While some other off-the-shelf solvers for various fragments of separation logic exist (Brotherston et al. 2012; Iosif et al. 2014), using them for a logic like SteelCore's dependently typed, impredicative CSL is an open challenge.

Martínez et al. (2019) confront this problem and develop tactics to process a separation logic VC computed by their Dijkstra monad, AC-rewriting terms and solving for frame variables, and finally feeding a first-order logic goal to an SMT solver. However, this scales poorly even on their simpler logic, with the verification time of a program dominated by the tactic simply discovering fragments of a VC that involve non-trivial separation logic reasoning, introducing existentially bound variables for frames, solving them and rewriting the remainder of the VC iteratively.

Our solution over the next several sections addresses these difficulties by developing a verification condition generator for quintuples, and automatically discharging the computation of frames using a combination of AC-matching tactics and SMT solving, while requiring the programmer to write invariants and to provide lemmas in the form of imperative ghost procedures.

7.2 A Type-and-Effect System for Separation Logic Quintuples

Our goal is to shallowly embed the SteelCore logic as a DSL for programming in F*'s type theory. This involves exposing the proof rules of the logic as dependently typed combinators and instructing F*'s typechecker to apply those combinators during type inference and elaboration, coupling the program with a proof of its correctness. To make this process algorithmic, we rely on F*'s user-defined effect system, which through our encoding, provides the needed structure. Rather than focus on the syntactic details of our encoding, we present a more abstract view of the elaboration problem and our solution in standard mathematical notation, relating back to the specifics of F* and SteelCore only when essential. In this section, we thus present our elaboration and VC generation strategy as a small, idealized calculus. We transcribe the rules omitting some side conditions (e.g., on the well-typedness of some terms) when they add clutter—such conditions are all captured formally in our mechanization. As such, these rules are implemented as combinators in F*'s effect system and mechanically proven sound against SteelCore's logic in F*.

7.2.1 Syntax

Figure 7.3 presents the syntax of a subset of the internal, desugared, monadic language of Steel in F*. Our implementation supports the full F* language, including full dependent types, inductive types, pattern matching, recursive definitions, local let bindings, universes, implicit arguments, a module system, typeclasses, etc. This is the

$$\begin{array}{llcl}
\text{constant} & T & ::= & \mathsf{unit} \mid () \mid \mathsf{Type} \mid \mathsf{prop} \mid \mathsf{vprop} \mid \ldots \\
\text{term} & e, t & ::= & x \mid T \mid \lambda x{:}t.\ e \mid e_1\ e_2 \mid x{:}t \to C \mid \mathsf{ret}\ e \mid \mathsf{bind}\ e_1\ x.e_2 \\
 & & \mid & e_1 \ast e_2 \mid e_1 \mathrel{-\!\!\ast} e_2 \mid e_1 \wedge e_2 \mid \forall x.e \mid \ldots \\
\text{computation type} & C & ::= & \mathsf{Tot}\ t \mid \{\ P \mid R\ \}\ y{:}t\ \{\ Q \mid S\ \} \\
\text{program} & d & ::= & \mathsf{val}\ f\ (x : t)\ :\ C\ =\ e
\end{array}$$

Figure 7.3: Simplified syntax for Steel

advantage of a shallow embedding: Steel inherits the full type system of F*. For the purpose of our minimalistic presentation, the main constructs of interest are vprops and computation types, which we describe next.

What is a vprop? A vprop is a typeclass that extends the slprops presented in Chapter 6. A vprop encapsulates two things: an interpretation as a separation logic proposition (inherited from the underlying slprops), and a self-framing memory representation called a *selector*. Specifically, it supports the following operations:

- An interpretation, defined as an affine predicate on memories, namely, a function interp (_:vprop) : mem → prop which satisfies the following affinity property interp p m ∧ disjoint m m' $\Longrightarrow$ interp p (join m m'). Similarly to Section 6.2.4, we write fpmem (p:vprop) for a memory validating p, i.e., m:mem { interp p m }.
- A selector type, type_of (p:vprop) : Type
- A selector, sel (p:vprop) (m:fpmem p) : type_of p, with the property that sel depends only on the p fragment of m, i.e., mimicking predicates from Section 6.2.4, (∀(m0:fpmem p) m1. disjoint m0 m1 $\Longrightarrow$ sel p m0 = sel p (join m0 m1)).
- vprops inherit all the usual connectives from slprops, including $\ast$, $-\!\!\ast$, ∧, ∨, ∀, ∃ etc. We observe that the selectors provide a form of linear logic over memory fragments as resources. For instance, the selector type for p ∗ q corresponds to a linear pair type_of p ∗ type_of q, while the selector type for p −∗ q is a map from memories validating the p ∗ (p −∗ q) to the type of q. However, we do not yet exploit this connection deeply, except to build typeclass instances for ∗ and −∗ and to derive the double implication p ∗−∗ q, a bidirectional coercion on selectors.

It is trivial to give a degenerate selector for any vprop, simply by picking the selector type to be unit. But, more interesting instances can be provided by the programmer depending on their needs. For example, given a reference r:ref a, the interpretation of ptr r : vprop could be that r is present in a given memory; type_of (ptr r) = a, and sel (ptr r) m : a could return the value of the reference r in m.

Computation Types. The type Tot t is the standard F* type of total computations and is not particularly interesting. The main computation type is the quintuple $\{\ P \mid R\ \}\ x{:}t\ \{\ Q \mid S\ \}$, where

$$\textsc{App}\quad \frac{\Gamma \vdash e : \mathsf{Tot}\ t \qquad \Gamma \vdash f : x{:}t \to C}{\Gamma \vdash f\ e : C[e/x]}$$

$$\textsc{Frame}\quad \frac{\Gamma \vdash e : \{\ P \mid R\ \}\ y{:}t\ \{\ Q \mid S\ \}}{\Gamma \vdash_F e : \{\ P * ?F \mid \lambda(s_{p_0}, s_{f_0}).R\ s_{p_0}\ \}\ y{:}t\ \{\ Q * ?F \mid \lambda(s_{p_0}, s_{f_0})\ (s_{q_1}, s_{f_1}).S\ s_{p_0}\ s_{q_1} \land \mathit{seleq}\ ?F\ s_{f_0}\ s_{f_1}\ \}}$$

$$\text{let } \mathit{pre}\ \chi\ R_1\ S_1\ R_2\ ?a\ ?b = \lambda s_{p_1}.\ R_1\ s_{p_1} \land \forall x\ s_{p_2}.\ ?a \land (S_1[x/y]\ s_{p_1}\ (\chi\ s_{p_2}) \implies R_2\ s_{p_2} \land \forall z.\ ?b)$$
$$\text{let } \mathit{post}\ \chi_1\ \chi_2\ S_1\ S_2 = \lambda s_{p_1}\ s_q.\ \exists x\ s_{p_2}.\ S_1\ s_{p_1}\ s_{p_2} \land S_2\ (\chi_1\ s_{p_2})\ (\chi_2\ s_q)$$

$$\textsc{Bind}\quad \frac{\begin{array}{c}\Gamma \vdash_F e_1 : \{\ P_1 \mid R_1\ \}\ y{:}t_1\ \{\ Q_1 \mid S_1\ \} \qquad \Gamma, x{:}t_1 \vdash_F e_2 : \{\ P_2 \mid R_2\ \}\ z{:}t_2\ \{\ Q_2 \mid S_2\ \} \\ \Gamma, x{:}t_1, ?a \vDash_{tac} Q_1[x/y] \ast\!\!-\!\!\ast P_2 : \chi_1 \qquad \Gamma, x{:}t_1, z{:}t_2, ?b \vDash_{tac} Q_2 \ast\!\!-\!\!\ast ?Q : \chi_2 \qquad x \notin FV(t_2, ?Q)\end{array}}{\Gamma \vdash_F \mathsf{bind}\ e_1\ x.e_2 : \{\ P_1 \mid \mathit{pre}\ \chi_1\ R_1\ S_1\ R_2\ ?a\ ?b\ \}\ z{:}t_2\ \{\ ?Q \mid \mathit{post}\ \chi_1\ \chi_2\ S_1\ S_2\ \}}$$

$$\textsc{Val}\quad \frac{\begin{array}{c}\Gamma, x : t_1 \vdash_F e : \{\ P' \mid R'\ \}\ y{:}t_2\ \{\ Q' \mid S'\ \} \\ \Gamma, x{:}t_1, ?a \vDash_{tac} P \ast\!\!-\!\!\ast P' : \chi_p \qquad \Gamma, x{:}t_1, y{:}t_2, ?b \vDash_{tac} Q' \ast\!\!-\!\!\ast Q : \chi_q \\ \Gamma \vDash_{smt} \forall x\ s_p.\ (R\ s_p \implies ?a \land R'\ (\chi_p\ s_p)) \land (\forall y\ s_q.\ S'\ (\chi_p\ s_p)\ (\chi_q\ s_q) \implies ?b \land S\ s_p\ s_q)\end{array}}{\Gamma \vdash \mathsf{val}\ f\ (x : t_1)\ :\ \{\ P \mid R\ \}\ y{:}t_2\ \{\ Q \mid S\ \}\ =\ e}$$

Figure 7.4: Core rules of Steel's type and effect system

- P : vprop is a separation logic precondition
- R : fppred P, where R is a predicate on P's selector, i.e., fppred p = type_of p → prop, where the predicate is applied to sel p on the underlying memory.
- x : t binds the name x to the t-typed return value of the computation.
- Q : vprop is a postcondition, with x:t in scope.
- S : fppost P Q is an additional postcondition, relating the selector of P in the initial memory, to the result and the selector of Q in the final memory, i.e., fppost (p:vprop) (q:vprop) = type_of p → type_of q → prop. It also has x:t in scope.

With its encoding of implicit dynamic frames, SteelCore's logic also provides support for a form of quintuples, but with one major difference: instead of operating on selectors, SteelCore uses memory predicates with proof obligations that they are self-framing, i.e., that they depend only on the appropriate part of memory. Proving that a memory predicate is self-framing can be tedious, and doing so for each specification significantly hampers the usability of SteelCore's quintuples. In contrast, the abstraction provided by Steel's quintuples with selectors frees the user from proof-obligations on the framing of memory predicates, while also being proven sound in the model of SteelCore's "raw" quintuples—proof-oriented programming at work!

7.2.2 VC Generation for Steel

Figure 7.4 presents selected rules for typechecking Steel programs. There are 3 main ideas in the structure of the rules.

First, there are two kinds of judgments, $\vdash$ and $\vdash_F$. The $\vdash$ judgment applies to terms on which no top-level occurence of framing has been applied. Dually, the $\vdash_F$ judgment marks terms that have been framed. We use this modality to ensure that frames are applied at the leaves, to effectful function calls only, and nowhere else. The application of framing introduces metavariables to be solved and introduces equalities among framed selector terms.

Second, the rule of consequence together with a form of framing is folded into sequential composition. Both consequence and framing can also be triggered by a user annotation in a val. Although Steel's separation logic is affine, Steel aims at representing and modeling a variety of concurrent programs, especially including low-level programs implemented in a language with manual memory management, such as C. To this end, we need to ensure that separation logic predicates do not implicitly disappear. As such, our VC generator uses equivalence $*\!\!-\!\!*$ where otherwise a reader might expect to see implications ($-\!\!*$). Programmers are expected to explicitly *drop* separation logic predicates by either freeing memory or calling ghost functions to drop ghost resources.

Finally, the proof obligations corresponding to the VCs in the rules appear in the premises in two forms, $\vDash_{tac}$ and $\vDash_{smt}$. The former involves solving separation logic goals using a tactic, which can produce auxiliary propositional goals to interact with SMT. The latter are SMT-encodeable goals—all non-separation logic reasoning is collected in the other rules, and eventually dispatched to SMT at the use of consequence triggered by a user annotation.

We now describe each of the rules in turn.

App. This is a straightforward dependent function application rule. F* internal syntax is already desugared into a monadic form, so we need only consider the case where both the function f and the argument e are total terms. Of course, the application may itself have an effect, depending on C. The important aspect of this rule is that it is a $\vdash$ judgment, indicating that this is a raw application—no frame has been added.

Frame. This rule introduces a frame. Its premise requires a $\vdash$ judgment to ensure that no repeated frames are added, while the conclusion is, of course, in $\vdash_F$, since a frame has just been applied. The rule involves picking a fresh metavariable $?F$ and framing it across the pre- and postconditions. The effect of framing on the memory postcondition S is particularly interesting: we strengthen the postcondition with $\mathsf{seleq}\ ?\mathsf{F}\ s_{f_0}\ \ s_{f_1}$, which is equivalent to $\mathsf{sel}\ ?\mathsf{F}\ s_{f_0} = \mathsf{sel}\ ?\mathsf{F}\ s_{f_1}$. We will present this predicate in detail in Section 7.2.5.

Bind. The most interesting rule is Bind, with several subtle elements. First, in order to sequentially compose e_1 and e_2, in the first two premises we require $\vdash_F$ judgments, to ensure that those computations have already been framed. The third premise encodes an application of consequence, to relate the vprop-postcondition Q_1 of e_1 to the vprop-precondition P_2 of e_2. Strictly speaking, we do not need a double implication here, but we generate equivalence constraints to ensure that our constraint solving heuristics do not implicitly drop resources. The premise $\Gamma, x{:}t_1, ?a \vDash_{tac} Q_1[x/y] \ast\!\!-\!\!\ast P_2 : \chi_1$ is a VC that is discharged by a tactic and, importantly, $?a$ is a propositional metavariable that the tactic must solve. For example, in order to prove $\Gamma, x{:}t_1, ?a \vDash_{tac} (r \to u) \ast\!\!-\!\!\ast (r \to v)$, where the interpretation of $r \to u$ is that the reference r points to u, a tactic could instantiate $?a := (u = v)$, i.e., the tactic is free to pick a hypothesis $?a$ under which the entailment is true. The fourth premise is similar, representing a use of consequence relating the postcondition of e_2 to a freshly picked metavariable $?Q$ for the entire postcondition, again not dropping resources implicitly. A technicality is the use of the selector coercions χ_1, χ_2 witnessing the equivalences, which are needed to ensure that the generated pre- and postconditions are well-typed. Notice that this rule does not have an SMT proof obligation. Instead, we gather in the precondition the initial precondition R_1 and the relation between the intermediate post- and preconditions, S_1 and R_2. Importantly, we also include the tactic-computed hypotheses $?a$ and $?b$, enabling facts to be proved by the SMT solver to be used in the separation logic tactic. Finally, in the postcondition, we gather the intermediate and final postconditions.

Val. The last rule checks that the inferred computation type for a Steel program matches a user-provided annotation, and is similar to most elements of Bind. As shown by the use of the entailment $\vdash_F$, it requires its premise to be framed. The next two premises are tactic VCs for relating the vprop pre- and postconditions, with the same flavor as before, allowing the tactic to abduct a hypothesis under which the goal is validated. Finally, the last premise is an SMT goal, which includes the freshly abducted hypotheses, and a rule of consequence relating the annotated pre- and postcondition to what was computed. Annotated computation types are considered to not have any implicit frames, hence the use of $\vdash$ in the conclusion.

As an example, typechecking the swap program presented in Figure 7.2 proceeds as follows: The App rule is applied to each of the read and write function applications. Each application of App is followed by an application of Frame; this enables the composition of the function applictions using the Bind rule, whose premises require $\vdash_F$ judgments. Finally, an application of the Val rule ensures that the annotated Steel computation type—an F*, user-friendly syntax for the computation type $\{\ P \mid R\ \}\ x{:}t\ \{\ Q \mid S\ \}$—is admissible for this swap program.

By structuring this calculus with proofs in mind, we have set the stage for tactics to focus on efficiently solving vprop goals, while building carefully crafted SMT-friendly VCs that can be fed as is to F*'s existing, heavily used SMT backend.

7.2.3 Why it Works: Proof-Oriented Programming

In Section 7.1, we claimed that prior attempts at using a WP-based VC generator for separation logic in F* did not scale. Here, we discuss some reasons why, and why the design we present here fares better.

As a general remark, recall that we want the trusted computing base (TCB) of Steel to be the same as SteelCore, i.e., we trust F* and its TCB. As such, our considerations for the scalability of one design over another will be based, in part, on the difficulty of writing efficient, untrusted tactics to solve various kinds of goals. Further, we aim for a Steel verifier to process an entire procedure in a single go and respond in no more than a few seconds or risk losing the user's attention. In contrast, in fully interactive verifiers, users analyze just a few commands at a time and requirements on interactive performance may be somewhat less demanding.

WP-Based VCs are Large and Require Non-Reflective Tactics. Separation logic provides a modular way to reason about memory, but properties about memory are only one of several concerns when proving properties about a program. VCs for programs in F* contain many other elements: exhaustiveness checks for case analysis, refinement subtyping checks, termination checks, and several other facts. In many existing F* developments, a VC for a single procedure can contain several thousand logical connectives, and the VC itself includes arbitrary pure F* terms. Tactics for separation logic proposed by Martínez et al. (2019) process this large term, applying verifiable but slow proof steps just to traverse the formula—think repeated applications of inspecting the head symbol of the goal, introducing a binder, splitting a conjunction, introducing an existential variable—even these simple steps are not cheap, since they incur a call to the unifier on very large terms—until, finally, a vprop-specific part of a VC is found, split from the rest and solved, while the rest of the VC is rewritten into propositional form and fed to the SMT solver.

Although F*'s relatively fresh and unoptimized tactic system bears some of the blame, tactics like this are inherently inefficient. Anecdotally, in conversations with some Iris users, we are told that running its WP-computations on large terms would follow a similar strategy to Martínez et al.'s tactics, and can also be quite slow. Instead, high-performance tactics usually make use of techniques like proof-by-reflection (Gonthier et al. 2016), but a reflective tactic for processing WP-based VCs is hard, since one would need to reflect the entire abstract syntax of pure F* terms and write certified transformations over it—effectively building a certified solver for separation logic.

Structured VCs Separate Concerns. A proof-oriented programming mindset suggests that producing a large unstructured VC and trying to write tactics to later recover structure from it is the wrong way to go about things. Instead, we propose to *program* our VC generator with the subsequent proofs of the VCs in mind: our proof rules are designed to produce VCs that have the right structure from the start, separating vprop reasoning and other VCs by construction. The expensive unification-based tactics to process large VCs are no longer needed. We only need to run tactics on very specific, well-identified sub-goals and the large SMT goals can be fed as is by F* to the SMT solver, once the tactics have completed.

Reflective Tactics for vprop Goals. Our tactics that focus on vprop implications are efficient because we use proof-by-reflection. Rather than reflect the entire syntax of F*, we only reflect the vprop skeleton of a term, and then can use certified, natively compiled decision procedures for rewriting in commutative monoids and AC-matching to (partially) decide vprop equivalence and solve for frames. What calls we make to the unifier are only on relatively small terms.

7.2.4 Correspondence to our Implementation

To encode the two judgments $\vdash$ and $\vdash_F$, we define a new F* computation type, SteelBase (a:Type) (framed:bool) (p:vprop) (q:a → vprop) (r:fppred p) (s:fppost p a q). In this effect, the indices a, p, q, r, s reflect the computation type $\{\ P \mid R\ \}\ x{:}a\ \{\ Q \mid S\ \}$ from the calculus previously presented while the framed boolean distinguishes between $\vdash$ (when framed = false) and $\vdash_F$ (when framed = true).

F*'s effect system provides hooks to allow us to elaborate terms written in direct style, let x = e in e' to bind_M [[e]] (λx → [[e']]) when e and e' elaborate to [[e]] and [[e']] respectively, with a computation type whose head constructor is M (Rastogi et al. 2021). We rely on this feature to encode the different rules presented in Figure 7.4. For instance, we define a bind monadic combinator for SteelBase computations as follows, omitting selector predicates for readability:

```
val bind
  (_ : squash (if framed_f then frame_f == emp else ⊤))
  (_ : squash (if framed_g then frame_g == λ_ → emp else ⊤))
  (_ : squash (a_prop ⟹ (λ x → post_f x ∗ frame_f) ∗−∗ (λ x → pre_g x ∗ frame_g x))
  (_ : squash (b_prop ⟹ (λ x → post_g x y ∗ frame_g x) ∗−∗ post))
  (f: steelbase a framed_f pre_f post_f)
  (g: (x:a → steelbase b framed_g (pre_g x) (post_g x)))
  : steelbase b true (pre_f ∗ frame_f) post
```

This combinator captures both the Bind and the Frame rule from our calculus. The squashed arguments correspond to verification conditions that F* must discharge when applying bind. When composing computations f and g, this combinator adds frames frame_f and frame_g to f and g respectively, with the restriction that a frame is the empty assertion emp if its corresponding computation was already framed; i.e., if the computation has the $\vdash_F$ judgment. Building upon this effect, we finally define the user-facing Steel effect, hiding framing reasoning from a Steel programmer.

```
effect Steel (a:Type) (p:vprop) (q:a → vprop) (r:fppred p) (s:fppost p a q)
  = SteelBase a false p q r s
```

$$\begin{tikzcd}
\text{Steel} \\
\text{SteelAtomic} \arrow[u, dotted] \\
\text{SteelGhost} \arrow[u, dotted]
\end{tikzcd}$$

Steel also has two other kinds of computation types, for atomic computations and for ghost (proof-only) computation steps. We apply the same recipe to generate VCs for them, inserting frames at the leaves, and including consequence and framing in copies of Bind and Val used for these kinds of computations. Ultimately, we have three user-facing computation types, Steel, SteelAtomic, and SteelGhost. Behind the scenes, each of these relies on a computation type (e.g., SteelBase) which handles framing reasoning. These computation types are ordered in an effect hierarchy, enabling smooth interoperation between different kinds of computations; SteelAtomic computations are implicitly lifted to Steel computations when needed, while SteelGhost can be used transparently as either SteelAtomic or Steel.

7.2.5 An SMT-Friendly Encoding of Selectors

Soundness of Quintuples. We prove the soundness of our quintuples with selectors by reducing them to raw quintuples in SteelCore. In SteelCore quintuples $\{\ P \mid R\ \}\ x{:}t\ \{\ Q \mid S\ \}_{raw}$, presented in detail in Section 6.2.4, we have R:fp_prop P and S:fp_prop_2 P Q, capturing that R and S depend only on the P and Q fragments of the initial and final memories, respectively. Thus, every user annotation in SteelCore's raw quintuples comes with an obligation to show that the R and S terms depend only on their specified footprint—these relational proofs on specifications can be overwhelming, and require reasoning about disjoint and joined memories, breaking the abstractions that separation logic offers.

In comparison, selector predicates are self-framing by construction: the predicates R and S can only access the selectors of P and Q instead of the underlying memory, which are themselves framing. By defining selector predicates as a suitable abstraction on top of the SteelCore program logic, we thus hide the complexity of the self-framing property from both the user and the SMT solver, leading to more efficient verification.

A More Efficient Encoding of seleq. To preserve the modularity inherent to separation logic reasoning when using selector predicates, the postcondition of the Frame rule previously presented contains the proposition seleq ?F s_{f_0} s_{f_1}, capturing that sel ?F s_{f_0} = sel ?F s_{f_1}.

Using this predicate, the SMT solver can derive that the selector of any vprop contained in the frame is the same in the initial and final memories, leveraging the fact that, for any m:fpmem (p $*$ q), sel (p $*$ q) m = (sel p m, sel q m). But as the size of the frame grows, this becomes expensive; the SMT solver needs to deconstruct and reconstruct increasingly large tuples.

Instead, we encode seleq as the conjunciton of equalities on the *atomic* vprops selectors contained in the frame, where an *atomic* vprop does not contain a $*$. For instance, p and q are the atomic vprops contained in p $*$ q. Our observation is that most specifications pertain to atomic vprops; the swap function presented in the introduction for instance is specified using the selectors of ptr r1 and ptr r2, instead of (ptr r1 $*$ ptr r2).

Once the frame has been resolved using the approach presented in Section 7.3, generating these equalities is straightforward using metaprogramming; we can flatten the frame according to the star operator, and generate the conjunction of equalities to pass to the SMT solver.

Limitations of Selectors. Selectors can alleviate the need for existentially quantified ghost variables; the value stored in a reference for instance can be expressed as a selector, decluttering specifications. However, not all vprops have meaningful selectors, nor do we expect that they should. For example, when using constructions like PCMs to encode sharing disciplines, it is not always possible to define a selector that returns the partial knowledge of a resource. In Chapter 8, we will demonstrate the wide range of possible styles to write and verify Steel programs, especially illustrating that, when applicable, selectors can significantly simplify specifications and proofs.

7.3 Automatically Discharging Steel Verification Conditions

In this section, we show how to discharge separation logic VCs generated during the elaboration of Steel programs; namely, how to solve frame metavariables $?F$ and separation logic entailments $\models_{tac}$.

We start by presenting a quick overview of our methodology on the simple (though artificial) example below:

```
val write (r:ref a) (x:a) : Steel unit (ptr r) (λ _ → ptr r)

let two_writes (r1 r2:ref int)
  : Steel unit (ptr r1 ∗ ptr r2) (λ _ → ptr r1 ∗ ptr r2)
  = write r1 0; write r1 1
```

For clarity, we will omit the $\models_{smt}$ constraints when typechecking this program; they are irrelevant to this example.

First, the App rule from Figure 7.4 is applied to both writes. The function applications are then sequentially composed using the Bind rule. This rule requires $\models_F$ computations as premises; frames ?F1 and ?F2 are automatically inserted by applying the Frame rule to each application. When composing sequentially, a fresh metavariable ?Q is generated, as well as the two constraints ptr r1 $*$?F1 $*\!\!-\!\!*$ ptr r2 $*$?F2 and ptr r2 $*$?F2 $*\!\!-\!\!*$?Q. Finally, the Val rule ensures that the inferred type matches the user-provided signature, generating the constraints ptr r1 $*$ ptr r2 $*\!\!-\!\!*$ ptr r1 $*$?F1 and ?Q $*\!\!-\!\!*$ ptr r1 $*$ ptr r2.

Determining whether two terms containing an arbitrary number of metavariables can be unified up to associative-commutative rewriting is a hard problem (Fages 1984). Constraints for Steel programs fit this description, using only one AC-function: the separation logic $*$. Our observation is that we can instead reduce constraint solving in Steel to a simpler problem, namely, AC-matching.

By solving constraints in a particular order, it is possible to only consider constraints that contain at most one metavariable. This happens when considering constraints generated by a linear traversal, in either forward or backward program order. In our example, going forward, we would first solve ?F1 in ptr r1 $*$ ptr r2 $*\!\!-\!\!*$ ptr r1 $*$?F1, followed by ?F2 in ptr r1 $*$?F1 $*\!\!-\!\!*$ ptr r2 $*$?F2 (?F1 having been solved previously), and finally ?Q through ptr r2 $*$?F2 $*\!\!-\!\!*$?Q. Once we reach the last constraint ?Q $*\!\!-\!\!*$ ptr r1 $*$ ptr r2, the metavariable ?Q has already been solved, and checking AC-equivalence is straightforward.

We first formalize this intuition in Section 7.3.1, proving that the rules presented in Figure 7.4 ensure that a scheduling suitable for AC-matching exists for any well-typed

Steel program. We then present our approach to AC-matching in the context of our auto-active prover. Compared to complete, but expensive algorithms for AC-matching previously proposed (Kapur and Narendran 1987), our algorithm is efficient, providing quick feedback to the programmer when unification fails. In exchange, it is incomplete; working within a proof assistant, when results are inconclusive, the programmer can provide hints and annotations to help the unifier make progress.

7.3.1 A Unitriangular AC-Matching Problem

In this section, we show that the constraints generated in Figure 7.4 can be split into a *unitriangular* set and an unrestricted set of equations. This property ensures that a scheduler can always pick an AC-matching constraint, while guaranteeing progress and termination.

For notational purposes, we write the metavariables ?`F`, ?`Q` etc. as variables ?`u` in this section. We also write $\Gamma \vdash e : C \mid U; \mathcal{X}$ to mean that a typing judgment $\Gamma \vdash e : C$ from Figure 7.4 generates the set of metavariables U and the set of $\vDash_{tac}$ constraints over them denoted by $\mathcal{X}$ (similarly for $\vdash_F$ judgments).

We begin by defining a unitriangular system of equations. Such systems of equations can be represented using unitriangular matrices, i.e., triangular matrices where all elements in the main diagonal are 1.

Definition 7.1 (Unitriangular system of equations)**.** *An ordered set of metavariables* $U = \{?u_i\}_{i\in[1,n]}$ *and an ordered set of equations* $\mathcal{X} = \{\mathcal{X}_i\}_{i\in[1,n]}$ *form a unitriangular system of equations if*

1. $\forall i \in [1, n]$*.* $?u_i$ *occurs exactly once in* $\mathcal{X}_i$*, and*
2. $\forall i, j \in [1, n]$*.* $?u_j$ *does not occur in* $\mathcal{X}_i$ *if* $j > i$*.*

Our main theorem is then as follows:

Theorem 7.2. *If* $\Gamma \vdash e : \{\ P \mid R\ \}\ z{:}t\ \{\ Q \mid S\ \} \mid U; \mathcal{X}$ *then* $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$ *and there exists an ordering of* U *and* $\mathcal{X}_1$ *such that* $(U, \mathcal{X}_1)$ *is unitriangular.*

An important observation is that two unitriangular systems of equations $(U, \mathcal{X})$ and $(U', \mathcal{X}')$ where $U \cap U' = \varnothing$ can be concatenated to form a single unitriangular system of equations. Borrowing terminology from the linear algebra community, this concatenation corresponds to creating a block diagonal matrix where the two blocks on the diagonal are $\mathcal{X}$ and $\mathcal{X}'$, which are unitriangular.

To prove Theorem 7.2, we need to reason about the metavariables and constraints generated by the frame computations. To that end, we work with a notion of *once-removed*-unitriangular system of equations. Intuitively, given a unitriangular system of equations, we can obtain a once-removed-unitriangular system of equations by removing the first constraint.

Definition 7.3 (Once-removed-unitriangular system of equations)**.** *An ordered set of metavariables* $U = \{?u_i\}_{i\in[1,n]}$ *and an ordered set of equations* $\mathcal{X} = \{\mathcal{X}_i\}_{i\in[2,n]}$ *form a once-removed-unitriangular system of equations if*

1. *$\forall i \in [2, n]$. $?u_i$ occurs exactly once in $\mathcal{X}_i$, and*

2. *$\forall i \in [1, n], j \in [2, n]$. $?u_i$ does not occur in $\mathcal{X}_j$ if $i > j$.*

With this definition in hand, we prove the following theorem about the framed computations.

Theorem 7.4. *If $\Gamma \vdash_F e : \{ \ P \mid R \ \} \ z{:}t \ \{ \ Q \mid S \ \} \mid U; \mathcal{X}$ then $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$ and there exists an ordering of U and $\mathcal{X}_1$ such that $(U, \mathcal{X}_1)$ is once-removed-unitriangular with exactly one occurrence of $?u_1$ in Q.*

The proofs for these theorems rely on the following auxiliary lemmas.

Lemma 7.5. *If $\Gamma \vdash e : \{ \ P \mid R \ \} \ z{:}t \ \{ \ Q \mid S \ \}$, then P and Q do not contain any metavariables.*

Lemma 7.6. *If $\Gamma \vdash_F e : \{ \ P \mid R \ \} \ z{:}t \ \{ \ Q \mid S \ \}$, then P and Q each contain exactly one occurrence of a metavariable.*

The proofs for Theorem 7.2 and Theorem 7.4 proceed by simultaneous induction on the two typing derivations. For readability, we omit here the complete proofs of the theorems and lemmas; they are presented fully in Appendix B. The intuition behind this proof is that we can construct the unitriangular system by traversing the derivation backwards, starting from the postcondition. The choice is arbitrary but convenient for the proof—one could also structure the proof to go forwards instead.

A Constraint Scheduler for Steel. The unitriangular shape of the set of equations allows us to solve the $?u_i$ sequentially while finally verifying that the solutions are consistent with the equations C_i. In practice, when typechecking a Steel program, we do not reorganize the generated constraints to extract a unitriangular system. We instead implement a simpler scheduling, solving the first remaining constraint which contains only one occurence of a metavariable. The existence of the unitriangular system ensures the progress and termination of this scheduling.

7.3.2 Solving AC-Matching Instances

In the previous sections, we showed how we could schedule equations to be solved sequentially, so that each scheduled equation contains at most one frame metavariable. In this section, we present the last missing piece of the puzzle: how to actually solve such an equation.

Consider below a simplified grammar for a subset of vprops.

$$v \ ::= \ c \mid ?v \qquad t \ ::= \ \mathsf{emp} \mid f \ v \mid ?u \mid t * t$$

We assume the existence of a set of constants c and uninterpreted function symbols f. Terms t can be the unit emp, a function f whose argument is either a constant of metavariable $?v$, a vprop-metavariable $?u$ (typically the frame variable), or a separating conjunction of two terms. The other connectives are uninterpreted by our AC-matching solver. Our main point of interest here is that we have two sorts of metavariables, $?v$ variables may arise due to implicit arguments in a program and, unlike the $?u$ variables, may have a type other than vprop.

When considering a scheduled equation, we will denote the frame metavariable $?u$, and assume without loss of generality that it is on the left-hand-side of the equality. Similarly to AC-matching algorithms in the literature, we consider flattened representations of both sides of the equation.

Our algorithm proceeds by trying to match each symbol on the left with a rigid head symbol (c or f) with a term on the right, and finally sets the metavariable $?u$ to the conjunction of the remaining terms on the right once matching is over, or the unit emp if no such term is left.

To fix the intuition, let us first present the simple case where all terms different from $?u$ are constants. If a constant on the left has no counterpart on the right, then the equation cannot be solved, raising a unification error. We present below some simple examples to illustrate—constants c_i, c_j are distinct and non-matchable.

- $c_1 * ?u = c_2 * c_1 * c_3$. The c_1 on both sides match and are removed. $?u$ is set to the conjunction of leftover terms on the right side, i.e. $c_2 * c_3$
- $c_1 * ?u = c_1$. The c_1 on both sides match and are removed. There is no leftover term on the right side, so $?u$ is set to the unit emp
- $c_1 * ?u = c_2$. A unification error is raised, since c_1 cannot be matched with any term on the right side.

Now consider the equation f ?v $*$ f v2 $*$?u $=$ f v2 $*$ f v1. By applying naively the matching algorithm presented previously, the first term on the left, f ?v, would be matched with the first term on the right, f v2. This would prevent the second term on the left, f v2, from being matched with the remaining terms on the right, which in turn would return a unification error to the programmer, pointing to f v2 not being matched, and leading to confusion since a matching term does exist on the right side.

A natural solution to this problem would be backtracking, attempting to match f ?v with a different term on the right side. As previously stated, this is a solution we wish to avoid; the cost of backtracking can become prohibitive, and hinder the interactivity required for program verification. We instead only match a left-hand-side term t_l if there is a *unique* term on the right that it can be unified with. If this is not the case, we delay the matching of t_l and attempt to match the rest of the terms on the left side. If no progress was made once we retry matching t_l, an error message prompts the programmer to instantiate more implicit arguments.

Again, we present below several examples illustrating the behaviour of our algorithm.

- f v1 $*$ f ?v $*$?u $=$ f v2 $*$ f v1. We first attempt to match f v1 with a term on the right. There is a unique solution, so matching is performed. We then attempt to match the second term, f ?v. Only f v2 is left on the right, so there is a unique solution and we can set ?u to emp.

- f ?v $*$ f v1 $*$?u $=$ f v2 $*$ f v1. We first attempt to match f ?v with a term on the right. Both terms on the right are valid solutions, so we delay this matching. We then attempt to match f v1, which has a unique solution. Matching on all terms on the left different from ?u has been attempted and progress has been made, we retry with the delayed terms. f ?v now has a unique solution on the right side. We finally set ?u to emp.

- f ?v $*$ f ?v' $*$?u $=$ f v1 $*$ f v2. Both f ?v and f ?v' have several solutions on the right side. Since no progress is made after attempting matching for terms on the left side, an error is raised.

7.3.3 Cooperating with the SMT Solver

Our AC-matching algorithm is entirely implemented as an F* tactic. It relies on the F* unifier to determine whether two terms can be matched, thus solving relevant metavariables when matching occurs. As in other systems, F* tactics are not trusted—the terms they generate are guaranteed to be well-typed, thus ensuring soundness of the decision procedure.

Since the decision procedure for AC-matching is not trusted, there is no need to restrict its complexity. As such, our decision procedure is designed to be easily extensible with additional heuristics and user customization. In this section, we present one such extension, which enables equality rewriting during AC-matching by querying the SMT solver.

Consider the simple case where we wish to solve the equation f b $=$ f true. Here, the unification procedure presented so far would fail even if the equality b $=$ true is valid, due to, say, a control flow hypothesis. The only solution would be for a programmer to manually trigger a rewrite by calling a ghost procedure.

Instead, we implement heuristic abduction of equalities (the ?a and ?b in $\vDash_{tac}$ in Figure 7.4) in our tactic that allows us to match f b $=$ f true via an SMT provable equality. This is powerful and allows an interplay between arbitrary theories and AC-matching, allowing, for example, our algorithm to match f (x - x) $=$ f 0 or f (2 $*$ x) $=$ f (x $+$ x), i.e., a kind of AC-matching modulo theories.

However, this is in tension with the basic structure of our AC-matching algorithm. If every $c_i = c_j$ is possible with theory reasoning, even the most basic steps of our matching algorithm will always fail, since there is no unique solution even to a simple problem like $c_i * ?u = c_i * c_j$. Hence, deciding which equalities to abduct and delegate to SMT requires program-specific knowledge, which we allow the programmer to configure.

When defining a separation logic predicate, the programmer can annotate some arguments to mark them as candidates for SMT-based rewritings. Consider for instance the standard separation logic predicate pts_to r v, which indicates that the reference r stores the value v. When reasoning about the functional correctness of a program using this predicate, equalities on the value v are common while equalities on the reference r itself are rarer. As such, a programmer could decide to mark values v as candidates for SMT-based rewriting, using the smt_rewrite attribute as follows, but not references r in pts_to predicates.

```
val pts_to (r:ref a) (#[@@ smt_rewrite] v:a) : vprop
```

Thus, unifying pts_to r v1 = pts_to r v2 would automatically succeed if the SMT solver can prove v1 = v2, but unifying pts_to r1 v = pts_to r2 v would require a manual rewrite when r1 = r2.

A library designer can make some of these choices once and for all, so that all clients benefit from smart equality abduction; our library for references with fractional permissions does indeed mark the value and the permission as abduction candidates.

When the AC-unifier cannot make progress anymore, ideally after having matched some of the left-hand-side terms that had a unique solution on the right-hand-side, it retries a similar unification while generating equalities to be discharged by SMT. These equalities propagate to the SMT solver using the $?a$ and $?b$ abduction variables from Figure 7.4. When falling back on the SMT solver is not necessary, these metavariables are set to $\top$, ensuring that no metavariable is left unsolved. Similarly to the main algorithm, solutions that are not unique are not accepted, so as to provide accurate error reporting to the programmer.

- pts_to r v1 $*$ pts_to r v2 = pts_to r v1 $*$ pts_to r v3. The decision procedure first attempts exact matching, and removes pts_to r v1 from both sides. It is then left with two terms that cannot be unified, and falls back on SMT rewritings. Since the value argument of pts_to has been marked as candidate for SMT-based rewriting, the AC-unifier queries the SMT solver to check whether v2 = v3.
- pts_to r v1 $*$ pts_to r v2 = pts_to r v3 $*$ pts_to r v4. No exact matching is possible, the AC-unifier falls back directly on SMT rewritings. Since both v1 and v2 could possibly be rewritten into v3 or v4, the unicity of the solution cannot be guaranteed. The AC-unifier fails, and asks the programmer to provide manual rewrites to disambiguate.

7.4 Summary

In this chapter, we have developed a full-fledged language embedded in F* for concurrent programming with semi-automated proofs, without extending the trust assumptions of the SteelCore logic we presented in Chapter 6 and while preserving its expressive power. By applying a proof-oriented mindset to the design of the language itself, we automatically separate verification conditions between separation logic predicates and selector predicates. This allows us to provide automation tailored to each type of predicate: we use a mixture of tactics and SMT to solve the former as AC-unification constraints modulo theories, while relying on standard SMT solving for the latter. In the next chapter, we put Steel to work, reaping the benefits of semi-automated proofs for a highly expressive program logic to implement a wide range of verified libraries.

Chapter 8

Working the Anvil: Smelting Verified Steel Libraries

To evaluate the usefulness of the proof-oriented methodology we advocate for, we present in this chapter verified libraries demonstrating the expressiveness and programmability of the Steel framework. Through different examples, we showcase the large variety of styles of programming and proving that Steel offers, ranging from Viper-style permission accounting with implicit dynamic frames, to more dependently typed libraries for various concurrency idioms, concurrent data structures, and message-passing protocols.

Contributions. The examples described in this chapter were presented in Swamy et al. (2020) and Fromherz et al. (2021). I was primarily involved in the implementation of the balanced trees (Section 8.2) and 2-party session types libraries (Section 8.7), and also with the design of the invariant for the Michael-Scott 2-lock queue (Section 8.6). I also reimplemented plain SteelCore versions of the synchronization primitives in Steel (Section 8.3), and assisted with the development and debugging of the other case studies presented in this chapter.

8.1 Notations and Basic Concepts

To start this chapter, we first summarize the key concepts and constructs presented in Chapters 6 and 7, but from a Steel programmer's perspective.

Three Kinds of Computations. As described in Section 7.2.4, Steel offers three computation types with the signatures below, where `fppred p = type_of p → prop` and `fppost p a q = type_of p → x:a → type_of (q x) → prop`:

```
Steel (a:Type) (p:vprop) (q:a → vprop) (r:fppred p) (s:fppost p a q)

SteelAtomic (a:Type) (i:inames) (p:vprop) (q:a → vprop) (r:fppred p) (s:fppost p a q)

SteelGhost (a:Type) (i:inames) (p:vprop) (q:a → vprop) (r:fppred p) (s:fppost p a q)
```

SteelAtomic and SteelGhost have the same signature and carry an additional index (i:inames) corresponding to the set of currently opened invariants (Section 6.3.4). SteelAtomic is used to classify computationally relevant code whose effects on memory are atomic; e.g., an atomic compare-and-set (CAS) instruction would have type SteelAtomic. SteelGhost describes code that has no observable computational effect; e.g., this could involve a proof step such as unrolling a recursive predicate, calling a lemma, or reading, writing, or allocating to ghost state. Steel is the general purpose computation type for Steel code, and involves a mixture of pure computations, multiple atomic steps composed in sequence or parallel, and ghost code—SteelAtomic and SteelGhost are implicitly lifted to Steel. Reflecting the Par node from the SteelCore action trees (Section 6.2.1), Steel provides parallel composition through the following combinator

```
par (f : unit → Steel a p q r s) (g: unit → Steel a' p' q' r' s')
  : Steel (a & a') (p ∗ p') (λ (x,x') → q x ∗ q' x')
    (requires λ(t,t') → r t ∧ r' t')
    (ensures λ(t,t') (x,x') (u,u') → s t x u ∧ s' t' x' u')
```

In case the r:fppred p or q:fppost p a q are trivial, we simply omit them; otherwise, as in par above, we tag the selector predicates with requires and ensures to improve readability.

Steel References. In its most basic form, the memory of a Steel program contains a map from abstract typed references pref (a:Type) (p:pcm a) to values of type a, where p:pcm a is some partial commutative monoid (PCM) over the carrier type a. By choosing suitable PCMs, Steel's libraries provide various flavors of derived reference types, the most commonly used of which are references with fractional permissions, with the signatures below.

- ref t, is a reference to a t-typed value
- f : frac is an erased, real-valued fraction between 0 and 1.
- $\mathsf{r} \overset{\mathsf{f}}{\mapsto} \mathsf{v}$: vprop asserts ownership of an f-fraction of r pointing to v, while $\mathsf{r} \mapsto \mathsf{v}$ is defined as $\mathsf{r} \overset{1.0}{\mapsto} \mathsf{v}$.
- ptr r f is equivalent to $\exists \mathsf{v}.\ \mathsf{r} \overset{\mathsf{f}}{\mapsto} \mathsf{v}$, with the selector type type_of (ptr r f) = t, when r:ref t.
- pure p is equivalent to emp in a context where the proposition p is valid.

Additionally, we have ghost references, ghost_ref (t:Type), which refer to erased t values in memory—both the references and the values they point to are computationally irrelevant. The vprop for ghost references is written $\mathsf{r} \overset{\mathsf{f}}{\dashrightarrow} \mathsf{v}$, but is otherwise identical to $\mapsto$. In Section 8.7 we show how to use references with other PCMs.

Steel Invariants. Any vprop can at some point in a computation be designated an *invariant*, and is from then on enforced by the logic to be maintained by all subsequent computation steps (Sections 6.3.3 and 6.3.4). The main constructs are shown below:

- inv (p:vprop) : Type, is the type of an invariant enforcing p. Note, inv p is a value that can be freely duplicated and shared among threads. It represents a kind of token witnessing the validity of p.
- Invariants are named, with name (i:inv p) : inv_name, and inames = set inv_name.
- new_inv (p:vprop) : SteelGhost (inv p) i p (λ_ → emp), consumes the initially valid p:vprop and returns a new token for it.
- Invariants can be opened and restored in atomic code using the following combinator, which states that f can assume p_i and restore it in an atomic step and return x, while also transforming p to q x. The index u:inames is used to ensure that f does not itself internally open i, which would be unsound. A similar combinator, with_inv_ghost, allows using and restoring an invariant in SteelGhost code.

  ```
  with_inv (i:inv pᵢ) (f: unit → SteelAtomic a u (pᵢ ∗ p) (λ x → pᵢ ∗ q x))
    : SteelAtomic a (name i ⊎ u) p q
  ```

8.2 Balanced Trees: Selectors at Work

As a first case study, we present a verified implementation of self-balancing AVL trees. To specify this implementation, our first step is to define a tree : vprop capturing the essence of a mutable tree. In the following code, Spec is the name of the F* module containing a standard, pure specification of binary trees, represented as an inductive datatype whose constructors are Leaf and Node data left right.

We begin by setting up the various types and representation invariants. The definitions of tree nodes and trees are mutually recursive: a tree node is a record containing a value data, as well as the left and right subtrees, while a tree is a pointer to a tree node.

```
type node (a: Type) = { data: a; left: t a; right: t a }

and t (a: Type) = ref (node a) (∗ The type of the binary linked trees ∗)

(∗ A recursive predicate for binary trees ∗)
let rec tree_interp' (ptr: t a) (n: Spec.tree (node a)) = match n with
  | Spec.Leaf → pure (ptr = null) (∗ Leaves are represented by null pointers ∗)
  | Spec.Node data left right →
      tree_interp' data.left left ∗ tree_interp' data.right right ∗ ptr ↦ data

(∗ We existentially quantify over the spec tree ∗)
let tree_interp (ptr:t a) = ∃n. tree_interp' ptr n
```

```
(* The selector only keeps the data in the nodes, returning a Spec.tree a *)
val tree_sel (ptr:t a) (m:fpmem (tree_interp ptr)) : Spec.tree a

(* We collect the components above to define the vprop tree, indexed by the root pointer *)
let tree (ptr:t a) : vprop = {
  interp = tree_interp ptr;
  type_of = Spec.tree a;
  sel = tree_sel ptr }
```

In particular, note that the interpretation of tree is an existentially quantified, recursive separation logic predicate. We expect to make ghost procedure calls to manipulate quantifiers and to roll and unroll recursive predicates. The signature of roll_tree is below.

```
val roll_tree (root: t a) (left: t a) (right: t a) : SteelGhost unit u
  (tree left ∗ tree right ∗ ptr ↦ root) (λ _ → tree root)
  (requires (λ s → s.[ptr].left == left ∧ s.[ptr].right == right))
  (ensures (λ s _ s' → s'.[ptr] == Spec.Node s.[ptr].data s.[left] s.[right]))
```

Proofs of such lemmas are a bit mechanical—we open the existentials for tree, instruct the F* normalizer to reduce the recursive function tree_interp and then fold it back to introduce tree_interp, then pack the existential, and return. In the future, we believe some of this boilerplate can be reduced through metaprogramming. Now with our roll and unroll lemmas in hand, we can turn to the code itself.

Using tree selectors, we can define concise specifications operating on pure F* trees. For example, the specification for height relates the returned value x to the height of the F* tree returned by tree_sel, while ensuring that the function did not modify the tree.

```
let rec height (ptr:t a)
  : Steel int (tree ptr) (λ _ → tree ptr)
      (requires λ_ → ⊤)
      (ensures λs x s' → s.[ptr] == s'.[ptr] ∧ Spec.height s.[ptr] == x)
  = if is_null ptr then (
      unroll_leaf ptr; 0
    ) else (
      let node = unroll_tree ptr in
      let hleft = height node.left in
      let hright = height node.right in
      roll_tree ptr node.left node.right;
      if hleft > hright then (hleft + 1) else (hright + 1)
    )
```

With the exception of three ghost calls that roll and unroll the definition of the vprop, the code is fairly canonical, and the proof is automated by the hybrid of tactics and SMT in about a second.

We require only five lemmas similar to roll_tree to obtain straightforward Steel implementations of our tree library operations: an unroll and roll operation for both the node and leaf cases, and a lemma node_is_not_null which establishes that a root pointer does not correspond to a leaf if the pointer is not null.

One of the main benefits of this approach is that tree can be used as a basis to define more involved notions of trees at a minimal cost. For instance, one can define mutable AVL trees as a pure predicate over the selector of a tree, without needing to modify the representation invariant. Using this predicate, we can specify a self-balancing insertion operation, insert_avl, which is parameterized by a comparison function cmp needed to perform a binary search. This operation builds on the same memory layout for the tree, but with a different logical layer over it (e.g., AVL balancing). The Spec.is_avl function checks that the tree meets our specification for an AVL; e.g., every subtree is balanced and the tree is a binary search tree.

The full Steel implementation of insert_avl, shown below, follows the flow of a textbook binary search tree insertion; it creates a new node if the tree is empty, or recursively inserts v in the correct subtree if not before finally rebalancing the tree. All verification conditions related to the shape of the tree are discharged automatically by SMT, while separation logic VCs only require minimal user interaction; calling stateful lemmas such as roll_tree in a few specific, predictable places is sufficient. The procedure is checked in about 6 seconds.

```
let rec insert_avl (cmp:Spec.cmp a) (ptr: t a) (v: a)
  : Steel (t a) (tree ptr) (λ ptr' → tree ptr')
    (requires λs → Spec.is_avl cmp s.[ptr])
    (ensures λs ptr' s' → Spec.insert_avl cmp s.[ptr] v == s'.[ptr'])
  = if is_null ptr then (
      unroll_leaf ptr; (* unroll the tree vprop *)
      let node = {data = v; left = ptr; right = null} in
      let new_tree = alloc node in
      (* roll the tree vprop and return tree *)
      roll_leaf ();
      roll_tree new_tree ptr null;
      new_tree
    ) else (
      let node = unroll_tree ptr in
      if cmp node.data v ≥ 0 then (
        let new_left = insert_avl cmp node.left v in
        let new_node = {data = node.data; left = new_left; right = node.right} in
        write ptr new_node;
        roll_tree ptr new_left node.right;
        rebalance_avl cmp ptr
      ) else (
        let new_right = insert_avl cmp node.right v in
        let new_node = {data = node.data; left = node.left; right = new_right} in
        write ptr new_node;
        roll_tree ptr node.left new_right;
        rebalance_avl cmp ptr)
    )
```

This verification style is not specific to self-balancing trees; we applied it to several other standard data structures such as singly and doubly linked lists. By splitting verification conditions between separation logic goals and selector predicates, we can thus reason separately about memory manipulations and about the logical meaning of these manipulations. Coupled with tactics and SMT automation, the readability of the implementations is greatly improved.

8.3 Synchronization Primitives

Moving on from sequential programming, we now build a few libraries of verified synchronization primitives in Steel. We start with a spin lock, built using invariants accessed by an atomic CAS instruction. Using a spin lock, we will then build a library for fork/join concurrency on top of structural parallelism (par) and general recursion.

Spin Locks. To implement a spin lock, we build on a primitive compare-and-set atomic action with the signature shown below. It states that given a reference r to a word-sized integer for which we have full permission, and old and new values, cas updates the reference to new if its current value is old, and otherwise leaves r unchanged. Note, cas takes an additional ghost parameter, v:erased uint32, which represents the value stored in the reference in the initial state.

```
val cas (r:ref uint32) (old new:uint32) (v:erased uint32) :
    SteelAtomic (b:bool{b=(v=old)}) u (r ↦ v) (λ b → r ↦ (if b then new else v))
```

A lock is represented as a pair of a reference and an invariant stating that the reference is in one of two states: either it holds the value available and the lock invariant p, a vprop, is true separately; or it holds the value locked.

```
let available = false

let locked = true

let lockinv (r:ref bool) (p:vprop) = (r ↦ available ∗ p) ∨ (r ↦ locked)

let lock_t = ref bool & inv_name

let protects (l:lock_t) (p:vprop) : prop = snd l ⇝lockinv (fst l) p

let lock p = l:lock_t { l `protects` p }
```

Using this representation, allocating a lock is then straightforward:

```
let new_lock p : Steel (lock p) p (λ _ → emp) =
    let r = alloc available in
    let i = new_inv (lockinv r p) in
    (| r, i |)
```

Releasing a lock requires opening the invariant to gain permission to the reference—we add comments to the code to show the relevant Hoare triples in the term using the notation {p} e {q}. Within the invariant, we use a ghost read to fetch the current value of the reference, then do a cas and can prove that it sets the reference to available.

In the case where the reference was already set to available, we use the affinity of our separation logic to forget the assertion if b then emp else p before closing the invariant. We could also return the resulting boolean to avoid losing information.

```
let release ((| r, i |):lock p) : Steel unit p (λ _ → emp) =
    let _ = with_inv i
(* {lockinv r p ∗ p} *)
    (* {((pts_to r 1.0 available ∗ p) ∨ pts_to r 1.0 locked) ∗ p} *)
        (let v = ghost_read r in
         cas r locked available v;
         drop (if b then emp else p))
    (* {λ b → pts_to r 1.0 available ∗ p} *)
(* {lockinv r p ∗ emp} *) in ()
```

Acquiring a lock is similar to releasing it: We try to set the lock reference to locked within the invariant using an atomic cas. If cas fails, we "spin" by repeatedly calling acquire until the lock becomes available. The function terminates once the reference has been set to locked and we successfully acquired the corresponding vprop.

```
let rec acquire ((| r, i |):lock p) : Steel unit emp (λ _ → p) =
  let b = with_inv i
    (* {lockinv r p} *)
    (let v = ghost_read r in cas r available locked v)
    (* {λ b → lockinv r p ∗ (if b then p else emp)} *)
  in
  if b then rewrite (if b then p else emp) p
  else (
    rewrite (if b then p else emp) emp;
    acquire (| r, i |)
  )
```

Fork/Join. Steel's only concurrency primitive is the par combinator for structured parallelism shown in Section 8.1. However, having just built a library for locks, we can code up a library for fork/join concurrency without too much trouble. As with locks, since F* is a higher-order language, we can easily abstract over computations and their specifications.

The interface we provide for forking and joining threads is shown below. The type thread p represents a handle to a thread which guarantees p upon termination. The combinator fork f g runs the thread f and continues with g in parallel, passing to g a handle to the thread running f. The join t combinator waits until the thread t completes and guarantees its postcondition.

```
val thread (p:vprop) : Type

val fork (f: (unit → Steel unit p (λ _ → q))) (g: (thread q → Steel unit r (λ _ → s)))
  : Steel unit (p ∗ r) (λ _ → s)

val join (t:thread p) : Steel unit emp (λ _ → p)
```

To implement this interface, we represent a thread handle as a boolean reference protected by a lock that guarantees the thread's postcondition p when the reference is set. Allocating a thread handle is easy, since the reference can initially be set to false.

```
let thread p = {
  r:ref bool;
  l:lock (∃ b. r ↦ b ∗ (if b then p else emp))
}

val new_thread (p:vprop) : Steel (thread p) emp (λ _ → emp)
```

To fork a thread, we create a new thread handle t, then in parallel, run g t and in the thread for f, we acquire the lock, run f (); then set the reference and release the lock.

```
let fork f g =
  let t = new_thread q in
  let _ = par
    (λ _ → acquire t.l; f(); write t.r true; release t.l)
    (λ _ → g t) in
  ()
```

Finally, to join, we acquire the lock and, if the reference is set, we can free the reference and return the postcondition p; otherwise, we release the lock and loop—F*'s existing support for general recursion makes it relatively easy.

```
let rec join (t:thread p) =
  acquire t.l;
  let b = !t.r in
  if b then free t.r
  else (
    release t.l;
    join t
  )
```

Note, to provide a C-style fork/join on top of our API requires a CPS-like transform, since fork expects separate continuations for the parent and child threads. To improve the usability of fork, one solution would be to layer another effect for continuations above the Steel effect to support fork/join in direct style.

8.4 A Library of Disposable Invariants

To illustrate how common proof idioms can be packaged as dependently typed libraries in Steel, we now present a library for disposable invariants, which are similar to Iris' cancellable invariants (Jung et al. 2018b). Disposable invariants, like invariants, package a vprop and provide a similar with_inv combinator to work with the vprop in the atomic code.

The main novelty of disposable invariants is that, similar to locks, they may be reclaimed, thereby returning the underlying vprop back to the context. But since disposable invariants are still computationally irrelevant, unlike locks, they do not have a computational overhead.

We present the library interface below. The main `vprop` provided by the library is `active perm i`, where `perm` is a permission over the disposable invariant `i`. `share` and `gather` may be used to split and collect the invariant permissions, while `dispose` enforces that the caller must have full permission over the invariant. We elide the `with_inv` combinator, its signature is similar to the signature shown in Section 8.1 with `active perm i` in the pre- and postcondition of the combinator.

```
val inv (p:vprop) : Type (* the type of disposable invariants *)

val name (i:inv p) : iname

val active (f:perm) (i:inv p) : vprop

val new_inv (p:vprop) : SteelGhost (inv p) _ p (λ i → active 1.0 i) (* consumes p *)

val share (i:inv p)
  : SteelGhost unit _ (active perm i) (λ _ → active perm/2 i ∗ active perm/2 i)

val gather (i:inv p)
  : SteelGhost unit _ (active perm0 i ∗ active perm1 i) (λ _ → active (perm0 + perm1) i)

val dispose (i:inv p{not (name i ∈ u)})
  : SteelGhost unit u (active 1.0 i) (λ _ → p) (* destroys i, recovers p *)
```

The implementation of the library packages a normal invariant with a `ghost_ref bool`. Depending on the value that the reference points to (`true` or `false` respectively), this invariant either encapsulates the underlying `p:vprop` or `emp`. Thus, a disposable invariant starts with the ghost reference pointing to `true`, while disposing it sets the value of the reference to `false`, returning the `vprop p` back to the context.

$$\texttt{let inv p} = \texttt{r:ghost\_ref bool}\ \&\ \texttt{Steel.Memory.inv}\ (\exists\ (\texttt{b:bool}).\ \texttt{r} \overset{0.5}{\dashrightarrow} \texttt{b} * (\texttt{if b then p else emp}))$$

$$\texttt{let active perm i} = \texttt{fst i} \overset{\texttt{perm/2}}{\dashrightarrow} \texttt{true}$$

8.5 Parallel Increment *à la* Owicki-Gries with Disposable Invariants

For our next case study, we present a Steel implementation of the well-known Owicki-Gries counter (Owicki and Gries 1976), using disposable invariants. In this example, the main thread spawns two worker threads, both of which increment a shared counter by 1. The goal is to prove in the main thread that once the worker threads finish, the value of the shared counter is incremented by 2. An interesting aspect of the problem is that, since the access to the shared counter is protected using a synchronization primitive (e.g., a spinlock), the threads do not even have read permission on the counter before they enter the critical section, and hence cannot provide a postcondition that relates the before and after values of the counter.

Owicki and Gries's solution is for each thread to use a ghost reference to track its contribution to the counter, with an invariant that the value of the shared counter is equal to the sum of the values of the two contribution variables. Since each ghost reference is incremented by 1, the main thread can now prove that the assertion about the counter value holds. The invariant is as follows:

```
let og' (ctr:ref int) (r1 r2:ghost_ref int) (w:erased int & erased int) =
  r1 $\overset{0.5}{\dashrightarrow}$ (fst w) ∗ r2 $\overset{0.5}{\dashrightarrow}$ (snd w) ∗ ctr ↦ (fst w + snd w)

let og ctr r1 r2 : vprop = ∃w. og' ctr r1 r2 w
```

Since the invariant needs to only be in place while the threads are active, this is a good candidate for disposable invariants, so long as the threads only use atomic instructions to increment the counter.

```
let incr_main (v:erased int) (ctr:ref int) : Steel unit (ctr ↦ v) (λ _ → ctr ↦ (v + 2)) =
  let r1 = ghost_alloc 0 in
  let r2 = ghost_alloc v in
  (∗ allocated the ghost refs:: r1 ⇢ 0 ∗ r2 ⇢ v ∗ ctr ↦ v ∗)

  ghost_share r1; ghost_share r2; (∗ split permissions ∗)
  intro_∃ (hide 0, v) (og' ctr r1 r2);
  (∗ r1 $\overset{0.5}{\dashrightarrow}$ 0 ∗ r2 $\overset{0.5}{\dashrightarrow}$ v ∗ og ctr r1 r2 ∗)

  let i = new_inv (og ctr r1 r2) in
  (∗ allocated the disposable invariant sealing og:: ... ∗ active 1.0 i ∗)

  share i;
  (∗ split the invariant permission:: active 0.5 i ∗ r1 $\overset{0.5}{\dashrightarrow}$ 0 ∗ active 0.5 i ∗ r2 $\overset{0.5}{\dashrightarrow}$ v ∗)

  let _ = par (∗ workers ∗)
    (incr_with_invariant ctr r1 r2 0 true i)
    (incr_with_invariant ctr r2 r1 v false i) in
  gather_invariant i;
  dispose i;
  (∗ disposed of the invariant:: r1 $\overset{0.5}{\dashrightarrow}$ 1 ∗ r2 $\overset{0.5}{\dashrightarrow}$ v + 1 ∗ og ctr r1 r2 ∗)

  let w = open_∃ () in
  (∗ r1 $\overset{0.5}{\dashrightarrow}$ 1 ∗ r2 $\overset{0.5}{\dashrightarrow}$ v + 1 ∗ og' ctr r1 r2 w ∗)

  ghost_gather (incr 0) r1; ghost_gather (incr v) r2;
  (∗ r1 ⇢ 1 ∗ r2 ⇢ v + 1 ∗ ctr ↦ v + 2 ∗)

  drop (r1 ⇢ 1 ∗ r2 ⇢ v + 1)
  (∗ dropped the ghost refs:: ctr ↦ v + 2 ∗)
```

The main thread creates the two ghost references and splits their permissions. It then creates the disposable invariant and spawns the two worker threads, passing them the invariant and the remaining half permission to their respective ghost reference. Once the worker threads finish, the main thread gathers the permissions of the invariant and the ghost references, and disposes them. With og back in the context, it is able to prove the required assertion about the counter.

Imperative Lemmas as Ghost Code. We show some of the relevant triples in comments to highlight how the use of ghost code manipulates ghost state as well as the logical context. Note, for instance, the use of intro_∃ and open_∃ to manipulate quantifiers. The drop function call at the end drops the frame, which in our case consists of the ghost_pts_to predicates for the two ghost references. Since our separation logic is affine, we can implement such a combinator in Steel. It is possible to restrict it so that drop is allowed only for certain predicates, e.g., those that describe ghost state only.

Finally, the worker threads open the invariant, thereby getting full permission to the counter and to their respective ghost reference. They increment the counter and the ghost reference, repackage the invariant, and return. Given an atomic operation to increment a reference (or a CAS), we implement the worker threads with the following signature.

```
let incr_with_invariant (ctr:ref int) (mine other:ghost_ref int) (n:erased int) (b:bool) (i:inv _)
  : Steel unit (active 0.5 i ∗ mine $\xrightarrow{0.5}$ n) (λ _ → active 0.5 i ∗ mine $\xrightarrow{0.5}$ (incr n))
  = with_inv i (incr ctr mine other n b (name i))
```

An alternate implementation of the atomic increment function using cas provided by the Steel standard library is straightforward. We also easily implement an Owicki-Gries counter using a spinlock instead of disposable invariants, similarly to Leino, Müller, and Smans (2009).

8.6 Michael-Scott 2-Lock Queues

In this section, we present a proof of safety of a more realistic concurrent data structure, a queue by Michael and Scott (1996) which enables enqueuers and dequeuers to proceed in parallel. We prove the main invariants of the algorithm, including that the queue is always connected and that the head and tail point to the first and last elements of the queue respectively.

The main idea of the data structure is illustrated by the diagram in Figure 8.1. A queue is implemented as a linked list that *always contains at least one element* (the last element cannot be dequeued) and a pair of pointers to the head and tail of the list. These head and tail pointers are each protected by a lock. Enqueuers take the tail lock, add a node at the end of the list, update the tail pointer, and release the lock. Dequeuers take the head lock, try to dequeue from the head of the list, and if successful, swing the head pointer to the next node, and release the lock.

The interesting case is when the queue has only one element in it. In this case, the head and tail pointers point to the same node. Enqueuing and dequeuing threads

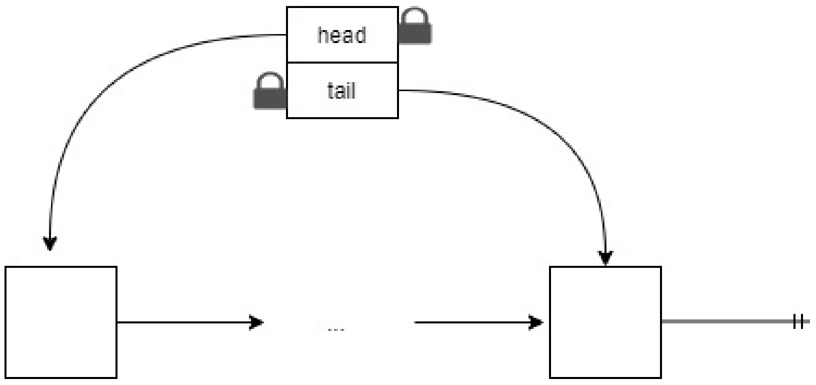


Figure 8.1: An overview of Michael and Scott's 2-lock queue

race on the next pointer of this node, with the enqueuer trying to update it while the dequeuer tries to read it. However, so long as reading or writing the next pointer is atomic, the algorithm correctly maintains the queue invariants.

To prove this in Steel, we follow a style similar in spirit to the Owicki-Gries parallel increment from the previous section, although this time we relate the invariants of the two locks with an atomic invariant on the queue itself by using two pieces of ghost state.

```
let lock_inv ptr ghost = ∃v. ptr ↦ v ∗ ghost ⇢^{0.5} v

let queue_invariant hd tl = ∃h t. hd.ghost ⇢^{0.5} h ∗ tl.ghost ⇢^{0.5} t ∗ Q.queue h t

type q_ptr a = {
  ptr : ref (Q.t a);
  ghost: ghost_ref (Q.t a);
  lock: lock (lock_inv ptr ghost) }

type t a = {
  head : q_ptr a;
  tail : q_ptr a;
  inv : inv (queue_invariant head tail) }
```

The type t a represents the structure of the two fields at the top of the picture. The head and tail pointers are q_ptrs, holding the concrete pointer ptr to a queue node Q.t a, a ghost pointer, and a lock relating the two. The queue itself bundles the head and tail q_ptrs with an invariant token inv.

The lock_inv holds full permission to the concrete pointer but only half the ghost pointer, while synchronizing them to hold the same value v. Meanwhile, queue_invariant holds the other half of the ghost pointers together with the invariant Q.queue h t, which states that we have a valid, non-empty linked list from h to t. These types and invariants drive the code that follows.

Creating a Queue. To allocate a new queue, we allocate the underlying linked list with Q.new_queue and an initial element x. Then, we allocate the two queue pointers for head and tail, introduce the queue invariant, package it, and return.

To define the function `new_queue`, the main proof effort involved is in allocating and sharing the ghost state, and introducing the existential quantifiers for the lock and queue invariants. This is mostly done inside the auxiliary `new_qptr` function, which allocates a concrete and a ghost pointer to a queue node `q:Q.t a` and relates them by creating a `lock: lock (lock_inv ptr ghost)`, finally returning a queue pointer alongside the remaining half share of the ghost pointer.

```
let new_queue (x:a) : Steel (t a) emp (λ _ → emp)
  = let new_qptr (q:Q.t a) : Steel (q_ptr a) emp (λ qp → qp.ghost -0.5-> q) =
        let ptr = alloc q in
        let ghost = ghost_alloc q in
        ghost_share ghost;
        intro_∃ _ (λ q → ptr ↦ q ∗ ghost -0.5-> q); (∗ need to introduce ∃, explicitly ∗)
        let lock = Steel.SpinLock.new_lock _ in
        { ptr; ghost; lock}
    in
    let hd = Q.new_queue x in
    let head = new_qptr hd in
    let tail = new_qptr hd in
    pack_queue_invariant _ _ head tail; (∗ need to package the invariant, 2 intro_∃ ∗)
    let inv = new_invariant _ _ in
    { head; tail; inv }
```

Enqueuing. The enqueue procedure below is also fairly clean.

```
let enqueue (hdl:t a) (x:a) : Steel unit emp (λ _ → emp)
  = Steel.SpinLock.acquire hdl.tail.lock;
    let v = open_∃ () in
    let tl = read hdl.tail.ptr in
    let cell = Q.({ data = x; next = null}) in
    let node = alloc cell in
    let enqueue_core #inames () : SteelAtomic unit inames
        (queue_invariant hdl.head hdl.tail ∗ (hdl.tail.ghost -0.5-> tl ∗ node ↦ cell))
        (λ _ → queue_invariant hdl.head hdl.tail ∗ hdl.tail.ghost -0.5-> node)
      = let h = open_∃ () in
        let t = open_∃ () in
        ghost_gather tl hdl.tail.ghost; (∗ fuse the two half permissions and get t=tl ∗)
        Q.enqueue tl node;
        ghost_write hdl.tail.ghost node; (∗ update the ghost state ∗)
        ghost_share hdl.tail.ghost;
        pack_queue_invariant _ _ _ _
    in
    with_inv hdl.inv enqueue_core;
    write hdl.tail.ptr node;
    intro_∃ _ (λ n → hdl.tail.ptr ↦ n ∗ hdl.tail.ghost -0.5-> n);
    Steel.SpinLock.release hdl.tail.lock
```

We start by acquiring a lock on the tail pointer. Then, we call a ghost computation from the library, `open_∃`, to return a witness for the existentially quantified `lock_inv` as an erased value. We then read the tail pointer and allocate a new cell with its next pointer properly initialized to null and ready for enqueuing.

The main work is done by the atomic computation `enqueue_core`, which opens and restores the queue invariant, by calling `Q.enqueue`, which itself is an atomic update of `tl→ next := node`, but the proof involves exploiting the synchronization of the ghost and concrete state, updating it and restoring the invariant. Once we exit the atomic block, we update the tail pointer, introduce the lock invariant's existential, and release the lock. The implementation of dequeue follows a similar approach.

Overall, with some carefully chosen types and invariants, the code mostly just writes itself, echoing Brady's type-define-refine slogan, but with Steel's CSL specifications. The proof overhead compares favorably with other automated F* developments—the framing is entirely automated, quantifier instantiation requires some manual intervention but the style we have here is very predictable, rather than relying on E-matching triggers for SMT. Yet, the interplay between SMT and tactics is profitable, with many small proofs done automatically behind the scenes. The whole procedure verifies in around 2 seconds including solving 25 SMT goals due to equality abduction in around 300 milliseconds.

8.7 PCMs for 2-Party Session Types

As a final example, we illustrate how Steel can be used to build dependently typed libraries that provide both a foundational semantics as well as usable abstractions for embedded session-typed programming. To this end, we propose a Steel library for duplex channels. Our work shares similarities with Actris (Hinrichsen et al. 2019), a full system that provides duplex channels with more features than we do here. But being a library in Iris, Hinrichsen, Bengtson, and Krebbers stop short of providing dependently typed libraries for programming. On the other hand, while Actris provides implementations of the channel API using low-level operational primitives, our Steel implementation is just a model of duplex channels showing that they can be realized by designing an appropriate PCM for 2-party dependent session types.

In summary, what is proved by our PCM model is that the channel is represented by a trace of messages, where the trace is a word in the language accepted by the protocol state machine; and that the participants' knowledge of the channel are mutually compatible and represents agreement on a prefix of the trace.

As much as demonstrating the applicability of Steel to message-passing programs, this example is meant to illustrate Steel's hybrid tactic/SMT-based automation at work with other features of Steel's logic, including user-defined PCMs. To set the goal posts, we offer the following interface for duplex channels.

```
val ch : Type

val ep (name:party) (c:ch) (p:prot) : vprop
val new (p:prot) : Steel (ch & ch) emp (λ (cA, cB) → ep A cA p ∗ ep B cB p)

let msg_t (p:prot) : Type = match hnf p with
  | Msg _ a _ → a
  | Return #a _ → a


val send (c:ch { is_send_next n next }) (x:msg_t next)
  : Steel unit (ep n c next) (λ _ → ep n c (step next x))

val recv (c:ch { is_recv_next n next })
  : Steel (msg_t next) (ep n c next) (λ x → ep n c (step next x))

val close (c:ch) : Steel unit (ep n c done) (λ _ → emp)
```

A channel is associated with a protocol p:prot via ep n c p, a vprop governing the use of one of the channel's named endpoints. A protocol is a free monad over basic actions to send and receive messages:

```
type tag = | Send | Recv

type prot : Type → Type =
| Return : #a:Type → v:a → prot a
| Msg : tag → a:Type → #b:Type → k:(a → prot b) → prot b
```

For example, a simple two-message protocol could be

```
let reply_larger : prot unit =
  Msg Send int (λ x → Msg Recv (y:int{y>x}) (λ _ → Return ()))
```

The type msg_t p computes the type of the next message of the protocol p, by reducing a protocol to its head-normal form and returning the type of the next Msg or Return node.

To allocate a channel, one calls new and obtains two separate endpoints A and B—B interprets the protocol dually to A, flipping sends and receives.

If an endpoint's protocol p is Msg Send t k, then send c x can be called with x:t, and the endpoint transitions to the next state of the protocol, step p x = k x. Dually, recv c blocks until it can return a x:t when the protocol is currently Msg Recv t k and the protocol continues as k x. For instance, the following code typechecks, since the protocol type guarantees that B must reply with a value larger than what it received from A.

```
let pingpong (c:ch) : Steel unit (ep A c reply_larger) (ep A c done) =
  send c 17;
  let y = recv c in
  assert (y > 17)
```

This interface to channels is simple, intuitive, and also quite powerful—protocols are monadic terms over the basic actions, and so support arbitrary dependence on the *values* exchanged, including branching for internal and external choice, and recursion. The question that remains is how to implement this interface.

Reminder on PCMs Our main insight is that one can design a PCM for protocols to orchestrate the temporal sharing of resources. As explained in Sections 6.3.1 and 8.1, Steel's memory model includes PCM references, pref a p, a reference to a value of type a, where p:pcm a. The vprop associated with a pref is r:pref a p $\rightsquigarrow$(v:a), where r $\rightsquigarrow$v $*$ r $\rightsquigarrow$u is equivalent to r $\rightsquigarrow$v $\oplus$ u, where $\oplus$ is the composition operator of the PCM p. Further, r $\rightsquigarrow$v is validated by a memory m only when there exists a frame f that is composable with v and m(r) = f $\oplus$ v. In other words, PCMs offer a form of rely-guarantee reasoning: a thread can rely on r $\rightsquigarrow$v being stable, but must in turn guarantee that its actions on r preserve other threads' assertions on r.

A PCM for Temporal Sharing of Protocol Endpoints. At the core of our model of 2-party sessions is a PCM on t p, a type that captures each participant's knowledge of the partial traces of a protocol p. A channel allocated with protocol p is a pref (t p) q, where q: pcm (t p) is to be defined shortly.

```
type t (p:prot) =
  | Nil (* unit of the PCM: no knowledge *)
  | V : partial_trace_of p → t p (* full knowledge *)
  | A_R : q:prot {is_recv q} → trace p q → t p (* A to receive next *)
  | A_W : q:prot {is_send q} → trace p q → t p (* A to send next *)
  | B_R : q:prot {is_send q} → trace p q → t p (* B to receive next *)
  | B_W : q:prot {is_recv q} → trace p q → t p (* B to send next *)
  | A_Fin : q:prot{is_ret q} → trace p q → t p (* A is finished *)
  | B_Fin : q:prot{is_ret q} → trace p q → t p (* B is finished *)
```

Each case in t p is intended to represent some knowledge of the state of the channel from the perspective of some participant. For example, given a channel reference c : pref (t p) q, the assertion c $\rightsquigarrow$A_W p tr is intended to model the knowledge that c is currently in a state where the protocol trace so far is tr and the next action on the trace is for A to send a message. To complete the construction, we now need to define which elements of t p are composable, and how to compose them.

```
let composable #p : symrel (t p) = λt0 t1 → match t0, t1 with
  | _, Nil | Nil, _ → ⊤(* unit composes with everything *)
  | V, _ | V, _ → ⊥(* V with nothing *)
  (* Both sides finished, traces agree *)
  | A_Fin q s, B_Fin q' s' | B_Fin q s, A_Fin q' s' → q == q' ∧ s == s'
  (* A is finished, B still has to read *)
  | A_Fin q s, B_R q' s' | B_R q' s', A_Fin q s → ahead A q q' s s'
  (* A is writing, B is reading: A is ahead *)
  | A_W q s, B_R q' s' | B_R q' s', A_W q s → ahead A q q' s s'
  (* Both reading, either ahead *)
  | A_R q s, B_R q' s' | B_R q' s', A_R q s → ahead A q q' s s' ∨ ahead B q' q s' s
  (* B is finished, A still has to read *)
  | A_R q' s', B_Fin q s | B_Fin q s, A_R q' s' → ahead B q q' s s'
  (* B is writing, A is reading: B is ahead *)
  | B_W q s, A_R q' s' | A_R q' s', B_W q s → ahead B q q' s s'
```

The composable (symmetric) relation captures that the reader's knowledge of the state of the channel is a prefix of the messages that have been written so far; while the writer's knowledge is the entire partial trace so far. When the protocol is finished, both participants agree on the entire trace.

The final step in defining our PCM is to compose the participants' knowledge of the traces—since the traces are composable, one of the traces is always a prefix of the other, and so composition takes the longer of the two traces.

```
let compose (s0:t p) (s1:t p{composable s0 s1}) = match s0, s1 with
  | a, Nil | Nil, a → a

  (* Just build V with the longer of the two traces *)
  | A_Fin q s, _
  | _, A_Fin q s
  | B_Fin q s, _
  | _, B_Fin q s → V (mk_trace q s)

  | A_W q s, B_R q' s'
  | B_R q' s', A_W q s
  | B_W q s, A_R q' s'
  | A_R q' s', B_W q s → V (mk_trace q s)

  | A_R q s, B_R q' s'
  | B_R q' s', A_R q s → if len s ≥ len s' then V (mk_trace q s) else V (mk_trace q' s')
```

Taking these definitions as the basis of q:pcm (t p), the essence of our 2-party session typed channels is done. With the knowledge that, say, c ⇝A_W p tr, an endpoint can only advance the channel to an extension of the trace tr. Conversely, with the knowledge that, say, c ⇝A_R p tr, an endpoint can rely on the fact that either the current value of the channel is already or will be extended to be ahead of the protocol state p, and the value expected by the receiver can be read from the trace.

Representing a Channel. This PCM-based rely-guarantee reasoning enables a fairly straightforward implementation of the main API we presented at the start, though with several levels of abstraction. We start with our representation of channels, the type ch below.

```
let chan p = ref (t p) (pcm p)

type ch = {
  p:prot;
  chan:chan p;
  tr: ref (until:prot & trace p until)
}
```

A channel ch is a triple of a protocol index p; a reference chan holding the current state of the channel; and a reference tr containing a partial trace of the messages exchanged on the channel so far, i.e., from the start state of p until the current state of the protocol, until. This trace will allow us to state our main invariant and will also serve, operationally, as the queue of messages transmitted so far and to be dispatched to the other endpoint.

The key representation predicate, ep name c next (defined below), states that there exists a partial trace (| until, tr |), such that **(1)** the protocol's next state is until; **(2)** c.chan carries knowledge (from the perspective of the given endpoint) that the protocol's current state is an extension of the trace tr; and, **(3)**, relating the two, the trace reference c.tr points to (| until, tr |).

```
let k_of name (next:prot) (tr:trace p next) = match name with
| A →
    if is_send next then A_W next tr else if is_recv next then A_R next tr else A_Fin next tr
| B →
    if is_send next then B_R next tr else if is_recv next then B_W next tr else B_Fin next tr

let ep name c next =
  ∃(| until, tr |). until == next ∗ c.chan ↦ k_of name next tr ∗ c.tr ↦ (|until, tr|)
```

Writing To and Reading From a Channel. Finally, the core of the top-level API to send and recv messages is implemented by the two functions shown below, write_a and read_a—similar functions exist for B. The top-level send and recv simply multiplex between these functions (and update the trace references) to present a single API for both participants.

To write a message on the channel, write_a expects the channel to be in the A_W next tr state. The main work here is calling a generic action for frame-preserving updates, upd_gen_action. Operationally, upd_gen_action c x y f reads the memory cell corresponding to c obtaining the current complete value for the channel v, which is provably equal to compose frame x, for some frame. The total function f updates v to v' = compose frame y, i.e., updating the local knowledge of the channel from x to y, while preserving frames.[1] In this case, write_a updates A's knowledge of the channel r from A_W next tr to v. The proof of write_a_f (the total, frame-preserving update function) is non-trivial and takes around 100 lines, but only involves pure reasoning about the PCM, and does not use any Steel-specific features.

```
let write_a (r:chan p { is_send next }) (tr:trace p next) (x:msg_t next)
  : Steel unit (r ↦ A_W next tr) (λ _ → r ↦ k_of A (step next x) (extend tr x))
  = let v = k_of A (step next x) (extend tr x) in
    upd_gen_action r (A_W next tr) v (write_a_f tr x)
```

It is worth repeating that while our library provides a semantic basis for dependent session types, and although it is executable, it is not a particularly realistic implementation of a channel in shared memory; e.g., it maintains the entire trace of interactions, and operations on the channel have to be protected by a lock.

In the future, we plan to integrate our channel API with the concurrent queue from Section 8.6, to only hold the messages that are yet to be delivered and to erase the

[1] upd_gen_action is expected to execute atomically. To execute it safely at runtime, one might use a lock to protect all accesses to memory cells that support this form of generic frame-preserving update.

traces by holding them in ghost references. Since with the session-typed API, write privilege on the channel is only ever held by one endpoint at a time, we speculate that, for this particular usage, we may actually be able to eliminate the use of locks in our two-lock queue.

More pragmatically, to focus on programming and proving distributed systems, we plan to use our session-typed channel API, proven here to be semantically justified in Steel, and to link it to a native implementation of sockets provided by the underlying execution platform.

8.8 Evaluating the Steel Automation

In the previous sections, we demonstrated the expressiveness of Steel on a large variety of verified libraries, using various styles of program proof. We now discuss how the automation we proposed in Chapter 7 impacts the usability and programmability of the Steel framework.

To this end, we implement several libraries both in Steel and in plain SteelCore, namely the spin locks and fork/join parallelism presented in Section 8.3, as well as a simpler version of our message passing library from Section 8.7 operating on unidirectionnal channels, also called *simplex channels*.

For comparison purposes, both our Steel and SteelCore implementations use the same specifications and only the proofs of those libraries differ; in particular, the Steel versions do not make use of the selector predicates presented in Section 7.2. As presented in Table 8.1, the improved automation in Steel shrinks the proofs dramatically; e.g., the Steel proof of simplex channels is several times shorter than the SteelCore equivalent, which, like `swap` from Chapter 7, is utterly overwhelmed by manual proof steps for framing and `vprop` rewriting.

Table 8.1: Comparison of the number of manual proof steps in SteelCore and Steel.

	SteelCore	Steel	Total file size (LoC, SteelCore implementation)
SpinLock	34	13	150
Fork/Join	33	9	130
Simplex	340	70	933

The Steel proofs use the same invariants as in SteelCore, but are thankfully significantly more maintainable. We find that our hybrid tactic- and SMT-based program verifier eliminates *all* mundane proof steps related to framing. Equality abduction in our tactics automatically delegates extensional conversions of `vprops` to SMT in many though not all cases. Finally, Steel is not fully automated and programmers must still perform some specific proof steps manually. For instance, they need to invoke lemmas to roll and unroll recursive `vprop` predicates and to trigger certain rewritings that equality abduction cannot handle, and also call ghost precedures to operate on the ghost state and maintain stateful invariants. We believe that the overhead is comparable to other SMT-based program verifiers, though, in comparison to prior developments in F* (verifying sequential imperative programs), Steel proofs are significantly more abstract due to the more powerful logic and the automated support for framing.

8.9 Discussion and Summary

In this chapter, we evaluated both the expressiveness of the SteelCore program logic presented in Chapter 6 and the programmability of the Steel framework relying on the automation presented in Chapter 7. Through several examples, we showcased the diversity of styles of program proof and the level of proof automation that Steel offers to users.

While we are not the first to attempt to automate separation logic reasoning, much of the prior work on automation has focused on *full automation*. Tools like Smallfoot (Berdine et al. 2005) or Cyclist (Brotherston et al. 2012) or the heap shape analysis proposed by Yang et al. (2008) aim to automate *fragments* of separation logic, including handling recursive predicates and quantifiers while aiming to scale lightweight, automated analyses and bug finding tools to large codebases. In contrast, aiming to co-develop programs and proofs, we focus on practical automation for user-assisted proofs of functional correctness, without restricting the expressiveness of our concurrent separation logic. In that sense, our work is closer to frameworks like RefinedC (Sammler et al. 2021), Viper (Müller et al. 2016) or VeriFast (Jacobs et al. 2011).

RefinedC is a framework to verify C programs, and is the work closest to us. Similarly to Steel, RefinedC is a foundational separation-logic-based framework: the soundness of the framework is established using Iris (Jung et al. 2018b), which is embedded in the Coq proof assistant. Additionally, RefinedC also provides practical automation for separation logic reasoning that does not require backtracking. To achieve this, Sammler et al. (2021) restrict their separation logic to a carefully chosen subset where predictable, goal-directed proof search is possible. Nevertheless, the logic is expressive enough to reason about a variety of C programming idioms including pointer arithmetic and concurrency with data races, and to verify a range of C programs such as memory allocators or efficient hashmap implementations. RefinedC's type system separates between ownership reasoning using separation logic and functional reasoning, which is expressed using pure side conditions discharged using Coq tactics. Steel adopts a similar methodology through our quintuples formalism, but with a small difference: Steel's selector predicates operate directly on heap fragments, often alleviating the need for auxiliary ghost variables to, for instance, specify which value is stored in a reference. While both Steel and RefinedC rely on their distinction between ownership and functional reasoning to provide practical separation logic automation, RefinedC's automation provides several interesting features that Steel does not currently possess. In particular, it is easily extensible with user-defined rewriting rules, allowing for instance to automatically roll and unroll certain recursive predicates, instead of manually calling lemmas such as `roll_tree` as we did when implementing balanced trees in Steel (Section 8.2).

Some of our specifications closely mimic Viper's implicit dynamic frames style, with an access permission and a heap-fragment refinement. For instance, in our verified implementation of balanced trees in Section 8.2, the validity of the `tree ptr` separation logic predicate represents permission to access the mutable tree at reference `ptr`, while the functional correctness of the data structure is specified using selector predicates. Similarly to Steel's selectors, specification assertions in Viper must be proven to be self-framing. However, heap predicates in Viper are based on accesses to references and object fields; self-framedness in Viper thus can be reduced to a permission to the memory locations that the predicate reads, which Viper can automatically determine. In contrast, Steel's selector predicates are more general than Viper's heap predicates. In exchange, Steel's definition of self-framedness is more complex, possibly leading to tedious proofs when defining a selector. Furthermore, compared to Viper, Steel also offers different verification styles beyond access permissions reasoning, for instance based on PCMs or invariants, which we use in several other examples. In a similar spirit to the quintuples we presented in Section 7.2, to verify C and Java programs, VeriFast automatically splits verification conditions between separation logic assertions and pure predicates during its symbolic execution. But while this automatic splitting is a nice feature, pure assertions in VeriFast do not depend on the heap, compared to Viper's heap-fragment refinements or to Steel's selector predicates.

Furthermore, while Viper, VeriFast and Steel all rely on an SMT solver to discharge verification conditions, another distinction with Viper and VeriFast is that Steel is built on top of SteelCore's foundational CSL in F*. As such, the TCB of the Steel framework corresponds exactly to that of F* itself. In particular, Steel leverages F*'s generic support for SMT solving—there is no specialized, trusted SMT encoding for Steel verification conditions. In this regard, Steel is closer to Low*, which we extensively used in Chapter 5 to implement the EverCrypt cryptographic provider. Low* and Steel both share a similar goal: reasoning about low-level programs. As a shallow embedding of a well-behaved subset of C into F*, Low* also relies directly on F*'s syntax, typing rules, and SMT encoding to ensure that programs are well-formed and to discharge verification conditions. But while Low* encodes a monolithic verification condition to SMT, Steel's proof-oriented design instead separates between separation logic and selector-based reasoning, enabling better automation for framing. By reducing the reliance on the SMT solver, and simplifying the queries that it receives, Steel proofs thus become more stable and deterministic than those in Low*. Nevertheless, while Steel provides a highly expressive separation logic and supports concurrency—which Low* does not—some program idioms are easier to specify and use in Low*. For instance, stating that two references *might alias*, i.e., that they are disjoint or equal, is straightforward in Low* and pervasive throughout our cryptographic code; in contrast, while this is expressible in separation logic using a disjunction, manipulating such a predicate involves eliminating and introducing a disjunction at each reference operation, which quickly becomes overwhelming. In the future, we aim to provide means to verifiably interoperate between Low* and Steel, to make reasoning about such forms of aliasing more palatable, while also enabling Steel programmers to make use of existing verified Low* projects, such as the EverCrypt library from Chapter 5.

Chapter 9

Conclusion

Motivated by the fast increasing reliance on software in critical settings, we advocate in this thesis for a proof-oriented programming paradigm to develop high-assurance software. Program proofs enable programmers to obtain strong, formal guarantees about the security and correctness of their programs. In this thesis, we show how co-developing programs and proofs simplifies, and thus increases the scalability of verification, but also how structuring programs with proofs in mind can simplify programming and improve the quality of software.

Using the F* proof assistant, we first develop EverCrypt, a verified, comprehensive, industrial-grade cryptographic provider. EverCrypt provides application developers with the functionality, ease of use, and speed they expect from existing cryptographic providers, while guaranteeing correctness and security of the code. Through abstraction and zero-cost generic programming, our methodology allows us to increase code and proof reuse. With EverCrypt, we thus demonstrate that verification does not need to preclude performance, and that verified software can meet the demands of real-world applications, leading to its deployment in high-profile, security-critical projects such as Mozilla Firefox or the Linux kernel.

Going beyond verified cryptography, and aiming to implement and verify low-level concurrent programs, we then present Steel, a verification framework based on a state-of-the-art concurrent separation logic shallowly embedded in F*. With Steel, we show how applying a proof-oriented mindset to the design of the framework can help structuring proof obligations, enabling practical, domain-specific automation. The result is a full-fledged, dependently typed language for concurrent programming with semi-automated proofs, with a high expressivity sufficient to reason about both concurrent, low-level data structures and richer programming idioms such as message-passing concurrency.

Together, these two case studies demonstrate the benefits of a proof-oriented programming style, and raise high hopes about increased adoption in industrial, high-assurance software development.

Appendix A

Secret Independence for Hybrid Low*/Vale programs

In this section, we restate and prove Theorem 4.2 presented in Section 4.2.3, establishing secret-independence for hybrid Low*/Vale programs.

Theorem (Secret-Independence for Hybrid Low*/Vale programs)**.** *Given configurations (H_1, e_1) and (H_2, e_2), where $\Gamma \vdash (H_1, e_1) : \tau$, $\Gamma \vdash (H_2, e_2) : \tau$, $H_1 \equiv_\Gamma H_2$ and $e_1 \equiv_\Gamma e_2$, and a secret-independent implementation of the secret integer interface P_s, either both the configurations cannot reduce further, or $\exists\, \Gamma' \supseteq \Gamma$ s.t. $P_s \vdash (H_1, e_1) \to^+_{\ell_1} (H'_1, e'_1)$, $P_s \vdash (H_2, e_2) \to^+_{\ell_2} (H'_2, e'_2)$, $\Gamma' \vdash (H'_1, e'_1) : \tau$, $\Gamma' \vdash (H'_2, e'_2) : \tau$, $\ell_1 = \ell_2$, $H'_1 \equiv_{\Gamma'} H'_2$, and $e'_1 \equiv_{\Gamma'} e'_2$,*

The proof of this theorem heavily relies on the proof proposed for Low* (Protzenko et al. 2017, Appendix F). We present below our additions to this proof to cover the interoperation between Vale and Low*.

Notations. To model Vale programs, we extend Low*'s formal syntax with an extern c expression form, that denotes the Vale code c embedded within Low*, and Low*'s trace language with a Vale trace z. Both the Vale code c and the Vale trace z are abstract elements, disjoint from the existing Low* formal syntax.

$$\begin{array}{rlll}
\text{Vale code} & c & & \\
\text{Expression} & e & ::= & \cdots \mid \mathsf{let}\ v = \mathsf{extern}\ c\ \mathsf{in}\ e \\
\text{Vale trace} & z & & \\
\text{Trace} & \ell & ::= & \cdots \mid z
\end{array}$$

Expression Typing. Low*'s typing judgements are of the shape $P; \Sigma; \Gamma \vdash e : t$, where P contains the signatures of the top-level functions in the context, Σ is the store typing, and Γ is the usual context of variables.

To enable typing of hybrid programs, we extend the Low* expression typing rules to support typing the extern expression. The first premise requires the Vale taint analyzer, here denoted as A_extern, to succeed, ensuring that the Vale procedure c is secret-independent. The second premise is standard, requiring for the expression e to be well-typed with the bound variable v in the context.

$$\frac{\mathsf{A\_extern}(\Sigma, \Gamma, c) = \mathsf{success} \quad P; \Sigma; \Gamma, v : \mathsf{uint64} \vdash e : t}{P; \Sigma; \Gamma \vdash \mathit{let}\ v = \mathsf{extern}\ c\ \mathsf{in}\ e} \text{ (T-EXTERN)}$$

Dynamic Semantics. Low* relies on small-step semantics, with judgments of the form $P \vdash (H, e) \longrightarrow_l (H', e')$, modeling that under the program P, the configuration (H, e) steps to $(H'e')$, emitting a trace of observations l, with H and H' being stacks of frames.

We extend Low*'s dynamic semantics to support the reduction of extern expressions. The premise corresponds to a Vale evaluation, using its definitional interpreter eval_code. This rule therefore models a lifting of Vale's semantics to Low*, executing a Vale program atomically.

$$\frac{H, c \overset{Vale}{\longrightarrow}_z H'}{P \vdash (H, \mathit{let}\ v = \mathsf{extern}\ c\ \mathsf{in}\ e) \longrightarrow_z (H', e)} \text{ (EXTERN)}$$

Proof of Theorem. The original Low* proof relates configurations using templates of the form (H, e). The relation $(H_1, e_1) \equiv_{(\Sigma,\Gamma)} (H_2, e_2)$ holds if and only if there exists a template (H, e) and two substitutions p_1 and p_2 of secret variables, where $H_i = H[p_i]$, $e_i = e[p_i]$, and the substitutions p_1 and p_2 are well-typed in the context Σ, Γ.

Using this formalism, the secret-independence theorem is then reformulated as the following bisimulation:

Theorem (Secret-independent Low* traces)**.** *If $P; \Sigma; \Gamma \vdash (H, e) : t$, $\Sigma; \Gamma \vdash p_1$ and $\Sigma; \Gamma \vdash p_2$, then either e is a value or there exists $m > 0, n > 0$, $\Sigma' \supseteq \Sigma$, $\Gamma' \supseteq \Gamma$, H', e', as well as two substitutions $p'_1 \supseteq p_1$ and $p'_2 \supseteq p_2$ such that*

a) $\Sigma'; \Gamma' \vdash p'_1$

b) $\Sigma'; \Gamma' \vdash p'_2$

c) $P \vdash (H, e)[p_1] \longrightarrow^m_{tr} (H', e')[p'_1]$

d) $P \vdash (H, e)[p_2] \longrightarrow^n_{tr} (H', e')[p'_2]$

The Low* proof of this theorem proceeds by induction on the expression e. To extend this proof to hybrid Vale/Low* programs, we thus need to provide a proof for the external Vale calls.

To this end, we rely on the following assumption on Vale executions, which models that Vale programs accepted by the static taint analyzer are secret-independent. The contexts Σ and Γ, as well as the secret substitutions p_1 and p_2 remain the same, encapsulating that Vale executions do not add new variable bindings, nor allocate memory.

Proposition (Assumption on external Vale code). *If*

1. $\mathsf{A\_extern}(\Sigma, \Gamma, c) = \mathsf{success}$
2. $\Sigma; \Gamma \vdash p_1$
3. $\Sigma; \Gamma \vdash p_2$
4. $P; \Sigma; \Gamma \vdash H$

then there exists H' such that

a) $H[p_1], c[p_1] \xrightarrow{Vale}_z H'[p_1]$

b) $H[p_2], c[p_2] \xrightarrow{Vale}_z H'[p_2]$

c) $P, \Sigma, \Gamma \vdash H'$

By relying on this proposition, proving the case $e = \mathsf{let}\ v = \mathsf{extern}\ c\ \mathsf{in}\ e'$ is now straightforward:

By inverting the typing rule [$\mathsf{T{-}Extern}$], we obtain $\mathsf{A\_extern}(\Sigma, \Gamma, c) = \mathsf{success}$ and $P; \Sigma; \Gamma \vdash e : t$.

Applying the assumption about external Vale code evaluation, we obtain

1. $H[p_1], c[p_1] \xrightarrow{Vale}_z H'[p_1]$
2. $H[p_2], c[p_2] \xrightarrow{Vale}_z H'[p_2]$
3. $P, \Sigma, \Gamma \vdash H'$

Clauses a) and b) from the theorem are trivial from hypotheses by choosing $\Sigma' = \Sigma$ and $\Gamma' = \Gamma$, since the substitutions p_1 and p_2 remained identical. We conclude by deriving clauses c) and d) from rule [Extern] applied to 1. and 2., choosing $m = n = 1$.

Appendix B

A Unitriangular System of Equations for Steel

In this section, we restate and prove the theorems presented in Section 7.3.1, establishing that any well-typed Steel computation yields a unitriangular system of equations.

Theorem 7.2. *If* $\Gamma \vdash e : \{\ P \mid R\ \}\ z{:}t\ \{\ Q \mid S\ \} \mid U; \mathcal{X}$ *then* $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$ *and there exists an ordering of* U *and* $\mathcal{X}_1$ *such that* $(U, \mathcal{X}_1)$ *is unitriangular.*

Theorem 7.4. *If* $\Gamma \vdash_F e : \{\ P \mid R\ \}\ z{:}t\ \{\ Q \mid S\ \} \mid U; \mathcal{X}$ *then* $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$ *and there exists an ordering of* U *and* $\mathcal{X}_1$ *such that* $(U, \mathcal{X}_1)$ *is once-removed-unitriangular with exactly one occurrence of* $?u_1$ *in* Q*.*

As stated in Section 7.3.1, the proofs of these theorems rely on the following lemmas, which we will prove first.

Lemma 7.5. *If* $\Gamma \vdash e : \{\ P \mid R\ \}\ z{:}t\ \{\ Q \mid S\ \}$*, then* P *and* Q *do not contain any metavariables.*

Lemma 7.6. *If* $\Gamma \vdash_F e : \{\ P \mid R\ \}\ z{:}t\ \{\ Q \mid S\ \}$*, then* P *and* Q *each contain exactly one occurrence of a metavariable.*

Proof of Lemmas 7.5 and 7.6. We prove these two lemmas by simultaneous induction on the typing derivations $\Gamma \vdash e : \{\ P \mid R\ \}\ z{:}t\ \{\ Q \mid S\ \}$ and $\Gamma \vdash_F e : \{\ P \mid R\ \}\ z{:}t\ \{\ Q \mid S\ \}$.

App. The App case is trivial by induction hypothesis.

Frame. By induction hypothesis, P and Q do not contain any metavariables. Hence, $P * ?F$ and $Q * ?F$ contain each exactly one occurence of a metavariable, which is $?F$.

Bind. By induction hypothesis on e_1, P_1 contains exactly one occurence of a metavariable. Furthermore, the postcondition of the conclusion of Bind is $?Q$, which is a metavariable.

Val. The predicates P and Q are an annotation provided by the user, they thus do not contain any metavariables. □

Building upon these lemmas, we can now formally establish the two theorems previously stated.

Proof of Theorems 7.2 and 7.4. Similarly to the proof of the lemmas above, we prove theorems 7.2 and 7.4 by simultaneous induction on the two typing derivations.

App. The induction hypothesis on e and f gives us two unitriangular systems of constraints, $(U, \mathcal{X})$ and $(U', \mathcal{X})$. These two systems can be concatenated to form a single unitriangular system of equations, which conclues this case.

Frame. The induction hypothesis gives us a unitriangular system of constraints $(\{?u_1, \ldots, ?u_n\}, \{\mathcal{X}_1, \ldots, \mathcal{X}_n\})$. The rule generates a new metavariable $?F$, which we rename as $?u_0$, and no additional constraint. The system $\{?u_0, ?u_1, \ldots, ?u_n\}$ and $\{\mathcal{X}_1, \ldots, \mathcal{X}_n\}$ thus is once-removed-triangular. Furthermore, by Lemma 7.5, Q does not contain any metavariable. Thus, the postcondition $Q * ?u_0$ contains exactly one occurence of $?u_0$.

Bind. Using the induction hypothesis on e_2, we get a once-removed-unitriangular system of constraints, $(\{?u_1, \ldots, ?u_n\}, \{\mathcal{X}_2, \ldots, \mathcal{X}_n\})$, such that Q_2 contains exactly one occurrence of $?u_1$. Similarly, the induction hypothesis on e_1 gives us another once-removed-unitriangular system of constraints, $(\{?u_{n+1}, \ldots, ?u_{n+k}\}, \{\mathcal{X}_{n+2}, \ldots, \mathcal{X}_{n+k}\})$, such that Q_1 contains exactly one occurrence of $?u_{n+1}$. The rule generates a new metavariable $?u_0$ for $?Q$ and two new constraints $Q_2 \mathrel{-\!\!*} ?u_0$ and $Q_1[x/y] \mathrel{-\!\!*} P_2$. It is now easy to see that the metavariables $\{?u_0, ?u_1, \ldots, ?u_n, ?u_{n+1}, \ldots, ?u_{n+k}\}$ and equations $\{(Q_2 \mathrel{*\!\!-\!\!*} ?u_0), \mathcal{X}_2, \ldots, \mathcal{X}_n, (Q_1[x/y] \mathrel{*\!\!-\!\!*} P_2), \mathcal{X}_{n+2}, \ldots, \mathcal{X}_{n+k}\}$ form a once-removed-unitriangular system of equations. For the remaining constraints from e_1 and e_2, we return their concatenation.

Val. The induction hypothesis gives us a once-removed-unitriangular system of equations $(U, \mathcal{X}_1)$ with remaining constraints $\mathcal{X}_2$ and Q' containing exactly one occurrence of $?u_1$. The rule adds two more constraints, $Q' \mathrel{*\!\!-\!\!*} Q$ and $P \mathrel{*\!\!-\!\!*} P'$, while the set of metavariables U remains the same. By lemma 7.5, P and Q do not contain any metavariables, and hence we conclude with $(U, \{Q' \mathrel{*\!\!-\!\!*} Q, \mathcal{X}_1\})$ as the unitriangular system of equations, and with $\{P \mathrel{*\!\!-\!\!*} P', \mathcal{X}_2\}$ as the remaining constraints. □

Bibliography

[Abel 2006] Andreas Abel. “A polymorphic lambda-calculus with sized higher-order types”. PhD thesis. Ludwig-Maximilians-Universität München, 2006.

[Abrial et al. 1991] Jean-Raymond Abrial, Matthew K.O. Lee, David Neilson, Peter N. Scharbach, and Ib Holm Sørensen. “The B-method”. In: *Proceedings of the International symposium of VDM Europe*. 1991.

[Aciiçmez et al. 2010] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. “New Results on Instruction Cache Attacks”. In: *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. 2010.

[ACM FORUM 1979] ACM FORUM. “Comments on social processes and proofs”. In: *Communications of the ACM* 22.11 (1979).

[Ahman et al. 2017] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. “Dijkstra Monads for Free”. In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2017.

[Ahman et al. 2018] Danel Ahman, Cédric Fournet, Cătălin Hriţcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. “Recalling a Witness: Foundations and Applications of Monotonic State”. In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2018.

[Ahmed et al. 2018] Amal Ahmed, Deepak Garg, Catalin Hritcu, and Frank Piessens. “Secure Compilation (Dagstuhl Seminar 18201)”. In: *Dagstuhl Reports* 8.5 (2018), pp. 1–30.

[Almeida et al. 2016] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. “Verifying Constant-Time Implementations”. In: *Proceedings of the USENIX Security Symposium*. 2016.

[Almeida et al. 2017] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. “Jasmin: High-Assurance and High-Speed Cryptography”. In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2017.

[Almeida et al. 2020] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. "The Last Mile: High-Assurance and High-Speed Cryptographic Implementations". In: *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*. 2020.

[Appel and Blazy 2007] Andrew W. Appel and Sandrine Blazy. "Separation Logic for Small-step Cminor". In: *Proceedings of the International Conference on Theorem Proving in Higher-Order Logics (TPHOL)*. 2007.

[Appel 2011] Andrew W. Appel. "Verified Software Toolchain". In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2011.

[Appel et al. 2014] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[Appel 2015] Andrew W. Appel. "Verification of a Cryptographic Primitive: SHA-256". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)*. 2015.

[Archinoff et al. 1990] G.H. Archinoff, R.J. Hohendorf, A. Wassyng, .B Quigley, and M.R. Borsch. "Verification of the shutdown system software at the Darlington nuclear generating station". In: *Proceedings of the International Conference on Control & Instrumentation in Nuclear Installations*. 1990.

[Astrauskas et al. 2019] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. "Leveraging Rust Types for Modular Specification and Verification". In: *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 2019.

[Atkey 2009] Robert Atkey. "Parameterised notions of computation". In: *Journal of functional programming* 19.3-4 (2009).

[Aumasson] Jean-Philippe Aumasson. *Guidelines for low-level cryptography software*. https://github.com/veorq/cryptocoding.

[Bach Poulsen et al. 2018] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. "Intrinsically-Typed Definitional Interpreters for Imperative Languages". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2018.

[Barnes and Bhargavan 2020] Richard Barnes and Karthik Bhargavan. *Hybrid Public Key Encryption. IRTF Internet-Draft*. https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hpke-06. 2020.

[Barnett et al. 2006] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. "Boogie: A modular reusable verifier for object-oriented programs". In: *Proceedings of Formal Methods for Components and Objects (FMCO)*. 2006.

[Barthe et al. 2011] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. "Computer-Aided Security Proofs for the Working Cryptographer". In: *Proceedings of the International Cryptology Conference (CRYPTO)*. 2011.

[Barthe et al. 2014] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. "System-level Non-interference for Constant-time Cryptography". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2014.

[Barthe et al. 2018] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. "Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"". In: *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*. 2018.

[Barthe et al. 2020] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. "Formal Verification of a Constant-Time Preserving C Compiler". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2020.

[Barthe et al. 2021] Gilles Barthe, Sunjay Cauligi, Benjamin Gregoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. "High-Assurance Cryptography Software in the Spectre Era". In: *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*. 2021.

[Bengtson et al. 2012] Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. "Charge! A Framework for Higher-Order Separation Logic in Coq". In: *Proceedings of the International Conference on Interactive Theorem Proving (ITP)*. 2012.

[Berdine et al. 2005] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. "Smallfoot: Modular Automatic Assertion Checking with Separation Logic". In: *Proceedings of Formal Methods for Components and Objects (FMCO)*. 2005.

[Berdine et al. 2011] Josh Berdine, Byron Cook, and Samin Ishtiaq. "SLAyer: Memory Safety for Systems-level Code". In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 2011.

[Beringer et al. 2015] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. "Verified Correctness and Security of OpenSSL HMAC". In: *Proceedings of the USENIX Security Symposium*. 2015.

[Bernstein 2006] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *Proceedings of the IACR Conference on Practice and Theory of Public Key Cryptography (PKC)*. 2006.

[Bernstein et al. 2011] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. “High-Speed High-Security Signatures”. In: *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. 2011.

[Bernstein et al. 2014] Daniel J. Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. “TweetNaCl: A crypto library in 100 tweets”. In: *Proceedings of the International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*. 2014.

[Beurdouche et al. 2015] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. “A Messy State of the Union: Taming the Composite State Machines of TLS”. In: *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*. 2015.

[Biryukov et al. 2016] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. “Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications”. In: *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 2016.

[Blanchet et al. 2003] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “A static analyzer for large safety-critical software”. In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2003.

[Blazy and Leroy 2009] Sandrine Blazy and Xavier Leroy. “Mechanized Semantics for the Clight Subset of the C Language”. In: *Journal of Automated Reasoning* 43.3 (2009), pp. 263–288.

[Blazy et al. 2017] Sandrine Blazy, David Pichardie, and Alix Trieu. “Verifying Constant-Time Implementations by Abstract Interpretation”. In: *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. 2017.

[Bond et al. 2017] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. “Vale: Verifying High-Performance Cryptographic Assembly Code”. In: *Proceedings of the USENIX Security Symposium*. 2017.

[Bornat et al. 2005] Richard Bornat, Cristiano Calcagno, Peter O’Hearn, and Matthew Parkinson. “Permission Accounting in Separation Logic”. In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2005.

[Bosamiya et al. 2020] Jay Bosamiya, Sydney Gibson, Yao Li, Bryan Parno, and Chris Hawblitzel. “Verified Transformations and Hoare Logic: Beautiful Proofs for Ugly Assembly Language”. In: *Proceedings of the International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*. 2020.

[Bourgeat et al. 2020] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. “The Essence of Bluespec: A Core Language for Rule-Based Hardware Design”. In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2020.

[Bowen and Hinchey 1995] Jonathan P. Bowen and Michael G. Hinchey. “Seven more myths of formal methods”. In: *IEEE software* 12.4 (1995), pp. 34–41.

[Boyland 2003] John Boyland. “Checking Interference with Fractional Permissions”. In: *Proceedings of the International Conference on Static Analysis (SAS)*. 2003.

[Brady 2013] Edwin Brady. “Idris, a general-purpose dependently typed programming language: Design and implementation”. In: *Journal of Functional Programming* 23 (05 Sept. 2013), pp. 552–593.

[Brady 2016] Edwin Brady. *Type-driven Development With Idris*. Manning, 2016. URL: http://www.worldcat.org/isbn/9781617293023.

[Brookes 2007] Stephen Brookes. “A Semantics for Concurrent Separation Logic”. In: *Theoretical Computer Science (TCS)*. 2007.

[Brotherston and Kanovich 2010] James Brotherston and Max Kanovich. “Undecidability of Propositional Separation Logic and Its Neighbours”. In: *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*. 2010.

[Brotherston et al. 2011] James Brotherston, Dino Distefano, and Rasmus Lerchedahl Petersen. “Automated Cyclic Entailment Proofs in Separation Logic”. In: *Proceedings of the International Conference on Automated Deduction (CADE)*. 2011.

[Brotherston et al. 2012] James Brotherston, Nikos Gorogiannis, and Rasmus Petersen. “A Generic Cyclic Theorem Prover”. In: *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*. 2012.

[Brumley and Boneh 2003] David Brumley and Dan Boneh. “Remote Timing Attacks Are Practical”. In: *Proceedings of the USENIX Security Symposium*. 2003.

[Brumley et al. 2012] Billy B. Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. “Practical Realisation and Elimination of an ECC-Related Software Bug Attack”. In: *Topics in Cryptology – CT-RSA*. 2012.

[Buisse et al. 2011] Alexandre Buisse, Lars Birkedal, and Kristian Støvring. “Step-indexed Kripke model of separation logic for storable locks”. In: *Electronic Notes in Theoretical Computer Science* 276 (2011).

[Butler et al. 2010] Ricky W. Butler, George Hagen, Jeffrey Maddalon, César A. Munoz, Anthony Narkawicz, and Gilles Dowek. “How Formal Methods Impels Discovery: A Short History of an Air Traffic Management Project”. In: *Proceedings of the International Conference on NASA Formal Methods (NFM)*. 2010.

[Calcagno and Distefano 2011] Cristiano Calcagno and Dino Distefano. "Infer: An Automatic Program Verifier for Memory Safety of C Programs". In: *Proceedings of the International Conference on NASA Formal Methods (NFM)*. 2011.

[Calcagno et al. 2007] Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis. "Modular Safety Checking for Fine-Grained Concurrency". In: *Proceedings of the International Conference on Static Analysis (SAS)*. 2007.

[Calcagno et al. 2009] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. "Compositional Shape Analysis by Means of Bi-Abduction". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2009.

[Calcagno et al. 2015] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. "Moving Fast with Software Verification". In: *Proceedings of the International Conference on NASA Formal Methods (NFM)*. 2015.

[Cauligi et al. 2020] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. "Constant-time foundations for the new Spectre era". In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2020.

[Chajed et al. 2019] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. "Verifying Concurrent, Crash-Safe Systems with Perennial". In: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 2019.

[Chang et al. 2007] Bor-yuh Evan Chang, Xavier Rival, and George C. Necula. "Shape analysis with structural invariant checkers". In: *Proceedings of the International Conference on Static Analysis (SAS)*. 2007.

[Charguéraud and Pottier 2017] Arthur Charguéraud and François Pottier. "Temporary Read-Only Permissions for Separation Logic". In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2017.

[Charguéraud 2011] Arthur Charguéraud. "Characteristic Formulae for the Verification of Imperative Programs". In: *Proceedings of the International Conference on Functional Programming (ICFP)*. 2011.

[Checkoway et al. 2011] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. "Comprehensive experimental analyses of automotive attack surfaces." In: *Proceedings of the USENIX Security Symposium*. 2011.

[Chen et al. 2014] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. “Verifying Curve25519 Software”. In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2014.

[Chen et al. 2015] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. “Using Crash Hoare Logic for Certifying the FSCQ File System”. In: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 2015.

[Chlipala 2013] Adam Chlipala. “The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier”. In: *Proceedings of the International Conference on Functional Programming (ICFP)*. 2013.

[Chou 2016] Tung Chou. “Sandy2x: New Curve25519 Speed Records”. In: *Proceedings of the International Conference on Selected Areas in Cryptography (SAC)*. 2016.

[Clarke et al. 2004] Edmund Clarke, Daniel Kroening, and Flavio Lerda. “A tool for checking ANSI-C programs”. In: *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2004.

[Clint and Hoare 1972] Maurice Clint and CAR Hoare. “Program proving: Jumps and functions”. In: *Acta informatica* 1.3 (1972), pp. 214–224.

[Cohen et al. 2009] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. “VCC: A Practical System for Verifying Concurrent C”. In: *Proceedings of the International Conference on Theorem Proving in Higher-Order Logics (TPHOL)*. 2009.

[Coq Development Team 2017] Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.7*. Oct. 2017. URL: http://coq.inria.fr.

[Cousot and Cousot 1977] Patrick Cousot and Radhia Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 1977.

[Cousot et al. 2005] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “The Astrée analyzer”. In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2005.

[Cremers et al. 2016] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. “Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication”. In: *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*. 2016.

[Damgard 1989] Ivan Damgard. “A Design Principle for Hash Functions”. In: *Proceedings of the International Cryptology Conference (CRYPTO)*. 1989.

[De Millo et al. 1979] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. “Social Processes and Proofs of Theorems and Programs”. In: *Communications of the ACM* 22.5 (1979), pp. 271–280.

[Debian Bug Tracker 2016] Debian Bug Tracker. *libssl1.0.0: illegal instruction crash on amd64*. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=793557#132. 2016.

[Delignat-Lavaud et al. 2021] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. “A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer”. In: *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*. 2021.

[Denis 2013] Frank Denis. *The Sodium cryptography library*. 2013. URL: https://download.libsodium.org/doc/.

[Dettman 2018] Peter Dettman. *Problems with field arithmetic*. https://github.com/armfazh/rfc7748_precomputed/issues/5. 2018.

[Dijkstra 1972] Edsger W. Dijkstra. “The Humble Programmer”. In: *Communications of the ACM* 15.10 (1972), pp. 859–866.

[Dinsdale-Young et al. 2010] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. “Concurrent Abstract Predicates”. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 2010.

[Dinsdale-Young et al. 2013] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. “Views: Compositional Reasoning for Concurrent Programs”. In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2013.

[Dinsdale-Young et al. 2017] Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. “Caper: Automatic Verification for Fine-grained Concurrency”. In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2017.

[Distefano and Parkinson 2008] Dino Distefano and Matthew Parkinson. “JStar: Towards Practical Verification for Java”. In: *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 2008.

[Dockins et al. 2016] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. “Constructing Semantic Models of Programs with the Software Analysis Workbench”. In: *Proceedings of the International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*. 2016.

[Dodds et al. 2009] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. “Deny-Guarantee Reasoning”. In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2009.

[Dodds et al. 2016] Mike Dodds, Suresh Jagannathan, Matthew J Parkinson, Kasper Svendsen, and Lars Birkedal. “Verifying custom synchronization constructs using higher-order separation logic”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)*. 2016.

[Donenfeld 2017] Jason A. Donenfeld. “WireGuard: Next Generation Kernel Network Tunnel”. In: 2017.

[Donenfeld 2018a] Jason A. Donenfeld. *kBench9000 - simple kernel land cycle counter*. https://git.zx2c4.com/kbench9000/about/. 2018.

[Donenfeld 2018b] Jason A. Donenfeld. *new 25519 measurements of formally verified implementations*. http://moderncrypto.org/mail-archive/curves/2018/000972.html. 2018.

[Dowek et al. 2005] Gilles Dowek, César Munoz, and Victor Carreño. “Provably safe coordinated strategy for distributed conflict resolution”. In: *Proceedings of the AIAA guidance, navigation, and control conference and exhibit*. 2005.

[Doychev et al. 2015] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. “CacheAudit: A Tool for the Static Analysis of Cache Side Channels”. In: *ACM Transactions on Information and System Security (TISSEC)*. 2015.

[Düll et al. 2015] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. “High-Speed Curve25519 on 8-Bit, 16-Bit, and 32-Bit Microcontrollers”. In: *Design, Codes, Cryptography* 77.2–3 (2015), pp. 493–514.

[Dumitrescu 2020] Victor Dumitrescu. *HACL* v2 integration (#589) · Issues · Tezos / tezos*. https://gitlab.com/tezos/tezos/-/issues/589. 2020.

[Durumeric et al. 2014] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. “The Matter of Heartbleed”. In: *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*. 2014.

[Ebner et al. 2017] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. “A Metaprogramming Framework for Formal Verification”. In: *Proceedings of the International Conference on Functional Programming (ICFP)*. 2017.

[Edelman 1997] Alan Edelman. “The Mathematics of the Pentium Division Bug”. In: *Society for Industrial and Applied Mathematics (SIAM) Review* 39 (1997), pp. 54–67.

[Eilers and Müller 2018] Marco Eilers and Peter Müller. “Nagini: A Static Verifier for Python”. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 2018.

[Erbsen et al. 2019] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. “Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises”. In: *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*. 2019.

[Fages 1984] François Fages. “Associative-commutative unification”. In: *Proceedings of the International Conference on Automated Deduction (CADE)*. 1984.

[Fankhauser et al. 2000] George Fankhauser, Christian Conrad, Eckart Zitzler, and Bernhard Plattner. “Topsy - a teachable operating system”. In: 2000.

[Fisher et al. 2017] Kathleen Fisher, John Launchbury, and Raymond Richards. “The HACMS program: using formal methods to eliminate exploitable bugs”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375.2104 (2017).

[Floyd 1967] Robert W Floyd. “Assigning meanings to programs”. In: *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*. 1967.

[Foley and Hoare 1971] Michael Foley and Charles Antony Richard Hoare. “Proof of a recursive program: Quicksort”. In: *The Computer Journal* 14.4 (1971), pp. 391–395.

[Fox and Myreen 2010] Anthony Fox and Magnus O. Myreen. “A trustworthy monadic formalization of the ARMv7 instruction set architecture”. In: *Proceedings of the International Conference on Interactive Theorem Proving (ITP)*. 2010.

[Fragoso Santos et al. 2017] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūniene, Thomas Wood, and Philippa Gardner. “JaVerT: JavaScript Verification Toolchain”. In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2017.

[Franklin et al. 2007] Jason Franklin, Vern Paxson, Adrian Perrig, and Stefan Savage. “An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants”. In:

Proceedings of the ACM Conference on Computer and Communications Security (CCS). 2007.

[Fromherz et al. 2018] Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. “Static value analysis of Python programs by abstract interpretation”. In: *Proceedings of the International Conference on NASA Formal Methods (NFM)*. 2018.

[Fromherz et al. 2019] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. “A Verified, Efficient Embedding of a Verifiable Assembly Language”. In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2019.

[Fromherz et al. 2021] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. “Steel: Proof-Oriented Programming in a Dependently Typed Concurrent Separation Logic”. In: *Proceedings of the International Conference on Functional Programming (ICFP)*. 2021.

[Gandolfi et al. 2001] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. “Electromagnetic Analysis: Concrete Results”. In: *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. 2001.

[Gleissenthall et al. 2019] Klaus von Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. “IODINE: Verifying Constant-Time Execution of Hardware”. In: *Proceedings of the USENIX Security Symposium*. 2019.

[Goguen and Meseguer 1982] Joseph A. Goguen and José Meseguer. “Security Policies and Security Models”. In: *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*. 1982.

[Gonthier et al. 2016] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. “A small scale reflection extension for the Coq system”. PhD thesis. Inria Saclay Ile de France, 2016.

[Gordon et al. 2013] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. “Rely-Guarantee References for Refinement Types over Aliased Mutable Data”. In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2013.

[Gotsman et al. 2007] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. “Local reasoning for storable locks and threads”. In: *Proceedings of the Asian Conference on Programming Languages and Systems (ASPLAS)*. 2007.

[Grossman et al. 2002] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. “Region-Based Memory Management in Cyclone". In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2002.

[Gueron and Krasnov 2014] Shay Gueron and Vlad Krasnov. "The Fragility of AES-GCM Authentication Algorithm". In: *Proceedings of the Conference on Information Technology: New Generations (ITNG)*. 2014.

[Gueron 2012] Shay Gueron. *Intel® Advanced Encryption Standard (AES) New Instructions Set*. https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf. 2012.

[Guiho and Hennebert 1990] Gerard Guiho and Claude Hennebert. "SACEM software validation". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. 1990.

[Gulley et al. 2013] Sean Gulley, Vinodh Gopal, Kirk Yap, Wajdi Feghali, Jim Guilford, and Gil Wolrich. *Intel® SHA Extensions*. https://software.intel.com/sites/default/files/article/402097/intel-sha-extensions-white-paper.pdf. 2013.

[Gurfinkel et al. 2015] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. "The SeaHorn verification framework". In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 2015.

[Haas et al. 2017a] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. "Bringing the Web up to Speed with WebAssembly". In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2017.

[Haas et al. 2017b] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. "Bringing the web up to speed with WebAssembly". In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2017.

[Hall 1990] Anthony Hall. "Seven myths of formal methods". In: *IEEE software* 7.5 (1990), pp. 11–19.

[Hancock and Setzer 2000] Peter Hancock and Anton Setzer. "Interactive programs in dependent type theory". In: *Proceedings of the International Workshop on Computer Science Logic (CSL)*. 2000.

[Harrison 2000] John Harrison. "Formal verification of IA-64 division algorithms". In: *Proceedings of the International Conference on Theorem Proving in Higher-Order Logics (TPHOL)*. 2000.

[Hawblitzel et al. 2014] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. "Ironclad Apps: End-to-End Security via Automated Full-System Verification". In: *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014.

[Hernandez-Castro et al. 2020] Julio Hernandez-Castro, Anna Cartwright, and Edward Cartwright. "An economic analysis of ransomware and its welfare consequences". In: *Royal Society Open Science* 7 (Mar. 2020), p. 190023.

[Hinrichsen et al. 2019] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. "Actris: Session-Type Based Reasoning in Separation Logic". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2019.

[Hoare 1969] Charles Antony Richard Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[Hoare 1976] Charles Antony Richard Hoare. "Parallel programming: an axiomatic approach". In: *Language Hierarchies and Interfaces*. Springer, 1976, pp. 11–42.

[Hobor and Villard 2013] Aquinas Hobor and Jules Villard. "The ramifications of sharing in data structures". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2013.

[Hobor et al. 2008] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. "Oracle Semantics for Concurrent Separation Logic". In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2008.

[Hoffmann 1978] Christoph M Hoffmann. "Design and correctness of a compiler for a non-procedural language". In: *Acta Informatica* 9.3 (1978), pp. 217–241.

[Honda et al. 1998] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. "Language primitives and type discipline for structured communication-based programming". In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 1998.

[Iosif et al. 2013] Radu Iosif, Adam Rogalewicz, and Jiri Simacek. "The Tree Width of Separation Logic with Recursive Definitions". In: *Proceedings of the International Conference on Automated Deduction (CADE)*. 2013.

[Iosif et al. 2014] Radu Iosif, Adam Rogalewicz, and Tomáš Vojnar. "Deciding entailments in inductive separation logic with tree automata". In: *Proceedings of the International Symposium on Automated Technology for Verification and Analysis (ATVA)*. 2014.

[Ishtiaq and OHearn 2001] Samin S. Ishtiaq and Peter W. O'Hearn. "BI as an Assertion Language for Mutable Data Structures". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2001.

[Jacobs and Beurdouche 2020] Kevin Jacobs and Benjamin Beurdouche. *Performance Improvements via Formally-Verified Cryptography in Firefox*. https://blog.mozilla.org/security/2020/07/06/performance-improvements-via-formally-verified-cryptography-in-firefox/. 2020.

[Jacobs and Piessens 2011] Bart Jacobs and Frank Piessens. "Expressive Modular Fine-Grained Concurrency Specification". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2011.

[Jacobs et al. 2011] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *Proceedings of the International Conference on NASA Formal Methods (NFM)*. 2011.

[Jensen and Birkedal 2012] Jonas Braband Jensen and Lars Birkedal. "Fictional separation logic". In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2012.

[Jones 1983] Cliff Jones. "Specification and Design of (Parallel) Programs." In: *Proceedings of the IFIP Congress*. 1983.

[Jourdan et al. 2015] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. "A Formally-Verified C Static Analyzer". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2015.

[Jung et al. 2015] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2015.

[Jung et al. 2016] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. "Higher-Order Ghost State". In: *Proceedings of the International Conference on Functional Programming (ICFP)*. 2016.

[Jung et al. 2018a] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the Foundations of the Rust Programming Language". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2018.

[Jung et al. 2018b] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28 (2018).

[Kapur and Narendran 1987] Deepak Kapur and Paliath Narendran. "Matching, unification and complexity". In: *ACM SIGSAM Bulletin* 21.4 (1987), pp. 6–9.

[Kassios 2006] Ioannis T. Kassios. “Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions”. In: *Proceeding of the International Symposium on Formal Methods (FM)*. 2006.

[Kästner et al. 2018] Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. “CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler”. In: *Proceedings of the European Congress on Embedded Real Time Software and Systems (ERTS)*. 2018.

[Kaufmann et al. 2016] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. “When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015”. In: *International Conference on Cryptology and Network Security (CANS)*. 2016.

[Kennedy et al. 2013] Andrew Kennedy, Nick Benton, Jonas B. Jensen, and Pierre-Evariste Dagand. “Coq: The World’s Best Macro Assembler?” In: *Proceedings of the Symposium on Principles and Practice of Declarative Programming (PPDP)*. 2013.

[Kern and Greenstreet 1999] Christoph Kern and Mark R. Greenstreet. “Formal Verification in Hardware Design: A Survey”. In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)*. 1999.

[King 1976] James C King. “Symbolic execution and program testing”. In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[Kiselyov and Ishii 2015] Oleg Evgenievich Kiselyov and Hiromi Ishii. “Freer monads, more extensible effects”. In: *Proceedings of the ACM SIGPLAN Symposium on Haskell*. 2015.

[Klein et al. 2009] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. “seL4: Formal Verification of an OS Kernel”. In: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 2009.

[Klein et al. 2018] Gerwin Klein, June Andronick, Matthew Fernandez, Ihor Kuz, Toby Murray, and Gernot Heiser. “Formally verified software in the real world”. In: *Communications of the ACM* 61.10 (2018), pp. 68–77.

[Kocher 1996] Paul C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *Proceedings of the International Cryptology Conference (CRYPTO)*. 1996.

[Kocher et al. 2019] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas

Prescher, Michael Schwarz, and Yuval Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*. 2019.

[Kolanski and Klein 2009] Rafal Kolanski and Gerwin Klein. “Types, Maps and Separation Logic”. In: *Proceedings of the International Conference on Theorem Proving in Higher-Order Logics (TPHOL)*. 2009.

[Koscher et al. 2010] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. “Experimental Security Analysis of a Modern Automobile”. In: *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*. 2010.

[Krawczyk and Eronen 2010] Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. 2010.

[Krebbers and Wiedijk 2011] Robbert Krebbers and Freek Wiedijk. “A Formalization of the C99 Standard in HOL, Isabelle and Coq”. In: *Proceedings of the International Conference on Intelligent Computer Mathematics (CICM)*. 2011.

[Krebbers et al. 2017a] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. “The Essence of Higher-Order Concurrent Separation Logic”. In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2017.

[Krebbers et al. 2017b] Robbert Krebbers, Amin Timany, and Lars Birkedal. “Interactive Proofs in Higher-Order Concurrent Separation Logic”. In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2017.

[Krogh-Jespersen et al. 2020] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. “Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems”. In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2020.

[Lammich and Meis 2012] Peter Lammich and Rene Meis. “A Separation Logic Framework for Imperative HOL”. In: *Archive of Formal Proofs* (2012).

[Lammich 2016] Peter Lammich. “Refinement Based Verification of Imperative Data Structures”. In: *Proceedings of the International Conference on Certified Programs and Proofs (CPP)*. 2016.

[Langley 2008] Adam Langley. *curve25519-donna: Implementations of a fast Elliptic-curve Diffie-Hellman primitive*. https://github.com/agl/curve25519-donna. 2008.

[Langley 2014] Adam Langley. *A shallow survey of formal methods for C code*. https://www.imperialviolet.org/2014/09/07/provers.html. 2014.

[Langley et al. 2016] Adam Langley, Mike Hamburg, and Sean Turner. *Elliptic Curves for Security*. https://tools.ietf.org/html/rfc7748. 2016.

[Lattner and Adve 2004] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 2004.

[Le et al. 2017] Quang Loc Le, Makoto Tatsuta, Jun Sun, and Wei-Ngan Chin. "A Decidable Fragment in Separation Logic with Inductive Predicates and Arithmetic". In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 2017.

[Leino and Moskal 2010] K. Rustan M. Leino and Michał Moskal. "Usable Auto-Active Verification". In: *Usable Verification Workshop*. 2010.

[Leino et al. 2009] K. Rustan M. Leino, Peter Müller, and Jan Smans. "Verification of Concurrent Programs with Chalice". In: *Foundations of Security Analysis and Design V*. 2009.

[Leino 2010] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 2010.

[Leroy 2006] Xavier Leroy. "Formal certification of a compiler back-end, or: programming a compiler with a proof assistant". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2006.

[Leroy et al. 2016] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. "CompCert – A Formally Verified Optimizing Compiler". In: *Proceedings of the European Congress on Embedded Real Time Software and Systems (ERTS)*. 2016.

[Leurent and Peyrin 2020] Gaëtan Leurent and Thomas Peyrin. "SHA-1 is a Shambles: First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust". In: *Proceedings of the USENIX Security Symposium*. 2020.

[Lewis and Martin 2003] Jeffrey R. Lewis and Brad Martin. "CRYPTOL: high assurance, retargetable crypto development and validation". In: *Proceedings of the IEEE Military Communications Conference (MILCOM)*. 2003.

[libjc 2019] libjc. *Combined AEAD ChaCha/Poly*. https://github.com/tfaoliveira/libjc/issues/2. 2019.

[Lions et al. 1996] Jacques-Louis Lions, Lennart Luebeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. *Ariane 5 flight 501 failure report by the inquiry board*. 1996.

[Lipp et al. 2018] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *Proceedings of the USENIX Security Symposium*. 2018.

[Magill et al. 2008] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. "THOR: A Tool for Reasoning about Shape and Arithmetic". In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 2008.

[Marlinspike and Perrin 2016] Moxie Marlinspike and Trevor Perrin. *The X3DH Key Agreement Protocol*. `https://signal.org/docs/specifications/x3dh/`. 2016.

[Marti et al. 2006] Nicolas Marti, Reynald Affeldt, Akinori Yonezawa, and Department Of Computer Science. "Formal Verification of the Heap Manager of an Operating System using Separation Logic". In: *Proceedings of the International Conference on Formal Engineering Methods (ICFEM)*. 2006.

[Martínez et al. 2019] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. "Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms". In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2019.

[Masti et al. 2015] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. "Thermal Covert Channels on Multi-core Platforms". In: *Proceedings of the USENIX Security Symposium*. 2015.

[Mehnert 2020] Hannes Mehnert. *Blog :: TLS 1.3 support for MirageOS*. `https://mirage.io/blog/tls-1-3-mirageos`. 2020.

[Merkle 1989] Ralph C. Merkle. "A certified digital signature". In: *Proceedings of the International Cryptology Conference (CRYPTO)*. 1989.

[Michael and Scott 1996] Maged M. Michael and Michael L. Scott. "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms". In: *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. 1996.

[Microsoft 2018] Microsoft. *Cryptographic API: Next generation - Windows Applications*. `http://https://docs.microsoft.com/en-us/windows/desktop/seccng/cng-portal`. 2018.

[Montgomery 1987] Peter L. Montgomery. “Speeding the Pollard and elliptic curve methods of factorization”. In: *Mathematics of Computation* 48 (1987), pp. 243–264.

[Moore et al. 2009] Tyler Moore, Richard Clayton, and Ross Anderson. “The Economics of Online Crime”. In: *Journal of Economic Perspectives* 23.3 (2009), pp. 3–20.

[Mostrous and Vasconcelos 2014] Dimitris Mostrous and Vasco Thudichum Vasconcelos. “Affine Sessions”. In: *Proceedings of the International Conference on Coordination Languages and Models (COORDINATION)*. 2014.

[Moura and Bjørner 2008] Leonardo de Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2008.

[Mozilla 2018] Mozilla. *Measurement Dashboard*. `https://mzl.la/2ug9YCH`. 2018.

[Müller et al. 2016] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. “Viper: A Verification Infrastructure for Permission-Based Reasoning”. In: *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2016.

[Myreen and Gordon 2007] Magnus O. Myreen and Michael J. C. Gordon. “Hoare Logic for Realistically Modelled Machine Code”. In: *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2007.

[Nanevski et al. 2008] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. “Hoare type theory, polymorphism and separation1”. In: *Journal of Functional Programming* 18.5-6 (2008), pp. 865–911.

[Nanevski et al. 2010] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. “Structuring the verification of heap-manipulating programs”. In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2010.

[Nanevski et al. 2014] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. “Communicating State Transition Systems for Fine-Grained Concurrent Resources”. In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2014.

[Nanevski et al. 2019] Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. “Specifying concurrent programs in separation logic: morphisms and simulations”. In: *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 2019.

[National Vulnerability Database 2014] National Vulnerability Database. *Heartbleed bug*. CVE-2014-0160 http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160. 2014.

[Naur and Randell 1969] Peter Naur and Brian Randell. "Software engineering: Report of a conference sponsored by the nato science committee, garmisch, germany, 7th-11th october 1968". In: (1969).

[Nelson and Oppen 1979] Greg Nelson and Derek C. Oppen. "Simplification by cooperating decision procedures". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)*. 1979.

[Ni and Shao 2006] Zhaozhong Ni and Zhong Shao. "Certified assembly programming with embedded code pointers". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2006.

[Nikhil 2004] Rishiyur S. Nikhil. "Bluespec System Verilog: efficient, correct RTL from high level specifications." In: *Proceedings of the ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. 2004.

[Nipkow et al. 2002] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media, 2002.

[Nir and Langley 2015] Yoav Nir and Adam Langley. *ChaCha20 and Poly1305 for IETF protocols*. https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-chacha20-poly1305-10. 2015.

[NIST] NIST. *National Vulnerability Database*. https://nvd.nist.gov/general.

[NIST 2007] NIST. *Recommendation for Block Cipher Modes of Operation: Galois/-Counter Mode (GCM) and GMAC*. NIST Special Publication 800-38D. 2007.

[Norell 2008] Ulf Norell. "Dependently Typed Programming in Agda". In: *Proceedings of the International Conference on Advanced Functional Programming (AFP)*. 2008.

[OHearn et al. 2001] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs That Alter Data Structures". In: *Proceedings of the International Workshop on Computer Science Logic (CSL)*. 2001.

[OHearn 2007] Peter W. O'Hearn. "Resources, Concurrency, and Local Reasoning". In: *Theoretical Computer Science (TCS)*. 2007.

[Oliveira et al. 2017] Thomaz Oliveira, Julio López, Hüseyin Hışıl, Armando Faz-Hernández, and Francisco Rodríguez-Henríquez. "How to (pre-)compute a ladder: Improving the Performance of X25519 and X448". In: *Proceedings of the International Conference on Selected Areas in Cryptography (SAC)*. 2017.

[OpenSSL] OpenSSL. *Vulnerabilities.* https://www.openssl.org/news/vulnerabilities.html Retrieved June, 2021.

[OpenSSL Team 2005] OpenSSL Team. *OpenSSL.* http://www.openssl.org/. 2005.

[Owicki and Gries 1976] Susan Owicki and David Gries. "Verifying properties of parallel programs: An axiomatic approach". In: *Communications of the ACM* 19.5 (1976), pp. 279–285.

[Owre et al. 1992] Sam Owre, John Rushby, and Natarajan Shankar. "PVS: A prototype verification system". In: *Proceedings of the International Conference on Automated Deduction (CADE).* 1992.

[Parkinson and Summers 2012] Matthew J. Parkinson and Alexander J. Summers. "The Relationship Between Separation Logic and Implicit Dynamic Frames". In: *Logical Methods in Computer Science* (2012).

[Păsăreanu and Rungta 2010] Corina S. Păsăreanu and Neha Rungta. "Symbolic PathFinder: symbolic execution of Java bytecode". In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE).* 2010.

[Patterson et al. 2017] Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. "FunTAL: Reasonably Mixing a Functional Language with Assembly". In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI).* 2017.

[Pek et al. 2014] Edgar Pek, Xiaokang Qiu, and Parthasarathy Madhusudan. "Natural Proofs for Data Structure Manipulation in C Using Separation Logic". In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI).* 2014.

[Penninckx et al. 2012] Willem Penninckx, Jan Tobias Mühlberg, Jan Smans, Bart Jacobs, and Frank Piessens. "Sound Formal Verification of Linux's USB BP Keyboard Driver". In: *Proceedings of the International Conference on NASA Formal Methods (NFM).* 2012.

[Percival 2005] Colin Percival. "Cache missing for fun and profit". In: *Proceedings of BSDCan.* 2005.

[Perrin and Marlinspike 2016] Trevor Perrin and Moxie Marlinspike. *The Double Ratchet Algorithm.* https://signal.org/docs/specifications/doubleratchet/. 2016.

[Petcher and Morrisett 2015] Adam Petcher and Greg Morrisett. "The Foundational Cryptography Framework". In: *Proceedings of the International Conference on Principles of Security and Trust (POST).* 2015.

[Philippaerts et al. 2011] Pieter Philippaerts, Frédéric Vogels, Jan Smans, Bart Jacobs, and Frank Piessens. “The Belgian Electronic Identity Card: a Verification Case Study”. In: *Proceedings of the International Workshop on Automated Verification of Critical Systems (AVOCS)*. 2011.

[Piróg et al. 2018] Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. “Syntax and Semantics for Operations with Scopes”. In: *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*. 2018.

[Polubelova et al. 2020] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. “HACLxN: Verified Generic SIMD Crypto (for All Your Favourite Platforms)”. In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2020.

[Polyakov et al. 2018] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. “Verifying Arithmetic Assembly Programs in Cryptographic Primitives”. In: *Proceedings of the International Conference on Concurrency Theory (CONCUR)*. 2018.

[Pottier 2017] François Pottier. “Verifying a Hash Table and Its Iterators in Higher-Order Separation Logic”. In: *Proceedings of the International Conference on Certified Programs and Proofs (CPP)*. 2017.

[Protzenko et al. 2017] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. “Verified Low-Level Programming Embedded in F*”. In: *Proceedings of the International Conference on Functional Programming (ICFP)*. 2017.

[Protzenko et al. 2019] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. “Formally Verified Cryptographic Web Applications in WebAssembly”. In: *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*. 2019.

[Protzenko et al. 2020] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. “EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider”. In: *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*. 2020.

[Rakamaric and Emmi 2014] Zvonimir Rakamaric and Michael Emmi. “SMACK: Decoupling Source Language Details from Verifier Implementations”. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 2014.

[Randell and Buxton 1970] Brian Randell and John N. Buxton. “Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27th-31st October 1969”. In: (1970).

[Rastogi et al. 2021] Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy. *Programming and Proving with Indexed Effects*. In Submission. 2021. URL: https://www.fstar-lang.org/papers/indexedeffects/.

[Rescorla 2018] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3, Draft 28*. https://tools.ietf.org/html/draft-ietf-tls-tls13-28. 2018.

[Reynolds 2002] John Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures”. In: *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*. 2002.

[Rocha Pinto et al. 2014] Pedro Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. “TaDA: A Logic for Time and Data Abstraction”. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 2014.

[Rodrigues et al. 2016] Bruno Rodrigues, Fernando Pereira, and Diego Aranha. “Sparse Representation of Implicit Flows with Applications to Side-Channel Detection”. In: *Proceedings of the International Conference on Compiler Construction (CC)*. 2016.

[Rushby and Von Henke 1993] John Rushby and Friedrich Von Henke. “Formal verification of algorithms for critical systems”. In: *IEEE Transactions on Software Engineering* 19.1 (1993), pp. 13–23.

[Rushby 1992] John Rushby. “Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems”. In: *Proceedings of the International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*. 1992.

[Sammler et al. 2021] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. “RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types”. In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2021.

[Seacord 2018] Robert Seacord. *Implement abstract data types using opaque types*. https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87151966. 2018.

[Sergey et al. 2015] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. “Mechanized Verification of Fine-Grained Concurrent Programs”. In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2015.

[Smans et al. 2012] Jan Smans, Bart Jacobs, and Frank Piessens. “Implicit Dynamic Frames”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)*. 2012.

[Souyris 2014] Jean Souyris. *Industrial use of CompCert on a safety-critical software product*. http://projects.laas.fr/IFSE/FMF/J3/slides/P05_Jean_Souyiris.pdf. 2014.

[Sridharan et al. 2013] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. “Alias analysis for object-oriented programs”. In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, 2013, pp. 196–232.

[Stevens et al. 2007] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. “Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities”. In: *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. 2007.

[Svendsen and Birkedal 2014] Kasper Svendsen and Lars Birkedal. “Impredicative Concurrent Abstract Predicates”. In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2014.

[Svendsen et al. 2013] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. “Modular Reasoning about Separation of Concurrent Data Structures”. In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2013.

[Swamy et al. 2013] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. “Verifying higher-order programs with the Dijkstra monad”. In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2013.

[Swamy et al. 2016] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. “Dependent Types and Multi-Monadic Effects in F*”. In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2016.

[Swamy et al. 2020] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. “SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs”. In: *Proceedings of the International Conference on Functional Programming (ICFP)*. 2020.

[Swierstra 2008] Wouter Swierstra. “Data types à la carte”. In: *Journal of functional programming* 18.4 (2008).

[Timany et al. 2018] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. “A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of RunST”. In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2018.

[Tiwari et al. 2009] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. “Complete Information Flow Tracking from the Gates Up”. In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2009.

[Tofte and Talpin 1997] Mads Tofte and Jean-Pierre Talpin. “Region-Based Memory Management”. In: *Information and Computation* 132.2 (1997), pp. 109–176.

[Tsai et al. 2017] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. “Certified Verification of Algebraic Properties on Low-Level Mathematical Constructs in Cryptographic Programs”. In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2017.

[Tuch et al. 2007] Harvey Tuch, Gerwin Klein, and Michael Norrish. “Types, Bytes, and Separation Logic”. In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2007.

[Turon et al. 2013] Aaron Turon, Derek Dreyer, and Lars Birkedal. “Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency”. In: *Proceedings of the International Conference on Functional Programming (ICFP)*. 2013.

[Vafeiadis and Parkinson 2007] Viktor Vafeiadis and Matthew Parkinson. “A Marriage of Rely/Guarantee and Separation Logic”. In: *Proceedings of the International Conference on Concurrency Theory (CONCUR)*. 2007.

[Vassena et al. 2021] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcıand Ranjit Jhala, Dean Tullsen, and Deian Stefan. “Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade”. In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2021.

[Villard et al. 2010] Jules Villard, Étienne Lozes, and Cristiano Calcagno. “Tracking Heaps That Hop with Heap-Hop”. In: *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2010.

[Vogels et al. 2011] Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans. “Annotation Inference for Separation Logic Based Verifiers”. In: *Proceedings of the*

International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS). 2011.

[Werner 1994] Benjamin Werner. “Une théorie des constructions inductives”. PhD thesis. Université Paris-Diderot-Paris VII, 1994.

[Winterer et al. 2020] Dominik Winterer, Chengyu Zhang, and Zhendong Su. “Validating SMT solvers via semantic fusion”. In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2020.

[Wolf et al. 2021] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. “Gobra: Modular Specification and Verification of Go Programs”. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 2021.

[Woodcock 1989] Jim C.P. Woodcock. “Properties of Z specifications”. In: *ACM SIGSOFT Software Engineering Notes* 14.5 (1989), pp. 43–54.

[Xia et al. 2019] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. “Interaction trees: representing recursive and impure programs in Coq”. In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2019.

[Xu et al. 2016] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. “A Practical Verification Framework for Preemptive OS Kernels”. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 2016.

[Xu 2019] Herbert Xu. *[PATCH 3/17] zinc: introduce minimal cryptography library*. `https://lore.kernel.org/lkml/E1h7DgR-0001H9-UN@gondobar/`. 2019.

[Yang and Hawblitzel 2010] Jean Yang and Chris Hawblitzel. “Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System”. In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2010.

[Yang et al. 2008] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O’Hearn. “Scalable Shape Analysis for Systems Code”. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 2008.

[Yang et al. 2011] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. “Finding and understanding bugs in C compilers”. In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2011.

[Ye et al. 2017] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. “Verified Correctness and Security of mbedTLS HMAC-DRBG". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2017.

[Yu et al. 1999] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model checking TLA+ specifications". In: *Proceedings of the Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*. 1999.

[Zhang et al. 2015] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. "A hardware design language for timing-sensitive information-flow security". In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2015.

[Zinzindohoué-Marsaudon 2018] Jean-Karim Zinzindohoué-Marsaudon. "Secure, fast and verified cryptographic applications: a scalable approach". PhD thesis. PSL Research University, 2018.

[Zinzindohoué et al. 2016] Jean-Karim Zinzindohoué, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. "A Verified Extensible Library of Elliptic Curves". In: *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*. 2016.

[Zinzindohoué et al. 2017] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. "HACL*: A Verified Modern Cryptographic Library". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2017.