Secure, fast and verified cryptographic applications : a scalable approach

Jean-Karim Zinzindohoué-Marsaudon

THÈSE DE DOCTORAT

de l'Université de recherche Paris Sciences et Lettres
PSL Research University

Préparée à ENS Ulm – INRIA Paris

SECURE, FAST AND VERIFIED CRYPTOGRAPHIC APPLICATIONS: A SCALABLE APPROACH

IMPLEMENTATIONS CRYPTOGRAPHIQUES SURES, PERFORMANTES ET VERIFIEES : UNE APPROCHE PASSANT A L'ECHELLE

Ecole doctorale ED386

SCIENCES MATHEMATIQUES DE PARIS CENTRE

Spécialité INFORMATIQUE

COMPOSITION DU JURY :

Mme. CORTIER Véronique
LORIA, Rapporteur , Présidente du Jury

M. FOUQUE Pierre-Alain
Université Rennes 1, Rapporteur

M. SCHWABE Peter
Radboud University, Membre du jury

M. FOURNET Cédric
Microsoft Research, Membre du jury

M. STRUB Pierre-Yves
Ecole Polytechnique, Membre du jury

M. BHARGAVAN Karthikeyan
INRIA Paris, Membre du jury

Soutenue par JEAN-KARIM ZINZINDOHOUE-MARSAUDON
le 03 iuillet 2018

Dirigée par **KARTHIKEYAN BHARGAVAN**

ENS | PSL

SECURE, FAST AND VERIFIED CRYPTOGRAPHIC APPLICATIONS:

A SCALABLE APPROACH

Jean-Karim ZINZINDOHOUE-MARSAUDON

Ph. D. Thesis

President of the Jury: Ms. Véronique CORTIER

July 3rd 2018

Abstract

The security of Internet applications relies crucially on the secure design and robust implementations of cryptographic algorithms and protocols. This thesis presents a new, scalable and extensible approach for verifying state-of-the-art bignum algorithms, found in popular cryptographic implementations. Our code and proofs are written in F*, a proof-oriented language which offers a very rich and expressive type system, with dependent types, refinement types, effects and customizable memory models. The natural way of writing and verifying higher-order functional code in F* prioritizes code sharing and proof composition, but this results in low performance for cryptographic code. We propose a new language, Low*, a fragment of F* which can be seen as a shallow embedding of C in F* and safely compiled to C code. Nonetheless, Low* retains the full expressiveness and verification power of the F* system, at the specification and proof level. We use Low* to implement cryptographic code, incorporating state-of-the-art optimizations from existing C libraries. We use F* to verify this code for functional correctness, memory safety and secret independence. We present HACL*, a full-fledged and fully verified cryptographic library which boasts performance on par, if not better, with the reference C code. Several algorithms from HACL* are now part of NSS, Mozilla's cryptographic library, notably used in the Firefox web browser and the Red Hat operating system. Eventually, we apply our techniques to miTLS, a verified implementation of the Transport Layer Security protocol. We show how they extend to cryptographic proofs, state-machine implementations and message parsing verification.

Contents

Chapter 1

Introduction

The last few decades have seen a booming expansion of information technologies. In less than half a century, the Internet went from a confidential military network to an indispensable communication technology that we all rely on. This incredible development was backed by dramatic improvements in computational power, storage capacity and worldwide connectivity in such a way that sharing large amounts of data and computational resources across the planet is now an easy feat. New generations of children are raised in a connected world where everything is immediate and decentralized, and information technologies are now pervasive. The development of new usages and features has been the main drive for this technological and societal revolution, led by young information technology (IT) companies that have grown to be richer — and maybe more powerful — than certain states.

Unfortunately, other subjects of concern such as security and privacy have not received as much attention, mostly for business and marketing reasons: the hype supporting new products always originates from new, distinguishing features, while improvements on the overall quality of the products in terms of safety, security or privacy are much harder to show off and market. This is not without consequences. Current users of various communication systems around the globe are mostly unaware that they inherited many legacy protocols, software designs and more generally conceptual views from their predecessors.

Because of the pace at which new standards, technologies and usages appear and disappear, IT companies focus on maintaining compatibility between their latest most up-to-date products and legacy items. Thus low-level libraries, which are at the core of many of those products, tend to encompass a broad range of features, from legacy ones to the most recent or advanced, in a mix of old and recent code. These libraries are essential building blocks for software applications, for instance providing ways to access the kernel of the system, to use the graphical interfaces, to access cryptographic or networking functionalities etc. with new, non-interoperable versions of those components

every couple months or years. Hence, these libraries are regularly updated, integrating new features but retaining the old ones, which may still be useful to a number of products. This process results in always larger and more complex codebases, almost impossible to audit and fully review, and sometimes leads to major issues: software components are usually not completely isolated and a flaw in some remote unused part of a core library could be used by an attacker to corrupt the whole system.

It raises the following questions: should we trust legacy code which is seldom used, might not have been developed with the current security standards and may very well be flawed in a way that would compromise the whole codebase? And on the other hand, should we trust the latest implementations which have not yet received a lot of attention and may be similarly flawed? Obviously we should be extremely cautious in both cases, and still, this is under the assumption that the code is open-source so that the community can review it and contribute. Proprietary code that almost no one gets their hands on should be considered with the utmost care.

The research community is always keen on studying the security of new systems under different and often innovative angles, and thus for a long time now these evolving new technologies have been analyzed, criticized and new approaches have been proposed. Unfortunately, exchanges between academic researchers and the IT industry are limited. Although there are some ties between the two communities, in practice it is not so common for academia to consider industrial constraints or the industry the latest academic research innovations. No one is to blame, the industry has its own set of constraints which often do not apply to a research environment, while real companies are interested in improvements from the research community only as long as they can readily be integrated in their processes at a negligible cost or with a high return on investment, which does not happen so often. However, as computer science is still young and undergoing heavy changes, now may be a good time to encourage further cooperation, especially on security related aspects.

The ambition of this work is to improve security in software development by bringing state-of-the-art methods from academia together with the specific needs from the industry and real world applications, and build new ways to solve the later using the former. This idea of making research more practical, or making the industry more research aware is not novel. Nonetheless, as new concerns emerge from the security and privacy aspects of computer science, contributing to making communication systems more reliable, robust and trustworthy appears as a challenging topic, which will quickly produce results and significant long term impacts. Software security often relies on cryptography, a disciple of mathematics and computer science which aims to ensure confidentiality, authenticity and integrity of data. This is the right time to evaluate secure development methods and challenge our ability to set new quality standards for the next generation of cryptographic algorithms and cryptography-based applications — typically secure communication applica-

tions — which are now pervasive.

Both the correctness and performance of cryptographic code are crucial. The whole purpose of cryptographic constructions is to ensure certain security guarantees to its user. It collapses if the actual cryptographic code is flawed. Moreover, broken cryptography is arguably worse than no cryptography: the end user relying on an unsafe channel will adopt a more careful, security aware behavior, while she will be less careful on a channel she trusts, meaning that she would get more exposed using a flawed secure communication channel while trusting it than just using regular communication means in the clear. Coincidentally, cryptography is often performance critical. For instance, a large percentage of exchanges over the internet are now encrypted. The efficiency of the corresponding cryptographic code is essential on many levels: latency, service availability, bandwidth, power consumption etc. The economic impact of a slight performance drop in the cryptography used by Google servers for instance is likely to be very high, not to mention the potential dissatisfaction of its consumers witnessing a degraded quality of service.

This work aims at proposing new development methods based on formal methods for cryptography-based software which will meet the different criteria needed to be adopted by real world projects.

Building trustworthy software In order to build trustworthy software, developers typically build sophisticated testing and auditing frameworks and hope to catch all important bugs by repeated testing and manual reviews. A more ambitious goal to is use "formal methods" to mathematically prove the absence of all bugs of a certain class. In order to use a formal method, one has to specify the properties of the program she wishes to prove and use a dedicated tool to prove that these properties hold. Some of these steps can be automated. The advantage of formal methods is that the verification is done statically for all possible inputs, there is no impact at runtime. The disadvantage is that, depending on the complexity of the expected properties, the proof burden can be much higher than it would be for testing and manual auditing.

Consequently, the first part of this work has been dedicated to exploring the actual proof capabilities of existing tools and testing new approaches to get a better understanding of what formal methods are lacking to be used for real world applications. In this regard, cryptographic code appears to be a great test case for various programming languages and tools. On one hand, cryptography usually originates from mathematical objects which lend themselves quite well to a concise and easy to review formalization. In other words formal methods, which are about giving and proving the adherence between a formal specification and an algorithm, should be well-equipped to specify the underlying mathematics of cryptographic primitives such as *Curve25519* [34]. On the other hand, because performance and safety are essential to this kind

of code, standard cryptographic primitive implementations embed a large set of low-level optimization and coding patterns which are error-prone and challenging from a correctness perspective. It makes this kind of code an ideal candidate for verification, with clear and simple mathematical specifications and complex, optimized implementations to match them against. Also, because in cryptography the space of possible inputs makes it impossible to catch all possible bugs by unit-testing or fuzzing, formal methods appear as the only way to get reasonable confidence in the correctness of the optimized implementations.

In September 2014, a blog post from Adam Langley, a leading cryptography developer and expert, set out a new challenge. Writing in his blog Imperial Violet [1], he presented the result of a shallow survey he made of different verification tools to tackle C code verification for the aforementioned Curve25519 cryptographic primitive. Interestingly, although this particular primitive was designed with implementation in mind, meaning that its implementation is thought to be easy and very clearly guided from its published specification [3] and it has very few potential pitfalls compared to other more complex primitives, Langley found its formal analysis to be quite hard.

> *The conclusion is a bit disappointing really: Curve25519 has no side effects and performs no allocation, it's a pure function that should be highly amenable to verification and yet I've been unable to find anything that can get even 20 lines into it. Some of this might be my own stupidity, but I put a fair amount of work into trying to find something that worked.*
>
> *There seems to be a lot of promise in the area and some pieces work well (SMT solvers are often quite impressive, the Frama-C framework appears to be solid, Isabelle is quite pleasant) but nothing I found worked well together, at least for verifying C code. That makes efforts like SeL4 and Ironsides even more impressive. However, if you're happy to work at a higher level I'm guessing that verifying functional programs is a lot easier going.*

This post shows that although formal methods have been around for several decades, and academic research has been quite active in the domain, the technology is either not mature enough, or the entry cost is too high and seasoned programmers outside the tools' development teams cannot use them. Seemingly, even simple cryptographic primitives carefully designed to be implementation friendly, and thus are though easy to prove by developers, are still out of reach for existing verification tools, at least with a reasonable effort. Hence, our goal is to contribute to making such methods more accessible.

[1] https://www.imperialviolet.org/2014/09/07/provers.html

Building performant software Cryptographic code is not only challenging on the correctness side but also in terms of performance. Indeed, in many cases, typically to provide a secure, encrypted channel, cryptographic code is a bottleneck for the overall software performance. Usually, the more control one has over the low-level details of the code, the more optimized that one may be. As of today, machines have not yet gotten the best of human developers and the faster secure crypto code is still not compiled from high-level languages but hand-optimized at the assembly level. An immediate consequence, however, is that the more details a human developer has to deal with and the more error prone the development becomes, which happens to precisely be the pitfall formal methods aim at avoiding.

We need to find a middle-ground where the language is low-level enough to get good performance and yet high-level enough so that the proofs are still widely automated. Assembly code is too low-level, it is the fastest language available but it is not portable and very error-prone, meaning that the proof effort for large projects would be almost impossible to manage. It is more reasonable to target C code, which is one of the most portable languages available, exhibits decent performances and is very well known. In practice, the reference implementations of almost all cryptographic primitives are written in C, precisely for those reasons. The question remains open though as to what should be the source language.

Building large software The usability of formal methods for software verification is not only linked to the intrinsic verification capabilities of the chosen tool or language. The ability of those methods to scale to large and collaborative projects is absolutely crucial for the end users (the community of developers). Usually, with enough time and effort invested, almost anything is achievable. However, spending several expert years on a single cryptographic primitive may be worth the effort from a scientific point of view, to validate a method and measure its efficiency, but it is unreasonable for a viable business company. We do not want to solely focus on the verification achievements but also on code sharing and our ability to distribute already implemented and verified algorithms in such a way that they could benefit other programmers and reduce their development — and proof — burden. In an ideal world, developers would have access to formally verified Application Programming Interfaces (API). An API is a normalized set of functions exposed by a service which serves as a front-end to access the service functionalities. From such verified APIs, they would easily build similarly verified and correct software leveraging on the proven building blocks and their ability to use the formal methods tools. While this is not yet possible, significant progress has been made towards sharing proofs between implementations and the hope that, in a not so distant future, formal methods will take more importance in regular software development is now quite vivid.

A powerful proof methodology with the ability to scale to large projects would undoubtedly benefit the field. Nonetheless, if the entry cost is too high, there are but little chances that the said methodology will actually be used by people outside the team which developed it. To lower the entry cost, there are two distinct paths one could explore. The first one is to take the language everyone uses to develop software, and provide great tooling to support formal verification for code natively written in that language. Unfortunately, it is not obvious that such a language exists — people have been developing in a variety of languages, and those which are the most portable like C are unfortunately not the fastest which is return, are platform independent like assembly. Furthermore, a tool which would be developed around an existing language would suffer from the inherent limitations of the language. For instance, rather than building amazing tools to prove C code secure, which is basically impossible except in specific cases, the Mozilla Foundation decided to create a new language, Rust [95], with specific security features built in. In general, even for languages with great verification tools, it is way easier to verify code which was written with proofs in mind than to verify legacy, general purpose software.

Hence, we considered that if the easiest way to prove code correct was to re-implement it in a secure but verification friendly manner, we might as well do it in a language which lends itself well to formal methods. And as Adam Langley highlighted, functional programming languages are typically better suited, due to their more mathematical nature, to formal verification. Those languages however are not as broadly taught and used as imperative languages are, meaning that only few developers are familiar with them and that the entry cost for such languages in the industry is much higher. To compromise, we propose an approach which consists in using a high-level, functional and well-suited for verification programming language to implement the various algorithms, and then safely compile the code to a well-known lower-level language: C. From there, the code can be reviewed just like natively written C code currently is, it benefits from the formally proven guarantees and because C is very common and has bindings to almost every other languages, the code can easily be used, once compiled, in existing or new developments, without the need to maintain a new language and a new tool chain.

We show how to leverage on our ability to share large amounts of code to implement a full-fledged cryptographic library producing C code with performance on par with the state of the art C code out there in the real world, how to integrate that code into existing real world projects such as NSS, the cryptographic library from the Mozilla Foundation, used among others by the Firefox web browser and the Red Hat Linux distribution, as well as in other, larger verification projects such as an implementation of the full Transport Layer Security (TLS) stack.

1.1 Contributions

To summarize, the contributions of this work are illustrated through four sequential steps.

1. **Verifying Bignum Code in F***: chapter 2 presents the F* language and gives a flavor of the verification system. Chapter 3 illustrates how F*, a modern proof-oriented functional programming language can be used to prove the functional correctness of state of the art cryptographic algorithms. In this first contribution we also highlight how the modular system of the language and a principle approach towards factoring out common code patterns across different but similar cryptographic primitive can lower the proof burden of adding more big number-based cryptographic primitives to the library by more than 50%;

2. **Compiling Verified F* code in C**: Chapter 4 presents a subset of the F* language which, with a particular representation of its memory model and a restricted set of features, is safely compilable to C code. Its strength comes from the combination of the expressiveness of the language for the computationally irrelevant proof aspects of the implementation, and the ability to model all low-level algorithmic optimizations in the restricted compilable subset;

3. **Building a high-assurance cryptographic library**: chapter 5 shows how to combine the two prior ones into a full-fledged cryptographic library, which performs on par with the reference existing C libraries, and demonstrates how to make use of the expressiveness of the source language to encompass platform specific features to get close to the performance of assembly code on certain platforms;

4. **Extending verification to protocol code**: chapter 6 shows that our methodology is not restricted to low-level cryptographic code and that formal methods indeed lend themselves particularly well to other sensitive areas such as verifying protocol state machines or the correctness of parsers, both security critical components but which designs are completely different from computationally intensive cryptographic primitives; thus illustrating how core security essential components can be independently redesigned and proven using formal methods.

1.2 Related work

Functional programming verification

This work relies on F*, a functional, proof-oriented verification language for formal proofs. The F* type system includes dependent types, refinement types

and monadic effects. It relies on a weakest precondition calculus and on a SMT encoding backend to automatically discharge verification conditions. The new features of the language were presented in a series of papers [116, 9, 8]. A series of research works have been working towards integrating dependent types within a full-fledged, effectful programming language. Cayenne [17] was an early effort to integrated dependent types within a Haskell-like language. More recently, old-F* [114] adds *value-dependent* types to an ML-like language; Rondon et al. [108] add decidable, refinement types to OCaml and Vazou et al. [123] adapt that work to Haskell. Meanwhile, monadic old-F* [115] adds a single monad to a variant of old-F* without refinement types. Liquid Haskell only has non-termination as an effect and for soundness requires a termination check based on the integer ordering, which is less expressive than ours. All these languages provide SMT-based automation, but do not have the ability to support interactive proofs or to carry out functional correctness proofs of effectful programs.

The Zombie language [54] investigates the design of a dependently typed language that includes non-termination via general recursion. Zombie arose from a prior language, Trellys [81]. Zombie supports reasoning extrinsically about potentially divergent code, whereas in F*, proofs about divergent programs are carried out intrinsically, within its program logic. Zombie does not address other effects or provide proof automation.

Idris [53] is another recent clean-slate design which provides non-termination primitively and also an elegant style of algebraic effects. However, Idris also lacks SMT-based proof automation.

Another related language is ATS [55], which, like F*, aims to combine effectful programming and theorem proving. However, the design of ATS is substantially different from F*. Furthermore, ATS only has limited support for automated theorem proving, unlike F*'s SMT integration.

Nanevski et al. [100] develop Hoare type theory (HTT) as a way of extending Coq with effects. The strategy there is to provide an axiomatic extension of Coq with a single catch-all monad in which to encapsulate imperative code. Tools based on HTT have been developed, notably Ynot [59]. This approach is attractive in that one retains all the tools for specification and interactive proving from Coq. On the downside, one also inherits Coq's limitations, e.g., the syntactic termination check and lack of SMT-based automation.

Most dependent type theories rely crucially on normalization for consistency, many researchers have been investigating improving on Coq's syntactic termination check via more semantic approaches. Agda [103] offers two termination checkers. The first one is based on fœtus [6]. Contrary to fœtus, the F* termination checker does not aim to find an ordering automatically; nonetheless, our check is more flexible, since it is not restricted to a structural decreasing of arguments, but the decreasing of a *measure* applied to the arguments. The second one is based on sized types [19, 7], where the size on types approximates the depth of terms. In contrast, in F*, the measures are defined

by the user and are first-class citizens of the language and can be reasoned about using all its reasoning machinery. Isabelle/HOL [102] also supports semantic termination checking, however, the approach of Krauss et al. [86] seems very different from ours, and only applies to a first-order fragment.

Software verification frameworks, such as Why3 [69] and Dafny [90], also use SMT solvers to verify the logical correctness of (mostly) first-order programs. Unlike F*, they do not provide the expressiveness of dependent types and do not provide the flexibility of user-defined effects and memory models.

F*'s hyper-heap model is closely related to local stores in Euclid [87]. Local stores are also a partitioned heap abstraction realized on a flat heap. However, local stores lack the hierarchical scheme of hyper-heaps, which we find convenient for hiding from clients the details of the partitioning scheme used within an object. Utting [122] describes a variation on local heaps that supports a "transfer" operation, moving references dynamically from one region to another. This may be a useful variation on hyper-heaps as well, at the cost of losing the stable, state-independent invariants obtained by pinning a reference to a (dynamically chosen) region.

Verifying Efficient Low-Level Code

Many approaches have been proposed for verifying the functional correctness and security of efficient low-level code. A first approach is to build verification frameworks for C using verification condition generators and SMT solvers, as Frama-C [82], VCC [60] or Verifast [78] do. While this approach has the advantage of being able to verify existing C code, this is very challenging: one needs to deal with the complexity of C and with any possible optimization trick in the book. Moreover, one needs an expressive specification language and escape hatches for doing manual proofs in case SMT automation fails. So others have deeply embedded C, or C-like languages, into proof assistants such as Coq [29, 15, 56] and Isabelle [124, 110] and built program logics and verification infrastructure starting from that. This has the advantage of using the full expressive power of the proof assistant for specifying and verifying properties of low-level programs. This remains a very labor-intensive task though, because C programs are very low-level and working with a deep embedding is often cumbersome. Acknowledging that uninteresting low-level reasoning was a determining factor in the size of the seL4 verification effort [83], Greenaway et al. [75, 74] have recently proposed sophisticated tools for automatically abstracting the low-level C semantics into higher-level monadic specifications to ease reasoning. Compiling F* to C, we take a different approach: we give up on verifying existing C code and embrace the idea of writing low-level code in a subset of C shallowly embedded in F*. This shallow embedding has significant advantages in terms of reducing verification effort and thus scaling up verification to larger programs. This also allows us to port to C only the parts

of an F* program that are a performance bottleneck, and still be able to verify the complete program.

In order to prevent the most devastating low-level attacks, several researchers have advocated dialects of C equipped with type systems for memory safety [61, 79, 117]. Others have designed new languages with type systems aimed at low-level programming, including for instance linear types as a way to deal with memory management [13, 104, 95]. One drawback is the expressiveness limitations of such type systems: once memory safety relies on more complex invariants than these type systems can express, compromises need to be made, in terms of verification or efficiency. Low*, our shallow embedding of C in F*, can perform arbitrarily sophisticated reasoning to establish memory safety, but does not enjoy the benefits of efficient decision procedures [1] and currently cannot deal with concurrency.

We are not the first to propose writing efficient and verified C code in a high-level language. LMS-Verify [14] recently extended the LMS meta-programming framework for Scala with support for lightweight verification. Verification happens at the generated C level, which has the advantage of taking the code generation machinery out of the TCB, but has the disadvantage of being far away from the original source code.

Bedrock [58] is a generative meta-programming tool for verified low-level programming in Coq. The idea is to start from assembly and build up structured code generators that are associated verification condition generators. The main advantage of this "macro assembly language" view of low-level verification is that no performance is sacrificed while obtaining some amount of abstraction. One disadvantage is that the verified code is not portable.

Crypto Code Verification

Formal verification has been successfully used on large security-critical software systems like the CompCert C compiler [92] and the sel4 operating system kernel [85]. It has even been used to verify a full implementation of the Transport Layer Security (TLS) protocol [42]. However, until recently, formal methods had not been applied to the cryptographic primitives underlying these constructions and protocols.

Recently, several works have taken on this challenge. Hawblitzel et al. [76] wrote and verified new implementations of SHA, HMAC, and RSA in the Dafny programming language. Appel [16] verified OpenSSL's C implementation of SHA-256 in Coq, and Behringer et al. [30] followed up with a proof of OpenSSL's HMAC code. Later, Ye et al. [125] proved correct the mbedTLS's implementation of the HMAC-DRBG primitive. Chen et al. [57] used a combination of SMT solving and the Coq proof assistant to verify a qhasm implementation of Curve25519. In [129], we wrote and verified three elliptic curves P-256, Curve25519, and Curve448 in the F* programming language and compiled them to OCaml.

Barthe et al [21] and Almeida et al [10, 11] explained how to verify constant-time assembly code to mitigate side-channels. [23] and [24] take a different approach, using masking techniques to prevent information leaks.

Bond et al. [52] show how to verify assembly implementations of SHA-256, Poly1305, and AES-CBC using Vale. Cryptol and SAW [119] have been used to verify C and Java implementations of Chacha20, Salsa20, Poly1305, AES, and ECDSA. The Fiat-Crypto project [68] chooses a systematic synthesis of elliptic curve cryptographic code using a verified Coq based toolchain. Compared to F*, their trusted computing base is smaller. However, only the field arithmetic parts are automatically generated rather than the whole primitive. The Jasmin [12] framework aims to help programmers develop high-speed cryptographic code in assembly, with memory safety and side-channel resistance guaranties. Although verified in Coq, it still lacks the expressiveness and proof capabilities of a verification language such as F*. In [121], Tsai et al. present a technique to translate a mathematical construct into an algebraic problem and prove it using SMT solvers, notably computing output ranges and checking for the absence of overflows.

Compared to these works, we use a different methodology, by verifying code in F* and compiling it to C. Furthermore, unlike these prior works, our goal is to build a self-contained cryptographic library, so we focus on a complete set of primitives and we aggressively share code between them.

Crypto Protocol Verification

This work is not primarily focused on cryptographic protocol verification. Several surveys [97, 77, 62, 47, 63, 49] detail the different techniques used to formally analyze and verify the security of cryptographic protocols, and many tools are available, Tamarin [98], ProVerif [48], EasyCrypt [20] and CryptoVerify [46] being the main ones.

However, these works do not attempt to prove the correctness of the actual cryptographic primitives these protocols use. [67] proposed a methodology to verify cryptographic protocols written in C, [40] designed a language and tool to verify XML web services and [106] targeted Java-written protocols. The miTLS project [42, 44] verifies an implementation of the TLS protocol, however the code of the underlying cryptographic library is trusted and unverified.

In this thesis we show how to extend the miTLS project, composing the functional correctness and memory safety properties of the cryptographic primitives with security proofs, all of it written in F*.

Chapter 2

Verified Programming in F*

Parts of the text are taken from [116], a paper that appeared in POPL 2016 and was co-authored by me along with Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss and Santiago Zanella-Béguelin.

2.1 Raising the Level of Trust in Software Development

Software bugs stem from the fallible nature of human developers and as such have plagued software development since its early days. Over the past decades, several approaches have been explored to raise the level of trust one can put in software and lower the likeliness of bug occurrences. Following each of these approaches, a collection of tools has been implemented and steadily improved. Unfortunately, although there have been significant improvements in the expressiveness of the tools and the scale of the projects they can tackle, a lot remains to be done. With the end goal of verifying cryptographic primitives, which are complex, performance critical and security critical to communications, the question of which path to follow and which tools to rely on remains open.

2.1.1 Pros and Cons of Verification Software

Nowadays, the choices available in programming languages and associated tools are overabundant, making it difficult to decide which ones to explore further. To differentiate between them we suggest four different axes:

Safety Programming languages can be differentiated based on their inherent *safety*. At one end of the spectrum, interpreted and runtime based languages such as Java or OCaml handle memory management automatically. This dramatically decreases the risk of introducing critical memory safety vulnerabili-

ties. At the other end, languages such as C leave full control over the memory to the programmer, thus providing her with more flexibility, at the cost of potentially introducing flaws.

Efficiency Low-level languages typically provide the programmer with a lot of control and expressiveness which enables fine-grained optimization of programs. High-level languages typically make development easier by providing high-level constructs and features which hide the low-level details, but do not allow for too sophisticated optimization.

Domain of application Domain-specific programming languages can be distinguished from general purpose languages. The former are tuned to perform particularly well on their domains but may be unusable for other purposes; the later offer a broader range of applications but less performance for specific tasks than dedicated tools.

Effects Some languages are natively designed to describe *computational effects*, such as interacting with the program's memory or handling exceptions. Others only support pure, effect-free computations. The later are much easier to reason about while the usability and performance of the former is greater.

The general belief is that safe and easy-to-verify languages usually offer poor performance, while, conversely, unsafe languages tend to provide higher performance, but are quite difficult to prove correct. As an example, a tool like Coq [18], a proof assistant, is very much biased towards proofs but is not meant to generate efficient code. On the other hand, assembly is the fastest language available, but it seems almost unreasonable to target assembly code verification for large projects given the complexity and the many unsafe features of the language. Similarly, domain specific languages are thought to be easier to verify and perform better than general purpose languages applied to their specific domain, but present a high entry cost as one needs to learn the specifics of each specialized language rather than one single, general purpose, one.

In the recent years, a new generation of languages has been aiming for the sweet spot between all of those. Two such languages, both successful yet remarkably different in their approach, are Rust and F*.

While Rust is not a general purpose verification language, it is memory safe and yet offers performance comparable to C or C++. On the other hand F* aims at providing the programmer with rich, expressive and flexible semantics so that she can implement and verify anything and yet rely on a substantial level of automation thanks to interactions with automated solvers. Those two examples highlight the fact that verification tools are more effective if the languages have been designed with verification in mind.

2.2 F*, a proof-oriented programming language

To tackle a new and ambitious verification project, the choice of the programming language and the associated tools is crucial. As previously outlined, there is *a priori* a wide range of solutions with their pros and cons. Nevertheless, considering the available languages and tools more closely, the F* programming language and verification system eventually appears as a natural choice, for several reasons.

First, F* is very expressive, with a rich effectful type system, and was designed to be general purpose and not domain specific. This makes it a more natural candidate than other recent and popular secure languages like Rust. In particular, Rust's design has been very much biased towards automated safety (memory safety, thread safety etc.) at no performance cost. As such, although it boasts amazing performance results compared to unsafe languages like C or C++ at no security cost, it does not offer the level of expressiveness of a general purpose verification language such as F*. For instance, it does not provide the user with enough means to tackle verification of cryptography-based software, in which the mathematical details are almost as important as the safety of the software as a whole, and definitely more important than raw performance. In the later sections of this chapter we will present the main aspects of the F* programming language.

Second, F* has already been deemed a worthy verification language for security critical software through the miTLS line of research [42, 45, 44]. The miTLS project has shown how to make use of the verification system to verify extensive and complex programs. Furthermore, this project revolves around cryptographic protocols; since it is our goal to develop an extensible framework for cryptography, the miTLS codebase, which relies on unverified cryptography, appears as an immediate consumer of the F* code we could produce. It makes sense to adopt a common language and syntax so that the two projects may coexist and benefit from each other's strengths.

Eventually, many new languages were initially designed as proofs of concept, to explore new technologies and new programming paradigms. They typically illustrate strengths and weaknesses of different approaches, and, often, end up being abandoned in favor of their successor, a language more mature, which will benefit from the new advances and latest research results while not carrying the mistakes of its ancestor. Such short-lifetime research languages are not good candidates for a project that hopes to span over several years, as those may end up relying on out-of-date or unmaintained systems. In contrast, F* is already the heir of a line of programming languages (Fine [113], F7 [41]). Furthermore, it has a vibrant community and it aims to be used in real-world software (see Everest Project [1]).

[1] https://project-everest.github.io/

2.2.1 A Weakest Pre-Condition Calculus

In 1975, Dijsktra [65] showed that program semantics could be defined via the program's *weakest precondition* predicate transformer. For any predicate on the outcome of the computation this predicate transformer returns the most liberal precondition the input has to verify in order for the post-condition to hold.

Through a series of works Swamy et al. [115, 116, 9] introduced the idea that this mechanism could be used to reason about the specifications of effectful programs. They note that the weakest precondition predicate transformer can be seen as a monad at the level of types where the return and bind combinators respectively return the weakest precondition of an effectful computation and the sequential composition of two weakest preconditions, leading to a deterministic encoding of program specifications into verification conditions. Going further, they show that inductive types, recursion and higher order specifications can be encoded as first order logical formulas and automatically discharged by automated theorem provers such as Satisfiability Modulo Theories (SMT) solvers. This concept has been practically implemented in the F* verification system.

F* is a general purpose, proof-oriented programming language which lets the programmer decorate her code with logical specifications about its properties. Its verification system combines different verification algorithms. The F* *typechecker* ensures that the program is well typed at the level of ML types, similarly to other functional programming languages, such as OCaml or F# for example. It also infers and checks the validity of combination of effects used by the programmer. Second, the *normalizer* proceeds to certain reductions to either prove some properties without the help of the external solver (for instance for pure computations over constants), and pre-processes some of the verification conditions to simplify them. Eventually, the program specifications are turned into verification conditions using the weakest pre-condition calculus and discharged to an external automated solver.

As a general purpose functional programming language, F* expressions are close to those of F# or OCaml (see figures 2.1 and 2.6. Its proof-oriented nature however shows in its type system (figures 2.2, 2.3, 2.5, richer than those of the aforementioned languages. In the coming sections, we describe the basic syntax of F* and some of its semantics. For the full semantics of the languages the reader should refer to [9].

2.3 Syntax

Figure 2.1 shows the main constructs of F*'s syntax. The language is functional and its syntax is very close to OCaml's or F#'s.

The F* programming language relies on a system of modules to structure and organize the code. Each module contains a set of declarations (functions,

Syntax	**Description**
(* ... *)	Comment
//	Single line comment
module module_name	Declaration of a new module
open module_name	Imports the visible definitions of module_name
let c = e	Top-level constant declaration
val f : t	Top-level function type declaration
let f a b = e	Top-level function declaration
let rec f a b = e	Recursive function declaration
type x = e	Top-level type declaration
let v = e in ...	Variable assignment or declaration
f a b	Function call or partial application
fun a b -> e	Anonymous function
e ;	Single line statement (expression e returns unit)
=, <>	Equality operators
if e then e' else e''	if-then-else control statement
begin ... end	Scoped block
match v with | $v1$ -> ... | $v2$ | $v3$ -> ... | _ -> ...	Pattern matching (switch-like control statement)

Figure 2.1: F* general syntax

types or constants) which can either be exposed outside the module (the default), or remain private to the module's implementation (via the private keyword). F* also offers the possibility to split a module into an interface (a *moduleName.fsti* file) and its implementation (a *moduleName.fst* file), in which case all the declarations in the *.fst* implementation file are private while the declarations in the interface *.fsti* file are public. The declarations of the implementation file are used to verify to code of the module and later run it. In any case, all F* modules start with the module keyword followed by the name of the module and end at the end of the file.

In order to make those declarations visible from other modules, programmers can either use the fully qualified module names, or import entire modules. To that intent, F* has a concept of namespace: a module name is built using a sequence of strings, starting with a capital letter and separated by dots. For instance, This.Is.A.Module is an acceptable F* module name. To make all the declarations of a module visible in another module without fully qualifying them, one can *import* the module, using the open directive. This will effectively make all the public declarations of that module within scope in the new module. Note that there is no overloading in F*, only shadowing. Hence, if two modules provide functions with identical names, the last import with

be the one visible. It is also possible to import partial namespaces, in which case only the non-imported part will have to be specified. Eventually, modules can be locally renamed to use shorter notations. As an example, if the This.Is.A.Module module contains a function f, to call is from another module the F* programmer can either use:

```
let v = This.Is.A.Module.f x
```

```
open This.Is.A.Module
let v = f x
```

```
open This.Is
let v = A.Module.f x
```

```
module M = This.Is.A.Module
let v = M.f x
```

Modules themselves are composed of a sequence of declarations. The ordering of those declaration matters: only the preceding ones are made available to the following ones. In a style similar to OCaml, function declarations are composed of a val declaration which declares the type of the function, and a let body which contains the actual code of the function. In the absence of a val declaration, the F* system will rely on type inference and the type annotations in the let body to determine the type of the object. A type keyword can be used to declare a new type. The if-then-else and the pattern matching are those traditionally available in functional languages. For let body function declarations, the rec keyword must be specified when the function is recursive. F* has no notion of indentation for scopes and blocks. Those are delimited either by parentheses or begin ... end blocks.

As a concrete illustration, the listing below shows the syntax of the recursive definition of the append function on lists.

```
module FStar.List.Tot.Base

[...]

val append: list α→ list α→ Tot (list α)
let rec append x y = match x with
  | [] → y
  | a::tl → a::append tl y
```

The function is comprised of the separate val and let declarations. The val declaration is straightforward: the append function takes two lists of elements of some type α and returns a new list of the same type. The let body illustrated

Type	Description
int (or $\mathbb{Z}$)	Mathematical unbounded integer
nat (or $\mathbb{N}$)	Natural integer
pos	Strictly positive integer
bool	Boolean
b:t{P(b)}	Refinement type
'a -> 'b -> 't	Lambda type
{ x : t_x; ... ; z : t_z }	Record type
| $Const_1$: x_1:'a -> ... -> x_m:'a_m -> 't	
| ... | $Const_n$: y_1:'b_1 -> ... -> y_n:'b_n -> 't	Sum type
M 't ...	Computation type

Figure 2.2: F* general types

the use of pattern matching: matching on either an empty of a non-empty list. The function is recursive, hence the rec keyword.

2.4 Types

Lambdas, binders and applications F* being a functional language, functions are first order values. The syntax λ(b$_1$) ... (b$_n$) → t introduces a lambda abstraction from binders b$_i$ to t. Binders are of the form x:t for binding a variable x to type t. A binding occurrence may be preceded by an optional #-mark, indicating the binding of an implicit parameter. Applications are written using juxtaposition, as usual. Implicit parameters are not mandatory in application as they are automatically inferred by the type system. Should the inference algorithm fail, those arguments can be inserted also using a #: f #i x ....

Arrows Function types are written b → m t. The variable bound by b is in scope to the right of the arrow. When the co-domain does not mention the formal parameter, the name of the parameter may be omitted. For example, we may write int → m int. For now we will consider that m if the effect Tot are pure, side-effect free and terminating computations.

So, the polymorphic identity function has type #a:Type → a → Tot a for instance.

Inductive types Aside from arrows and primitive types like int, the basic building blocks of types in F* are recursively defined indexed datatypes. For example, we show below an inductive type that defines polymorphic lists.

```
type list (α:Type) =
  | Nil : list α
  | Cons : hd:α → tl:list α→ list α
```

The type of each constructor is of the form $\mathsf{b}_1 \to ... \to \mathsf{b}_n \to \mathsf{T}\ \tau_1\ ...\ \tau_m$, where T is type being constructed. This is syntactic sugar for $\mathsf{b}_1 \to ... \to \mathsf{b}_n \to \mathsf{Tot}\ (\mathsf{T}\ \tau_1\ ...\ \tau_m)$, i.e., constructors are total functions.

Given a datatype definition, F* automatically generates a few auxiliary functions: for each constructor C, it provides a *discriminator* C?; and for each argument a of each constructor, it provides a *projector* C?.a. We also use syntactic sugar for records, tuples and lists, all of which are encoded as datatypes. The programmer directly writes fixpoints and general recursive functions, and a semantic termination checker ensures consistency.

Refinement types A refinement of a type t is a type $\mathsf{x}{:}\mathsf{t}\{\phi\}$ inhabited by expressions $\mathsf{e} : \mathsf{Tot\ t}$ that additionally validate the formula $\phi[e/\mathsf{x}]$. For example, F* defines the type $\mathsf{nat} = \mathsf{x}{:}\mathsf{int}\{\mathsf{x} \geq 0\}$. Using this type, we can write the following program:

```
let abs : int → Tot nat = λn → if n < 0 then −n else n
```

Unlike strong sums $\Sigma x@t.\phi$ [112] in other dependently typed languages, F*'s refinement types $\mathsf{x}{:}\mathsf{t}\{\phi\}$ are subtypes of t (as such, they more closely resemble predicate subtyping [109]); for example, $\mathsf{nat} <: \mathsf{int}$. Furthermore, $\mathsf{n}{:}\mathsf{int}$ can be implicitly refined to nat whenever $\mathsf{n} \geq 0$. Specifically, the representations of nat and int values are identical—the proof of $\mathsf{x} \geq 0$ in $\mathsf{x}{:}\mathsf{int}\{\mathsf{x} \geq 0\}$ is never materialized. As in other languages with refinement types, this is convenient in practice, as it enables data and code reuse, proof irrelevance, as well as automated reasoning.

A new subtyping rule allows refinements to better interact with function types and effectful specifications, further improving code reuse. For example, the type of abs declared above is equivalent by subtyping to the following refinement-free type:

```
x:int → Pure int (requires true) (ensures (λ y → y ≥ 0))
```

We also introduce syntactic sugar for mixing refinements and dependent arrows, writing $\mathsf{x}{:}\mathsf{t}\{\phi\} \to \mathsf{m\ t}$ for $\mathsf{x}{:}(\mathsf{x}{:}\mathsf{t}\{\phi\}) \to \mathsf{m\ t}$.

Refinement types are more than just a notational convenience: nested refinements within types can be used to specify properties of unbounded data structures, and other invariants. For example, the type list nat describes a list whose elements are all non-negative integers, and the type ref nat describes a heap reference that always contains a non-negative integer.

Expression	Description
==>	Logical implication
<==>	Logical equivalence
/\, \/	Logical conjunction and disjunction
forall (x:t) ... (x':t'). P x ... x'	Universal quantification
exists (x:t) ... (x':t'). P x ... x'	Existential quantification
assert(P)	Asserts that the logical predicate P is correct
assume(P x)	Assumes the logical predicate P
assert_norm(e)	Asserts that the expression e is true using the F* normalizer
abstract e	Abstract declaration, which definition is hidden from the solver

Figure 2.3: F* specification and proof expressions and operators

2.5 Proofs

Logical specifications The language of logical specifications ϕ and predicate transformers wp is included within the language of types. F* provides syntactic sugar for the logical connectives $\forall$, $\exists$, $\wedge$, $\vee$, $\Longrightarrow$, and $\Longleftrightarrow$, which can be encoded in types. These connectives are also overloaded for use with boolean expressions—F* automatically coerces booleans to Type as needed.

The core of the F* programming language is pure (effect-free) and functional. This core of the language is usable for concrete code, which is the code intended to be compiled to a target language (natively F# and OCaml) and run, as well as for specifications, which are all the runtime irrelevant annotations provided around concrete code to explicit its properties.

Therefore, although it misses some main features of the language (in particular the monadic effects), this pure functional core is already a full-fledged general purpose language which can be used for programming and verification in the same spirit as Galina for the Coq proof assistant. In particular, this subset of the language is the only one accepted by the F* system for proofs and specifications. In this core, for simplicity, we will only consider the effect Tot which stands for *Total*, i.e. that the code as no side effects and is guaranteed by virtue of typing to deterministically terminate. Another effect is available, the *ghost* effect, which as a first approximation has no difference with the Tot effect except to be erased by the F* compiler, and so we will omit it for now and go back to it later. Also, note that Tot being the default, it may sometimes be omitted.

Intuitively, a function

```
val f: x:α{P(x)} → y:b'{Q(y)} → Tot (z:γ{R(z)})
```

```
let f x y = ...
```

should be read "f is a function which takes inputs of type α and β respectively satisfying the logical properties P and Q and is guaranteed to return a value z, of type γ such that z satisfies the logical property R".

Abstract declarations Because it is not always relevant to pass information on some implementation details to the SMT solver, F* provides the programmer with the abstract keyword. This keyword can annotate any function or type declaration. In that case, the definition of the object is not passed to the proof system which therefore treats it as an abstract object:

- function: with an abstract function, only the type of the function is passed to the proof system for obvious typechecking reasons, but the let body definition of the function is left opaque;
- type: with an abstract type, no information about the type definition is passed to the proof system, except its kind.

This functionality has two main benefits: the first on is to allow the programmer to create new datastructures with a layer of abstraction which let him control what can be done with the datastructure and thus how it can be compiled for instance. The second benefit is to have control on the amount of details passed to the SMT solver. As it is very sensitive to the proof context, it is often valuable to restrain the details of the modules implementations from being exposed, and rely on explicit lemma calls to add the information to the proof context when necessary.

2.5.1 Illustrative example: lists

A Recursive Data Structure Lists are textbook recursive data structures available in all functional programming languages, F* is no exception. The user is provided with a rich variety of API functions and lemmas she can use either in concrete code to manipulate data or in specifications and proofs to reason about it. In F*, those are defined in the FStar.List.Tot module[2] of the F* standard library.

In F*, the type list is recursive and classically defined as the sum of two constructors: one for the empty list and another one which extends an existing list by placing a new element at its head. A list object is therefore either empty or built from a *head* element and an already existing list, the *tail*. The listing below shows current F* syntax for the type list:

[2] https://github.com/FStarLang/FStar/blob/master/ulib/FStar.List.Tot.fst

```
type list (a:Type) =
  | Nil : list a
  | Cons : hd:a → tl:list a → list a
```

As explained in the previous section, binders, when not used in dependent types, are optional in F*. Therefore, the Cons constructor type is strictly equivalent to Cons: a → list a → list a. Here the hd and tl binders serve as documentation to describe what each field means. They also provide better, more understandable syntax for *projectors*. The Cons constructor bundles two elements, hd and tl. If one wants to retrieve either the head or the tail of a Cons list object, F* provides syntax via the Cons?._i l notation where l is the list argument and _i refers to the i-th field of the constructor. If provided with field names, F* also provides the Cons?.hd l and Cons?.tl l syntax, which makes the use of projectors much more readable. In particular, it avoids confusions with regard to the order of the arguments in the constructor.

As it is common in programming languages, F* supports syntactic sugar [] for the empty list (Nil == []) and :: for the other constructor (Cons hd tl == hd::tl).

List library functions As aforementioned, the standard library module provides many useful functions and lemmas to manipulate and reason about lists. Among those we could cite:

These functions are all pure and implemented solely based on the list type definition. Many of them take advantage of the recursive nature of the list type. For instance, the listing below shows the definition of the length function (which returns the length of the list passed in argument), and the append function, which takes two lists in arguments and returns the concatenation of the two:

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

let rec append (x:list α) (y:list α) : Tot (list α) =
   match x with
  | [] → y
  | a::tl → a::append tl y
```

The definitions of those two list functions are self-explanatory in the following sense: because the functions are pure, their whole definitions — the types and the body of the let — are encoded to the SMT solver which can then reason about what they do. Because of that, *intrinsic* refinements, although they may be useful in some cases, are optional and, in a way, redundant.

Intrinsic reasoning The type declaration of the length function contains an implicit refinement. The result of the length function is annotated to be of type nat, which is a refinement of the type int : nat is an alias for type nat = x:int{x ≥ 0}. The following declaration for length:

```
let length (l:list α) : Tot int = (...)
```

would have verified and worked just as well. Adding a refinement to the type definition of a total function and verifying it is called *intrinsic verification*. This proof style has its strengths and weaknesses. The main advantage is that intrinsic refinements are systematically propagated to the context at every function call, without having to rely on external lemmas or the cleverness of the automatic solver. The main drawback however, is that it complexifies the proof environment with hypotheses that may not be necessary, in which case they may hinder the proof process. Furthermore, because the full definition of a total function is encoded to the external solver, intrinsic properties could be locally re-proven as needed.

The intrinsic refinement on the positivity of value returned by the length function is a typical example of a useful case of the intrinsic style. In mathematics and computer science lengths are usually positive. Therefore, in many F* programs the length of a list will be expected to be positive, as a prerequisite to other computations of proofs. Thereon is it legitimate to systematically carry around the positivity property: rather than clobbering the proof context it will remove some verification conditions. We could use a similar method to let F* prove useful properties about the append function. For instance, the F* proof system verifies the following version of append:

```
let append (x:list α) (y:list α) : Tot (z:list α{length z ≥ length x}) =
  match x with
  | [] → y
  | a::tl → a::append tl y
```

However, this property, although useful, is less crucial than the fact that the length function returns only positive values. For instance, we may need the property length z ≥ length y or the more general property length z = length x + length y, in which case the weaker intrinsic refinement shown above is not pertinent. In the standard library, and more generally for large F* developments, we cannot presume of the use that will later be made of functions and thus cannot overload them with unnecessary logical refinement. In such cases, more detailed properties should be left to separate lemmas.

```
1  val lemma_append_length:
2    x:list α→ y:list α→ Lemma (length (append x y) ≥ length x)
3    (* (decreases (length x)) *)
4  let rec lemma_append_length x y =
5    match x with
6    | [] → (* assert(length x == 0);
7                 assert(append [] y == y);
8                 assert(length y ≥ 0); *)
9              ()
10   | hd::tl → (* assert(length x = 1 + length tl);
11                    assert(append x y = hd::(append tl y)); *)
12                lemma_append_length tl y
13                (* ; assert(length (append tl y) = length tl + length y) *)
14   (* | _ → assert(false) // this branch is irrelevant *)
```

Figure 2.4: Example of F* lemma proven recursively

2.5.2 Lemmas

In F*, lemmas are not specific constructs but special instances of total functions: they always return unit. It implies that they are computationally irrelevant. Because of the absence of side effect, a lemma call simply reduces to () ("unit") and thus corresponds to a no-operation and should be ignored. They are, however, very relevant to the proofs. The resulting unit value carries a logical refinement which gets added to the proof context and then can be used by F* and the external solver.

Extrinsic style Going back to the previous example of the append function that shows that the length of the result is greater than the length of first argument, because the property is too specific we now want to prove the property *extrinsically*. This means that we want a lemma which, from the arguments of the append function, guarantees that the result of the append function satisfies the said property. The listing below illustrates how to write such a lemma:

This example lemma also constitutes a good illustration of the recursive proof process in F*. The lemma verifies within a couple milliseconds. However, it is worth noticing that although it seems like a very simple lemma to prove, there are a few subtleties that the F* verification system has to handle.

Termination check As mentioned before, lemmas (with the Lemma syntax notation) are syntactic sugar for Tot unit. Therefore, the body of any lemma must be proven to terminate. In our example, the F* verification system is able to determine automatically that the length of the x argument strictly decreases with each recursive call. Given that the length is always positive, the sequence of calls is guaranteed to terminate with a last call where the x argument is Nil,

and thus return. In more complex cases, the F* verification system may need additional guidance to figure out how to prove termination. To that intent, a decreases clause can be added at the end of a recursive lemma or function type declaration (commented out because unnecessary in our example) which will specify what has to be proven to strictly decrease. Of course, the argument of the decreases clause has to be of a type on which a total order is defined — a condition satisfied in the example by the natural integer type.

Initial step This lemma can easily be proven by induction. The first step is to show that in the initial case — if the x argument is the empty list, which is the only non-inductive case — the condition is satisfied. Because our example is simple enough, F* does not need any guidance to prove that the length property holds in this particular case. In more complex settings however, F* may struggle and require some help. The inlined comments on lines 6-8 illustrate how to provide the verification system with extra information and *guide* the proof. These assert calls are translated by F* into intermediate verification conditions which are discharged to the external solver for verification, and then added to the proof context. Intuitively, instead of letting the solver dwell into an unguided search, assert is a way to direct it by adding specific properties known by the programmer to be useful and potentially easier to prove to its context and which, hopefully, will be useful for later proofs.

Inductive step The complexity in an inductive proof typically resides in the induction step. Just like in mathematics, in order to perform this demonstration the F* system assumes that the goal holds for all steps prior to a step n, arbitrarily chosen (different from the initial step) and attempts to demonstrate that it holds for the chosen step. This is baked into the recursive call of the lemma: because termination is provable and the initial case as well, it is sound to recursively call the lemma, which provides the goal of the lemma for any *smaller* argument. Namely, that the length of the concatenation of the tail of x and y is greater than the length of the tail of x. Given the definition of append when the first argument is a Cons list, F* can prove that the length property is indeed satisfied. Note that similarly to the initial step, many of the proof steps are automated by F* and the external SMT solver. This automation feature is one of the key strengths of the language as it saves a lot of time and annotation effort from the programmer. However, in order to make the proofs more robust, faster, or simply to go through, it is sometimes valuable to add a few extra annotations, examples of which are given in comment in this inductive step.

Exhaustive pattern matching One last important point the verification system has to tackle is the exhaustiveness of the pattern matching: in order for the proof to succeed the verification system has to ensure that all cases

have been taken into account. Here we match exactly against the two different list constructors, without constraining them, so proving that all cases have been handled is straightforward for F*. It may not always be so. If not, it may be valuable to use reasoning by contradiction. Here, if we uncomment the third branch (with the wildcard "_"), we can prove that reaching that branch implies that ⊥ holds and thus that it is impossible, meaning that the pattern matching was, indeed, exhaustive.

Visibility and syntax Lemmas bodies are of no interest to the verification system once they are proven. The purpose of lemmas is to introduce new hypothesis into the proof context, not to pollute it with unnecessary proof steps. Therefore, the F* system will only treat Lemmas as abstract. Also, although the syntax in the example is the most used one, F* offers specific desugaring to lemmas to simplify degenerated cases. For instance, an effect signature Lemma (requires (⊤)) (ensures (P x)) may be written more concisely Lemma (P x).

Calling a lemma Because lemmas are total functions, calling them in F* code is similar to calling any other function. For functions which return unit, F* offer syntactic sugar where the function gets called like in a statement, with a semi-colon at the end and without explicit let-binding. The let _= lemma_call () in ... and lemma_call (); notations are semantically strictly equivalent. The listing below illustrates how to use a lemma in a function's body:

```
let test () =
  let x = [1; 2; 3] in
  let y = [4; 5] in
  let z = append x y in
  (* assert(length z = length x + length y); // fails *)
  lemma_append_length x y;
  (* assert(length z = length x + length y) // succeeds *)
```

The lemma call introduces the property attached to the ensures clause into the proof context, thus making it available to the solver for future goals.

These few notes and examples are meant to give the reader a flavor of the proof process when programming in F*. The system could be described as *semi-automated*. The combination of the F* typechecker and the external SMT solver will handle most of the simple — sometimes even complex — cases. Nonetheless, because full automation for such proofs is still an active area of research and may require an arbitrary amount of time for the SMT solver, F* offers ways to subdivide goals and guide the solver through the proof process, a bit in the spirit of proof assistants.

Effect	Description
Pure 't pre post	Effect of side-effect free, terminating computations
Tot 't	Effect of pure computations with trivial pre and post-conditions
Lemma pre post	Effect of lemmas
Div 't pre post	Effect of side-effect free computations which may not terminate
ST t' pre post	Effect of stateful computations
Ghost t' pre post	Effect of computationally irrelevant computations

Figure 2.5: F* main effects

Syntax	Description
ref v	Declaration of a reference pointing to value v
!r	Dereferences (reads) the value pointed by reference r
r <− v	Assigns v to reference r

Figure 2.6: F* stateful syntax

2.6 Effects

Computation types Computation types m t have the form M t $\tau_1 \ldots \tau_n$, where M is an effect constructor, t is the result type, and each τ_i is a term (e.g., a type or an expression). For primitive effects, computation types have the shape M t wp, where the index wp is a predicate transformer. We also use a number of derived forms. For example, the primitive computation-type PURE (t:Type) (wp:PURE.WP t) has two commonly used derived forms, shown below. For terms that are unconditionally pure, we have already introduced Tot in the previous sections:

```
effect Tot (t:Type) = PURE t (λ post → ∀x. post x)
```

When writing specifications, it is often convenient to use traditional pre- and post-conditions instead of predicate transformers—the abbreviation Pure defined below enables this.

```
effect Pure (t:Type) (p:PURE.Pre) (q:PURE.Post t)
       = PURE t (λ post → p ∧ ∀x. q x ⟹ post x)
```

For better readability, we write Pure t (requires p) (ensures q)≜Pure t p q; "requires" and "ensures" are semantically insignificant.

The functional core of F* is well suited for proofs and specifications. However, apart from the machine integers which are at the core of cryptography,

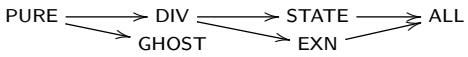
the other data structures we presented in the previous sections are not always ideal for programming. For instance the best performance is the main goal.

The main drawback of lists for instance is that their recursive nature makes the complexity of accessing their elements linear in the depth of the element in the list, while ideally these accesses would be constant-time. Sequences have a similar downfall in that they are immutable, which implies that all returned sequences are fresh values and that the runtime system has to deal with a considerable number of allocations and de-allocations automatically.

A key strength of F* is its ability to manipulate and reason about effects. F* effects are diverse and customizable and thus can adapt to the need of the programmer. Because this work focuses on reference implementations of cryptographic code, the rest of this work will focus mainly on the *state* effect, which takes changes to the program's memory into account, and its several instantiations in F*. But of course the F* effect system contains many other effects worth notice but less relevant for this work (such as divergence or exception handling), we refer the reader to the main F* papers [115, 116, 9] for more details on those.

2.6.1 Lattice of effects

F* defines a lattice of primitive effects, which can be customized and refined by the programmer. Although all of F* programs could be written in the ALL effect, which encompasses statefulness, divergence and exceptions, it is convenient to use more specific monads for computations which do not exhibit all the aforementioned effects. For instance, it would be unnecessarily heavy to propagate state constraints for pure computations. Furthermore, as the verification conditions are automatically generated from the weakest precondition predicate transformer, sequential computations with complex effects lead to an exponentially large verification conditions while PURE computations for instance are straightforward to compose.



F* primitively defines six different effects that are hierarchically placed over a lattice: PURE computations which are terminating and have no side-effects, DIV computations which do not exhibit side-effects but might not terminate, GHOST for computationally irrelevant code, STATE for stateful computations, EXN for exception throwing computations and ALL for computations that may exhibit all effects but the GHOST one. Effects lower on the lattice can be lifted to those placed higher, while the converse is forbidden. At the very top of the lattice is an implicit extra effect `T` which is for mutually incompatible effects—it is implicit because F* rejects all computations in that effect so it cannot be used concretely. The lifting functions from an effect to another higher in

```
PURE.Post a = a → Type
PURE.Pre = Type
PURE.WP a = PURE.Post a → PURE.Pre
PURE.return a (x:a) (post:PURE.Post a) = post x
PURE.bind a b (wp1:PURE.WP a) (wp2: a → PURE.WP b) : WP b =
  λ(post:PURE.Post b) → wp1 (λ x → wp2 x post)
```

Figure 2.7: F* definition of the PURE monad

the lattice are already defined in F* and the verification system performs the lifting automatically so that, for example, PURE functions can freely be used STATE code without explicit lifts being required.

PURE At the bottom of the lattice is PURE, the effect terminating, side effect free computations. The post-condition is indexed by the returned value of the computation and the weakest pre-condition calculus for PURE programs is defined below:

Since effects can be lifted to effect higher than them in the lattice, PURE can composed with any other F* effect. In particular, it can be used in specifications, which F* constraints to be pure terms.

STATE Another important effect is the STATE effect. Such computations are *stateful*: they carry an implicit state which they can read from and update. This implicit state therefore parametrizes the pre- and post-conditions of any STATE computations, as shown below:

```
STATE.Post a = a → state → Type
STATE.Pre = state → Type
STATE.WP a = STATE.Post a → STATE.Pre
STATE.return a (x:a) (post:STATE.Post a) = λs → post x s
STATE.bind a b (wp1:STATE.WP a) (wp2: a → STATE.WP b) : WP b =
  λ(post:STATE.Post b) s0 → wp1 (λ x s1 → wp2 x post s1) s0
```

Figure 2.8: F* definition of the STATE monad

Intuitively the implicit state represents the memory of the program. The pre-condition depends on the state of the memory when the function is called, while the post-condition specifies how the state was updated. The definition of the weakest precondition monadic calculus on the PURE and STATE effects shows that STATE can indeed supersede PURE: any pure computation can be

seen as a stateful one leaving the state untouched. PURE is therefore a sub-effect of STATE and the automated effect lifting performed by F* is guided by:

```
PURE.lift_state a (wp:PURE.WP a) : STATE.WP a =
    λ(post:STATE.Post a) s → wp (λ x → post x s)
```

GHOST The GHOST effect defines terms that are not computationally relevant. GHOST is on a separate branch of the lattice and cannot be lifted to any other effect but the implicit top T which is rejected by the verification system. This mechanism guarantees that GHOST code is never used in computationally relevant code, but only in the proof world. Therefore, when a valid F* program is compiled to executable code, GHOST code can be safely erased, having provable non-interference with terms with any other effect. This feature is useful to manipulate proof witnesses in concrete code for instance. These proof witnesses are helpful for intermediate lemmas and proofs steps, but need to be erased at compile time. This specificity aside, GHOST is identical to PURE in its definition and the second effect to be usable in F* specifications.

Other Effects F* also exposes primitive effect for pure, diverging computations (DIV), exception throwing computations (EXN) and the standard ML effect of OCaml programs (ALL).

Syntactic sugar and definition of new effects Effectful signatures which effects are of the form M a wp where a is the return type and wp the weakest precondition predicate transformer are not ideal to make specifications immediately explicit to human readers. Rather, it is more intuitive to write the specifications in the style of *contracts*. Contracts expose separate pre- and post-conditions. For pure computations, the pre-condition only depends on the bounded arguments while the post-condition is also parametrized by the result of the computation. For stateful computations the pre-condition depends on the state of the program when the function is called while the post-condition is expressed with regard to the initial state, the result and the final state. This syntactic sugar around effects can be further refined by the programmer. Below we give examples for Pure and ST. The Tot effect corresponds to a degenerated case of PURE where the weakest precondition predicate transformer is trivial (the pre-condition is ⊤ and the post-condition λres → ⊤).

For better readability, F* provides syntactic sugar for those pre- and post-conditions, identifiable to their respective requires and ensures keywords. Following those notations, a typical F* stateful function has a signature of the form

```
effect Pure (a:Type) (pre:pure_pre) (post:pure_post a) =
        PURE a (λ (p:pure_post a) → pre ∧ (∀ (x:a). post x ⟹ p x))

effect Tot (a:Type) = PURE a (λ p → ∀(x:a). p x)

effect ST (a:Type) (pre:st_pre) (post: (heap → Tot (st_post a))) =
       STATE a (λ (p:st_post a) (h:heap) → pre h ∧ (∀ a h1. post h a h1 ⟹
p a h1))
```

Figure 2.9: Examples of F* effect abbreviations

```
val f: x1:t1 → ... → xn:tn → ST t
 (requires (λ h → pre x1 ... xn h))
 (ensures (λ h0 r h1 → Post x1 ... xn h0 r h1))
```

where `h` and `h0` correspond to the program's state when the function is called, `r` is the result of the computation and `h1` denotes the state of the program when the function returns.

2.6.2 Customizable memory model

As illustrated in the previous section, F* `STATE` effect carries an implicit state which records persistent changes to the program's memory. While the effect itself is primitive to the language, the type of the actual *state* object is freely customizable by the programmer. The complete type signature of the `STATE` effect in F* is given below:

```
STATE_h (heap:Type) : result:Type → wp:st_wp_h heap result → Effect
```

`STATE_h` takes the type `heap` of the state as a parameter on which the weakest precondition predicate transformer also depends. This `heap` type fully characterizes the memory layout considered. The F* standard library proposes two different models but as we will see, new `heap` types can be defined to model more specific memory disciplines.

Heap The default memory model F* exposes with the `STATE` effect is called `Heap`. It is a simple representation of the memory as a map from references (keys) to values, described in figure 2.10. Its *target* memory model is the OCaml and F# memory state, languages to which F* code can be compiled by default. In those languages memory management is completely automated. The runtime system takes care of the allocations and relies on a garbage collector to automatically reclaim memory locations that shall no longer be used.

```
val heap :Type u#1
val emp :heap
val ref (a:Type0) :Type0
val addr_of: #a:Type0 → ref a → GTot nat
val is_mm: #a:Type0 → ref a → GTot bool
val compare_addrs: #a:Type0 → #b:Type0 → r1:ref a → r2:ref b →
Tot (b:bool{b = (addr_of r1 = addr_of r2)})
val contains: #a:Type0 → heap → ref a → Type0
val unused_in: #a:Type0 → ref a → heap → Type0
let fresh (#a:Type) (r:ref a) (h0:heap) (h1:heap) =
  r `unused_in` h0 ∧ h1 `contains` r
val sel: #a:Type0 → heap → ref a → GTot a
val upd: #a:Type0 → heap → ref a → a → GTot heap
val alloc: #a:Type0 → heap → a → mm:bool → GTot (ref a * heap)
val free_mm: #a:Type0 → h:heap → r:ref a{h `contains` r ∧ is_mm r} → GTot heap
let modifies_t (s:tset nat) (h0:heap) (h1:heap) =
  (∀ (a:Type) (r:ref a).{:pattern (sel h1 r)}
                         ((¬ (TS.mem (addr_of r) s)) ∧ h0 `contains` r) ⟹
sel h1 r == sel h0 r) ∧
  (∀ (a:Type) (r:ref a).{:pattern (contains h1 r)}
                         h0 `contains` r ⟹ h1 `contains` r) ∧
  (∀ (a:Type) (r:ref a).{:pattern (r `unused_in` h0)}
                         r `unused_in` h1 ⟹ r `unused_in` h0)
let modifies (s:set nat) (h0:heap) (h1:heap) = modifies_t (TS.tset_of_set s) h0 h1
```

Figure 2.10: API of the Heap memory model

Because the runtime of the target language automates everything, the constraints which need to be enforced to guaranty the memory safety of the program are minimal. Namely, one has to make sure that references which are dereferenced point to valid memory locations. To make this process simple the logic of Heap relies on the fact that the mapping from references to values is monotonic: since the runtime system will automatically free unused variables and the programmer has no control over the memory management it is not necessary to provide a *freeing* mechanism of the F* level. From a specification perspective, a freshly allocated reference will thus remain accessible for the whole existence of the program (although of course the garbage collector can reclaim it sooner). As a corollary, as only a specific API call of Heap can return fresh valid references, a program has no way to generate such references and thus any existing reference is guaranteed to be valid, even in the absence of explicit predicates over the corresponding state.

HyperHeap The Heap memory model has the advantage to be extremely simple to use and natively produces safe OCaml or F# code. Its main draw-

```
abstract let rid = list (int * int)

let reveal (r:rid) : GTot (list (int * int)) = r

abstract let color (x:rid): GTot int =
  match x with
  | [] -> 0
  | (c, _)::_ -> c

type t = Map.t rid heap
```

Figure 2.11: Type definition of the HyperHeap memory model

back is that it has no built-in features for reasoning about which parts of the memory have been updated, and which have not. Intuitively, the memory, which is a single map, gets modifies as a whole when it is updated and the programmer is responsible for making explicit which references (keys) have been updated and which have not. This leads to scalability issues in the presence of many simultaneously modified references. For instance, suppose that some program has stored some data under some reference r, which should not be changed throughout the computation. With the Heap model, for every update to the state the programmer will have to prove that the modified reference is different from the r reference. This adds a significant proof burden: the invariant has to be carried everywhere. To tackle this issue, F* natively offers a second memory model called HyperHeap. Informally HyperHeap divides Heap into *regions*. The idea is that different regions are either nested or distinct. References from two distinct regions automatically enjoy separation: updating a region is guaranteed to leave distinct regions unchanged. Therefore, separation between references is lifted to separation between regions. Because those regions can be arbitrarily subdivided into further subregions, separation can be as coarse or as precise as needed.

In practice HyperHeap is implemented as a map of Heap objects, on top of which a tree structure is enforced, which represents the hierarchy of regions.

2.7 Proofs with effects

2.7.1 Example: swapping references

The example on figure 2.12 illustrates how to verify stateful code in F*.

In this pre- and post-condition syntax the specification of the swap function reads as follows: assuming that x and y are valid memory locations—the back ticks allows infix notations for 2-ary functions f x y ⟺ x `f` y—swap guarantees than it modified only the reference x and the reference y, and that the values

```
let swap (x:ref int) (y:ref int) : ST unit
  (requires (λ h → h `contains` x ∧ h `contains` y))
  (ensures (λ h0 _h1 → h0 `contains` x ∧ h1 `contains` x
                     ∧ h0 `contains` y ∧ h1 `contains` y
                     ∧ sel h1 x = sel h0 y ∧ sel h1 y = sel h0 x
                     ∧ modifies (Set.union (Set.singleton (addr_of x))
                                           (Set.singleton (addr_of y))) h0 h1))
  = let h0 = ST.get() in
    let tmp = x inx := y;
    y := tmp;
    let h1 = ST.get() in
    assert(Heap.sel h1 x == Heap.sel h0 y ∧ Heap.sel h1 y == Heap.sel h0 x)
```

Figure 2.12: Example of F* code and specification in the STATE monad

of the references have been swapped: in the resulting state `h1` `x` points to the initial value of `y` and vice-versa.

In this example, the `STATE` monad is parametrized by the standard F* `Heap`.

```
assume val alloc: #a:Type → init:a → ST (ref a)
                                        (λ h → ⊤)
                                        (λ h0 r h1 → (r, h1) ==
                    let r = { addr = h.next_addr; init = x; mm = mm } in
                    r, upd #a h r x)

assume val recall: #a:Type → r:ref a → STATE unit
                                  (λ 'p h → Heap.contains h r ⟹ 'p () h)

assume val read: #a:Type → r:ref a → STATE a
                                  (λ 'p h → 'p (sel h r) h)

assume val write: #a:Type → r:ref a → v:a → ST unit
                                        (λ h → ⊤)
                                        (λ h0 x h1 → h0 `contains` r ∧ h1==upd h0 r v)

assume val get: unit → ST heap (λ h → ⊤) (λ h0 h h1 → h0==h1 ∧ h==h1)
```

Because the `Heap` is the simplest memory model, there is little logic behind it and it provides an easy-to-use API:

- allocate (`alloc`) a fresh reference on the heap pointing to an initial value `init`;
- recall (`recall`) that a reference exists in memory: this is a meta argument that leverages on the fact that since the only function that may return a new reference is the `alloc` function and since there is no way to *free* a reference, if one is given a reference as argument then it must be that the heap contains it;

```
let test_swap (w:ref int) : ST int
  (requires (λ h → ⊤))
  (ensures (λ h0 r h1 → r = 1))
  = let x = alloc 0 in
    let y = alloc 1 in
    (* swap w x; // fails to typecheck *)
    swap x y;
    x
```

Figure 2.13: Example of F* code and specification in the STATE monad

- dereference (read) a reference to access the value it points to;
- update (write) a reference, replacing its previous value with a new one;
- retrieve (get) the underlying implicit heap object to reason about it in proofs and specifications.

Each of these API functions carries predicates on the transformations that occur in the underlying heap. In the body of the swap function first the initial value of x is retrieved and stored in the tmp variable. The ! ("bang") operator is usual syntactic sugar when dereferencing, i.e. !x is syntactic sugar for FStar.ST.read x. The read function accesses a value in memory but does not modify it. The next call to := which again is usual syntactic sugar for assignments (x := v is sugar for FStar.ST.write x v) which specifies that the underlying heap that models memory has changed: the memory (viewed as a map) has been updated at its key x with the value read at key y. Eventually the initial value of x (which now points to the same value as y) which was stored in tmp is written in reference y resulting in a perfect swap of the references.

Because it may be useful to get access to the actual memory to reason about it is proofs, the FStar.ST module which defines the STATE effect parametrized by the FStar.Heap.heap object, defines a proof only primitive, get. It returns the heap object at the point in the program where the primitive is called, thus returning a similar value to those in the pre and post-conditions of the stateful functions. Here given that we retrieve the memory at very beginning and very end of the function body, h0 and h1 correspond to the same objects both in the type declaration (val) and in the let definition.

The call to assert shows how to use those objects in the body in the function just as they can be manipulated in the specifications.

The test_swap function in figure 2.13 illustrates how to leverage on an existing proof. The w reference passed to the test_swap function has no specification. Hence the F* system cannot prove that it points to a valid memory location

Module	Description
Int<N>, UInt<N>	Machine integer libraries
List	Lists
Seq	Immutable sequences
Array	Mutable arrays

Figure 2.14: F* standard library modules

and will refuse to read or to use it as an argument for the swap function (illustrated on line 6 of the example). The references x and y however are allocated in the body of the function, the system can thus prove that they are appropriately in memory when swap is called. Then, although the body of the swap function is not encoded to the SMT solver, it can rely on its specification to reflect on how the function call impacted the memory. As the values of x and y have been swapped, F* is able here to automatically prove that the value returned by the function is 1.

These examples are simple but they show have the proof system works, and how the programmer can guide the SMT solver so that the proofs are run in a semi-automated setting. The whole process is incremental: be it in the context of PURE or STATEful computations, the verification system ensures that the specification of a constant or function matches its actual implementation, and then let the programmer rely on those to prove things on code which calls into this function. The coming chapters will go more in depth into the proof process for larger, more interesting programs, but at the end of the encoding to the solver, it all boils down to the core components and mechanisms which were described here.

2.8 Libraries

2.8.1 Sequences

An abstract module The previous list examples illustrated how to use F* with functions which definitions are transparent. As explained, in that case the system encodes both the type signature and the let body definition in the proof context, allowing the solver and the programmer to freely reason about both. This feature is not always desirable. The *sequence* (from the FStar.Seq standard library module) data structure for instance does not follow this pattern. In F*, sequences are immutable arrays for which the standard library offers the usual operators: create to build a new immutable array, index to access one of its elements, upd to update an element with a new value, append to concatenate two arrays and slice to get only a slice of the array. As one may notice, those library functions are identical to those available for lists. There is however a key difference in the intended representation of the data structure:

```
type seq (a:Type)

val length: #a:Type → seq a → Tot nat
val index: #a:Type → s:seq a → i:nat{i < length s} → Tot a
val create: #a:Type → nat → a → Tot (seq a)
val init: #a:Type → len:nat → contents: (i:nat { i < len } → Tot a) → Tot (seq a)
val of_list: #a:Type → list a → Tot (seq a)
val createEmpty: #a:Type → Tot (s:(seq a){length s=0})
val upd: #a:Type → s:seq a → n:nat{n < length s} → a → Tot (seq a) (decreases (length s))
val append: #a:Type → seq a → seq a → Tot (seq a)
abstract val slice: #a:Type → s:seq a → i:nat → j:nat{i ≤ j && j ≤ length s} →
Tot (seq a) (decreases (length s))

(* Lemmas about length *)
val lemma_create_len: #a:Type → n:nat → i:a → Lemma
  (requires ⊤)
  (ensures (length (create n i) = n))
  [SMTPat (length (create n i))]

val lemma_init_len: #a:Type → n:nat → contents: (i:nat { i < n } → Tot a) → Lemma
  (requires ⊤)
  (ensures (length (init n contents) = n))
  [SMTPat (length (init n contents))]

[...]
```

Figure 2.15: F* sequence type and main function definitions

lists are recursive and offer powerful mathematical ways to inductively reason about. They are, however, not a very efficient data structure. Accessing or updating an element of a list is made slow by the fact that each element is allocated individually and thus the list can only be accessed through its head, by sequentially reading each individual element. Arrays are typically meant to be contiguous blocks of data which aim at making operations such as reading and writing to any of its elements constant-time and independent from the size of the array.

Therefore, the sequence data structure, although its implementation details do rely on the list data structure (internally the `sequence` type is defined as a wrapper around a list) for soundness, the whole module abstracts it away so that only the `val` type signatures are made available to the SMT solver and the module can link at compile time against array-like libraries such as the OCaml `Array` module. To that intent, F* provides a special keyword, `abstract`. Note that lemmas' bodies are never encoded to the proof context, because the F* system natively handles them as *abstract*. From the viewpoint of the programmer, the definitions of the `FStar.Seq` API are shown in figure 2.15.

Because of the abstraction the solver gets no details about the concrete

implementation of those functions. In particular, the structure of the seq type is only defined through its API; the type itself is abstract and has no definition other than its name.

Since the abstract keyword is only effective outside the module, soundness is guaranteed by the fact that the local implementation and proofs are sound, and thus that the exposed val declarations are verified. This mechanism allows for more lightweight modules and, often, a better automation for specific use cases. Indeed, because the developer has full control over the API and lemmas provided, such modules are good candidates for *pattern instantiated* lemmas. Patterns are an SMT solver functionality which essentially serves two purposes. The first one is to help the solver decide when a logical property guarded by a universal or existential quantifier should be instantiated. The second is to allow the solver to trigger the use of certain lemmas when it recognizes certain patterns in the hypothesis or the goals of a verification condition. F* provides the programmer with syntax to add her own patterns of her lemmas, in order to help them be automatically fired on the SMT solver's end.

2.8.2 Machine Integers

For programs in general, and cryptographic applications in particular, unbounded integers and machine integers are central components, both to the concrete, runnable code and to the specifications and proofs. The F* verification system has a native understanding of mathematical unbounded integers ($\mathbb{Z}$). Those are made available to any F* program and benefit from a specific encoding to the SMT solver. On the other hand, machine integers — which fit into a fixed number of bits, usually 32 or 64 — are not native to the language. Instead, they are defined via the F* standard library where the exact behavior of each of their operators is fully specified. Their definitions are built from two other data structures. Machine integers themselves are wrappers around mathematical integers which refinements specify that they fit within appropriate bounds with regard to their signedness and their width. This representation lets us implement the arithmetic operations based on standard modular arithmetic, using F* native mathematical operators. For logical operators (& ^ etc.) however, a second, bit-based representation is required. F* machine integers of n-bits are in bijection with F* bitvectors of size n. The logical operators themselves are defined as the composition of the conversion to and back from bitvectors, and the actual bitvector operators. The bitvectors themselves are defined as specialized instances of sequences of booleans.

Note that machine integers cannot be defined immediately as refinements over mathematical integers because of implicit coercions to mathematical integers. For instance, consider figure 2.18, assuming bytes of 8-bits.

```
let max_int (n:nat) : Tot int = pow2 n - 1
let min_int (n:nat) : Tot int = 0

let fits (x:int) (n:nat) : Tot bool = min_int n ≤ x && x ≤ max_int n
let size (x:int) (n:nat) : Tot Type0 = b2t(fits x n)

(* Machine integer type *)
type uint_t (n:nat) = x:int{size x n}

(* Addition primitives *)
val add: #n:nat → a:uint_t n → b:uint_t n → Pure (uint_t n)
  (requires (size (a + b) n))
  (ensures (λ _ → ⊤))
let add #n a b =
  a + b

abstract val add_underspec: #n:nat → a:uint_t n → b:uint_t n → Pure (uint_t n)
  (requires ⊤)
  (ensures (λ c →
    size (a + b) n ⟹ a + b = c))
let add_underspec #n a b =
  if fits (a+b) n then a + b else magic ()

val add_mod: #n:nat → uint_t n → uint_t n → Tot (uint_t n)
let add_mod #n a b =
  (a + b) % (pow2 n)

(* Casts *)
val to_vec: #n:nat → num:uint_t n → Tot (bv_t n)
let rec to_vec #n num =
  if n = 0 then Seq.createEmpty #bool
  else Seq.append (to_vec #(n - 1) (num / 2)) (Seq.create 1 (num % 2 = 1))

val from_vec: #n:nat → vec:bv_t n → Tot (uint_t n)
let rec from_vec #n vec =
  if n = 0 then 0
  else 2 * from_vec #(n - 1) (slice vec 0 (n - 1)) + (if index vec (n - 1) then 1 else 0)

val inverse_vec_lemma: #n:nat → vec:bv_t n →
  Lemma (requires ⊤) (ensures equal vec (to_vec (from_vec vec)))
        [SMTPat (to_vec (from_vec vec))]
let inverse_vec_lemma #n vec = ()

val inverse_num_lemma: #n:nat → num:uint_t n →
  Lemma (requires ⊤) (ensures num = from_vec (to_vec num))
        [SMTPat (from_vec (to_vec num))]
let inverse_num_lemma #n num = to_vec_lemma_2 #n num (from_vec (to_vec num))

val logand: #n:pos → a:uint_t n → b:uint_t n → Tot (uint_t n)
let logand #n a b = from_vec #n (logand_vec #n (to_vec #n a) (to_vec #n b))
```

Figure 2.16: F* internal definition of the machine integer operators

```
let n = 32

private type t' = | Mk: v:uint_t n → t'
type t = t'

let v (x:t) : Tot (uint_t n) = x.v

let v_inj (x1 x2: t): Lemma (requires (v x1 == v x2)) (ensures (x1 == x2)) = ()

val add: a:t → b:t → Pure t
  (requires (size (v a + v b) n))
  (ensures (λ c → v a + v b = v c))
let add a b =
  Mk (add (v a) (v b))

val add_underspec: a:t → b:t → Pure t
  (requires ⊤)
  (ensures (λ c →
    size (v a + v b) n ⟹ v a + v b = v c))
let add_underspec a b =
  Mk (add_underspec (v a) (v b))

val add_mod: a:t → b:t → Pure t
  (requires ⊤)
  (ensures (λ c → (v a + v b) % pow2 n = v c))
let add_mod a b =
  Mk (add_mod (v a) (v b))
```

Figure 2.17: F* exposed definition of the machine integers

```
1  type byte = b:int{ − 128 ≤ b ∧ b < 128}
2
3  (* This λction does not catch the overflow, undefined for signed words *)
4  let identity (x:byte) : Tot (x':byte) =
5    x + 1 − 1
6
7  let ( + ) (x:byte) (y:byte{− 128 ≤ x + y ∧ x + y < 128}) : Tot (z:byte) =
8    x + y
9
10 (* This λction appropriately fails to typecheck *)
11 let identify' (x:byte) : Tot (x':byte) =
12   x + 1 − 1
```

Figure 2.18: Example of implicit problematic casts

Such casts may be responsible for uncaught overflows. If x is encoded as a simple refinement over mathematical integers, then the proof system will deduce that $(x+1)-1$ also satisfies the refinement, and thus that the operation is correct. However, suppose that $x = 2^7 - 1$, meaning that x is the largest value possible. Then $x+1$ will overflow, overflow which may not be defined for signed integers in the target language, for instance in the C standard. If those integers are to be compatible with a variety of target languages, we have to make sure that every single operation will be checked for overflows. As illustrated on line 7 and 8 of figure 2.18, F* allows the programmer to overload operators, here the addition. However, doing it this way results in a readability issue as it becomes difficult to know whether the native unbounded "+" operator or the machine integer one is used. To avoid that, we rely on a different syntax for machine integer's operators and mathematical operators.

To avoid explicit casts, the solution consists in wrapping those refined integers in a data constructor. Thanks to the wrapper, the machine integers are no longer subtypes of the mathematical integers and thus the mathematical integers' operators are no longer at risk to be implicitly used on machine integers. The only thing that remains is to properly define at each integer's module level the operation appropriately, as on line 3 of figure 2.18.

Mapping to bitvectors The standard library provides a bitvector library which lets the user manipulate sequences of booleans. Although those bitvectors have no concrete representation in F* (they would be compiled to actual sequences of booleans and not to OCaml words), they provide the user with an easy way to reason about logical operators on machine words.

Indeed, while arithmetic operators are naturally represented as modular operations over mathematical unbounded values, there are no such intuitive representations for logical operators over integers. As those are better seen and understood as computations over strings of bits, F* defines a mapping from machine words of n-bits to bitvectors of length n which correspond to the binary representation of the given word. The specification proves that the mapping is invertible and thus defines logical shifts and operators as the transposition of the corresponding operation on bitvectors, but in the machine word's space. The result is a complete, sound and fully specified machine word library which a programmer can use to produce more idiomatic low-level code after compilation. In the absence of machine words, to ensure correctness F* translates mathematical integers into unbounded integers in the target language (for instance into ZArith big integers in OCaml). Depending on the code, this fallback will result in lower performance and potential security issues due to side channels.

```
let rec factorial (x:int{x ≥ 0}) : Tot (y:int{y > 0}) =
  if x = 0 then 1
  else
    x * factorial (x - 1)
```

Figure 2.19: Example of F* recursive code and specification in the PURE monad

2.9 A feel for the SMT encoding

The previous sections introduced some core data structures every F* development relies on and gave a flavor of how the proof process is carried on the programmer's side. The purpose of this section is to present in some more details how the internals of F* operate and interact with the external SMT solver on a simple, pure example.

In 2.19 we show the definition and a specification of the factorial in the F* language. The factorial is defined by induction over natural numbers, as indicated by the rec keyword and the refinement on the argument x:int{x≥ 0}. The function is typed to return a strictly positive integer in the Tot effect. Remember that Tot is not a new effect but merely an abbreviation of PURE when the post condition is trivial ($\top$). Notice that for PURE computations the pre- and post-conditions can be moved to the refinements: pre-conditions will lead to refinements on the arguments while post-conditions will impose refinements on the result. In this example, the factorial computation is typed x:int{x ≥ 0} → Tot (y:int{y > 0}), which could also be read:

$$\text{x:int} \rightarrow \text{PURE (y:int) } (\lambda \text{ post} \rightarrow \text{x} \geq 0 \wedge \forall \text{y. y} > 0 \Longrightarrow \text{post y})\}.$$

Our goal is therefore to prove that for all positive inputs, the result of the factorial is strictly positive. To that intent F* will derive the weakest precondition predicate transformer for this computation, and discharge to the SMT solver a verification condition to ensure that the specification is at least as strong as the weakest derived pre-condition. Here the verification condition is $\forall \text{post. } (\text{x} \geq 0 \wedge \forall \text{y. y} > 0 \Longrightarrow \text{post y}) \Longrightarrow \text{post (factorial x)}$.

To prove the validity of the specification — that the factorial of a natural integer is a strictly positive integer — F* relies on several mechanisms. Because the function is recursive, the proof has to be performed recursively. To that intent F* infers or requires the programmer to specify a lexical order on the arguments of the recursive calls to guaranty that the function will eventually terminate. Because the factorial has only one natural integer argument F* is able to infer that the invariant is that this value strictly decreases with each recursive call and thus will return when 0 is reached. In more complex cases

however, the inference algorithm may fail in which case the programmer can specify the order relation and the arguments on which to verify them.

Then F* will add to the verification condition that the recursive call(s) to factorial x' must verify that x' < x, which is discharged to the solver and easily verified. Eventually, it will assume that the recursive call verifies the specification (i.e. factorial (x–1) > 0 and thus discharges the following verification condition:

$$\forall \mathsf{post}.\ (\mathsf{x} \geq 0 \wedge \mathsf{factorial}\ (\mathsf{x}{-}1) > 0 \wedge \forall \mathsf{y}.\ \mathsf{y} > 0 \Longrightarrow \mathsf{post}\ \mathsf{y}) \Longrightarrow \\ ((\mathsf{x} = 0 \Longrightarrow \mathsf{post}\ 1) \wedge (\mathsf{x} \neq 0 \Longrightarrow \mathsf{post}\ (\mathsf{x} * \mathsf{factorial}\ (\mathsf{x}{-}1))))$$

The solver automatically validates the verification condition. More complex properties will obviously lead to more complicated encodings but the mechanisms remain the same: the weakest precondition calculus directs how to derive the weakest pre-condition of the computation and then the SMT solver checks that the provided pre- and post-conditions satisfy the requirements.

Conclusion

F* is a new, viable alternative to existing formal verification tools. It is a full-fledged programming language, with an active community and ongoing, large projects [3]. Its strength resides in the combination of a standard functional language core and syntax (à la OCaml or F#) with a much richer and more expressive type system. In contrast with most other tools and programming languages, it provides both proof automation via the SMT solver backend and interactive proof mechanisms. Very few types are native to the language; the module system and the support for abstraction allow the programmer to implement custom libraries, thus redesigning types and effects as she sees fit. The automated encoding of the verification conditions to the SMT solver reduces the proof burden while the programmer retains some control over the proof unrolling to help the solver where it struggles. Future work will aim at improving proof automation and reducing the trusted computing base (TCB). SMT solvers have important limitations in specific areas, such as non-linear arithmetic. The F* compiler should be improved to handle more of the proofs (using its normalizer) and to allow other proof backends to compensate for the weaknesses of SMT solvers. The current trusted computing base encompasses the F* compiler, the SMT encoding and the SMT solver itself. We aim to provide a mechanized proof of soundness for the core of F*, although this represents a significant amount of work. In order to remove the SMT solver from the TCB, in future work we will extract proof certificates from the solver's

[3] https://project-everest.github.io

computation, and verify their correctness with external verified tools. That way the solver itself does not need to be trusted and we shall keep the best of both worlds — high automation and high levels of trust.

Chapter 3

An Extensible Bignum Framework for Cryptography

This chapter presents the proof methodology for bignum algorithms in elliptic curve algorithms. Parts of the text are taken from [129].

3.1 A Dire Need for Trust in Cryptography

One major challenge for software verification resides in its need to scale to large, active and evolving projects. Over the past years, several academic verification works have proven to be amazingly successful, some of them tackling large and complex codebases. The CompCert C compiler [93] for instance has been entirely verified in Coq. Other projects [84, 76] have tackled entire kernels. These are concrete evidence that when experts invest enough time and effort, wonders can happen.

Nonetheless, such projects require several expert-months or expert-years to come through and the value in terms of research is often in the original proof of concept, rarely in the ability of the project and formal method tools to be updated and supported for an extended period of time. With the ambition of building long term projects, viable and maintainable by a heterogeneous and not necessarily expert team of developers, in this chapter we explore different paths towards making formal methods and software verification scale more easily.

Verification projects usually tackle security critical software where even the most careful peer reviewing and auditing has proven unable to remove all flaws [1]; previous works on trusted kernels and compilers typically highlight this dire need for more security in key software components. Following this very idea we illustrate a new verification design on cryptographic code, a great candidate for large scale verification frameworks for several reasons.

[1] See the CVEs of the OpenSSL project for instance: https://www.openssl.org/news/vulnerabilities.html

Security critical The first reason why cryptographic code is a good candidate for formal verification is that it is now security critical in almost all our daily communication activities. While this may seem obvious, as the goal of cryptography is to provide additional security guarantees to its user, we argue that broken cryptography is actually worse than no cryptography because it gives the user a false sense of protection and immunity, and which may lead her to adopt behaviors she might not have had, had she not thought the communication channel to be secure.

Widespread use Second, the use of cryptography has spread everywhere over the past years, be it storage, communication channels, access control etc. And of course, the more uses, the more external attackers have to gain at finding flaws to break it; cryptography now represents a large and worthwhile attack surface and should be protected accordingly.

Vulnerable code Third, although they are probably among the most carefully reviewed codebases, cryptographic libraries have a history of critical vulnerabilities, which seem to indicate that, for the level of trust they require, conventional auditing methods are not satisfying.

A moving target Cryptography, very much like computer science in general, is evolving at a very fast pace. Industry constraints impose that the code is always faster and more optimized while new primitives are regularly standardized and old primitives deprecated. As such it is difficult to invest a large amount of work from expert developers to implement and verify a specific version of a specific primitive for a specific platform, knowing that it may go to waste in only a couple years.

Mathematically precise A significant part of modern cryptography relies on simple and well formalized mathematical objects, such as large finite fields or geometrical constructions. These are great for verification inasmuch as the mathematical specification of what the program's execution should be is orders of magnitude shorter than the actual code, and much cleaner and easier to review.

3.2 Prime fields Arithmetic

In computer science, cryptography is a gigantic field and current methods are not mature enough to efficiently verify a significant share of it. OpenSSL contains hundreds of thousands of lines of unverified cryptographic code [2], and it is but a small portion of the cryptography commonly used in our daily

[2] https://github.com/openssl/openssl

activities. Cryptographic primitives and their mathematical grounds are as diverse as the creativity and usages devised by cryptographers and developers can be. Hence, some of those primitives share very little from an algorithmic perspective. From a standardization standpoint, the different primitives could be split into two (and maybe somewhat intersecting) categories:

1. **The algorithm-based primitives** are those which, although their design has been guided by mathematical constraints, are specified in the literature directly as algorithms to be implemented. For those, the code or pseudo-code from the specification is the reference, and typically developers do not have to give the concrete implementation too many thoughts because the literature is transparent about it. Most cryptographic symmetric ciphers or hash functions fall under this category. Some optimizations are still possible of course on the number of memory reads and writes, using platform specific features such as vectors etc., but those remain quite marginal;

2. **Mathematics-based primitives** are standardized as mathematical objects, which makes their specifications more abstract and leaves the programmer with many more implementation decisions to make, which would typically depend on her language of choice, target platform, efficiency constraints etc. For instance, elliptic curve based or polynomial-based primitives are such.

The second category is the most challenging one for formal methods tools. The programmer has real implementation choices to make and has to balance the risk of introducing security critical flaws with the perspective of getting faster code. Some tools have already been proposed to tackle verification of primitives of the first category [52, 119] and have proven quite successful with it. Our aim is to extend formal methods to more complex algorithms, driven by mathematical constraints for which it becomes critical to relate the actual implementation algorithms to the mathematical description of the cryptographic primitive. A target of choice among those are big integers (or *bignums*), which are used nowadays in many cryptographic systems, for different purposes but with the same mathematical grounds and computational constraints.

3.2.1 Many Different Primitives, a Common Ground

A significant share of modern cryptographic primitives (for instance those standardized in the TLS 1.3 secure communication protocol) rely on large prime fields ($\mathbb{Z}/p\mathbb{Z}$ where p is prime).

Another common pattern to those prime-field based cryptographic primitives is the fact that, because they are designed to handle security sensitive data, correctness is not the only requirement for the algorithms which are implemented. Those algorithms also have to satisfy at least some minimal

countermeasures against side-channel information leakage. Side-channels are information channels through which sensitive information may be leaked, irrespective of the functional correctness of the related algorithm. The most common, easiest to exploit side-channel is information leakage through timing measurements.

Algorithm standards typically do not specify computational costs (complexity, timing) as part of their correctness specification. A simple illustration can be brought forth by a comparison function. The naive implementation would start by comparing the first bit of each value and then if equal pursue, if not immediately return *false*. In the event where the first bit of both values are indeed equal and the comparison proceeds to the next bit the computation time will be slightly greater than if not. If the attacker is able to perform multiple queries with multiple values, he will then guess the first bit value through a statistical timing analysis, and then proceed to the next bit after setting the first one to the guessed right value. This process will leak the key and even worse, the time taken is just linear in the size of the key, even though there are no *bugs* or functional flaws. The countermeasure simply consists in performing the comparison on all bits in all cases and in only returning the result at the very end. Of course, this constant time implementation has a performance cost, as it, intuitively, always follows the *worst case* scenario (from a computation time perspective).

Cryptographers and developers face a difficult trade-off: cryptographic code, which is often the bottleneck of cryptographic applications, has to be optimized as much as possible yet without introducing any bug or side-channel vulnerability.

3.2.2 Optimization-friendly primes

In $\mathbb{Z}/p\mathbb{Z}$ the division and the modulo operations are generally the most resource intensive operations in a constant-time algorithm setting. The generic algorithms for those operations, for instance the *Barrett reduction* are not trivial to implement, and costly. Therefore, people have been using primes with specific shapes, so as to benefit from constant-time algorithmic optimizations, in particular for the modulo operator which has to be computed on almost every operation in the prime field.

Because in $\mathbb{Z}/p\mathbb{Z}$ the prime p is large (typically over a 100-bit for security reasons), such values cannot be natively stored into machine words. They have to be spread across several *limbs*: an element of the field is actually an array of smaller integers and thus field operations are implemented on top of these ad-hoc, prime and platform specific representations. In this context, a certain family of integers lays itself particularly well to the exercise: the primes from the *Mersenne* family. Mersenne primes have a very specific shape: $2^n - 1$ for n such that the value is indeed a prime number. 7 for instance is a Mersenne prime (for $n = 3$). Pseudo-Mersenne primes are prime values of the form $2^n - c$

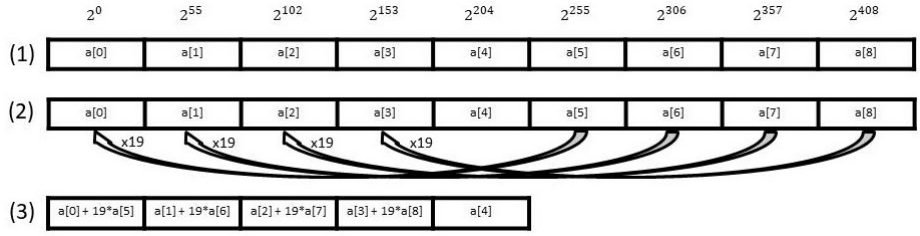


Figure 3.1: Example of pseudo-Mersenne prime reduction

where c is small enough (for instance $2^{130} - 5$ is such a value). Generalized-Mersenne primes are of the form $f(2^n)$ where f is a low degree polynomial with small coefficients (typically ± 1). For instance the NIST-P256 elliptic curve [80] relies on the prime field where $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 = f(2^{32})$ with $f(x) = x^8 - x^7 + x^6 + x^3 - x^0$ is a generalized Mersenne prime. All those have in common that they enable computationally more efficient prime modulo algorithms.

Indeed, let us consider the example of Curve25519 [34], an elliptic curve over of the field $\mathbb{Z}/(2^{255} - 19)\mathbb{Z}$. The prime of this field, $2^{255} - 19$, is a pseudo-Mersenne prime as we just mentioned. Storing such a value in this field requires 255 bits, and all computations in the field are meant to be modular, modulo that prime. Notice that in this field, $2^{255} = 19$. More generally, for any n such that $n \geq 255$, observe that $2^n = 19 * 2^{n-255}$. It implies that, for a number that has more than 255 bits, a first modular reduction can be easily carried by trimming all the bits above the 255th — we assume for now that the bignum is stored as contiguous bits in its binary representation on the machine —, multiplying those by 19 and adding them to the initially trimmed number. Depending on how large the initial value was, a couple of such operations will swiftly ensure that the result fits within 255-bit, which may be passed as a pre-condition for further computations.

Figure 3.1 illustrates the modular reduction process for a specific representation of Curve25519's bignum elements: the bignums are split into limbs of 51 bits, which implies that 5 limbs are used to store the original value. The result of a textbook multiplication will thus have 9 limbs, and the value of this element (1) is $value(a) = a[0] + 2^{51} * a[1] + 2^{102} * a[2] + 2^{153} * a[3] + 2^{204} * a[4] + 2^{255} * a[5] + 2^{306} * a[6] + 2^{357} * a[7] + 2^{408} * a[8]$. Because $2^{255} = 19$ in this prime field, $2^{255} * a[5] = 19 * a[5]$, $2^{306} * a[6] = 19 * 2^{51} * a[5]$ etc. Hence a very easy reduction process, where the top limbs are multiplied by 19, added to the bottom limbs and discarded.

This idea extends to all primes of the Mersenne family, with more or less complexity depending on their shape. In practice, a quick survey among modern cryptographic primitives suggests that the majority of them do rely on such primes, for instance Curve25519, P256, Curve448, Poly1305, Ed25519

and many more implement those optimizations.

In the following sections we present two different approaches to implement and verify bignum code in F*: one, presented in Section 3.3, is very generic and relies on the functional and higher order nature of the F* language; we show how to leverage on the modular system of F* and the idea of *templates* to factor out proofs which become broadly reusable, thus lowering significantly the proof effort to implement new bignum-based primitives. The other, presented in Section 3.4, shows a different balance between code sharing and efficiency, and highlights how verify state-of-the-art C-like algorithms, with no losses in complexity.

3.3 A verified generic bignum library

Since big integers are at the core of many widespread cryptographic primitives, a modern cryptographic big integer library will be mainly evaluated on two criteria. The first one is efficiency. In most cases, typically for elliptic curves, the whole primitive relies on the core prime field implementation. Any performance loss there would greatly impact the overall primitive performance. The second one is its resistance to side channel attacks, in particular timing attacks. In this section, we detail how to build a generic big integer framework in F*, proven functionally correct, easily extensible to new primes values, and satisfying the two aforementioned goals: implementing and verifying state-of-the-art fixed prime algorithms and enforcing a systematic side-channel mitigation discipline.

To tackle both of these goals without sacrificing code sharing and scalability we propose a novel approach relying three distinct features:

1. We present *templates* to provide a generic encoding for the representations of bignums (their *radix*), thus making the framework parametric and general purpose both in code and proofs;

2. We devise a type abstraction mechanism to ensure the equivalence of execution traces of bignum computations irrespective of the secret input values;

3. We survey the optimized algorithms implemented in popular C cryptographic libraries, incorporate them in our setup and verify their correctness; the modular structure of the code allows for efficient prime specific functions to be plugged in the code without breaking the proof. This lets us implement prime specific functions securely, without loss of performances and at a low incremental cost since all the generic algorithms have already been implemented and verified.

```
1  module Spec.Poly1305
2  [...]
3  (* Poly1305 prime number *)
4  let prime = pow2 130 - 5
5  (* An element of the field *)
6  type elem = e:int{e ≥ 0 ∧ e < prime}
7  (* Field addition and multiplication laws *)
8  let fadd (e1:elem) (e2:elem) = (e1 + e2) % prime
9  let fmul (e1:elem) (e2:elem) = (e1 * e2) % prime
10 (* Neutral elements for addition and multiplication *)
11 let zero : elem = 0
12 let one : elem = 1
13 [...]
```

Figure 3.2: Poly1305 bignum specification

3.3.1 Simple specifications

The F* language natively supports unbounded integers. Therefore, reasoning on prime fields and bignums at the specification level is quite trivial. In fact, specifying the mathematical operations which are at the core of a cryptographic primitive such as *Poly1305* [33] in F* is extremely simple and only takes a couple lines of code.

Figure 3.2 specifies the field operations in Poly1305. The elements of the field are the integers greater than or equal to zero and less than $2^{130} - 5$, the addition and multiplication are simply specified as modular operations. The Poly1305 algorithm does not rely on inverses, but it is equally trivial to specify the opposite ($-a = 2^{130} - 5 - a$) and the inverse ($1/a = a^{2^{130}-3}$).

Therefore the complexity of the proof does not come from the specification of the mathematical operation being implemented but rather from the fact that the state-of-the-art algorithms are optimized, rely on the prime's shape and the platform the algorithm will run on and side-channel mitigation constraints. Therefore we will not focus on naive implementations of the bignum algorithms which would be easy enough to prove, but rather on those which are found in the reference implementations used worldwide. However, if needed the code presented in figure 3.2 can be compiled and run in F* which used the *ZArith* OCaml library to represent unbounded integers.

3.3.2 Prioritizing Code Sharing

Our main ambition here, besides verifying cryptographic software, is to show how parametric proofs and modular implementations can significantly lower the proof burden, even on complex cryptographic primitives. Software verification usually has prohibitive development costs, even for experts. Although these are sustainable for researchers willing to push their tools to the edge

to demonstrate their capabilities on the largest projects possible, these development processes involving formal methods remain out of reach for industry developers who have to deal with time and maintenance constraints.

Since the community keeps proposing new primitives, we propose a verification approach which favors code sharing over specialization with the aim of providing an easily extensible framework for elliptic curves and other prime-field based primitives.

A Generic Bignum API

We make certain design choices which, although they present certain drawbacks, allow us to quickly set up a verified extensible library for prime-field bignums. In particular, although we decide to implement low-level algorithms which devise state-of-the-art optimizations for bignum computations, our objective of code sharing and the resulting lack of primitive-driven specialization prevents us from reaching the best algorithmic complexity for a given primitive. Intuitively the reason for that is that we do not presume anything on future uses of the manipulated bignums, which means that, for instance, intermediate results which could have been of interest for a given primitive are simply discarded, while a specialized implementation would have kept them and reused them. For instance, suppose a situation where one has to compute two modular additions in a row (e.g. $(((a+b)\%p)+c)\%p$). We could imagine a setup where there would be no need to perform the modulo operation between the two additions: $((a+b)+c)\%p$ would work, because the number a, b et c are small enough for instance. Unfortunately, such a setting would be completely specific to a certain sequence of calls in a given primitive and because in the generic bignum library setting we cannot presume how it will be used, only the modular operations are exposed. Therefore, while the generic approach does implement the best-in-class algorithms for a general purpose fixed prime modular bignum library, the invariants we provide with the generic code are too conservative for a specific cryptographic primitive to reach its very best performance using it.

On the other hand, the library is easy to use and versatile. It presents itself as an F* `Bignum` module which exposes the standard finite field operations: addition, subtraction, multiplication and inverse (see figure 2). Because we cannot presume of the use of the bignums and because the purpose of the generic bignum framework is to quickly sketch new verified primitives, all the bignum operations take and return bignums in the same format, which we call their canonical form. That form is a systematic pre- and post-condition of the generic bignum library which guarantees for the user of the bignum library that she can freely run arbitrary sequences of data without having to carry any extra invariants.

Of course, the library cannot be completely generic. Fixed prime modular arithmetic is meant to be optimized (the primes are chosen accordingly) and

our bignum library does it by cleanly separating — using the F* modular system — the generic and parametric modules from Modulo, the prime-specific one. The idea is that the Modulo module only exposes a generic interface on which the rest of the library relies and allows us to implement a complete and easily extensible bignum library. It remains the programmer's duty when introducing a new prime field-based primitive to implement the prime specific version for that particular prime and prove that it matches the provided interface. Since the implementation of a module which has an interface is left abstract by the type system once the module has been verified (only the interface is encoded to the verification system), this module implementation, which will be detailed later in this section, is nonetheless the only proof burden left to the programmer.

3.3.3 Representation and Templates

Motivations

Big integers for cryptography are too large to be stored on single words, they instead span over several limbs. Therefore, in the rest of the chapter we will assimilate low-level bignums to arrays of limbs. Now, to functionally reason about those arrays the programmer needs a function mapping the sequences of machine integers to their actual mathematical value. Implementing such a function requires knowledge of the radix used to represent the big integer. For general purpose, unbounded big integer libraries the radix used is typically the size of the limbs (32-bit or 64-bit on common platforms). However, for prime field bignum computations radixes can be different in order to leverage on specific prime structures for further constant-time optimizations. We call *packed* a representation where the radix takes all the bits of the limbs, and *unpacked* a representation in which the radix is strictly smaller than the size of the limbs.

The former is inconvenient from a verification standpoint: the packed representation is such that the bignum is stored as an array of words where every bit is used. The mathematical value of the bignum is intuitively the concatenation of all the limbs of the array in little-endian representation, which makes it very easy to convert to and from bytes. However, since any computation on those limbs may lead to integer overflows, those must be tracked and the corresponding carries propagated. The fact that the carry steps cannot be separated from the rest of the computation makes it harder to verify such algorithms, while this is precisely what the unpacked representation enables.

Given our constraints, it becomes more interesting to represent bignums as unpacked arrays. It means that, when in their canonical form, the most significant bits of each of its words are left empty. As an example, an unpacked representation for a bignum of 255 bits (as used in Curve25519) is an array of 5 64-bit words in which only the first 51 bits would be used, when

the packed representation only requires 4 words of 64-bit. Now, because the unpacked representation has some additional space, several computations may be run consecutively without having to worry about overflows and carries. For instance, a 64-bit word can store the addition of 8191 51-bit values before overflowing. Even though the unpacked representation is not as memory efficient as the packed one, when properly used it provides more efficient algorithms in a constant-time setting. This representation is by far the most common for all bignum-based primitives on 32 or 64-bit platforms, as the additional memory consumption (more space is used to store each bignum) is balanced by the computational perform improvements. In practice, all mainstream cryptographic libraries for such platforms use unpacked representations to represent low-level bignums. More exotic platforms, such as 8-bit machines, may use different coding patterns but we will ignore them to only consider the mainstream ones.

F* Templates

Templates are used in the F* code to encode the representation of bignums. Their type definition is as follows:

```
type template = ℕ → Tot ℕ*
```

It specifies the radix of the representation for each index, in other terms the number of bits they should be encoded on when the bignum is in its *canonical* form, independently from the size of the platform. Assuming a platform of n-bit, if $\forall i \in \mathbb{N}, t(i) = n$ then the representation is packed, and such representations are useful, even in our setting, to reason about inputs and outputs for instance, which are packed byte arrays (specified by the template $t_{bytes} : i \to 8$).

If $\forall i \in \mathbb{N}, t(i) < n$, then the representation is unpacked. This case corresponds to the internal representation of all bignums for internal uses (after deserialization) in all popular cryptographic libraries. From this point on, we will be considering the unpacked representation and the details of the internal bignum algorithm implementations.

Evaluating Low-level Bignums

The template of a bignum specifies how to interpret its mathematical value. Given b a bignum, $i \in \mathbb{N}^*$:

$$w(b,n) = \sum_{i=0}^{n-1} t_b(i)$$
$$eval(b,n) = \sum_{i=0}^{n-1} 2^{w(b,i)} * b[i]$$

where t_b is the template associated to the bignum b. Intuitively, the representation of a bignum with a certain template corresponds to its decomposition into the template's base. Just as the hexadecimal representation of a number corresponds to writing it in base 2^4, the template $t_{56} : n \to 51$ used for Curve25519 for instance corresponds to the equivalent base 2^{51}. Although we will consider mostly constant templates, more sophisticated ones could be used, like $t_{26/25} : n \to 26 - (n\%2)$ for Curve25519 on a 32-bit platform[3]. $w(b,i)$ computes the log_2 of the weight of the value stored in the i-th limb of the bignum b from its template, while $eval(b,i,len)$ is the weighted sum of the len first limbs of the same bignum.

The *eval* function is the inverse of the decomposition of the integer on a base: from the concrete representation it computes back the mathematical value. Interestingly, the correctness of the *eval* function does not depend on the concrete value of the limbs being smaller that indicated by the template. The eval function is surjective: two different bignum encodings may represent the same mathematical integer.

Stateful Implementation

We encode bignums, w and eval in F* as follows:

```
noeq type biginteger (size:pos) =
  | Bigint: data:array (usint size) → t:template → biginteger size

let rec w (t:template) (n:nat) : Tot nat =
  match n with
  | 0 → 0
  | _ → t (n−1) + w t (n−1)

let rec eval (#size:pos) (b:seq (usint size)) (t:template) (n:nat{n ≤ Seq.length b}): GTot nat =
  match n with
  | 0 → 0
  | _ → pow2 (bitweight t (n−1)) ∗ v (Seq.index b (n−1)) + eval b t (n−1)
```

A word of syntax: the noeq keyword specifies that there is no decidable equality on the biginteger type. This is because the biginteger is mutable: it wraps a value of type array which value depends on the memory state of the

[3]https://github.com/agl/curve25519-donna/

program. The biginteger type, and thus the eval function are parametrized by a size value, which indicates the size of the targeted platform in number of bits. In the eval function, the size parameter is annotated as implicit using the # notation. This means that when applying the eval function to a sequence of data, the programmer does not have to provide the actual size of the targeted platform. Rather it can rely on the inference of the type system to determine its value. Of course, should the inference algorithm fail, F* would complain and require these extra parameters to be explicitly provided by the programmer (again using the # notation).

The template-based eval function maps the representation of bignums as sequences of machine integers to mathematical integers, which lets the programmer reason about the algorithmic operations on bignums, not at the representation-dependent low level but directly at abstract and platform independent mathematical level. This whole reasoning happens in the *pure* core of F* as this is the only setting in which proofs and specification can live. While it would be possible to implement whole cryptographic algorithms in this setting, such a choice would not be comprehensible. Pure computations when compiled to real-life code lead to numerous implicit allocations and deallocations of fresh values, which typically results in inefficient code both in terms of memory consumption and computational speed. Furthermore, the purpose of this work is to explore the capabilities of a new language such as F* to tackle large and evolving cryptographic verification projects. One of the key aims is to prove the correctness of the various low-level optimizations introduced by crypto developers, and such developers always target low-level languages (typically C or assembly) where mutability is inherent to the language. Relying on one of the key strengths of the F* language — its effect system — we chose to make our library stateful, using mutable data types and in particular the native F* array type.

In the chapter 2 we saw several ways to encode a memory state in F*, using either the single Heap model or the more refined region-based HyperHeap model. Here we chose to use the ST effect with a single heap because it provides sufficient granularity for our framework and proofs. Because in our setting the bignums' data is represented as a mutable array, the stateful encoding of the code is crucial to the functional correctness proof.

Generic and Specialized Bignum Definitions

Consider the F* definition of bignums:

```
noeq type biginteger (size:pos) = | Bigint: data:array (usint size) → t:template → biginteger size
```

Bignums are represented as a data constructor wrapping both an array of unsigned secret integers of a certain platform size (the size parameters) and their template (mapping to their mathematical value). Given that the

```
val prime: erased pos // Prime value, e.g. 2^255 - 19
val platform_size: pos // Default word size on the platform, e.g. 64 (bits)
val platform_wide: w:pos{w = 2 * platform_size }
val templ: t:(nat → Tot pos) // Actual template, e.g. λx → 51
val norm_length: pos // Default length of the bignums, e.g. 5 limbs for Curve25519

// Necessary conditions for the parameters' values
val lemma_0: unit → Lemma (∀ (i:nat). i < 2*norm_length - 1 ⟹
templ i < platform_size)
val lemma_1: unit → Lemma (platform_wide - 1 ≥ log_2 norm_length)
```

Figure 3.3: Signature of the Parameters module which specializes the generic bignum library for a specific prime

biginteger type depends on both the size of the platform limbs and the values of its template, each bignum type must be instantiated using primitive-specific values. For instance, for the elliptic curve Curve25519, on a 64-bit platform, the community agrees that the most appropriate template is $t_{25519} : n \to 51$.

For simplicity, using the module system of F*, bignums are specialized using values from a generic Parameters (see its signature in figure 3.3):

```
type bigint = b:biginteger Parameters.platform_size{b.t = Patameters.templ}
```

The w and eval functions are defined in the listing above. w returns the log (in base 2) of the weight of each limb, in other terms the number of bits preceding that limb. For instance, in Curve25519, the bitweight of the lowest limb is 0 (hence a weight of $2^0 = 1$), the one of the second lowest limb is 51 (hence a weight of 2^{51}), the one of the third is 102 etc. Relying on w, the recursive eval function maps the bignum to its mathematical value. In that intent, it takes a sequence of integers — corresponding to the value of a bignum in a certain memory state h — and a template, which should of course be the one of the said bignum, and it evaluates the sequence of integers, weighting each of the limbs according to the w value.

eval is purely aimed at specification and should not appear in concrete code; it is used to prove the validity of the low-level constant-time algorithms, it would have no meaning to extract it, instead it would likely defeat the side-channel mitigation discipline we want to set up, and render efforts to restrict ourselves to low-level representations of the bignums useless. Using the F* effect system, the function is annotated as *ghost* and is therefore erased by the compiler. The F* verification system guarantees that the function shall never be executed and thus has no impact on performance or secrecy. Those properties are all proven statically at compile time, and thus have absolutely no impact on the resulting compiled code. In particular such specifications do not introduce any dynamic checks.

```
let len = Parameters.norm_length // = 5 for Curve25519

let isSum (h0:heap) (h1:heap)
                  (a:bigint{live h0 a ∧ live h1 a ∧ getLength h0 a ≥ len
                    ∧ getLength h0 a = getLength h1 a})
                  (b:bigint{live h0 b ∧ getLength h0 b ≥ len})
                  (ctr:nat) =
  (∀ (i:nat). {:pattern (v (getValue h1 a i))}
          (i≥ ctr ∧ i<len) ⟹ (v (getValue h1 a i) = v (getValue h0 a i)
                                                  + v (getValue h0 b i)))

val fsum:
  a:bigint → b:bigint{similar a b} → ST unit
    (requires (λ h → normalized h a ∧ normalized h b))
    (ensures (λ h0 u h1 →
      normalized h0 a ∧ normalized h0 b ∧ normalized h1 b
      ∧ (live h1 a) ∧ (modifies (Set.singleton (Array.addr_of a.data)) h0 h1)
      ∧ (getLength h1 a = getLength h0 a) ∧ (getLength h0 b = getLength h1 b)
      ∧ (eval h1 a len = eval h0 a len + eval h0 b len)
      ∧ (isSum h0 h1 a b 0) ))
```

Figure 3.4: Specification of the addition function

3.3.4 Verifying Generic Bignum Operations

Using the previously defined low-level stateful representation of the bignums, it is possible to implement and verify the standard prime field operations on those low-level constructs. In rest of this section, the examples use the parameters of Curve25519 ($x \to 51$ template and arrays of length 5) for 64-bit platforms. The listing below shows the specification of a limb to limb addition operation which takes two bignums `a` and `b` as input and performs an in-place limb-to-limb addition of their contents, storing the results into `a`, the first argument.

The canonical length for bignums for Curve25519 64-bit is 5, with a template for 51-bit for each limb. Given that the corresponding field is $\mathbb{Z}/(2^{255} - 19)\mathbb{Z}$, 255 bits are indeed sufficient to encode all the values in the field.

`isSum` defines a functional predicate over two heaps and two bignums. As we are in the `ST` effect, the pre and post-conditions are parametrized by a `heap`. `h0` corresponds to the initial memory state when `h1` corresponds to the resulting memory state when the function returns. This predicate states that the value of each limb of `a` in `h1` is the sum of the value of the corresponding limbs of `a` and `b` in `h0`. When used in the specification of the function `sum`, the predicate implies that `sum` performs an in-place, limb-to-limb addition between `a` and `b`, with the result stored in `a`.

Obviously such a simple addition function requires some pre-conditions. In particular the predicate `isSum` specifies that each element of `a` after the `sum`

function returns is the *mathematical* sum of the corresponding elements in the original values of a and b, which can only be true if no overflow occurred.

The normalized predicate holds for both bignums a and b in the pre-state h.

```
let normalized (h:heap) (#size:pos) (b:biginteger size{ live h b }) =
  (∀ (n:nat). {:pattern (v (getValue h b n))} n < getLength h b ⟹
       v (getValue h b n) < pow2 (getTemplate b n))
```

It specifies that the bignum's data is canonically formatted under the template's unpacked representation (all limb values fit within the number of bits indicated by the template). In our example it requires the bignum's array to be a live reference to a memory block, to be at least 5 limbs long, each limbs to be both greater than or equal to 0 and less than 2^{51} and the associated template to be t_{25519}.

The refinement on b, the similar predicate

```
type similar (#size:pos) (a:biginteger size) (b:biginteger size) = a.t == b.t ∧ a.data == b.data
```

specifies that both a and b must be defined with the same template and that they must refer to disjoint memory blocks. It enforces the memory separation condition between a and b that is necessary to ensure memory safety.

In other terms, the type declaration of the sum function specifies that if provided with two inputs a and b which correspond to two distinct arrays of at least five limbs with values in range $\{0; 2^{51}\}$, then the function mutates a in-place such that each of its limbs contains the sum of the initial limbs of the inputs.

Proving Functional Correctness

Given those properties, the verification system guarantees that the post-condition holds in the resulting memory state h1. The modifies clause states that only a's data is modified through the execution of the function. Given the separation condition between a and b enforced by the similar predicate — a and b encapsulate different references pointing to different memory blocks — it implies that b is left untouched by sum, and thus that the normalized predicate still holds. Now, as the reference a has been modified, we need to indicate and prove that it still exists and points to valid data when the function returns. That is what the live condition gives us. This liveness condition is also a prerequisite to be allowed to express logical properties on a's data in the h1 environment, such as the fact that the length of the underlying array has been left unchanged.

Next are the functional correctness properties. First, the isSum predicate expresses what was computed by the function: the l first limbs of a in state h1 contain the sum of the corresponding limbs of a and b in state h0. Knowing that the bignums satisfied the normalized predicate in the initial h0 state, it allows

for further proofs on the size of the limbs of a in h1 as it keeps track of the ranges of possible values in each limb, which is essential to track and prove the absence of overflows through the computations.

Second, the equality on the eval function is the one which does guaranty the functional correctness of the sum function. Indeed, it shows that the result of the sum function maps to a mathematical integer which is the (integer) sum of the original value of the inputs in the mathematical integer space. In a second step, after computing the modulo function, the modular addition will be proven correct, not just with regard to integer arithmetic, but in finite field $\mathbb{Z}/p\mathbb{Z}$.

The code snippet in Figure 3.5 illustrates how the functional correctness proofs are carried in F*.

```
1  abstract let notModified (h0:heap) (h1:heap)
2                           (a:bigint{live h0 a ∧ live h1 a ∧
   getLength h0 a = getLength h1 a
3                             ∧ getLength h0 a ≥ norm_length})
4                           (ctr:nat) =
5    (∀ (i:nat). {:pattern (getValue h1 a i)}
6      ((i ≠ ctr ∧ i < getLength h0 a) ⟹ getValue h1 a i == getValue h0 a i))
7
8  val fsum_index:
9    a:bigint → b:bigint{similar a b} → ctr:nat{ ctr ≤ norm_length } →
10   ST unit
11     (requires (λ h →
12       (live h a) ∧ (live h b)
13       ∧ (norm_length ≤ getLength h a ∧ norm_length ≤ getLength h b)
14       ∧ (∀ (i:nat). (i ≥ ctr ∧ i < norm_length) ⟹
15            (v (getValue h a i) + v (getValue h b i) < pow2 platform_size)) ))
16     (ensures (λ h0 _h1 →
17       (live h0 a) ∧ (live h0 b) ∧ (live h1 a) ∧ (live h1 b)
18       ∧ (norm_length ≤ getLength h0 a ∧ norm_length ≤ getLength h0 b)
19       ∧ (modifies (Set.singleton (Array.addr_of a.data)) h0 h1)
20       ∧ (getLength h0 a = getLength h1 a)
21       ∧ (isSum h0 h1 a b ctr)
22       ∧ (notModified2 h0 h1 a ctr) ))
23 let rec fsum_index a b ctr =
24   match l − ctr with
25   | 0 → ()
26   | _ →
27     let ai = index_limb a ctr in let bi = index_limb b ctr in
28     let z = add_limb ai bi in
29     upd_limb a ctr z;
30     fsum_index a b (ctr+1)
```

Figure 3.5: Implementation and proof of the recursive fsum_index function

```
val addition_lemma:
  h0:heap → h1:heap → a:bigint{live h0 a ∧ live h1 a} →
  b:bigint{live h0 b ∧ b.t = a.t} →
  len:ℕ{len ≤ getLength h0 a ∧ len ≤ getLength h0 b
        ∧ len ≤ getLength h1 a
        ∧ (∀ (i:ℕ). i < len ⟹
          v (get h1 a i) = v (get h0 a i) + v (get h0 b i)) } →
  Lemma (eval h0 a len + eval h0 b len = eval h1 a len )
```

Figure 3.6: Functional correctness lemma for the bignum addition

Using the information available on the different functions called (for instance index_limb which returns the value of the i-th limb of bigint or add_limb which returns the machine integer sum of its operands) and a system of *triggers* to guide the SMT solver in its instantiation of quantifiers — see the :pattern (getValue h1 a i) trigger in the notModified predicate — the F* system is able to discharge the verification conditions for the fsum_index function automatically, meaning that the pre-conditions from the requires clause are sufficient to guaranty the ensures post-condition.

Hence, calling sum_index a b 0 on two bignums satisfying the requires clause will return a new memory state in which IsSum h0 h1 a b 0 holds as required in the post-condition of the sum function above. Additionally, if the Normalized predicate holds for a and b in the initial state h0, then using the properties of the unpacked representation, we can show that the pre-condition a[i]+b[i]$<2^{SIZE}$ initially holds.

Eventually, the lemma in figure 3.6, proven by induction and based on the isSum predicate, demonstrates the equality on the eval functions in the sum function post-condition. Calling this lemma after the fsum_index function gives us the sum complete specification. The concrete code has to provide some additional intermediate lemmas to help the prover and make it more efficient and more flexible to amend the code without breaking the proof, but these are the key steps.

3.3.5 Extensible Bignum Library

To implement a generic and extensible prime field library we need to define the constraints imposed on the bignum operations. In particular, to run arbitrary sequences of computations the library API has to enforce that every bignum operation result satisfies the precondition of every bignum operation. The template defines this common ground: both input and output data in the bignum library have to fit into the canonical form and thus satisfy the normalized invariant.

The previous section highlighted how to implement and prove the correctness of a straightforward textbook addition operation: provided guarantees

that no overflow can occur, the verification system is easily able to prove that the addition operation we just described returns a bignum which maps to a field element which is indeed the field addition of the field elements corresponding to the inputs.

In the same spirit, we implement other bignum textbook operations: subtraction, scalar multiplication and multiplication. Each of those bignum operations does assume that the input bignum in its canonical state. Their result, however, does not meet the criteria we just defined, and legitimately: each of these operations considers a bignum in $\mathbb{Z}$ and does not account for the modular reduction. Indeed, the modular reduction is specific to each prime and cannot be presumed in advance. Rather, generic algorithms exist which could allow for modular reduction of any prime, but for efficiency reasons these are (most of the time) not the ones used in real world cryptography and thus not the ones we want to implement and verify.

To handle this difficulty we rely on the modularity of the F* code. F* benefits from a system of modules which lets the programmer specify the interface of a certain module while leaving its actual implementation abstract. The correctness of the implementation is checked by the F* verification system against the interface but it is never made available outside the module. We structure the library code as follows:

- a Parameters module contains platform and prime specific values (such as the size of the words, the value of the prime, the template, etc.) which will be used to instantiate different invariants other modules
- generic modules for addition, subtraction, scalar multiplication and multiplication are provided and produce results for which generic invariants hold
- a prime-specific Modulo module contains the missing functions and proofs for fill the holes of the generic modules
- a top-level Bignum module combines the prime-specific and the generic code into the finished API, usable by cryptographic primitives

3.3.6 Prime Specific Code

We implement constant-time modular arithmetic on bignums in a given prime field. In modern cryptography, and in particular for the curves we are considering, the primes have been carefully chosen so as to allow for efficient constant time modulo reductions. Still, the way these reduction operations can be efficiently implemented depends on the value of each prime and cannot be parametrized.

Inspired from existing libraries, we implement five distinct functions relying on the prime value, and which have to be provided by the developer if she

wishes to extend our existing framework with new primitive and new prime fields.

freduce_degree is a function which takes a bignum of size 2∗l−1 where l is the length in the canonical unpacked representation, and returns a bignum of size l mapping to the same value in the field. The purpose of this function is to provide a first reduction step after a textbook multiplication. Indeed, the multiplication of two bignums of size l leads to a new bignum of size 2∗l−1 which does not fit the rest of the computation settings: to ensure the genericity of our bignum implementation all bignum operations take bignums in canonical form (satisfying the normalized predicate), and return values satisfying the same predicate. In particular, primes of the Mersenne family (which are the most common in modern cryptographic primitives) lay themselves well to such a function (see section 3.2.2).

freduce_coefficients serve a similar purpose as freduce_degree except that instead of reducing the number of limbs of a bignum, it reduces the number of bits of each limb of a bignum which is already at its canonical length. It will typically proceed to two carry passes on the bignum of size l to return a new bignum of size l, which maps to the same field value through the eval function but satisfies the normalized predicate. The combination of both freduce_degree and freduce_coefficients is necessary to implement the modular reduction of the multiplication (which make the number of limbs grow), while the addition or subtraction for instance only need to call the freduce_coefficient function to get their result back to the canonical normalized shape.

freduce_complete supposes that the input is a bignum which is already in its canonical form. Its purpose is to further reduce the improper values which may remain. Remember that the eval function is surjective and that two bignums with different values may map to the same field value, even though they are in canonical form. For instance, in the Curve25519 example, 1 and $2^{255} - 18$ which can be respectively represented in the t_{25519} template in little-endian as 1|0|0|0|0 and 0x7fffffffffffee|0x7fffffffffffff|0x7fffffffffffff|0x7fffffffffffff|0x7fffffffffffff. The later, however, is good for internal use but not to return a final value, since it is not a value of the finite field. It has to be reduced to the former instead, which is precisely the purpose of the freduce_complete function.

crecip is a prime specific inversion function. It is quite challenging to implement a division function in constant-time in our setting, and indeed the reference cryptographic libraries instead chose to use the finite field property $\frac{1}{a} = e^{p-2}$ in which without a division specific algorithm, relying only on multiplication and squaring an inversion operation can be implemented. Of course

since p depends on the chosen primitive, such an inversion function has to be provided for each primitive.

add_big_zero is the last prime specific function of our setting. It takes a bignum in canonical form, and adds a multiple of the prime to it such that it will prevent underflows in limb to limb subtraction with another bignum in canonical form. Indeed our specification of the subtraction function on the unsigned integers does not allow underflows, so one always has to prove that $a \geq b$ before computing $a - b$. Hence when subtracting two bignums limb to limb, it is necessary to have that $\forall i \in \mathbb{N}, i < l \implies a[i] \geq b[i]$. To get this property while not modifying the encoded values of the bignums, we add to the bignum a a multiple of the prime, typically $2p$ or $4p$ encoded in such a way that $\forall i \in \mathbb{N}, i < l \implies p_{multiple}[i] \geq 2^{56}$ in our example, and then compute the value of a' such that $\forall i \in \mathbb{N}.i < l \implies a'[i] = a[i] + p_{multiple}[i]$ minus b limb to limb. As we encode the prime multiple to meet those specific constraints on each of its limbs, we cannot do it generically and the programmer has to provide such an encoding. For instance for Curve25519, the multiple of the prime $2^{256} - 38$ can be written in little endian form 0xfffffffffffda|0xffffffffffff|0xffffffffffff|0xffffffffffff|0xffffffffffff, representation in which each of its limbs is greater than 2^{51} and hence it is fine to subtract any bignum in canonical form from the addition of this multiple of $2^{255} - 19$ with any bignum in canonical form.

It is not mandatory to split the modulo function into the three reduction functions. We chose to adopt this pattern which is already used in the standard implementations of the curves because each phase is relatively costly and needs not be executed after each bignum operation. Indeed the addition, subtraction and scalar multiplication functions on the bignums do not modify the size of the input arrays. They take normalized arrays (in canonical form) as inputs and return results in arrays of the same standard length. The multiplication operation, however, is different: it returns an array of size 2*l−1 where l is the standard length. So when given the result of a multiplication, both freduce_degree and freduce_coefficient are required to get back a normalized bignum. On the other hand, the freduce_complete function is only needed before serializing an internal value. Therefore, splitting the reduction into three bits allows for more efficient algorithms and has no impact on the correctness.

Primes Specific Invariants Our library already provides interfaces for all the functions above, shown in figure 1. For instance the functional correctness of the freduce_degree or freduce_coefficients function is already specified:

```
val freduce_degree: b:bigint_wide → ST unit
    ...
    (ensures (λ h0 _h1 → ... ∧
       eval h1 b Parameters.norm_length % reveal Parameters.prime =
```

```
        eval h0 b (2*Parameters.norm_length-1) % reveal Parameters.prime))

val freduce_coefficients: b:bigint_wide{b.t == Parameters.templ} → ST unit
    ...
    (ensures (λ h0 _h1 → ... ∧
        eval h1 b Parameters.norm_length % reveal Parameters.prime =
        eval h0 b Parameters.norm_length % reveal Parameters.prime))
```

The implementation has to, of course, satisfy this precise specification, because the top-level Bignum module relies on it for the correctness of the top-level bignum API. In our experience however, functional correctness does not constitute the challenging part of the proof. Rather, tracking that the bounds are appropriate is the difficult part. Indeed, the functional correctness relies only on the field mathematical properties, which are easily discharged at the mathematical level (e.g. $2^{255}\%(2^{255} - 19) = 19$), while the bounds depend on the reduction algorithm, as well as the platform size, the prime shape and the bounds which result from the provided generic functions. For instance, a naive implementation of the freduce_coefficient might work after a call to the addition fsum function, but not after a call to the subtraction fdifference function (remember that the subtraction carries the implicit addition of a multiple of the prime and thus that its limbs actually grow faster than in the addition case).

To account for those constraints without knowing the prime shape or the platform size in advance, the interface of the Modulo module we provide assumes abstract bound predicates and lemmas which enforce that the combination of those bound constraints with the bound predicates after addition, subtraction scalar multiplication and multiplication is right. The programmer has to implement both the functions and the bound specific predicates and prove the provided lemmas.

For instance the actual implementation of the satisfies_modulo_constraints predicate for Curve25519 is

```
let satisfies_modulo_constraints h b =
  getLength h b ≥ 2*Parameters.norm_length-1 && b.t == Parameters.templ
  && maxValue h b * 20 < pow2 (Parameters.platform_wide - 1)
```

To prove the feasibility of these proofs we provided examples for Curve25519, Curve448 and P256 which all exhibit primes with different structures [4].

Functionally Correct API

For the internal intermediate computations the first two reduction functions are sufficient to implement functionally correct modular arithmetic. This is true because freduce_coefficients returns a normalized bignum. Nevertheless, given

[4] https://github.com/mitls/hacl-star/tree/master/snapshots/ecc-star/curve_proof

that our bignum library encodes prime field elements, the final returned encoding must be unique for each element of the prime field. And a normalized bignum does not satisfy this condition. Sticking to the example of Curve25519, the prime has value $p_{25519} = 2^{255} - 19$. The unpacked representation guarantees the uniqueness of the representation of integer values between 0 and $2^{255} - 1$ included, which means that values greater than or equal to p_{25519} and less than 2^{255} have two different valid encodings. The freduce_complete function takes care of this issue making sure that returned values are in bijection with the prime field. However, as it is costly to implement in constant time and not required for the correctness of the internal computations, freduce_complete will only be computed once, when serializing data.

Therefore, the modular internal addition operation of bignums of limbs is eventually exposed as

```
val fsum: a:bigint → b:bigint{Similar a b} → ST unit
    (requires (λ h →
      (Normalized h a) ∧ (Normalized h b)
    ))
    (ensures (λ h0 _h1 →
      Normalized h0 a ∧ Normalized h1 a ∧ Normalized h0 b
      ∧ (valueOf h1 a = (valueOf h0 a ^+ valueOf h0 b))
      ∧ (modifies (getRef a) h0 h1) ))
```

where the functional correctness relies entirely on the correctness of the field operator ^+ defined in the Field module, and the modifies clause which composed with the similar predicate gives enough information on the memory states for further proofs.

At this point, the concrete values of the bignum limbs as well as all the details of the algorithm are hidden by the interface, and the rest of the code will rely solely on the exposed high level specifications, thus leading to modular and extensible proofs.

3.4 A second approach: balancing code sharing and performance

3.4.1 Inefficiencies of a Generic Bignum Library

In the previous sections we presented a new approach towards building a generic and easily extensible framework for fixed-prime bignum modular arithmetic. This line of work was initiated by a simple but important ascertainment: most of the flaws — and thus the verification efforts — from prime field based cryptographic primitives come from the bignum code. Indeed, for such primitives the bignum code is usually at the core of the computation and the various modular reduction phases can be tricky to implement correctly and in constant-time. Our approach shows that it is possible to follow a principled discipline to implement new prime fields, relying on the framework we

designed, implementing only the very prime-specific details with the help of several similar examples and formal methods as a safety check. This successfully led to a verified generic bignum library which incorporates state-of-the-art implementation tricks in F* and reuses most of the code across several fields. We estimate to about 80% the amount of code shared across several cryptographic primitives, which does lower the proof effort significantly and should help people who wish to quickly scheme new functionally correct cryptographic primitives to do so easily.

One unfortunate drawback of this methodology is that is exhibits poor performance when measured on cryptographic primitives built on top of the bignum code. The generic fixed-prime bignum library aims at providing the most efficient API possible *at the bignum level*. The API of the library is intended to serve as a building block for any application which wishes to rely on verified prime field operations, therefore all the invariants provided by the different bignum operations have to be consistent with one another so that the application can easily implement arbitrary sequences of operations on top of them. However, these choices hurt when competing against code which has not only been specialized for a given prime as the library does, but also for a given primitive. In such cases, the additional specialization enables new optimizations such as reusing intermediate values or skipping some unnecessary reduction steps which make the generic code seem inefficient in comparison.

Indeed, remember that the modular reductions are the main cost of the algorithms. In particular, on recent platforms, limb to limb operations (like addition or multiplication) are heavily optimized by compilers, using vector instructions when available for instance, and thus become extremely fast. The constant-time modulo is more complex and depends on the prime and in particular its shape. To illustrate the flexibility of the proof system of F* and the libraries and verification techniques we developed to implement the extensible bignum library, we show how to specialize our bignum code for specific primitives, at the cost of some additional proof effort but incurring no performance hit compared to the best-in-class algorithms we can see in real world cryptographic libraries. Of course, although this code does not share as much code as it does for the prime-field bignum library, a significant share of it remains shared.

3.4.2 Fixed sequences of operations

The first and most straightforward way to cut some computations is to track unnecessary reductions in a primitive, and remove them. For generic code, one should be free to run arbitrary sequences of field computations using the bignum API, and thus all the bignums will be systematically reduced to their normalized state, however for real world primitives this is unnecessary in most cases. In Curve25519 for instance, our leading example in this chapter, the primitive being an elliptic curve point scalar multiplication, the whole com-

putation revolves around doubling and adding points together. Those elliptic curve operations boil down to operations on the point coordinates which live in the underlying prime field. And since these sequences of operations on coordinates are fixed, the operations can be optimized to remove some unnecessary extra carry passes or reduction steps for instance.

```
A = x_2 + z_2
AA = A^2
B = x_2 - z_2
BB = B^2
E = AA - BB
C = x_3 + z_3
D = x_3 - z_3
DA = D * A
CB = C * B
x_3 = (DA + CB)^2
z_3 = x_1 * (DA - CB)^2
x_2 = AA * BB
z_2 = E * (AA + a24 * E)
```

Figure 3.7: Specification of the doubling (x2,z2) and adding (x3,z3) of two points on a Montgomery curve

In the listing above, extracted from the RFC 7748 "Elliptic Curves for Security" which standardizes Curve25519, the sequence of operations describe how to compute the *doubling* of the point (x_2, z_2) (stored in (x_2, z_2) at the end of the sequence), and the *addition* of (x_2, z_2) and (x_3, z_3) (stored in (x_3, z_3)). Although it may not be straightforward to notice, the sequences of field operations always follow the same discipline: each multiplication or squaring operation sees its operands being the result of either an addition or a subtraction. Contrary to the multiplication and the squaring, the addition and subtraction do not grow the size of the limbs of a normalized bignum very much. We thus use a *non-modular* version of the addition and the subtraction, which is faster, and only perform the modular reduction operation after the multiplication or squaring happen. Moreover, we show that the fix-point for such operation combinations is that in the Curve25519 example, we can get rid of 5 carries in the freduce_coefficients function, and still not overflow.

3.4.3 Prime specific optimizations for multiplication and squaring

In order to make the code as prime agnostic as possible for the generic bignum API, we chose to follow the method we outlined previously: each field operation (addition, subtraction etc.) is generically implemented without its associated modular reduction and an interface is provided for the modulo operation which

splits the reduction in three steps. The first one reduces a bignum (seen as an array) to its canonical length and is useful only for multiplication and squaring, the second one reduces the bignum to its canonical shape, trimming the superfluous bits in each limb, and is useful for all modular reductions, the last one ensures that the encoding of the result is right and is only useful once, at the end of a cryptographic computation.

Multiplication

Now, because the prime is known in advance, both the multiplication and the squaring functions can be specialized. In the previous setting, the bignum operations were implemented using recursive functions because the length of the arrays is unknown, and the reduction happened in at least two steps. This pattern does not lead to a minimal number of computations. Taking the Curve25519 example, the original textbook multiplication is implemented the scalar multiplication of a bignum a by each of the limbs of b, shifting the resulting array accordingly to the left and adding them together. Overall this means 25 64x64-bit multiplications and 20 128-bit additions. Then the freduce_degree consists of 4 128-bit multiplications by 19 and 4 128-bit additions. We omit the copies corresponding to shifting values in an array.

In a slightly different implementation, the same operation could be implemented using a *shift-reduce* mechanism. The idea being that, instead of running a textbook multiplication and shifting the result of the scalar multiplication of a by the i-th limb of b, we first shift and reduce a and then only multiply it by the $i-th$ limb of b. The algorithm is as follows:

```
shift_reduce(b) =
  b = (b[4]*19) @ (b[0..4])

mul(a,b) =
  uint64 c[5] = {0};
  for (i = 0; i < 5; i++)
    c += b[i] * a
    if (i < 4) a = shift_reduce(a)
  c
```

Because the shift and the reduction happen before the scalar multiplication, the operations happen on 64-bit values instead of 128-bit once and there is no more need to add the top limbs to the bottom limbs to perform the first step of the reduction. This new algorithm is thus more efficient: 25 64x64-bit multiplications, 20 128-bit additions and 4 64-bit multiplications by 19, which means that overall there are 4-less 128-bit additions and that 4 128-bit multiplications by 19 were turned into 4 64-bit multiplications, leading to better performance.

Interestingly this algorithm is still quite generic, provided that there exists a way to efficiently implement the shift_reduce function. In particular, for

Table 3.1: Code size, verification effort, and performance

Function	Specs	Code	Annotations	Verification	Computation
Math	150	0	10	-	-
Zsum	22	11	311	8s	1.5μs
Zmul	55	11	1144	57m	28μs
Modulo (25519)	65	60	551	15m11s	36μs
Ladder	13	40	527	2m50s	176ms

Mersenne or pseudo-Mersenne prime (such as $2^{255} - 19$) as shown above the shift-reduce function is trivial to implement. Therefore, this algorithm still allows a lot of code sharing (the mul function), only the *reduce* part of the shift_reduce function has to be specifically implemented for each prime. For Mersenne or pseudo-Mersenne primes, this algorithm has the best known performance, it is as fast as fully specialized code (without recursive calls etc.), however in the general case it may not be and the shift_reduce function can prove to be quite complex to implement, which is why we did not implement is as part of the generic bignum library.

The squaring function benefits from the same improvements as the multiplication, but can be optimized even further. Indeed, because of the symmetry of the operation, some values can be cached and reused.

Conclusion

The high (almost prohibitive) time investment required to use formal methods to verify state-of-the-art implementations has been a major limiting factor to their adoption. Only in specific areas — such as cryptography — could we justify such an effort, due to the disastrous security impact of potential flaws. In this chapter, however, we showed that the entry cost for formal methods is getting lower. The keys to make them more accessible are modularity and automation. As seen in Chapter 2, automation is one of the strengths of F*. However, SMT solvers are still heuristic tools and proofs still require some efforts from the programmer. Fortunately, F* also supports a system of modules, as well as abstraction. The combination of those two features lets the programmer precisely define the semantics of the interfaces shared across modules. This process is convenient to build large proofs: one module can be easily replaced by another, provided that both implement the same interface, and automation is preserved by the fact that the unnecessary details are kept hidden from the SMT solver. We illustrated those mechanisms on a concrete hard example: bignum computations for cryptography. In cryptography, for security and efficiency reasons, bignums typically have similar shapes — we saw that for elliptic curves theses are primes of the Mersenne family. Therefore, we wrote the bignum library and its modules to factor out the common patterns

and parameterize them with the primes' shapes.

To that intent, we used the concept of *templates*, which describe the link between the mathematical and the concrete physical representation of the bignums, and we presented two different approaches to implementing the bignum library. This first approach is focused towards code sharing while the second shows that without sacrificing too much of this code sharing, we can prove the correctness of state-of-the-art algorithms in terms of complexity. However, this high-level of code and proof reuse heavily relies on high-order features of F*. While the algorithms are efficient, the languages they compile to are not the best in class in terms of performance, and are difficult to protect against side-channel attacks.

Chapter 4

Low*, a Low-Level Programming Subset of F*

Parts of the text are taken from [107], a paper that appeared in ICFP 2017 and was co-authored by me along with Jonathan Protzenko, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet and Nikhil Swamy.

4.1 Verification Approach

4.1.1 Lessons from previous experiments

Chapter 3 illustrated how verifying state of the art cryptographic algorithms for bignums is feasible in F*, with a moderate proof effort. In addition to the generic bignum API on top of which one can easily build verified cryptographic primitives, we showed that, provided an extra proof effort, implementing and verifying the best-in-class algorithms for bignum-based cryptographic primitives was achievable.

While the proof methodology is promising, the overall performance remains disappointing. Although the algorithms implemented are the efficient, reference ones, and match a low-level C-like representation of the bignums as arrays, OCaml, the language F* natively extracts to, is not the ideal candidate for fast cryptographic implementations.

Garbage collection First, the OCaml language is a high-level functional language. In particular it hides the memory management details to the programmer, and handles function allocation and memory recollection automatically. In a normal setting this is a great feature since it guarantees the memory safety of compiled OCaml programs — which is typically hard to get with lower-level languages, such as C. The OCaml runtime system chooses where to allocate data structures and relies on a garbage collector to automatically

collect ad free memory addresses which are no longer in use. Unfortunately garbage collectors behaviors are difficult to anticipate and are known to introduce side-channels [111, 89]. This makes the OCaml language an unsatisfactory candidate for cryptographic implementations. Furthermore, the bignum generic API shows that the kind of code needed for cryptography does not require automated memory management. Rather, as the code was originally inspired from reference C implementations, the algorithms follow very consistent allocation and freeing disciplines, and never rely on complex data structures such as lists or trees, which are typically those which functional languages are great at working with.

Integer representation The OCaml language uses an ad-hoc representation for machine integers. Indeed, to distinguish between and optimize the implementation of integers and pointers, OCaml restricts the size of actual integers by 1 bit (i.e. 63 bits on a 64-bit platform, 31 on a 32-bit platform), the remaining bit being used to tag the object as either an integer or a pointer. This restriction is annoying in a cryptographic implementation setting where powers of two are heavily used. Furthermore, some algorithms for 64-bit or 32-bit words cannot be adapted to 63 or 31-bit without a performance loss. The OCaml library also offers modules for 32 or 64-bits integers, but these are wrappers which introduce extra level of indirection and do not offer the same performance as native integers.

Array representation In OCaml, arrays are *boxed*, which means that an array object is actually pointing to a first memory block, composed of two distinct parts: a header, which contains the type of the data stored, the length of the array etc., and the actual memory address of the data. Compared to a C buffer, which is just a memory address, there is an extra indirection which impacts the performance of memory accesses. This feature is great in some settings, because it allows the runtime system to dynamically check that the accesses are within the bounds of the array, which prevents all buffer overruns. But this is precisely what our methodology aims at statically enforcing through a low-level oriented verification framework. Also, because the programmer does not control the management of the program's memory, it is not possible to perform pointer arithmetic natively in F* arrays. This restriction can be worked around using implementation tricks, for instance writing a wrapper around the Array library which records the length of the array and the current address the pseudo-buffer is pointing to. But this introduces yet an extra indirection and, paradoxically, while making the algorithm even closer to the reference C code, hurts the overall performance even more.

4.1.2 Targeted code

For this verification work to get attention from real world cryptographic library users, the resulting code has to be fast, portable and easy to review. The fastest algorithms target the simplest memory model, where memory is freely accessible to the program and the programmer for reading and writing. They do not rely on sophisticated data structures to avoid complexity and potential side-channel flaws, and use the typical word length of common platforms, 32 and 64-bit. This makes C code the dedicated language for reference crypto implementations — assembly is faster but even more difficult to implement properly and very platform specific. Our generic bignum library demonstrates the ability of vanilla F* to verify large, complex and parametrizable low-level projects. However in order to produce more efficient and idiomatic imperative code, a more specialized back-end, which would get rid of the garbage collection and the different memory oddities and indirections, is needed.

In this chapter, we describe Low*, a subset of F* tailored for low-level programming and verification. In practice, Low* is a shallow embedding of a small, sequential, well-behaved subset of C in F*. Low* does not involve any garbage collection or implicit heap allocation; instead, it has a structured memory model à la CompCert, and it provides the control required for writing efficient low-level security-critical code. By virtue of typing, any Low* program is memory safe. In addition, the programmer can make full use of the verification power of F* to write high-level specifications and verify the functional correctness of Low*. At extraction time, specifications and proofs are erased, and the remaining code enjoys a predictable translation to C. We prove that this translation preserves semantics and side-channel resistance. And we provide KreMLin, a new compiler back-end from Low* to C.

4.1.3 Verification Goals

For security-critical low-level programs, there are three main properties which programmers, and of course verification tools, try to tackle to the best of their abilities.

Memory Safety is the most essential security condition for any program. The C programming language for instance is a double-edged sword. Its expressiveness and the low-level programming features it provides to the programmer have made it the language of choice for a wide range of security critical projects, for which portability and performance are essential. Of course, it also entails that its inherently unsafe nature makes C code development extremely error-prone. Unfortunately, memory safety related bugs, such as using dangling pointers or accessing arrays out of their bounds, may result in the worst exploits, such as arbitrary code execution. Sadly, OpenSSL CVEs [1] reveal that

[1] https://www.openssl.org/news/vulnerabilities.html

these are the most common vulnerabilities. As such, memory safety should be the primary focus of all verification efforts aimed at producing or proving secure low-level, manually managed code.

Functional correctness is particularly important in protocols and cryptography. A functional correctness bug will not allow an attacker to corrupt the whole program; it may however have dramatic security consequences as we showed on the TLS state machine [39]. The traditional methods to check for functional correctness of programs are unit-tests and fuzzing methods. However, in the context of cryptography or cryptographic protocols, it happens that the probability that certain events will occur at random is very close to zero. A bug with such a low probability of occurrence would be very unlikely to be caught by random testing or fuzzing, although it could be exploited by a malicious attacker. It is therefore essential that the low-level, platform specific optimized implementation of a program can be precisely related to a shorter, easy to read and to audit functional specification, so that the behavior of the program can be expressed and checked for correctness with minimal efforts.

Side-channel attacks have been more and more widespread over the past decade [105, 120, 128, 32]. They cover all the exploits that can lead to sensitive information leakage, even though the code is perfectly sound from an algorithmic perspective. For instance, if a comparison between an attacker controlled value and a sensitive value is ran bit per bit and fails immediately at the bitwise first difference, the attacker needs only a simple statistical timing analysis to get every bit right. In this scenario, he would in no time retrieve the sensitive data (see section 3.2. Such flaws are difficult to detect and test for, which is why some very low-level tools have been developed [88], to try and detect side-channel vulnerabilities in assembly code. And security-critical low-level program developers try to enforce a constant-time implementation discipline in their code. Similarly, in a formal verification context, as side-channel vulnerabilities cannot be easily captured by functional correctness mechanisms, our verification framework has to embed specific counter measures to account for potential side-channel information leaks.

4.1.4 Designing Low*, a Low-Level Oriented Subset of F*

F* is a general purpose, proof-oriented programming language, and Chapter 2 and 3 illustrated how to write complex functional correctness proofs using the F* proof system. In particular, F* has a customizable memory model and an effect system which will let the programmer specify the model of her choice. As such, F* is a natural candidate to experiment with a specific back-end for low-level verification. On the other hand, the low-level language of choice in this context is the C language, as it is the most portable and the most broadly used at this time, for this kind of code. We know how to prove functional

correctness of low-level algorithms as well as the absence of a certain class of side-channels. The remaining challenge is thus to design libraries that will ensure the correctness of the extraction from F* to C code, as well as a tool to perform such an extraction.

Chapter 2 described the effect mechanisms, and in particular the Ghost effect. By default F* compiles to OCaml code, a functional language very close to the source language. Therefore this compilation phase is mostly comprised of an *erasure* pass: types that do not natively exist in OCaml are appropriately compiled, while other values can immediately be translated to the target language. However, only a fragment of F* can be relevant for C compilation. The computationally irrelevant parts of the code, i.e. all types and proofs, will be erased and therefore will not impact the generated C code in any way. To account for those different remarks, our approach is the following: we design a custom memory model which abstracts the low-level C memory management. We implement libraries to expose C specific constructs such as buffers with pointer arithmetic and we prove the correctness of a translation from a restricted fragment of F* to C code. Eventually we implement a tool, KreMLin, which implements this translation and produces C code from this subset. In the rest of the thesis we will denote as F* the general language, and Low* the language in which computationally relevant code is restricted to the C-compilable subset of F*.

4.2 The Low* language

4.2.1 Designing an Abstraction of the C Memory Model: HyperStack

A Region-based Memory Model The C memory model is structured between a *heap* and a *stack*. In the C calling convention, new stack frames are pushed and popped at function call sites, when entering new blocks, in loops, etc. These stack frames come with particular lifetimes and layouts, and therefore they impact the scope and the lifetimes of the different variables pointing to memory blocks from the stack, and which are bound by the constraints of the frame in which they were initially allocated. The management of those stack frames — and of the data allocated there — is semi-automated in the sense that it is guided by the control flow of the program. On the other hand, the *heap* is designed for more refined memory management: the programmer has control over when heap-based memory blocks are allocated or freed, and thus a function may allocate an object on the heap which will outlive the function call.

Remember that F* provides a STATE monad which allows the programmer to reason about the memory state of the program. This monad is parametrized by the type state, which can be freely defined by the programmer. F* also natively provides two memory models, Heap and HyperHeap (see §2.6.2 for more

```
type mem
type ref : Type → Type
val region_of: ref a → Ghost rid
val `_ ∈ _` : ref a → mem → Tot Type (* a ref is contained in a mem *)
val `_ [_]` : mem → ref a → Ghost a (* selecting a ref *)
val `_ [_] ← _` : mem → ref a → a → Ghost mem (* updating a ref *)
val rref r a = x:ref a {region_of x = r} (* abbrev. for a ref in region r *)
```

Figure 4.1: HyperStack reference type definitions

details). In this setting, however, a more refined and specialized memory model is required to match the previously discussed C behavior. To that intent, in Low* the type state of the STATE monad (see Figure 2.8) is instantiated to HyperStack.mem (which we refer to as just "hyper-stack"), a new region-based memory model [118] covering both the stack and the heap. Hyper-stacks are a generalization of hyperheaps which provide lightweight support for memory separation and framing for stateful verification. They augment hyper-heaps with a shape invariant to indicate that the lifetime of a certain set of regions follows a specific stack-like discipline. They partition memory into a set of regions. Each region is identified by a region identifier rid and regions are classified as either stack regions or heap regions, according to the predicate is_stack_region—we use the type abbreviation sid for stack regions and hid for heap regions. A distinct stack region, root, outlives all other stack regions. The snippet below is the corresponding F* code.

```
type rid
val is_stack_region: rid → Tot bool
type sid = r:rid{is_stack_region r}
type hid = r:rid{¬ (is_stack_region r)}
val root: sid
```

The stack regions are *ephemeral*, and their lifetime is decided by the control flows of the program. Their stack shape invariant guarantees that new stack regions are created and destroyed similarly to stack frames in C, while the heap regions are *eternal* and never get destroyed once created.

The snippet in figure 4.1 shows the signature of mem, our model of the entire memory, which is equipped with a select/update theory [96] for typed references ref a. Additionally, we have a function to refer to the region_of a reference, and a relation r ∈ m to indicate that a reference is live in a given memory state.

Heap regions The heap set of regions is very close to the original HyperHeap model. All the *eternal* regions together form a tree structure, with the root at the top. The goal of this structure is to enable an easy separation mechanism. Indeed, in the case of a single region, the memory separation between

```
val alloc: r:hid → init:a → ST (rref r a) (ensures (λ m0 x m1 → x ∉ m0 ∧ x ∈ m1 ∧
m1 = (m0[x]← init)))
val free: r:hid → x:rref r a → ST unit (requires (λ m → x ∈ m)) (ensures (λ m0 _ m1 →
x ∉ m1 ∧ ∀y≠x. m0[y] = m1[y]))
val (!): x:ref a → ST a (requires (λ m → x ∈ m)) (ensures (λ m0 y m1 → m0 = m1 ∧
y = m1[x]))
```

Figure 4.2: Excerpt of the HyperStack memory API

several references implies predicates which size is quadratic in the number of references. Consequently, such predicates may quickly grow to be impossible to manage. In the *heap* part of the tree, regions are either in a *parent/child* relationship, belong to the same branch, or belong to distinct branches. Regions from distinct branches automatically benefit from memory separation properties: values allocated in distinct regions are also necessarily distinct. Regions on the same branch can be considered distinct or included into one another depending on whether the *distinct* relation is transitive over the parent-children relationship or not. One property of the eternal regions (the heap regions) is that, as its name suggests, they cannot be removed. Once a fresh region has been generated, it will stay forever.

By defining the ST effect over the mem type, we can program stateful primitives for creating new heap regions, and allocating, reading, writing and freeing references in those regions—we show some of their signatures in figure 4.2. Assuming an infinite amount of memory, alloc's pre-condition is trivial while its post-condition indicates that it returns a fresh reference in region r initialized appropriately. Freeing (the free function) and dereferencing (the ! "bang" operator) require their argument to be present in the current memory, eliminating double-free and use-after-free bugs.

Since we support freeing individual references within a region, our model of regions could seem similar to [28]'s *reaps*. However, at present, we do not support freeing heap objects *en masse* by deleting heap regions; indeed, this would require using a special memory allocator. Instead, for us heap regions serve only to *logically* partition the heap in support of separation and modular verification, as is already the case for hyper-heaps [116], and heap region creation is currently compiled to a no-op by KreMLin.

The stack has a more constrained structure; although its base structure is also that of a tree, it has a *stack* invariant which means that it contains only one branch, originating from the root. Each region on the stack is a frame and all frames are by default pairwise distinct. Moreover the stack can grow or decrease throughout the program's lifetime, based on the stack discipline: only the tip, the top most region of the stack can be popped, at the exception of the root.

Stack regions, which we will henceforth call *stack frames*, serve not just as a reasoning device, but provide the efficient C stack-based memory management mechanism. KreMLin maps stack frame creation and destruction directly to the C calling convention and lexical scope. To model this, we extend the signature of mem to include a tip region representing the currently active stack frame, ghost operations to push and pop frames on the stack of an explicitly threaded memory, and their effectful analogs, push_frame and pop_frame that modify the current memory. Finally, we show the signature of salloc which allocates a reference in the current tip stack frame.

```
val tip: mem → Ghost sid
val push: mem → Ghost mem
val pop: m:mem{tip m ≠ root} → Ghost mem
val push_frame: unit → ST unit (ensures (λ m0 () m1 → m1 = push m0))
val pop_frame: unit → ST unit (requires (λ m → tip m ≠ root))
                              (ensures (λ m0 () m1 → m1 = pop m0))
val salloc: init:a → ST (ref a) (ensures (λ m0 x m1 → x ∉ m0 ∧ x ∈ m1
                                  ∧ region_of x = tip m1 ∧ tip m0 = tip m1 ∧ m1 = (m0[x] ← init)))
```

```
let downward_closed (h:HH.t) =
  ∀(r:rid). r `is_in` h (* for any region in the memory *)
  ⟹ (r=HH.root (* either is the root *)
      ∨ (∀ (s:rid). r `is_above` s (* or, any region beneath it *)
          ∧ s `is_in` h (* that is also in the memory *)
          ⟹ (is_stack_region r = is_stack_region s))) (* must be of the same flavor
                                                          as itself *)

let is_tip (tip:HH.rid) (h:HH.t) =
  (is_stack_region tip ∨ tip=HH.root) (* the tip is a stack region, or the root *)
  ∧ tip `is_in` h (* the tip is active *)
  ∧ (∀ (r:sid). r `is_in` h ⟺ r `is_above` tip) (* ny other sid
activation is a above (or equal to)
 the tip *)

let hh = h:HH.t{HH.root `is_in` h ∧ HH.map_invariant h ∧ downward_closed h}
(* the memory itself, always contains the root region, and the parent of any active region
 is active *)

noeq type mem =
  | HS : h:hh
       → tip:rid{tip `is_tip` h} (* the id of the current top−most region *)
       → mem
```

Figure 4.3: HyperStack shape invariants

Custom effects are defined to account for the specificities of this memory model. The ST effect in this memory model is the default effect. It speci-

```
let equal_domains (m0:mem) (m1:mem) =
  m0.tip = m1.tip
  ∧ Set.equal (Map.domain m0.h) (Map.domain m1.h)
  ∧ (∀ r. Map.contains m0.h r ⟹ Heap.equal_dom (Map.sel m0.h r) (Map.sel m1.h r))
```

Figure 4.4: The equal_domains predicate which defines the Stack effect invariant

```
effect Stack (a:Type) (pre:st_pre) (post: (mem → Tot (st_post a))) =
       STATE a
             (λ (p:st_post a) (h:mem) → pre h ∧ (∀ a h1. (pre h ∧ post h a h1 ∧
equal_domains h h1) ⟹ p a h1)) (* WP *)
```

Figure 4.5: Definition of the Stack effect: the 'equal_domains' clause enforces that 1) both mem have the same tip, 2) both mem reference the same heaps (their map: rid -> heap have the same domain), 3) in each region id, the corresponding heaps contain the same references on both sides.

fies that the memory layout of the stack has been left unchanged, while the structure of the heap may have been modified: new regions may have been allocated and new objects may have been freed or allocated in that region. On the stack, this effect maintains an invariant which ensures that nothing has been allocated or freed. Existing values however may be mutated. This corresponds to the semantics of a function call or entering and leaving a new block in C: the structure of the heap may change but no that of the stack which will have the exact same structure before and after the call.

The Stack effect, which definition is shown on figures 4.4 is more restrictive that the default ST effect, but also simpler for the proofs: it maintains the same invariant as ST on the stack but also extends it to the heap. As such, a program annotated as a Stack computation can automatically be lifted to the default ST effect, while the opposite is not possible. Intuitively the Stack effect implies that the underlying computation is entirely stack-based and thus cannot suffer from any memory leaks. It also removes the need for constraining invariants on the heap-allocated objects as those are guaranteed to be live after a Stack computation is ran — as long as it was provably live before. A third effect, StackInline is more complex. It deals with computations which do not respect the Stack effect invariant, but may still be convenient in isolation for the proofs. This typically corresponds to the stack allocations in C, and they are designed in coherence with the fact that they will be automatically and forcefully inlined into the resulting C code.

Memory management is therefore semi-automated. In this model, both liveness and separation annotations have to be provided for the heap part of the memory which may be modified at will. On the other hand, we rely on the

correctness of the Stack invariant to ensure that our resulting code is memory safe and does not suffer from memory leaks. The programmer can freely control the allocation and de-allocation of frames on the stack using the push_frame and pop_frame primitives. Intuitively those correspond respectively to opening and closing *blocks* with curly braces in the C language. They must be called for all functions which allocate on the stack, and systematically matched. They have to be manually inserted by the programmer in such a way that the Stack invariants are maintained. This condition is necessary to typecheck any Low* program.

Although simplified, this abstract memory representation is detailed enough to specify a subset of the C semantics in F*. [107] details the correctness proof between the semantics of the heap and stack in the HyperStack memory model, and the *C-Ligh* CompCert [92] semantics.

Stack computations are ST computations which leave the stack tip unchanged (i.e., they pop all frames they pushed) and yield a final memory state with the same domain as the initial one. This ensures that Low* code with Stack effect has explicitly deallocated all heap allocated references before returning, ruling out memory leaks. As such, we expect all externally callable Low* functions to have Stack effect. External programs can safely pass pointers to objects allocated in their heaps into Low* functions with Stack effect since the definition of Stack forbids the Low* code from freeing these references.

4.2.2 Buffers

One key difference between automatically managed arrays in high level languages such as OCaml and low-level buffers is that the former are packaging the memory address pointing to the data with a header containing safety related information, such as the length of the allocated array. In the latter however, the buffer simply consists of a memory address which is not guaranteed to be readable. Of course, the length of the data it points to is known only to the programmer. The former is safe, but carries an extra level of indirection, the latter is unsafe but memory accesses are faster. The verification goal is thus to ensure a safe use of the later, to benefit from the same security guaranties as the former but with best performance.

In Low*, the encoding of buffer uses similar ideas to the one of arrays in regular F*, but with additional constraints.

```
abstract type buffer a =
  | MkBuffer: max_length:uint32
    → content:ref (s:seq a{Seq.length s = max_length})
    → idx:uint32
    → length:uint32 {idx + length ≤ max_length} → buffer a
```

Figure 4.6: Low* Buffer.buffer representation

```
let chacha20
  (len: uint32{len ≤ blocklen})
  (output: bytes{len = output.length})
  (key: keyBytes)
  (nonce: nonceBytes{disjoint [output; key; nonce]})
  (counter: uint32) : Stack unit
  (requires (λ m0 → output ∈ m0 ∧ key ∈ m0 ∧ nonce ∈ m0))
  (ensures (λ m0 _m1 → modifies1 output m0 m1 ∧
      m1.[output] ==
      Seq.prefix len (Spec.chacha20 m0.[key] m0.[nonce]) counter))) =
    push_frame ();
    let state = Buffer.create 0ul 32ul in
    let block = Buffer.sub state 16ul 16ul in
    chacha20_init block key nonce counter;
    chacha20_update output state len;
    pop_frame ()
```

```
void chacha20 (
  uint32_t len,
  uint8_t *output,
  uint8_t *key,
  uint8_t *nonce,
  uint32_t counter)
{
  uint32_t state[32] = { 0 };
  uint32_t *block = state + 16;
  chacha20_init(block, key, nonce, counter);
  chacha20_update(output, state, len);
}
```

Figure 4.7: A snippet from ChaCha20 in Low* (top) and its C compilation (bottom)

Modeling arrays Hyper-stacks separate heap and stack memory, but each region of memory still only supports abstract, ML-style references (see figure 4.1). A crucial element of low-level programming is control over the specific layout of objects, especially for arrays and structs. We describe first our modeling of arrays by implementing an abstract type for buffers in Low*, using just the references provided by hyper-stacks. Relying on its abstraction, KreMLin compiles our buffers to native C arrays.

The type 'buffer a' in figure 4.6 is a single-constructor inductive type with 4 arguments. Its main content argument holds a reference to a seq a, a purely functional sequence of a's which length is determined by the first argument max_length. The refinement type b:buffer uint32{length b = n} is translated to a C declaration uint32_t b[n] by KreMLin and, relying on C pointer decay, further referred to via uint32_t *. The last two arguments of a buffer are there

to support creating smaller sub-buffers from a larger buffer, via the Buffer.sub operation below. A call to 'Buffer.sub b i l' returning b′ is compiled to C pointer arithmetic b + i (as seen in Figure 4.7 line 13 in chacha20). To accurately model this, the content field is shared between b and b′, but idx and length differ, to indicate that the sub-buffer b′ covers only a sub-range of the original buffer b. The sub operation has computation type Tot, meaning that it does not read or modify the state. The refinement on the result b′ indicates its length and, using the includes relation, records that b and b′ are aliased.

```
val sub: b:buffer a → i:uint32 → len:uint32{i + len ≤ b.length}
         → Tot (b':buffer a{b'.length = len ∧ b `includes` b'})
```

We also provide statically bound-checked operations for indexing and updating buffers. The signature of the index function below requires the buffer to be live and the index location to be within bounds. Its postcondition ensures that the memory is unchanged and describes what is returned in terms of the logical model of a buffer as a sequence.

```
let get (m:mem) (b:buffer a) (i:uint32{i < b.length}) : Ghost a =
  Seq.index (m[b.content]) (b.idx + i)

val index: b:buffer a → i:uint32{i < b.length} → Stack a
  (requires (λ m → b.content ∈ m))
  (ensures (λ m0 z m1 → m1 = m0 ∧ z = get m1 b i))
```

All lengths and indices are 32-bit machine integers, and refer to the number of elements in the buffer, not the number of bytes the buffer occupies. This currently prevents addressing very large buffers on 64-bit platforms. (To this end, we may parameterize our development over a C data model, wherein indices for buffers would reflect the underlying (proper) ptrdiff_t type.)

Similarly, memory allocation remains platform-specific. It may cause a (fatal) error as it runs out of memory. More technically, the type of create may not suffice to prevent pointer-arithmetic overflow; if the element size is greater than a byte, and if the number of elements is 2^{32}, then the argument passed to malloc will overflow on a platform where sizeof size_t == 4. To prevent such cases, KreMLin inserts defensive dynamic checks (which typically end up eliminated by the C compiler since our stack-allocated buffer lengths are compile-time constants). In the future, we may statically prevent it by mirroring the C sizeof operator at the F* level, and requiring that for each Buffer.create operation, the resulting allocation size, in bytes, is no greater than what size_t can hold.

Modeling structs Generalizing 'buffer t' (abstractly, a reference to a finite map from natural numbers to t), we model C-style structs as an abstract reference to a 'struct key value', that is, a map from keys k:key to values whose type

'value k' depends on the key. For example, we represent the type of a colored point as follows, using a struct with two fields X and Y for coordinates and one field Color, itself a nested struct of RGB values.

```
type color_fields = R | G | B
type color = struct color_fields (λ R | G | B → uint32)
type colored_point_fields = X | Y | Color
type colored_point = struct colored_point_fields (λ X | Y → int32 | Color → color)
```

C structs are flatly allocated; the declaration above models a contiguous memory block that holds 20 bytes or more, depending on alignment constraints. As such, we cannot directly perform pointer arithmetic within that block; rather, we navigate it by referring to fields. To this end, our library of structs provides an interface to manipulate pointers to these C-like structs, including pointers that follow a path of fields throughout nested structs. The main type provided by our library is the indexed type ptr shown below, encapsulating a base reference content: ref from and a path p of fields leading to a value of type to.

```
abstract type ptr: Type → Type =
  Ptr: #from:Type → content: ref from → #to: Type → p: path from to → ptr to
```

When allocating a struct on the stack, the caller provides a 'struct k v' literal and obtains a 'ptr (struct k v)', a pointer to a struct literal in the current stack frame (a Ptr with an empty path).

The extend operator below supports extending the access path associated with a 'ptr (struct k v)' to obtain a pointer to one of its fields.

```
val extend: #key: eqtype → #value: (key → Tot Type) → p: ptr (struct key value) → fd: key →
  ST (ptr (value fd))
    (requires (λ h → live h p))
    (ensures (λ h0 p′ h1 → h0 == h1 ∧ p′ == field p fd))
```

Finally, the read and write operations allows accessing and mutating the field referred to by a ptr.

```
val read: #a:Type → p: ptr a → ST value
  (requires (λ h → live h p))
  (ensures (λ h0 v h1 → live h0 p ∧ h0 == h1 ∧ v == as_value h0 p))

val write: #a:Type → b:ptr a → z:a → ST unit
  (requires (λ h → live h b))
  (ensures (λ h0 _h1 → live h0 b ∧ live h1 b ∧ modifies_1 b h0 h1 ∧ as_value h1 b == z))
```

4.2.3 HACL* integers

In F*, unsigned machine words are implemented over bitvectors as well as mathematical integers. The root of their specification is the bijection relationship which exists between bitvectors of n-bit and natural integers in the interval $[0; 2^n[$. Therefore, at the F* level there are Tot API functions available to coerce machine words to and from natural numbers and bitvectors.

As we detailed in the previous section, it is critical to protect sensitive data against side-channel information leaks. In our setting, we are restricting ourselves to only a few C constructs, namely buffers, structs and machine words. Therefore, one of the keys to prevent information leakage is to use a tainting mechanism, very common in information flow theory. Coincidentally this is achievable very easily using F* type system, without involving the SMT solver. The idea is to rely on the type abstraction mechanism of F*. In F* the abstract keyword specifies that the definition of a term shall remain unknown to other module calling into this one. Moreover the noeq key word indicates that one does not wish to define the equality on a particular type. In this setting we will use the actual F* public machine words to specify and implement the Low* sensitive integers while abstracting the relationship between the two and restricting the operators exposed by the sensitive integer modules.

It is difficult to make assumptions on the actual information leakage of integers operators at the processor level. It is extremely dependent on the platform and the manufacturers themselves do not always seem to know precisely which guaranties they provide. Thus we make assumptions about what we deem susceptible to leak information and what we assume not. Should those assumptions be wrong on some platform, we cannot guaranty any form of side channel resistance on this particular platform. Also these assumptions hold at the compiled C level, we cannot provide any kind of guaranties on the compiled binary.

On the sensitive integers we assume the following (constant-time) primitives:

- addition, subtraction and multiplication;
- logical AND, OR, NOT and XOR;
- logical shift right and left;
- *masking* equality functions such as eq_mask specified in figure 4.8.

As shown on figure 4.8 these secret integers are only wrappers around F* integers, and they are implemented using the original F* machine word operations. As explained some operators are not exposed though, such the division and remainder arithmetic operations, too often not constant time, and any kind of comparison function returning a boolean over two secret integer values. Furthermore the function that maps machine words to natural numbers

```
let n = UInt32..n

noeq private type t' = | Mk: v:UInt32..t → t'
type t = t'

let v (x:t) : GTot (FStar.UInt.uint_t n) = UInt32..v x.v

val add: a:t → b:t{UInt.size (v a + v b) n} → Tot (c:t{v a + v b = v c})
let add a b =
  Mk (add (a.v) (b.v))

val add_mod: a:t → b:t → Tot (c:t{(v a + v b) % pow2 n = v c})
let add_mod a b =
  Mk (add_mod (a.v) (b.v))

[...]

assume val eq_mask: a:t → b:t → Tot (c:t{(v a = v b ⟹ v c = pow2 n − 1) ∧ (v a ≠
v b ⟹ v c = 0)})
```

Figure 4.8: Low* sensitive integer representation

is now specified to live in the Ghost effect, which means that any expression relying on that value will be computationally irrelevant and automatically erased. Eventually only there are two kinds of API functions exposed on secret integers: (1) is the kind of total functions over secret integers, which will only return secret integer values and are assumed to be constant time and (2) is the kind of functions which can return other F* types and are necessary for specifications, which are systematically annotated as Ghost. Therefore once a value is lifted to secret integers, it cannot go back to a public value.

4.3 A formal translation from Low* to Clight

Figure 4.9 on page 96 provides an overview of our translation from Low* to CompCert Clight, starting with EMF*, a recently proposed model of F* [9]; then λow*, a formal core of Low* after all erasure of ghost code and specifications; then C*, an intermediate language that switches the calling convention closer to C; and finally to Clight. In the end, our theorems establish that: (a) the safety and functional correctness properties verified at the F* level carry on to the generated Clight code (via semantics preservation), and (b) Low* programs that use the secrets parametrically enjoy the trace equivalence property, at least until the Clight level, thereby providing protection against side-channels.

Prelude: Internal transformations in EMF* We begin by briefly describing a few internal transformations on EMF*, focusing in the rest of this section

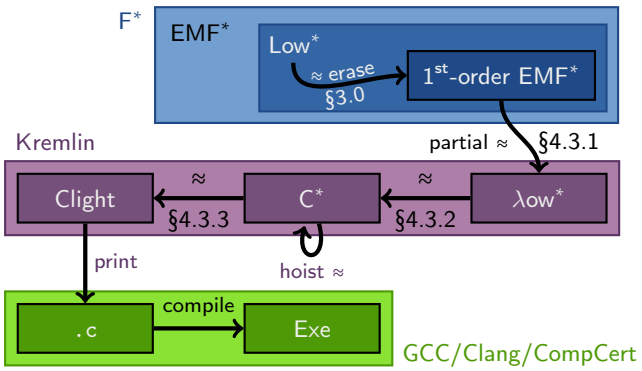


Figure 4.9: Low* embedded in F*, compiled to C, with soundness and security guarantees (details in §4.3)

$$
\begin{aligned}
\tau &::= \mathsf{int} \mid \mathsf{unit} \mid \{\overrightarrow{f = \tau}\} \mid \mathsf{buf}\ \tau \mid \alpha \\
v &::= x \mid n \mid () \mid \{\overrightarrow{f = v}\} \mid (b, n, \overrightarrow{f}) \\
e &::= \mathsf{let}\ x : \tau = \mathsf{readbuf}\ e_1\ e_2\ \mathsf{in}\ e \mid \mathsf{let}\ \_ = \mathsf{writebuf}\ e_1\ e_2\ e_3\ \mathsf{in}\ e \\
&\quad \mid \mathsf{let}\ x = \mathsf{newbuf}\ n\ (e_1 : \tau)\ \mathsf{in}\ e_2 \mid \mathsf{subbuf}\ e_1\ e_2 \\
&\quad \mid \mathsf{let}\ x : \tau = \mathsf{readstruct}\ e_1\ \mathsf{in}\ e \mid \mathsf{let}\ \_ = \mathsf{writestruct}\ e_1\ e_2\ \mathsf{in}\ e \\
&\quad \mid \mathsf{let}\ x = \mathsf{newstruct}\ (e_1 : \tau)\ \mathsf{in}\ e_2 \mid e_1 \rhd f \\
&\quad \mid \mathsf{withframe}\ e \mid \mathsf{pop}\ e \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
&\quad \mid \mathsf{let}\ x : \tau = d\ e_1\ \mathsf{in}\ e_2 \mid \mathsf{let}\ x : \tau = e_1\ \mathsf{in}\ e_2 \mid \{\overrightarrow{f = e}\} \mid e.f \mid v \\
P &::= \cdot \mid \mathsf{let}\ d = \lambda y : \tau_1.\ e : \tau_2, P
\end{aligned}
$$

Figure 4.10: λow* syntax

on the pipeline from λow* to Clight. To express computational irrelevance, we extend EMF* with a primitive Ghost effect. An erasure transformation removes ghost subterms, and we prove that this pass preserves semantics, via a logical relations argument. Next, we rely on a prior result [9] showing that EMF* programs in the ST monad can be safely reinterpreted in $\text{EMF}^{\star}_{\text{ST}}$, a calculus with primitive state. We obtain an instance of $\text{EMF}^{\star}_{\text{ST}}$ suitable for Low* by instantiating its state type with HyperStack.mem. To facilitate the remainder of the development, we transcribe $\text{EMF}^{\star}_{\text{ST}}$ to λow*, which is a restriction of $\text{EMF}^{\star}_{\text{ST}}$ to first-order terms that only use stack memory, leaving the heap out of λow*, since it is not a particularly interesting aspect of the proof. This transcription step is essentially straightforward, but is not backed by a specific proof. We plan to fill this gap as we aim to mechanize our entire proof in the future.

4.3.1 λow*: A Formal Core of Low* Post-Erasure

The meat of our formalization of Low* begins with λow*, a first-order, stateful language, whose state is structured as a stack of memory regions. It has a simple calling convention using a traditional, substitutive β-reduction rule.

Its small-step operational semantics is instrumented to produce traces that record branching and the accessed memory addresses. As such, our traces account for side-channel vulnerabilities in programs based on the program counter model [99] augmented to track potential leaks through cache behavior [21]. We define a simple type system for λow* and prove that programs well-typed with respect to some values at an abstract type produce traces independent of those values, e.g., our bigint library when translated to λow* is well-typed with respect to an abstract type of limbs and leaks no information about them via their traces.

Syntax Figure 4.10 shows the syntax of λow*. A program P is a sequence of top-level function definitions, d. We omit loops but allow recursive function definitions. Values v include constants, immutable records, and buffers $(b, n, [])$ and mutable structures $(b, n, \overrightarrow{f})$ passed by reference, where b is the address of the buffer or structure, n is the offset in the buffer, and $\overrightarrow{f}$ designates the path to the structure field to take a reference of (this path, as a list, can be longer than 1 in the case of nested mutable structures.) Stack allocated buffers (readbuf, writebuf, newbuf, and subbuf), and their mutable structure counterparts (readstruct, writestruct, newstruct, ▷), are the main feature of the expression language, along with withframe e, which pushes a new frame on the stack for the evaluation of e, after which it is popped (using pop e, an administrative form internal to the calculus). Once a frame is popped, all its local buffers and mutable structures become inaccessible.

Mutable structures can be nested, and stored into buffers, in both cases without extra indirection. However, the converse is not true, as λow* currently does not allow arbitrary nesting of arrays within mutable structures without explicit indirection via separately allocated buffers. We leave such generalization as future work.

Type system λow* types include the base types int and unit, record types $\{\overrightarrow{f = \tau}\}$, buffer types buf τ, mutable structure types struct τ, and abstract types α. The typing judgment has the form, $\Gamma_P; \Sigma; \Gamma \vdash e : \tau$, where Γ_P includes the function signatures; Σ is the store typing; and Γ is the usual context of variables. We elide the rules, as it is a standard, simply-typed type system. The type system guarantees preservation, but not progress, since it does not attempt to account for bounds checks or buffer/mutable structure lifetime. However, memory safety (and progress) is a consequence of Low* typing and its semantics-preserving erasure to λow*.

Semantics We define evaluation contexts E for standard call-by-value, left-to-right evaluation order. The memory H is a stack of frames, where each frame maps addresses b to a sequence of values $\overrightarrow{v}$. The λow* small-step semantics judgment has the form $P \vdash (H, e) \rightarrow_\ell (H', e')$, meaning that under

$$\frac{}{P \vdash (H, \mathsf{withframe}\ e) \rightarrow_{\cdot} (H; \{\}, \mathsf{pop}\ e)}\ \textsc{WF} \qquad \frac{}{P \vdash (H; \_, \mathsf{pop}\ v) \rightarrow_{\cdot} (H, v)}\ \textsc{Pop}$$

$$\frac{}{P \vdash (H, \mathsf{if}\ 0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \rightarrow_{\mathsf{brF}} (H, e_2)}\ \textsc{IfF}$$

$$\frac{P(f) = \lambda y : \tau_1.\ e_1 : \tau_2}{P \vdash (H, \mathsf{let}\ x : \tau = f\ v\ \mathsf{in}\ e) \rightarrow (H, \mathsf{let}\ x : \tau = e_1[v/y]\ \mathsf{in}\ e)}\ \textsc{App}$$

$$\frac{H(b, n + n_1, []) = v \qquad \ell = \mathsf{read}(b, n + n_1, [])}{P \vdash (H, \mathsf{let}\ x = \mathsf{readbuf}\ (b, n, [])\ n_1\ \mathsf{in}\ e) \rightarrow_{\ell} (H, e[v/x])}\ \textsc{Rd}$$

$$\frac{\begin{array}{c} b \notin \mathsf{dom}(H; h) \quad h_1 = h[b \mapsto v^n] \quad e_1 = e[(b, 0)/x] \\ \ell = \mathsf{write}(b, 0), \ldots, \mathsf{write}(b, n-1) \end{array}}{P \vdash (H; h, \mathsf{let}\ x = \mathsf{newbuf}\ n\ (v : \tau)\ \mathsf{in}\ e) \rightarrow_{\ell} (H; h_1, e_1)}\ \textsc{New}$$

Figure 4.11: Selected semantic rules from λow*

the program P, configuration (H, e) steps to (H', e') emitting a trace ℓ, including reads and writes to buffer references or mutable structure references, and branching behavior, as shown below.

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

Figure 4.11 shows selected reduction rules from λow*. Rule WF pushes an empty frame on the stack, and rule Pop pops the topmost frame once the expression has been evaluated. Rule IfF is standard, except for the trace brF recorded on the transition. Rule App is a standard, substitutive β-reduction. Rule Rd returns the value at the $(n + n_1)$ offset in the buffer at address b, and emits a $\mathsf{read}(b, n + n_1, [])$ event. Rule New initializes the new buffer, and emits write events corresponding to each offset in the buffer.

Secret independence A λow* program can be written against an interface providing secrets at an abstract type. For example, for the abstract type limb, one might augment the function signatures Γ_P of a program with an interface for the abstract type $\Gamma_{\mathsf{limb}} = \mathsf{eq\_mask} : \mathsf{limb}^2 \rightarrow \mathsf{limb}$, and typecheck a source program with free limb variables ($\Gamma = \mathsf{secret{:}limb}$), and empty store typing, using the judgment $\Gamma_{\mathsf{limb}}, \Gamma_p; \cdot; \Gamma \vdash e : \tau$. Given any representation τ for limb, an implementation for eq_mask whose trace is input independent, and any pair of values $v_0 : \tau, v_1 : \tau$, we prove that running $e[v_0/\mathsf{secret}]$ and $e[v_1/\mathsf{secret}]$ produces identical traces, i.e., the traces reveal no information about the secret v_i. We sketch the formal development next, leaving details to the appendix.

Given a derivation $\Gamma_s, \Gamma_P; \Sigma; \Gamma \vdash e : \tau$, let Δ map type variables in the interface Γ_s to concrete types and let P_s contain the implementations of the

functions (from Γ_s) that operate on secrets. To capture the secret independence of P_s, we define a notion of *equivalence modulo secrets*, a type-indexed relation for values ($v_1 \equiv_\tau v_2$) and memories ($\Sigma \vdash H_1 \equiv H_2$). Intuitively two values (resp. memories) are equivalent modulo secrets if they only differ in subterms that have abstract types in the domain of the Δ map—we abbreviate "equivalent modulo secrets" as "related" below. We then require that each function $f \in P_s$, when applied in related stores to related values, always returns related results, while producing *identical* traces. Practically, P_s is a (small) library written carefully to ensure secret independence.

Our secret-independence theorem is then as follows:

Theorem 4.3.1 (Secret independence)**.** *Given*

1. *a program well-typed against a secret interface, Γ_s, i.e, $\Gamma_s, \Gamma_P; \Sigma; \Gamma \vdash (H, e) : \tau$,*
2. *a well-typed implementation of the Γ_s interface, $\Gamma_s; \Sigma; \cdot \vdash_\Delta P_s$, such that P_s is equivalent modulo secrets,*
3. *a pair (ρ_1, ρ_2) of well-typed substitutions for Γ,*

then either:

1. *both programs cannot reduce further, i.e. $P_s, P \vdash (H, e)[\rho_1] \not\twoheadrightarrow$ and $P_s, P \vdash (H, e)[\rho_2] \not\twoheadrightarrow$, or*
2. *both programs make progress with the same trace, i.e. there exists $\Sigma' \supseteq \Sigma, \Gamma' \supseteq \Gamma, H', e'$, a pair (ρ_1', ρ_2') of well-typed substitutions for Γ', and a trace ℓ such that*
   a) *$P_s, P \vdash (H, e)[\rho_1] \to_\ell^+ (H', e')[\rho_1']$ and $P_s, P \vdash (H, e)[\rho_2] \to_\ell^+ (H', e')[\rho_2']$, and*
   b) *$\Gamma_s, \Gamma_P; \Sigma'; \Gamma' \vdash (H', e') : \tau$*

4.3.2 C*: An Intermediate Language

We move from λow* to Clight in two steps. The C* intermediate language retains λow*'s explicit scoping structure, but switches the calling convention to maintain an explicit call-stack of continuations (separate from the stack memory regions). C* also switches to a more C-like syntax, separates side effect-free expressions from effectful statements.

$$
\begin{array}{rl}
\hat{P} & ::= \overrightarrow{\mathsf{fun}\ f\ (x : \tau) : \tau\ \{\ \overrightarrow{s}\ \}} \\
\hat{e} & ::= n \mid () \mid x \mid \hat{e} + \hat{e} \mid \{\overrightarrow{f = \hat{e}}\} \mid \hat{e}.f \mid \&\hat{e} \to f \\
s & ::= \tau\ x = \hat{e} \mid \tau\ x = f(\hat{e}) \mid \mathsf{if}\ \hat{e}\ \mathsf{then}\ \overrightarrow{s}\ \mathsf{else}\ \overrightarrow{s} \mid \mathsf{return}\ \hat{e} \\
& \mid \{\overrightarrow{s}\} \mid \tau\ x[n] \mid \tau\ x = *[\hat{e}] \mid *[\hat{e}] = \hat{e} \mid \mathsf{memset}\ \hat{e}\ n\ \hat{e}
\end{array}
$$

The syntax is unsurprising, with two notable exceptions. First, despite the closeness to C syntax, contrary to C and similarly to λow*, block scopes are not required for branches of a conditional statement, so that any local variable or local array declared in a conditional branch, if not enclosed by a further block, is still live after the conditional statement. Second, non-array local variables are immutable after initialization.

$$\frac{}{\hat{P} \vdash (S, V, \{\overrightarrow{s_1}\}; \overrightarrow{s_2}) \rightsquigarrow (S; (\{\}, V, \square; \overrightarrow{s_2}), V, \overrightarrow{s_1})}\ \textsc{Block}$$

$$\frac{}{\hat{P} \vdash (S; (M, V', E), V, []) \rightsquigarrow (S, V', E\,[()])}\ \textsc{Empty}$$

$$\frac{[\![\hat{e}]\!]_{(V)} = 0}{\hat{P} \vdash (S, V, \mathsf{if}\ \hat{e}\ \mathsf{then}\ \overrightarrow{s_1}\ \mathsf{else}\ \overrightarrow{s_2}; \overrightarrow{s}) \rightsquigarrow_{\mathsf{brF}} (S, V, \overrightarrow{s_2}; \overrightarrow{s})}\ \textsc{CIfF}$$

$$\frac{\hat{P}(f) = \mathsf{fun}\ (y : \tau_1) : \tau_2\ \{\ \overrightarrow{s_1}\ \} \qquad [\![\hat{e}]\!]_{(V)} = v}{\hat{P} \vdash (S, V, \tau\ x = f\ \hat{e}; \overrightarrow{s}) \rightsquigarrow (S; (\bot, V, \tau\ x = \square; \overrightarrow{s}), \{\}[y \mapsto v], \overrightarrow{s_1})}\ \textsc{Call}$$

$$\frac{[\![\hat{e}]\!]_{(V)} = (b, n, \overrightarrow{f}) \quad \mathsf{Get}(S, (b, n, \overrightarrow{f})) = v \quad \ell = \mathsf{read}\ (b, n, \overrightarrow{f})}{\hat{P} \vdash (S, V, \tau\ x = *[\hat{e}]; \overrightarrow{s}) \rightsquigarrow_{\ell} (S, V[x \mapsto v], \overrightarrow{s})}\ \textsc{CRead}$$

$$\frac{S = S'; (M, V, E) \qquad b \notin S \qquad V' = V[x \mapsto (b, 0, [])]}{\hat{P} \vdash (S, V, \tau\ x[n]; \overrightarrow{s}) \rightsquigarrow (S'; (M[b \mapsto \bot^n], V, E), V', \overrightarrow{s})}\ \textsc{ArrDecl}$$

Figure 4.12: Selected semantic rules from C*

Operational semantics, in contrast to λow* A C* evaluation configuration C consists of a stack S, a variable assignment V and a statement list $\overrightarrow{s}$ to be reduced. A stack is a list of frames. A frame F includes frame memory M, local variable assignment V to be restored upon function exit, and continuation E to be restored upon function exit. Frame memory M is optional: when it is $\bot$, the frame is called a "call frame"; otherwise a "block frame", allocated whenever entering a statement block and deallocated upon exiting such block. A frame memory is just a partial map from block identifiers to value lists. Each C* statement performs at most one function call, or otherwise, at most one side effect. Thus, C* is deterministic.

The semantics of C* is shown to the right in Figure 4.12, also illustrating the translation from λow* to C*. There are three main differences. First, C*'s calling convention (rule Call) shows an explicit call frame being pushed on the stack, unlike λow*'s β reduction. Additionally, C* expressions do not have side effects and do not access memory; thus, their evaluation order does not matter and their evaluation can be formalized as a big-step semantics; by themselves, expressions do not produce events. This is apparent in rules like CIfF and CRead, where the expressions are evaluated atomically in the premises. Finally, newbuf in λow* is translated to an array declaration followed by a separate initialization. In C*, declaring an array allocates a fresh memory block in the current memory frame, and makes its memory locations available but uninitialized. Memory write (resp. read) produces a write (resp. read)

event. memset $\hat{e}_1$ m $\hat{e}_2$ produces m write events, and can be used only for arrays.

Correctness of the λow*-to-C* transformation We proved that execution traces are exactly preserved from λow* to C*:

Lemma 4.3.2 (λow* to C*)**.** *Let P be a λow* program and e be a λow* entry point expression, and assume that they compile: $\Downarrow(P) = \hat{P}$ for some C* program $\hat{P}$ and $\downarrow(e) = \overrightarrow{s}; \hat{e}$ for some C* list of statements $\overrightarrow{s}$ and expression $\hat{e}$.*

Let V be a mapping of local variables containing the initial values of secrets. Then, the C program $\hat{P}$ terminates with trace ℓ and return value v, i.e., $\hat{P} \vdash ([], V, \overrightarrow{s}; \mathsf{return}\ \hat{e}) \overset{\ell,*}{\to} ([], V', \mathsf{return}\ v)$ if, and only if, so does the λow* program: $P \vdash (\{\}, e[V]) \overset{\ell,*}{\to} (H', v)$; and similarly for divergence.*

In particular, if the source λow* program is safe, then so is the target C* program. It also follows that the trace equality security property is preserved from λow* to C*. We prove this theorem by bisimulation. In fact, it is enough to prove that any λow* behavior is a C* behavior, and flip the diagram since C* is deterministic. That C* semantics use big-step semantics for C* expressions complicates the bisimulation proof a bit because λow* and C* steps may go out-of-sync at times. Within the proof we used a relaxed notion of simulation ("quasi-refinement") that allows this temporary discrepancy by some stuttering, but still implies bisimulation.

4.3.3 From C* to CompCert Clight and Beyond

CompCert Clight is a deterministic (up to system I/O) subset of C with no side effects in expressions, and actual byte-level representation of values. Clight has a realistic formal semantics [50, 91] and tractable enough to carry out the correctness proofs of our transformations from λow* to C. More importantly, Clight is the source language of the CompCert compiler backend, which we can thus leverage to preserve at least safety and functional correctness properties of Low* programs down to assembly.[2]

Recall that we need to produce an event in the trace whenever a memory location is read or written, and whenever a conditional branch is taken, to account for memory accesses and statements in the semantics of the generated Clight code for the purpose of our noninterference security guarantees. However, the semantics of CompCert Clight *per se* produces no events on memory accesses; instead, CompCert provides a syntactic program annotation mechanism using no-op *built-in calls*, whose only purpose is to add extra events in the trace. Thus, we leverage this mechanism by prepending each memory

[2] As a subset of C, Clight can be compiled by any C compiler, but only CompCert provides formal guarantees.

access and conditional statement in the Clight generated code with one such built-in call producing the corresponding events.

The main two differences between C* and Clight, which our translation deals with as described below, are immutable local structures, and scope management for local variables.

Immutable local structures C* handles immutable local structures as first-class values, whereas Clight only supports non-compound data (integers, floating-points or pointers) as values.

If we naively translate immutable local C* structures to C structures in Clight, then CompCert will allocate them in memory. This increases the number of memory accesses, which not only introduces discrepancies in the security preservation proof from C* to Clight, but also introduces significant performance overhead compared to GCC, which optimizes away structures whose addresses are never taken.

Instead, we split an immutable structure into the sequence of all its non-compound fields, each of which is to be taken as a potentially non-stack-allocated local variable,[3] except for functions that return structures, where, as usual, we add, as an extra argument to the callee, a pointer to the memory location written to by the callee and read by the caller.

Local variable hoisting Scoping rules for C* local arrays are not exactly the same as in C, in particular for branches of conditional statements. So, it is necessary to hoist all local variables to function-scope. CompCert 2.7.1 does support such hoisting but as an unproven elaboration step. While existing formal proofs (e.g., Dockins' [66, §9.3]) only prove functional correctness, we also prove preservation of security guarantees, as shown below.

Proof techniques Contrary to the λow*-to-C* transformation, our subsequent passes modify the memory layout leading to differences in traces between C* to Clight, due to pointer values. Thus, we need to address security preservation separately from functional correctness.

For each pass changing the memory layout, we split it into three passes. First, we *reinterpret* the program by replacing each pointer value in event traces with the function name and recursion depth of its function call, the name of the corresponding local variable, and the array index and structure field name within this local variable. Then, we perform the actual transformation and prove that it exactly preserves traces in this new "abstract" trace model. Finally, we reinterpret the generated code back to concrete pointer values. We

[3] Our benchmark without this structure erasure runs 20% slower than with structure erasure, both with CompCert 2.7. Without structure erasure, code generated with CompCert is 60% slower than with `gcc -O1`. CompCert-generated code without structure erasure may even segfault, due to stack overflow, which structure erasure successfully overcomes.

successfully used this technique to prove functional correctness and security preservation for variable hoisting.

For each pass that adds new memory accesses, we split it into two passes. First, a reinterpretation pass produces new events corresponding to the provisional memory accesses (without actually performing those memory accesses). Then, this pass is followed by the actual trace-preserving transformation that goes back to the non-reinterpreted language but adds the actual memory accesses into the program. We successfully used this technique to prove functional correctness and security preservation for structure return, where we add new events and memory accesses whenever a C* function returns a structure value.

In both cases, we mean *reinterpretation* as defining a new language with the same syntax and small-step semantic rules except that the produced traces are different, and relating executions of the same program in the two languages. There, it is easy to prove functional correctness, but for security preservation, we need to prove an invariant on two small-step executions of the same program with different secrets, to show that two equal pointer values in event traces coming from those two different executions will actually turn into two equal abstract pointer values in the reinterpreted language.

Our detailed functional correctness and security preservation proofs from λow* to Clight can be found in the appendix.

Towards verified assembly code We conjecture that our reinterpretation techniques can be generalized to most passes of CompCert down to assembly. While we leave such generalization as future work, some guarantees from C to assembly can be derived by instrumenting CompCert [21] and LLVM [126, 127, 11] and turning them into *certifying* (rather than certified) compilers where security guarantees are statically rechecked on the compiled code through translation validation, thus re-establishing them independently of source-level security proofs. In this case, rather than being fully preserved down to the compiled code, Low*-level proofs are still useful to *practically* reduce the risk of failures in translation validation.

4.4 The KreMLin compiler

4.5 KreMLin: a Compiler from Low* to C

4.5.1 From Low* to Efficient, Elegant C

As explained previously, λow* is the core of Low*, post erasure. Transforming Low* into λow* proceeds in several stages. First, we rely on F*'s existing normalizer and erasure and extraction facility (similar to features in Coq [94]), to obtain an ML-like AST for Low* terms. Then, we use our new tool KreMLin

that transforms this AST further until it falls within the λow* subset formalized above. KreMLin then performs the λow* to C* transformation, followed by the C* to C transformation and pretty-printing to a set of C files. KreMLin generates C11 code that may be compiled by GCC; Clang; Microsoft's C compiler or CompCert. We describe the main transformations performed by KreMLin, beyond those formalized in §4.3, next.

Structures by value We described earlier (§4.2.2) our Low* struct library that grants the programmer fine-grained control over the memory layout, as well as mutability of interior fields. As an alternative, KreMLin supports immutable, by-value structs. Such structures, being pure, come with no liveness proof obligations. The performance of the generated C code, however, is less predictable: in many cases, the C compiler will optimize and pass such structs by reference, but on some ABIs (x86), the worst-case scenario may be costly.

Concretely, the F* programmer uses tuples and inductive types. Tuples are monomorphized into specialized inductive types. Then, inductive types are translated into idiomatic C code: single-branch inductive types (e.g., records) become actual C structs, inductives with only constant constructors become C enums, and other inductives becomes C tagged unions, leveraging C11 anonymous unions for syntactic elegance. Pattern matches become, respectively, switches, let-bindings, or a series of cascading if-then-elses.

Whole-program transformations KreMLin perform a series of whole-program transformations. First, the programmer is free to use parameterized type abbreviations. KreMLin substitutes an application of a type abbreviation with its definition, since C's typedef does not support parameters. (C++11 alias templates would support this use-case.) Second, KreMLin recursively inlines all StackInline functions, as required for soundness (cf. §**??**). Third, KreMLin performs a reachability analysis. Any function that is not reachable from the main function or, in the case of a library, from a distinguished API module, is dropped. This is essential for generating palatable C code that does not contain unused helper functions used only for verification. Fourth, KreMLin supports a concept of "bundle", meaning that several F* modules may be grouped together into a single C translation unit, marking all of the functions as static, except for those reachable via the distinguished API module. This not only makes the code much more idiomatic, but also triggers a cascade of optimizations that the C compiler is unable to perform across translation units.

Going to an expression language F* is, just like ML, an expression language. Two transformations are required to go to a statement language: *stratification* and *hoisting*. Stratification places buffer allocations, assignments and conditionals in statement position before going to C*. Hoisting, as dis-

cussed in §4.3.3, deals with the discrepancy between C99 block scope and Low* with_frame; a buffer allocated under a then branch must be hoisted to the nearest enclosing push_frame, otherwise its lifetime would be shortened by the resulting C99 block after translation.

Readability KreMLin puts a strong emphasis on generating readable C, in the hope that security experts not familiar with F* can review the generated C code. Names are preserved; we use enum and switch whenever possible; functions that take unit are compiled into functions with no parameters; functions that return unit are compiled into void-returning functions. The internal architecture relies on an abstract C AST and what we believe is a correct C pretty-printer.

Implementation KreMLin represents about 10,000 lines of OCaml, along with a minimal set of primitives implemented in a few hundred lines of C. After F* has extracted and erased the AEAD development, KreMLin takes less than a second to generate the entire set of C files. The implementation of KreMLin is optimized for readability and modularity; there was no specific performance concern in this first prototype version. KreMLin was designed to support multiple backends; we are currently implementing a WebAssembly backend to provide verified, efficient cryptographic libraries for the web.

4.5.2 Integrating KreMLin's Output

KreMLin generates a set of C files that have no dependencies, beyond a single .h file and C11 standard headers, meaning KreMLin's output can be readily integrated into an existing source tree.

To allow code sharing and re-use, programmers may want to generate a shared library, that is, a .dll or .so file that can be distributed along with a public header (.h) file. The programmer can achieve this by writing a distinguished API module in F*, exposing only carefully-crafted function signatures. As exemplified earlier (Figure 4.7), the translation is predictable, meaning the programmer can precisely control, in F*, what becomes, in C, the library's public header. The bundle feature of KreMLin then generates a single C file for the library; upon compiling it into a shared object, the only visible symbols are those exposed by the programmer in the header file.

We used this approach for our HACL* library. Our public header file exposes functions that have the exact same signature as their counterpart in the NaCL library. If an existing binary was compiled against NaCL's public header file, then one can configure the dynamic linker to use our HACL* library instead, without recompiling the original program (using the infamous "LD preload trick").

The functions exposed by the library comply with the C ABI for the chosen toolchain. This means that one may use the library from a variety of programming languages, relying on foreign-function interfaces to interoperate.

One popular approach is to generate bindings for the C library *at run-time* using the ctypes and the libffi [73] libraries. This is an approach leveraged by languages such as JavaScript, Python or OCaml, and requires no recompilation.

An alternative is to write bindings by hand, which allows for better performance and control over how data is transformed at the boundary, but requires writing and recompiling potentially error-prone C code. This is the historical way of writing bindings for many languages, including OCaml. We plan to have KreMLin generate these bindings automatically. We used this approach in miTLS, effectively making it a mixed C/OCaml project. We intend to eventually lower all of miTLS into Low*.

Conclusion

While F* is usually recognized for its expressiveness and its proof automation, we showed that it is also a suitable candidate to write efficient low-level code. Low-level languages give a lot of control to the programmer, which makes them fast, but also insecure and hard to verify. On the other hand, higher order functional languages are safe and provide many useful verification features, at the cost of complete control over the execution of the code.

In this Chapter, relying on Low* a shallow embedding of C in F*, we showed how to effectively write C (and thus as-fast-as-C) code in F*, while retaining the full proof capabilities of F*. The verification and compilation process effectively produces C code which is memory safe, functionally correct and provides secret-independence guaranties to mitigate side-channel attacks. The compilation process from F* to C is proven correct and a strong effort has been made to ensure that the resulting code is human readable and auditable.

One limitation of this approach to verify C code is that one has to reimplement everything in F*, we cannot directly prove the correctness of existing idiomatic C code. However, we claim that our approach is better suited to scale to large projects, as it is particularly hard to prove code that was not written with verification in mind, and that tools directly targeting low-level code are less expressive that high-level languages. Still, our Low* frontend is effectively F*, a functional language largely unknown to developers. For the computationally relevant code, a C or Rust front-end could be a substantial improvement to push the adoption of our verification framework, while the proof itself could stay in vanilla F*.

Chapter 5

HACL*, a Fast and Verified Reference Cryptographic Library

Parts of the text are taken from [130], a paper that appeared in CCS 2017 and was co-authored by me along with Jonathan Protzenko, Karthikeyan Bhargavan and Benjamin Beurdouche.

Chapter 3 presented a new approach towards generic cryptographic code for fixed prime fields in F* which minimizes the proof burden at no algorithmic performance cost. Chapter 4 showed that such code could safely be compiled to efficient low-level languages, in particular to C code, in which most common cryptographic libraries are implemented. Leveraging on these results, this chapter illustrates how to combine generic cryptographic proofs with low-level compilation to implement a full-fledged cryptographic library. We show that our methods are not limited to prime-field arithmetic and easily extend to symmetric ciphers or complex cryptographic constructions such as Ed25519, a twisted Edward Curve based digital signature algorithm. In particular, this cryptographic library implements the full NaCl API, a modern small but complete general purpose API for cryptographic applications.

5.1 A self-contained, modern and efficient cryptographic library

Implementing cryptographic code is notoriously hard and error-prone. This claim is easily backed by a quick search for vulnerabilities in open source cryptographic code over the internet. It may be surprising given how security critical cryptographic code is, and how many pairs of eyes one can hope to get on widely used open source projects such as the OpenSSL cryptographic

protocol library.

Unfortunately crypto developers have to balance between two antithetical goals. *Security* is obviously the main objective of cryptographic code and constructs, but *performance* is also essential. The TLS protocol for instance is one of the main secure protocols used over the internet. Its goal is to negotiate and create a secure channel over which two parties can safely communicate under confidentiality, integrity and authentication guarantees. In a world where all internet communications are slowly moving toward encryption, it is easy enough to understand why performance becomes critical for crypto code, especially for large content providers. Unfortunately the fastest algorithms are rarely the most secure. Certain optimizations may lead to either unwanted and potentially exploitable behaviors, or side-channel information leaks. From there, striking a balance between safe code and fast code becomes quite challenging and it is safe to say that only few people are actually trusted by the community to contribute new cryptographic code to popular cryptographic projects.

Our approach for this cryptographic library is thus a bit different. Rather than trusting the programmer, we advocate trusting the machine and letting the programmer rely on the proof checker to ascertain the correctness of her implementation. To that intent we leverage on the Low* subset of F* presented in Chapter 4 to ensure statically by typing that our compiled C code is safe and provides clear guaranties regarding its functional behavior, its memory discipline and information leakage by timing mitigation technique.

This library is not intended as a proof of concept but rather as a framework to build upon and to expand. Old and pervasive projects such as the OpenSSL library are too large for one to hope to thoroughly verify them in the coming years. More recent projects have aimed at offering a more restricted but carefully chosen set of cryptographic primitives which enable the main security guarantees needed by cryptographic applications. The NaCl API is such. Already implemented by libraries such as LibSodium or TweetNaCl, it relies on a small set of modern, recently standardized primitives. In HACL* we provide symmetric ciphers (Chacha20, Salsa20), hash functions (SHA2-256, SHA2-512), Message Authenticating Codes (MAC: Poly1305, HMAC), Elliptic Curve Diffie-Hellman (Curve25519) and Edwards Curve based digital signatures (Ed25519). Thanks to these primitives, not only can HACL* implement the NaCl API, but also one ciphersuite of the recently standardized TLS 1.3 secure communication protocol.

In the rest of the chapter we describe how the HACL* library is structured in order to be extensible, to provide a clear understanding of the security guaranties to external users and contributors and how it achieves state-of-the-art C performance on top of practical verification.

5.2 Methodology: Structuring the Library Between Specifications and Low-Level Code

The overall goal of this work is to illustrate the worth of formal methods for software development, democratize their use and improve the level of trust we can put into daily used software. Therefore, an essential milestone on the road to achieving this goal is to formulate the security and functionality guarantees in such a way that non F* expert users and contributors will be convinced — provided that they trust the formal method tools — that that they understand the security claims well and that they match their expectations. To that intent, we designed a self-contained, full-fledged cryptographic library built around three inter-dependent components.

F* specifications The central property of a verified program is functional correctness. In essence, HACL* claims that its C compiled code has a *correct* input/output behavior, whatever the input values may be. This notion of *correctness* is relative to a *reference program* which is trusted to have the intended behavior. All other functionally verified implementations are proved to be functionally equivalent to this reference program which we will designate as the *specification*.

Therefore in HACL*, a *specification* is a pure, simple and standalone F* program which aim is to provide to external readers a description of the primitive as clear, as concise and as readable as possible. Cryptographic primitive standards, for instance *Requests For Comments* (RFC) or NIST documents, tend to provide some pseudo-code or Python excerpts of the specified primitive in addition to the textual description in order to make it clearer, guide the developers through the implementation process and lift any potential ambiguity. HACL* specifications' aim is to fulfill the exact same goal. Because they are trusted and all other implementations of the same primitive in HACL* is functionally equivalent to them, they must receive special attention from external reviewers. Conveniently, they are the only pieces of code one has to look at to know the exact behavior of the library functions.

Low* code Limitations of the pure fragment of F*, in which the specifications are implemented, have been broadly discussed in Chapter 2. Therefore while specifications are great to provide concise and readable F* descriptions of a cryptographic primitive, the end goal of HACL* is to rival with the reference cryptographic libraries from the real world. To that intent, concrete HACL* code is written in the Low* subset and compiles to C code. It is memory-safe (by virtue of typing in the Low* subset) and its correctness comes from the soundness of the Low* to C compilation as well as the functional equivalence proof with the corresponding F* specification. Contrary to the specification, the Low* code implements all the low-level optimizations one could find in a

C implementation of the same primitive which is why it can be compiled to efficient C code. It can also be used in larger F* developments which may then benefit from all the invariants guaranteed for this code: functional correctness (by functional equivalence with the specifications), memory safety (by typing in Low*), and timing leakage mitigation techniques (by typing).

C code The Low* code is an intermediate proof artifact: a Low* program closely models the C memory model and thus can produce C code, and it belongs to the F* system which enables proofs. However it is not natively executable, and has to go through a compilation phase beforehand. For users who do not care about reusing the proven Low* invariants, this Low* code has little value. Rather, these users will be interested in the produced C code directly, which is guaranteed to be safe, and in the F* specification, which expresses the behavior of a primitive in a less verbose manner than optimized machine generated C code. HACL* distributes snapshots of already compiled from Low*, ready-to-use C code which can be integrated in existing developments. In particular HACL* exposes the NaCl API, a modern cryptographic API with few but carefully chosen cryptographic primitives.

5.3 High-Level Specifications

5.3.1 Reference code

The HACL* library is built around functional equivalence proofs between simple, pure F* programs and complex stateful Low* programs. The former, the specifications, are meant to be easy to read, to understand, to review and to trust. They use only the pure fragment of F* and a limited set of primitives which should make them very accessible and readable, even to users who are stranger to the F* language. Familiarity with functional programming would of course help with the syntax. The later, the C compilable code, is much more complex and has to be proven not only functionally correct, but also memory safe and timing-leakage resistant. The relationship between the high-level specification and the low-level code is shown on figure 5.1 and figure 5.2, which display the top-level function for the Curve25519 algorithm in both settings.

Curve25519 is an elliptic curve point scalar multiplication algorithm which enables a variant of the Diffie-Hellman algorithm on elliptic curves (ECDH). The goal of this cryptographic construct is to negotiate a shared secret on an insecure channel. The Curve25519 elliptic curve can be equipped with a group structure and through a series of coordinate computations, from the `x` coordinate of a point `P` on the curve and a scalar `k1` one can compute the `x` coordinate of the point `kP` where `kP` is `P`+`P`+...+`P` `k` times, and + is the group additive law. The key idea here is that the scalar multiplication is difficult to invert. Hence from `P` and `kP` retrieving `k` is difficult for an adversary. Thereon, two parties Alice and Bob may negotiate a shared secret if `P` being public,

```
module Spec.Curve25519
[...]

(* Type aliases *)
type scalar = lbytes 32 // Sequence of bytes of length 32
type serialized_point = lbytes 32

let scalarmult (k:scalar) (u:serialized_point) : Tot serialized_point =
let k = decodeScalar25519 k in
let u = decodePoint u in
let res = montgomery_ladder u k in
encodePoint res
```

Figure 5.1: F* top-level specification function for Curve25519

```
module Curve25519
[...]

val crypto_scalarmult:
  mypublic:uint8_p{length mypublic = 32} →
  secret:uint8_p{length secret = 32} →
  point:uint8_p{length point = 32} →
  Stack unit
    (requires (λ h → live h mypublic ∧ live h secret ∧ live h point))
    (ensures (λ h0 _h1 → live h1 mypublic ∧ modifies_1 mypublic h0 h1 ∧
     live h0 mypublic ∧ live h0 secret ∧ live h0 point ∧
     h1.[mypublic] == Spec.Curve25519.scalarmult h0.[secret] h0.[point]))
```

Figure 5.2: Low* top-level API for Curve25519

Alice sends ka.P to Bob and Bob sends kb.P to Alice, from which both of them can computation ka.kb.P = kb.ka.P. Knowing neither ka nor kb, an eavesdropping adversary will not be able to guess this value which will be used as a shared secret by Alice and Bob.

The scalarmult specification function implements this scalar multiplication algorithm. It takes as arguments two sequences of 32 bytes (the initial x coordinate of the point and the secret value), and returns a new sequence of 32 bytes, which is the little endian encoding of the x coordinate of the result. For more details about the different components of the F* specification of Curve25519 we refer to the full specification and the corresponding RFC [34, 3].

The low-level Low* equivalent function is a bit more verbose. As presented in Chapter 4, this function enforces the Stack effect, using the HyperStack memory model. Section 5.4 will discuss in more details the different invariants stated in this top-level function. For now, we want to focus on the last line of the post-condition:

```
h1.[mypublic] == Spec.Curve25519.scalarmult h0.[secret] h0.[point]
```

This line gives us the functional equivalence proof between the specification and the concrete code. Note that the excerpt uses the following type and syntax:

- uint8_p stands for *uint8 pointer*, objects of this type are byte arrays;
- the .[] notation designates an *access* to the HyperStack memory, in other terms, h1.[mypublic] is the value of the memory block pointed by mypublic in the memory state h1.

Therefore this part of the post-condition equates the value of the mypublic buffer content in the resulting memory state h1 with the application of the specification function Spec.Curve25519.scalarmult to the buffer contents of secret and point in the initial memory state h0. This equality is the core of the functional correctness of the HACL* library. Consumer of the library do not need to review the complex low-level Low* code, they can rely on a review of the top-level Low* API and the corresponding specification and the functional equivalence proof which will guaranty that for any value the two programs will have the same input/output behavior.

To that intent, the specification modules of HACL* have been put aside from the concrete code modules. They are the "truth" for the HACL* library, which means that whenever we consider a certain cryptographic primitive, the definition of that particular primitive is given by its specification module. This design aims at encouraging external users and contributors to review those specifications to have complete trust into the library, especially since those specifications have been designed specifically to be easy to read and to match the reference literature (for instance the corresponding RFC).

5.3.2 Auditable code

F* is a new programming language and it would be unreasonable to ask new or external users and contributors to be familiar with the corner cases of the language and its syntax. Rather, F* shares a large part of its syntax with functional programming languages from the ML family, such as OCaml or F#. Its functional aspect may seem a bit odd to imperative programming oriented developers, more accustomed to languages such as C or Java, but apprehending the syntax and differences between functional and imperative programming is easy for seasoned developers.

Therefore, assuming some familiarity with the basic concepts of functional programming (such as let-bindings and recursion), we limit the complexity of the HACL* specification by restricting ourselves to only the following constructs:

Pure code Because these F* functions are meant to be used in the pre- and post-condition of the type declaration of the concrete code API, they have to belong to the pure subset of the language. Furthermore, to further assert the correctness of the primitive specifications we want to be able to compile the specification code to OCaml, and run it on test vectors. Since ghost code cannot be extracted and run, HACL* specifications are all written in the pure, non-ghost fragment and thus the associated effect will systematically be `Tot`. As details in Chapter 2 `Tot` implies termination and the absence of side-effects, it is syntactic sugar for `Pure` with degenerated pre- and post-conditions.

Data structures The specifications only rely on a restricted subset of F* data structures, which belongs to those which were described in Chapter 2. Namely, depending on the primitive, the specifications will use:

- mathematical integers
- machine integers
- lists
- sequences
- non-recursive data constructors

In particular, the specification will make use of literals. For clarity sake, F* does not rely on type inference to distinguish between overloaded versions of the same operator on machine integers of different sizes or unbounded integers, nor does it use inference mechanisms to infer the type of integer literals. Rather F* uses a suffix notation to distinguish between the different types: `y` corresponds to 8-bits integers, `s` to 16-bits, `l` to 32-bits and `L` to 64-bits, all of which can be preceded by `u` to specify that the literal is of the corresponding unsigned type. Literals without suffixes are mathematical (unbounded) integers. Additionally F* uses the classical pre-fix notations `0b`, `0o` or `0x` to input respectively binary, octal or hexadecimal values. Hence `0xa`, `0o12uy` or `10l` are all literals for the value `10`, respectively for mathematical integers, unsigned 8-bits integers and signed 32-bits integers. To distinguish between the different operators, F* uses the following convention:

- `^` suffixes machine integer operators while non-prefixed operators are the unbounded integers (the default)
- % suffixes machine integer operators when they have wraparound semantics, while there is not suffix for the operator which prevents overflows (the default). Note that this is only for unsigned integers, signed integers have no wraparound semantics as overflow behaviors are not specified in the C standard for signed integers

For instance, when working with 32-bit unsigned integers:

```
let example a b c d =
  let _ = a + b in (* F* infers a & b as unbounded integers, and '+' as the addition in Z *)
  let _ = c +%^ d in (* F* infers c & d as 32bit unsigned integers and '+' as the wrapping
                        addition *)
  c +^ d (* Fails to typecheck as F* cannot ensure that c + d < 2^32 and the default '+^'
            operator forbids overflows *)
```

Note that infix notations can be locally redefined in F*, using the `let ( + ) = f in ...` to redefine the "plus" operator for instance. Therefore in general F* code those notations are susceptible to vary depending on the developer. HACL* and the rest of the thesis however will stick to the aforementioned notation.

Classically for lists, rather than using the `Cons` constructor, syntactic sugar is provided with the `[x1; ... ; xn]` notation. Sequence literals are entered using either specialized n-ary library functions of the form `create_n` which take n value and returns a sequence of such values, or using a list literal and a coercion function between lists and sequences.

HACL* specifications also make use of sum types to aggregate values together when convenient. For instance:

```
type proj_point = | Proj: x:elem → z:elem → proj_point
```

conveniently defines a point for Curve25519 as a constructor which takes two field elements.

Operations on integers, lists and sequences have been thoroughly presented in Chapter 2. Essentially HACL* specification will use the usual operators on integers (arithmetical, logical and comparison operators); creation, read and update functions on sequences, and lists as literals for large sequences of data. Sum types only use constructors and projectors.

Specification Specific Notations and Definitions To maximize code sharing, avoid repeating some definitions in every specification file and improve readability, shared definitions are factorized into a common module on which all the specification code depends. Notably the specification library contains functions to easily convert bytes into machine words and vice versa, as well as special combinators on sequences which mimic imperative behaviors.

Figure 5.3 shows an excerpt of the Chacha20 cipher specification. This piece of code does not present any complexity except for some syntactic specificities and library functions. Chacha20 is a symmetric stream cipher which relies on an internal state and a shuffling function to produce a stream of blocks of 64 bytes of random-looking data. This state is represented as a sequence of 16 32-bit integers, and the shuffling function is divided into different rounds: it alternatively shuffles the state by columns and by diagonal lines. The `state`

```
type state = m:seq UInt32.t {length m = 16}
type idx = n:nat{n < 16}
type shuffle = state → Tot state

let line (a:idx) (b:idx) (d:idx) (s:t{v s < 32}) (m:state) : Tot state =
  let m = m.[a] ← (m.[a] +%^ m.[b]) in
  let m = m.[d] ← ((m.[d] ^^ m.[a]) <<< s) in m

let quarter_round a b c d : shuffle =
  line a b d 16ul @
  line c d b 12ul @
  line a b d 8ul @
  line c d b 7ul

let column_round : shuffle =
  quarter_round 0 4 8 12 @
  quarter_round 1 5 9 13 @
  quarter_round 2 6 10 14 @
  quarter_round 3 7 11 15

let diagonal_round : shuffle =
  quarter_round 0 5 10 15 @
  quarter_round 1 6 11 12 @
  quarter_round 2 7 8 13 @
  quarter_round 3 4 9 14

let double_round: shuffle =
    column_round @ diagonal_round (* 2 rounds *)

let rounds : shuffle =
    Spec.Loops.iter 10 double_round (* 20 rounds *)

let chacha20_core (s:state) : Tot state =
    let s' = rounds s in
    Spec.Loops.seq_map2 (λ x y →x +%^ y) s' s
```

Figure 5.3: Example of Chacha20 code

type is the type of this internal state, while the shuffle function type designates a function mapping two states.

The building block of the shuffling algorithm is the line function. It takes 3 different indices from the state as well as an extra parameter for the depth of the *rotation* operation, and it changes two values of the state (out of 16) based on those parameters.

```
let line (a:idx) (b:idx) (d:idx) (s:t{v s < 32}) (m:state) : Tot state =
```

```
let m = m.[a] ←(m.[a] +%^ m.[b]) in
let m = m.[d] ←((m.[d] ^^ m.[a]) <<< s) in m
```

The m.[i] notation is syntactic sugar for the read access to sequences at index i (Seq.index m i) and let m' = m.[i] ←v in e' replaces the less readable let m' = Seq.upd m i v in. In an imperative language like C, the body of the line function would have been written

```
#define ROTL(x,s) (((x) << (s)) | ((x) >> (32−(s))))
m[a] = m[a] + m[b];
m[d] = ROTL(m[d] ^ m[a],s)
```

<<< is an infix operator for the *rotate* function (the ROTL macro in the C excerpt), which shift the bits of a 32-bits integer by s to the left and reinserts the trimmed bits to the right. Its semantics is defined in the specification library module of HACL*.

The @ operator is a functional substitute for a semicolon. It feeds the result of the application of the function on the left hand side to the function on the right hand side, its concrete definition being let (@) f g = λx →g (f x).

Eventually, lists and sequences are equipped with special combinators which can conveniently be used to specify imperative features such as loops. The excerpt shows two such examples: iter and seq_map2 from the Spec.Loops library module. The library module contains a third combinator, seq_map. iter consists in repeatedly applying a function f to a sequence of data s f (f (f ... (f s))) n times. Intuitively is corresponds to a loop of the form

```
for (int i = 0; i < n; i++) f(s);
```

The seq_map and seq_map2 functions instead iterate on each element of one (seq_map) or two (seq_map2) sequences of data in the spirit of

```
for (int i = 0; i < length(s); i++) f(s[i]);
```

5.3.3 RFC-based code

HACL* specification modules are self-contained programs: they provide a top-level API similar to those of the exposed low-level code. Since programming styles are heavily dependent on the programmer, we implemented those modules so that they would stay as close as possible to the corresponding reference specification in the literature. The cryptographic primitives of HACL* are all standardized either in the NIST standards, or in RFCs. Those precisely describe the internals of the primitives and often guide the developer through

the implementation process, dividing the primitive into functional blocks. In such cases, HACL* reuses the same names for clarity.

The goal is that a reviewer familiar with the reference textual specification will easily recognize the different implementation parts in HACL* specification code and thus easily assert its correctness. Figure 3 from Appendix B displays the full HACL* specification for Curve25519. The reference textual specification for this elliptic curve is the RFC 7748.

RFC 7748 uses a mix of Python code examples and text descriptions to fully specify the X25519 and X448 primitives. Asserting the equivalence between the RFC and the HACL* specification requires careful reviewing of the Python and F* code. The analogy between functions such as decodeScalar is straightforward. The differences stem from the imperative style of the Python code versus the recursive style of the F* code. The decodeLittleEndian RFC function is implemented in a library file of HACL* as a general purpose function called little_endian, which turns a sequence of bytes into its little endian integer value, and the converse is little_bytes, which takes an integer value and the number of bytes on which to encode it, and returns the corresponding sequence of bytes. The F* functions big_endian and big_bytes serve the exact same purpose but in the big endian setting.

Although the style of the two Montgomery ladder algorithms is different, it is straightforward to see that the behaviors are equivalent. The F* specification only aims at expressing the functional correctness of a primitive, and not to enforce any kind of side-channel protection, the constant-time implementation of the cswap function is unnecessary here. It is however crucial that the F* code takes into account the textual remarks from the RFC which do not necessarily reflect into the Python example code. For instance, the RFC 7748 indicates that:

> The u-coordinates are elements of the underlying field GF(2^255 - 19) [...] and are encoded as an array of bytes, u, in little-endian order such that u[0] + 256*u[1] + 256^2*u[2] + ... + 256^(n-1)*u[n-1] is congruent to the value modulo p and u[n-1] is minimal. When receiving such an array, implementations of X25519 [...] MUST mask the most significant bit in the final byte. This is done to preserve compatibility with point formats that reserve the sign bit for use in other protocols and to increase resistance to implementation fingerprinting.

It is therefore essential that two input values for Curve25519 which only differ by their most significant bit (the 256-th bit of a 32-byte array) are processed identically by a correct implementation. HACL* specification enforces this criterion at:

```
let decodePoint (u:serialized_point) =
```

```
  (little_endian u % pow2 255) % prime
```

Indeed, the little endian interpretation of `u` is taken modulo 2^{255} which obviously ignores the top-most bit. Similarly the RFC states that

> Implementations MUST accept non-canonical values and process them as if they had been reduced modulo the field prime. The non-canonical values are `2^255 - 19` through `2^255 - 1` for `X25519[...]`.

This again appears in the F* definition of `decodePoint` which does not reject non-canonical values but simply treats them as if they were canonical (with the reduction modulo the prime $2^{255} - 19$).

```
let decodePoint (u:serialized_point) =
  (little_endian u % pow2 255) % prime
```

5.3.4 Executable Specifications

Any F* program can be extracted and compiled to OCaml code, and this feature is used to further ascertain the correctness of the HACL* specifications. The literature always provides test vectors alongside with the description of the primitives, and we test that the pure F* specification program of the each of the primitives we implement in the library passes the RFC test vectors.

The pure subset of F* has many inefficiencies which would be problematic for real-life code. In this particular setting however, it proves to be very convenient. The compiled code runs in OCaml, a safe language with automatic memory management. It relies on existing libraries of the language (for instance the `Zarith` OCaml bignum library for F* mathematical integers) which makes running vanilla F* examples extremely easy. The whole implementation and compilation process of new specifications in F* is very straightforward, and because all the implementation details such as side-channel protection or low-level handling of memory of completely ignored in this setting, the full specification of a cryptographic primitive such as Curve25519 is less than 80 lines of code (see figure 3).

5.4 Low-level Low* code

The low-level Low* code is really the core of the HACL* library. It is at the crossroad between the high-level specifications which are necessary to ensure functional correctness but, although runnable, cannot be used for production software, and the resulting C code which, although it implicitly carries all the properties proven from the Low* code, does not support a type and proof system which allows the programmer to make changes to it safely.

Chapter 4 presented the memory model and compilation scheme from the Low* subset of F* to safe C code. This section demonstrates the usefulness of this setting in the context of performance critical, security sensitive cryptographic code.

5.4.1 Efficient Symmetric Cryptography

HACL* supports hash functions from the SHA2 family as well as stream ciphers from the Salsa family. Neither these algorithms are complex, but they are often performance critical: hash functions on large files can take time and symmetric ciphers are the bottleneck of any encryption system. The library provides Low* implementations of these algorithms which compile to efficient C code, competitive with the state of the art C code in existing popular cryptographic libraries.

The main difference between specifications and concrete code for such algorithms is the persistence of the state. Symmetric cryptographic algorithms typically work with blocks of data and follow this pattern:

- The *initialization*, which sets up an initial internal state from the key and the nonce for instance;
- The *update*, which shuffles the internal state to new pseudo-random values and optionally process a block of input data;
- The *finalization*, which cleans up and returns the result

The description of the algorithms typically collapses the *initialization* and *update* steps into one, repeated for each block. While this is simpler and clearer for the actual algorithm description, it is often not the most efficient to implement actual code.

The example of the Chacha20 stream cipher The Chacha20 stream cipher algorithm is an illustrative example of the above. It produces a block of 64-bytes of pseudo random data using the following procedure:

1. An initial state of 16 32-bit integers is created from the Chacha20 secret key (32-bytes), a nonce (12-bytes) and a 32-bit counter in the IETF version of the algorithm;
2. A copy is made of the initial state, which is shuffled internally using 20 iterations of a round column and diagonal shuffling routine;
3. The result of this shuffling procedure is summed with the initial state;
4. The resulting array of 16 32-bit integers is serialized into a stream cipher block of 64-bytes of data;

```
let chacha20_block (k:key) (n:nonce) (c:counter): Tot block =
    let st = setup k n c in
    let st' = chacha20_core st in
    uint32s_to_le 16 st'
```

Figure 5.4: F* Specification of the Chacha20 routine for each block

5. The counter is incremented and the algorithm resumes at the first step

Figure 5.4 shows the chacha20_block function from HACL* specifications, with the initial setup of the state from a key, a nonce and a counter, then the chacha20_core function which shuffles the state and sums it with its initial value, and finally the serialization function which returns 64-bytes of a stream.

The algorithm is very simple and relies on a systematic internal state setup. However two consecutive states (for two adjacent blocks of the stream cipher to produce) initially only differ by the value of the counter, which is stored in the internal state array at index 12. Therefore a trivial algorithmic optimization consists in keeping a copy of the originally setup state throughout the computation and only update its index value at each new block, rather than systematically making copies of the key and the nonce.

Nonetheless we need to prove the equivalence between the naive specification implementation and the persistent state optimized Low* implementation.

Figure 5.5 shows the methodology we use to implement this in HACL*. Because the state is persistent, the key and the nonce which were used to set it up are not passed in arguments to the Low* chacha20_block function like they were in the Low* specifications. In order to be able to easily reason about the original key and nonce that were used, we carry them as *ghost* parameters inside a log argument. This log, which carries sequences of bytes for proof purposes but not to be extracted in actual code, is annotated as an erased type in F*. The erased type is defined in the F* standard library Ghost module as an abstract wrapper around a type α. While the hide function which turns a value of type α into an erased α value is a total function (val hide : #a:Type u#a → a → Tot (erased a)), the only way to access the underlying data is via the val reveal : #a:Type u#a → erased a → GTot a function. Being annotated GTot, the F* verification system enforces that the returned value is never used in concrete code. In other terms, an erased type value such as the log in chacha20_block can be passed around in concrete functions for proof purposes, it will later on be erased by the compiler and will have no impact on the resulting code. This is precisely what we want here.

The invariant predicate binds the key and the nonce from the log to the actual state, it is maintained throughout the function. Indeed it holds for the state st when the function is called, so that the returned block of stream can be proven equal to a block of the chacha20_block function from the specification applied

```
type log_t_ = | MkLog: k:Spec.key → n:Spec.nonce → log_t_
type log_t = Ghost.erased log_t_

let invariant (log:log_t) (h:mem) (st:state) : GTot Type0 =
  live h st ∧ (let log = Ghost.reveal log in let s = as_seq h st in
    match log with | MkLog key nonce → reveal_h32s s ==
                                Spec.setup key nonce (H32.v (Seq.index s 12)))

val chacha20_block:
  log:log_t →
  stream_block:uint8_p{length stream_block = 64} →
  st:state{disjoint st stream_block} →
  ctr:UInt32.t →
  Stack log_t
    (requires (λ h → live h stream_block ∧ invariant log h st))
    (ensures (λ h0 updated_log h1 → live h1 stream_block ∧ invariant log h0 st
      ∧ invariant updated_log h1 st ∧ modifies_2 stream_block st h0 h1
      ∧ (let block = reveal_sbytes (as_seq h1 stream_block) in
          match Ghost.reveal log, Ghost.reveal updated_log with
          | MkLog k n, MkLog k' n' →
              block == chacha20_block k n (U32.v ctr) ∧ k == k' ∧ n == n')))
```

Figure 5.5: Low* declaration of the chacha20_block function

```
(* Field types and parameters *)
let prime = pow2 130 - 5
type elem = e:int{e ≥ 0 ∧ e < prime}
let fadd (e1:elem) (e2:elem) = (e1 + e2) % prime
let fmul (e1:elem) (e2:elem) = (e1 * e2) % prime
let zero : elem = 0
let one : elem = 1
let ( +@ ) = fadd // Infix operator definition
let ( *@ ) = fmul // Infix operator definition
```

Figure 5.6: F* specification of the prime field for Poly1305

to the given key and nonce, and the code also ensures that it holds when the function returns. The latter is not completely straightforward as the state has been updated to use the counter ctr of the current block. As previously mentioned however, this update only affects a single cell of the state array and nonce of those which depend on the key and nonce.

5.4.2 Efficient Asymmetric Cryptography

Asymmetric cryptographic algorithms commonly rely on prime-field arithmetic, that is, addition and multiplication modulo a prime p in $\mathbb{Z}_p$. In HACL*,

the Poly1305, Curve25519, and Ed25519 algorithms all compute on various prime fields. The mathematical specification for these field operations is very simple; Figure 5.6 depicts the F* spec for the Poly1305 field.

For security, the primes used by cryptographic algorithms need to be quite large, which means that elements of the field cannot be represented by machine integers, and instead need to be encoded as bignums, that is, arrays of integers. Consequently, bignum arithmetic becomes a performance bottleneck for these algorithms. Furthermore, well known bignum implementation tricks that work well for numerical computations are not really suitable for cryptographic code since they may leak secrets. For example, when multiplying two bignums, a generic bignum library may shortcut the computation and return zero if one of the arguments is zero. In a crypto algorithm, however, the time taken by such optimizations may leak the value of a key. Implementing an efficient and secure generic modulus function is particularly hard. Consequently, cryptographic implementations are often faced with a trade-off between efficient field arithmetic and side-channel resistance.

5.4.3 Efficient Bignum Libraries for Poly1305, Curve25519, and Ed25519

For algorithms like RSA that use large and unpredictable primes, implementations often choose to forego side-channel resistance. However, for modern fixed-prime primitives like Poly1305 and Curve25519, it is possible to choose the shape of the prime carefully so that field arithmetic can be both efficient and side-channel resistant. For instance, given a fixed Mersenne prime of the form $2^n - 1$, the modulo operation is easy to implement: all the bits beyond n-th bit can be repeatedly lopped off and added to the low n bits, until the result is an n bit value. Computing the modulo for the Poly1305 prime $2^{130} - 5$ or Curve25519 $2^{255} - 19$ in constant time is similar.

Once a suitable prime is picked, the main implementation choice is whether to represent the field elements as *packed* bignums, where each array element (called a *limb*) is completely filled, or to use an *unpacked* representation, where the limbs are only partially filled. For example, in the Poly1305 field, elements are 130-bit values and can be stored in 3 64-bit integers. The little-endian packed layout of these elements would be $64bits|64bits|2bits$, whereas a more evenly distributed unpacked layout is $44bits\ |44bits\ |42bits$. The main advantage of the unpacked layout is that when performing several additions in a sequence, we can delay the carry propagation, since the limbs will not overflow. In the packed representation, we must propagate carries after each addition. Optimizing carry propagation by making it conditional on overflow would not be safe, since it would expose a timing side-channel. Indeed, most efficient 64-bit implementations of Poly1305 and Curve25519 use unpacked representations; Poly1305 uses the 44-44-42 layout on 64-bit platforms and 5 26-bit limbs

on 32-bit platforms; Curve25519 and Ed25519 use 5 limbs of 51-bits each or 10 limbs of 25.5 bits each.

In summary, efficient implementations of Poly1305, Curve25519, and Ed25519 use prime-specific computations and different unpacked bignum representations for different platforms. Consequently, each of their implementations contains its own bignum library which must be independently verified. In particular, previous proofs of bignum arithmetic in Poly1305 [52] and Curve25519 [57] are implementation-specific and cannot be reused for other platforms or other implementations. In contrast, Zinzindohoue et al. [129] develop a generic verified bignum library in OCaml that can be used in multiple cryptographic algorithms. The cost of this genericity is significantly reduced performance. In the rest of this section, we present a novel approach that allows us to share verified bignum code across primitives and platforms, at no cost to performance.

5.4.4 Verifying a Generic Bignum Library

In HACL*, we uniformly adopt unpacked representations for our bignums. We define an evaluation function eval that maps a bignum to the mathematical integer it represents. This function is parametric over the base of the unpacked layout: for example, our Poly1305 elements are in base 2^{44}, which means that a bignum b represents the integer $eval(b) = b[0] + 2^{44} * b[1] + 2^{88} * b[2]$.

We observe that, except for modulo, all the bignum operations needed by our primitives are independent of the prime. Furthermore, generic bignum operations, such as addition, do not themselves depend on the specific unpacked representation; they only rely on having enough remaining space so that limbs do not overflow. Using these observations, we implement and verify a generic bignum library that includes modular addition, subtraction, multiplication, and inverse, and whose proofs do not depend on the prime or the unpacked representation. Each generic operation is parametric over the number of limbs in the bignum and requires as a pre-condition that each limb has enough spare room to avoid overflow. To satisfy these preconditions in a cryptographic primitive like Poly1305, the implementation must carefully interleave carry propagation steps and modular reduction with generic operations.

The only part of the bignum library that depends on the prime is the modular reduction, and this must be implemented and verified anew for each new prime. All other functions in the bignum library are written and verified just once. When compiling the code to C, the prime-specific code and the representation constants (e.g. the number of limbs, the evaluation base etc.) are inlined into the generic bignum code, yielding an automatically specialized bignum library in C for each primitive. As a result, our generated field arithmetic code is as efficient as the custom bignum libraries for each primitive. Hence, we are able to find a balance between generic code for verification and specialized code for efficiency. We are able to reuse more than half of the field arithmetic code between Poly1305, Curve25519, and Ed25519. We could

share even more of the code if we specialized our bignum library for pseudo-Mersenne primes. For primes which shapes do not enable optimized modulo computations, we also implement and verify a generic modulo function based on Barrett reduction, which we use in the Ed25519 signature algorithm.

When programming with unpacked bignums, carry propagation and modular reduction are the most expensive operations. Consequently, this style encourages programmers to find clever ways of delaying these expensive operations until they become necessary. Some implementations break long carry chains into shorter sequences that can be executed in parallel and then merged. These low-level optimizations are error-prone and require careful analysis. In particular, carry propagation bugs are the leading functional correctness flaws in OpenSSL crypto, with two recent bugs in Poly1305 [27, 51], and two others in Montgomery multiplication (CVE-2017-3732, CVE-2016-7055). A carry propagation bug was also found in TweetNaCl [38].

Our Curve25519 implementation is closely inspired by Adam Langley's `donna_c64` 64-bit implementation, which is widely used and considered the state-of-the-art C implementation. In 2014, Langley reported a bug in this implementation [1]: the implementation incorrectly skipped a necessary modular reduction step. In response, Langley explored the use of formal methods to prove the absence of such bugs, but gave up after failing to prove even modular addition using existing tools. This paper presents the first complete proof of a C implementation of Curve25519, including all its field arithmetic. In particular, our proofs guarantee the absence of carry propagation bugs in Poly1305, Curve25519, and Ed25519.

A surprising benefit of formal verification is that it sometimes identifies potential optimizations. When verifying Curve25519, we observed that `donna_c64` was too conservative in certain cases. Each multiplication and squaring operation had an unnecessary extra carry step, which over the whole Curve25519 scalar multiplication totaled to about 3400 extra cycles on 64-bit Intel processors. We removed these redundant carries in our code and proved that it was still correct. Consequently, the Curve25519 C code generated from HACL* is slightly (about 2.2%) faster than `donna_c64` making it the fastest C implementation that we know of.

5.4.5 Verifying elliptic curve operations

Curve25519

Curve25519 [34, 3] a Montgomery elliptic curve designed for use in a Diffie-Hellman (ECDH) key exchange. The key operation over this curve is the multiplication nP of a public curve point P by a secret scalar n. A distinctive property of this family of curves is only the x-coordinate of P is needed to

[1] https://www.imperialviolet.org/2014/09/07/provers.html

```
let prime = pow2 255 - 19
type elem = e:int{0 ≤ e ∧ e < prime}
type serialized_point = b:bytes{length b = 32}
type proj_point = | Proj: x:elem → z:elem → proj_point

let decodePoint (u:serialized_point) =
  (little_endian u % pow2 255) % prime

let encodePoint (p:proj_point) =
  let x = p.x *@ (p.z ** (prime - 2)) in
  little_bytes 32ul x
```

Figure 5.7: F* specification of Curve25519 point format

compute the x-coordinate of nP. This leads to both efficient computations and small keys.

The simplicity of the algorithm and its adoption in protocols like TLS and Signal have made it a popular candidate for formal verification. Several other works have been tackling Curve25519. However, our implementation is, to the best of our knowledge, the first implementation to verify the full byte-level scalar multiplication operation. Chen et al. [57] verified one step of the Montgomery ladder for a qhasm implementation, but did not verify the ladder algorithm or point encodings; Zinzindohoue et al. [129] implemented and verified the Montgomery ladder for Curve25519 and two other curves, but they did not verify the point encodings. Our Curve25519 implementation is verified to be fully RFC-compliant.

Figure 5.7 shows the F* specification for the point encoding and decoding functions that translate between curve points and byte arrays. Implementing and verifying these functions is not just a proof detail. Compliance with encodePoint avoids the missing reduction bug that Adam Langley described in `donna_c64`. The first line of encodePoint computes x as a result of the modular multiplication operation *@ (see Figure 5.6). Hence, the result of encodePoint is a little-endian encoding of a number strictly less than $2^{255} - 19$. Consequently, a Low* implementation of Curve25519 that forgets to perform a modular reduction before the little-endian encoding does not meet this specification and so will fail F* verification.

Ed25519 The Ed25519 signature scheme [37, 4] is an EdDSA algorithm based on the twisted Edwards curve birationally equivalent to Curve25519. Despite their close relation, the implementation of Ed25519 involves many more components than Curve25519. It uses a different coordinate system and different point addition and doubling formulas. The signature input is first hashed using the SHA-512 hash function, which we verify separately. The signature operation itself involves prime-field arithmetic over two primes:

the Curve25519 prime $2^{255} - 19$ and a second non-Mersenne prime $2^{252} + 27742317777372353535851937790883648493$. This second prime does not enjoy an efficient modulo operation, so we implement and verify a slower but generic modulo function using the Barrett reduction. We thus obtain the first verified implementation of Ed25519 in any language. In terms of size and proof complexity, Ed25519 was the most challenging primitive in HACL*; implementing and verifying the full construct took about 3 person-weeks, despite our reuse of the Curve25519 and SHA-512 proofs.

Our implementation is conservative and closely follows the RFC specification. It is faster than the naive Ed25519 reference implementation (`ref`) in TweetNaCl, but about 2.5x slower than the optimized `ref10` implementation, which relies on a precomputed table containing multiples of the curve base point. Our code does not currently use precomputation. Using precomputed tables in a provably side-channel resistant way is non-trivial; for example, [101] demonstrate side-channel attacks on Ed25519 precomputations on certain platforms. We leave the implementation and verification of secure precomputation for Ed25519 as future work.

5.5 Vectorization

In the previous section, we saw how we can implement cryptographic primitives in Low* by closely following their high-level F* specification. By including a few straight-forward optimizations, we can already generate C code that is as fast as hand-written C reference implementations for these primitives. However, the record-breaking state-of-the-art assembly implementations for these primitives can be several times faster than such naive C implementations, primarily because they rely on modern hardware features that are not available on all platforms and are hence not part of standard portable C. In particular, the fastest implementations of all the primitives considered in this paper make use of vector instructions that are available on modern Intel and ARM platforms.

Intel architectures have supported 128-bit registers since 1999, and, through a series of instruction sets (SSE, SSE2, SSSE3, AVX, AVX2, AVX512), have provided more and more sophisticated instructions to perform on 128, 256, and now 512-bit registers, treated as vectors of 8, 16, 32, or 64-bit integers. ARM recently introduced the NEON instruction set in 2009 that provides 128-bit vector operations. So, on platforms that support 128-bit vectors, a single vector instruction can add 4 32-bit integers using a special vector processing unit. This does not strictly translate to a 4x speedup, since vector units have their own overheads, but can significantly boost the speed of programs that exhibit single-instruction multiple-data (SIMD) parallelism.

Many modern cryptographic primitives are specifically designed to take advantage of vectorization. However, making good use of vector instructions

```
val uint32x4: Type0
val v: uint32x4 → GTot (s:seq UInt32.t){length s = 4}
val load32x4: x0:UInt32.t → x1:UInt32.t → x2:UInt32.t → x3:UInt32.t →
            Tot (r:uint32x4{v r = createL [x0;x1;x2;x3]})
val (+%^): x:uint32x4 → y:uint32x4 →
            Tot (r:uint32x4{v r = map2 UInt32.((+%^)) (v x) (v y)}
let (^^) : x:uint32x4 → y:uint32x4 →
            Tot (r:uint32x4{v r = map2 UInt32.((^^)) (v x) (v y)}
let (<<<): s:uint32x4 → n:UInt32.t{UInt32.v n < 32} →
            Tot (r:uint32x4{v r = map (λ x → x UInt32.((<<<)) n) (v s)})
val shuffle_right: s:uint32x4 → n:UInt32.{v r < 4} →
                    Tot (r:uint32x4{if v n == 1 then createL [s.[3];s.[0];s.[1];s.[2]]
                                    else if v n == 2 then ...})
```

Figure 5.8: (Partial) F* Interface for 128-bit vectors interpreted as 4 32-bit unsigned integers.

```
typedef unsigned int uint32x4 __attribute__ ((vector_size (16)));
uint32x4 load32x4(uint32_t x1, uint32_t x2, uint32_t x3, uint32_t x4){
  return ((uint32x4) _mm_set_epi32(x4,x3,x2,x1));
}
uint32x4 shuffle_right(uint32x4 x, unsigned int n) {
  return ((uint32x4) _mm_shuffle_epi32((__m128i)x,
                          _MM_SHUFFLE((3+n)%4,(2+n)%4,(1+n)%4,n%4)));
}
uint32x4 uint32x4_addmod(uint32x4 x, uint32x4 y) {
  return ((uint32x4) _mm_add_epi32((__m128i)x,(__m128i)y);
}
```

Figure 5.9: (Partial) GCC library for 128-bit vectors using Intel SSE3 intrinsics: (`https://software.intel.com/sites/landingpage/IntrinsicsGuide/`)

often requires restructuring the sequential implementation to expose the inherent parallelism and to avoid operations that are unavailable or expensive on specific vector architectures. Consequently, the vectorized code is no longer a straightforward adaptation of the high-level specification and needs new verification. In this section, we develop a verified vectorized implementation of ChaCha20 in Low*. Notably, we show how to verify vectorized C code by relying on vector libraries provided as compiler builtins and intrinsics. We do not need to rely on or verify assembly code. We believe this is the first verified vectorized code for any cryptographic primitive and shows the way forward for verifying other record-breaking cryptographic implementations.

5.5.1 Modeling Vectors in F*

In F*, the underlying machine model is represented by a set of trusted library interfaces that are given precise specifications, but which are implemented at runtime by hardware or system libraries. For example, machine integers are

represented by a standard library interface that formally interprets integer types like UInt32.t and primitive operations on them to the corresponding operations on mathematical integers int. When compiling to C, KreMLin translates these operations to native integer operations in C. However, F* programmers are free to add new libraries or modify existing libraries to better reflect their assumptions on the underlying hardware. For C compilation to succeed, they must then provide a Low* or C implementation that meets this interface.

We follow the same approach to model vectors in HACL* as a new kind of machine integer interface. Like integers, vectors are pure values. Their natural representation is a sequence of integers. For example, Figure 5.8 shows a fragment of our F* interface for 128-bit vectors, represented as an abstract type uint32x4. Each vector can be interpreted, via the v function, as a sequence of four 32-bit unsigned integers. (More generally, such vectors can be also interpreted as eight 16-bit or sixteen 8-bit integers, and we can make these representations interconvertible.) Many classic integer operations $(+, -, *, \&, \text{'} <<, >>)$ are lifted to uint32x4, and interpreted as the corresponding point-wise operations over sequences of integers. In addition, the interface declares vector-specific operations like load32x4 to load vectors, and shuffle_right, which allows the order of integers in a vector to be switched.

We provide C implementations of this interface for Intel SSE3 and ARM NEON platforms. Figure 5.9 shows a fragment of the Intel library relying on GCC compiler intrinsics. This C code is not verified, it is trusted. Hence, it is important to minimize the code in such libraries, and to carefully review them to make sure that their implementation matches their assumed specification in F*. However, once we have this F* interface and its C implementation for some platform, we can build and verify vectorized cryptographic implementations in Low*.

5.5.2 Verified Vectorized ChaCha20

The ChaCha20 stream cipher was designed by D. Bernstein [35] and standardized as an IETF RFC [2]. It is widely recommended as an alternative to AES in Internet protocols. For example, ChaCha20 is one of the two encryption algorithms (other than AES) included in TLS 1.3 [5]. The NaCl API includes Salsa20, which differs a little from ChaCha20 [35] but for the purposes of verification, these differences are irrelevant; we implemented both in HACL*.

Figure 5.10 depicts a fragment of our RFC-based F* specification of ChaCha20. ChaCha20 maintains an internal state that consists of 16 32-bit integers interpreted as a 4x4 matrix. This state is initialized using the encryption key, nonce, and the initial counter (typically 0). Starting from this initial state, ChaCha20 generates a sequence of states, one for each counter value. Each state is serialized as a key block and XORed with the corresponding plaintext (or ciphertext) block to obtain the ciphertext (or plaintext). To generate a key block, ChaCha20 shuffles the input state 20 times, with 10 column rounds

```
type state = m:seq UInt32.t{length m = 16}
type idx = n:nat{n < 16}

let line (a:idx) (b:idx) (d:idx) (s:t{v s < 32}) (m:state) =
  let m = m.[a] ← (m.[a] +%^ m.[b]) in
  let m = m.[d] ← ((m.[d] ^^ m.[a]) <<< s) in m

let quarter_round a b c d =
  line a b d 16ul @
  line c d b 12ul @
  line a b d 8ul @
  line c d b 7ul

let column_round =
  quarter_round 0 4 8 12 @
  quarter_round 1 5 9 13 @
  quarter_round 2 6 10 14 @
  quarter_round 3 7 11 15
```

Figure 5.10: RFC-based ChaCha20 specification in F*. The @ operator is serial function composition: (f @ g)(x) = g(f(x))

```
type state = m:seq uint32x4 {length m = 4}
type idx = n:nat{n < 4}

let line (a:idx) (b:idx) (d:idx) (s:UInt32.t{v s < 32}) (m:state) =
  let ma = m.[a] in let mb = m.[b] in let md = m.[d] in
  let ma = ma +%^ mb in
  let md = (md ^^ ma) <<< s in
  let m = m.[a] ← ma in
  let m = m.[d] ← md in m

let column_round =
  line 0 1 3 16ul @
  line 2 3 1 12ul @
  line 0 1 3 8ul @
  line 2 3 1 7ul
```

Figure 5.11: F* specification for 128-bit vectorized ChaCha20

and 10 diagonal rounds. Figure 5.10 shows the computation for each column round.

As we did for SHA-256, we wrote a reference stateful implementation for ChaCha20 and proved that it conforms to the RFC-based specification. The generated code takes 6.26 cycles/byte to encrypt data on 64-bit Intel platforms; this is as fast as the C implementations in popular libraries like OpenSSL and libsodium, but is far slower than vectorized implementations. Indeed, previous work (see [36, 72]) has identified two inherent forms of parallelism in ChaCha20 that lend themselves to efficient vector implementations:

Line-level Parallelism: The computations in each column and diagonal round can be reorganized to perform 4 line shufflings in parallel.

Block-level Parallelism: Since each block is independent, multiple blocks can be computed in parallel.

We are inspired by a 128-bit vector implementation in SUPERCOP due to Ted Krovetz, which is written in C using compiler intrinsics for ARM and Intel platforms, and reimplement it in HACL*. Krovetz exploits line-level parallelism by storing the state in 4 vectors, resulting in 4 vector operations per column-round, compared to 16 integer operations in unvectorized code. Diagonal rounds are a little more expensive (9 vector operations), since the state vectors have to be reorganized before and after the 3 line operations. Next, Krovetz exploits block-level parallelism and the fact that modern processors have multiple vector units (typically 3 on Intel platforms and 2 on ARM) to process multiple interleaving block computations at the same time. Finally, Krovetz vectorizes the XOR step for encryption/decryption by loading and processing 128 bits of plaintext/ciphertext at once. All these strategies requires significant refactoring of the source code, so it becomes important to verify that the code is still correct with respect to the ChaCha20 RFC.

We write a second F* specification for vectorized ChaCha20 that incorporates these changes to the core algorithm. The portion of this spec up to the column round is shown in Figure 5.11. We modify the state to store four vectors, and rearrange the line and column_round using vector operations. We then prove that the new column_round function has the same functional behavior as the RFC-based column_round function from Figure 5.10. Building up from this proof, we show that the vectorized specification for full ChaCha20 computes the same function as the original spec.

Finally, we implement a stateful implementation of vectorized ChaCha20 in Low* and prove that it conforms to our vectorized specification. (As usual, we also prove that our code is memory safe and side-channel resistant.) This completes the proof for our vectorized ChaCha20, which we believe is the first verified vectorized implementation for any cryptographic primitive.

When compiled to C and linked with our C library for uint32x4, our vectorized ChaCha20 implementation has the same performance as Krovetz's implementation on both Intel and ARM platforms. This makes our implementation the 8th fastest in the SuperCop benchmark on Intel processors, and the 2nd fastest on ARM. As we did with Krovetz, we believe we can adapt and verify the implementation techniques of faster C implementations and match their performance.

Algorithm	Spec (F* loc)	Code+Proofs (Low* loc)	C Code (C loc)	Verification (s)
Salsa20	70	651	372	280
Chacha20	70	691	243	336
Chacha20-Vec	100	1656	355	614
SHA-256	96	622	313	798
SHA-512	120	737	357	1565
HMAC	38	215	28	512
Bignum-lib	-	1508	-	264
Poly1305	45	3208	451	915
X25519-lib	-	3849	-	768
Curve25519	73	1901	798	246
Ed25519	148	7219	2479	2118
AEAD	41	309	100	606
SecretBox	-	171	132	62
Box	-	188	270	43
Total	801	22,926	7,225	9127

Table 5.1: HACL* code size and verification times

5.6 Evaluation (LoC, verification time, benchmarks etc.)

In this section, we assess the coding and verification effort that went into the HACL* library, and evaluate its performance relative to state-of-the-art cryptographic libraries.

Coding and Verification Effort Taking an RFC and writing a specification for it in F* is straightforward; similarly, taking inspiration from existing C algorithms and injecting them into the Low* subset is a mundane task. Proving that the Low* code is memory safe, secret independent, and that it implements the RFC specification is the bulk of the work. Table 5.1 lists, for each algorithm, the size of the RFC-like specification and the size of the Low* implementation, in lines of code. Specifications are intended to be read by experts and are the source of "truth" for our library: the smaller, the better. The size of the Low* implementation captures both the cost of going into a low-level subset (meaning code is more imperative and verbose) and the cost of verification (these include lines of proof). We also list the size of the resulting C program, in lines of code. Since the (erased) Low* code and the C code are in close correspondence, the ratio of C code to Low* code provides a good estimate of code-to-proof ratio.

One should note that a large chunk of the bignum verified code is shared across Poly1305, Curve25519 and Ed25519, meaning that this code is verified once but used in three different ways. The sharing has no impact on the quality of the generated code, as we rely on KreMLin to inline the generic code and

Algorithm	HACL*	OpenSSL	libsodium	TweetNaCl	OpenSSL (asm)
SHA-256	13.43	16.11	12.00	-	7.77
SHA-512	8.09	10.34	8.06	12.46	5.28
Salsa20	6.26	-	8.41	15.28	-
ChaCha20	6.37 (ref) 2.87 (vec)	7.84	6.96	-	1.24
Poly1305	2.19	2.16	2.48	32.65	0.67
Curve25519	154,580	358,764	162,184	2,108,716	-
Ed25519 sign	63.80	-	24.88	286.25	-
Ed25519 verify	57.42	-	32.27	536.27	-
AEAD	8.56 (ref) 5.05 (vec)	8.55	9.60	-	2.00
SecretBox	8.23	-	11.03	47.75	-
Box	21.24	-	21.04	148.79	-

Table 5.2: Intel64-GCC: Performance Comparison in cycles/byte on an Intel(R) Xeon(R) CPU E5-1630 v4 @ 3.70GHz running 64-bit Debian Linux 4.8.15. All measurements (except Curve25519) are for processing a 16KB message; for Curve25519 we report the number of cycles for a single ECDH shared-secret computation. All code was compiled with GCC 6.3. OpenSSL version is 1.1.1-dev (compiled with no-asm); Libsodium version is 1.0.12-stable (compiled with –disable-asm), and TweetNaCl version is 20140427.

specialize it for one particular set of bignum parameters. The net result is that Poly1305 and Curve25519 contain separate, specialized versions of the original Low* bignum library. Chacha20 and Salsa20, just like SHA-256 and SHA-512, are very similar to each other, but the common code has not yet been factored out. We intend to leverage recent improvements in F* to implement more aggressive code sharing, allowing us to write, say, a generic SHA-2 algorithm that can be specialized and compiled twice, for SHA-256 and SHA-512.

Our estimates for the human effort are as follows. Symmetric algorithms like Chacha20 and SHA2 do not involve sophisticated math, and were in comparison relatively easy to prove. The proof-to-code ratio hovers around 2, and each primitive took around one person-week. Code that involves bignums requires more advanced reasoning. While the cost of proving the shared bignum code is constant, each new primitive requires a fresh verification effort. The proof-to-code ratio is up to 6, and verifying Poly1305, X25519 and Ed25519 took several person-months. High-level APIs like AEAD and SecretBox have comparably little proof substance, and took on the order of a few person-days.

Finally, we provide timings, in seconds, of the time it takes to verify a given algorithm. These are measured on an Intel Xeon workstation without relying on parallelism. The total cost of one-time HACL* verification is a few hours; when extending the library, the programmer writes and proves code interactively, and may wait for up to a minute to verify a fragment depending on its complexity.

The HACL* library is open source and is being actively developed on GitHub. Expert users can download and verify the F* code, and generate the C library themselves. Casual users can directly downloaded the generated C code. The full C library is about 7Kloc and compresses to a 42KB zip file. Restricting the library to just the NaCl API yields 5Kloc, which compresses to a 25KB file. For comparison, the TweetNaCl library is 700 lines of C code and compresses to 6Kb, whereas libsodium is 95Kloc (including 24K lines of pure C code) and compresses to a 1.8MB distributable. We believe our library is quite compact, auditable, and easy to use.

Measuring Performance We focus our performance measurements on the popular 64-bit Intel platforms found on modern laptops and desktops. These machines support 128-bit integers as well as vector instructions with up to 256-bit registers. We also measured the performance of our library on a 64-bit ARM device (Raspberry Pi 3) running both a 64-bit and a 32-bit operating system.

On each platform, we measured the performance of the HACL* library in several ways. First, for each primitive, we uses the CPU performance counter to measure the average number of cycles needed to perform a typical operation. (Using the median instead of the average yielded similar results.) Second, we used the SUPERCOP benchmarking suite to compare HACL* with state-of-the-art assembly and C implementations. Third, we used the OpenSSL speed benchmarking tool to compare the speed of the HACL* OpenSSL engine with the builtin OpenSSL engine. In the rest of this section, we describe and interpret these measurements.

Performance on 64-bit Platforms Table 5.2 shows our cycle measurements on a Xeon workstation; we also measured performance on other Intel processors, and the results were quite similar. We compare the results from HACL*, OpenSSL, and two implementations of the NaCl API: libsodium and TweetNaCl. OpenSSL and libsodium include multiple C and assembly implementations for each primitive. We are primarily interested in comparing like-for-like C implementations, but for reference, we also show the speed of the fastest assembly code in OpenSSL. In the Appendix, Table .1 ranks the top performing SUPERCOP implementations on our test machine, and Table .5 displays the OpenSSL speed measurements.

For most primitives, our HACL* implementations are as fast as (and sometimes faster than) state-of-the-art C implementations in OpenSSL, libsodium, and SUPERCOP. Notably, all our code is significantly faster than the naive reference implementations included in TweetNaCl and SUPERCOP. However, some assembly implementations and vectorized C implementations are faster than HACL*. Our vectorized Chacha20 implementation was inspired by Krovetz's 128-bit vectorized implementation, and hence is as fast as that implementation, but slower than implementations that use 256-bit vectors. Our Poly1305 and Curve25519 implementations rely on 64x64 bit multiplication;

they are faster than all other C implementations, but slower than vectorized assembly code. Our Ed25519 code is not optimized (it does not precompute fixed-base scalar multiplication) and hence is significantly slower than the fast C implementation in libsodium, but still is much faster than the reference implementation in TweetNaCl.

Table .2 measures performance on a cheap ARM device (Raspberry Pi 3) running a 64-bit operating system. The cycle counts were estimated based on the running time, since the processor does not expose a convenient cycle counter. The performance of all implementations is worse on this low-end platform, but on the whole, our HACL* implementations remain comparable in speed with libsodium, and remains significantly faster than TweetNaCl. OpenSSL Poly1305 and SHA-512 perform much better than HACL* on this device.

Performance on 32-bit Platforms Our HACL* code is tailored for 64-bit platforms that support 128-bit integer arithmetic, but our code can still be run on 32-bit platforms using our custom library for 128-bit integers. However, we expect our code to be slower on such platforms than code that is optimized to use only 32-bit instructions. Table .3 shows the performance of our code on an ARM device (Raspberry Pi 3) running a 32-bit OS. In the Appendix, Table .4 ranks the top SUPERCOP implementations on this device.

For symmetric primitives, HACL* continues to be as fast as (or faster than) the fastest C implementations of these primitives. In fact, our vectorized Chacha20 implementation is the second fastest implementation in SUPERCOP. However, the algorithms that rely on Bignum operations, such as Poly1305, Curve25519, and Ed25519, suffer a serious loss in performance on 32-bit platforms. This is because we represent 128-bit integers as a pair of 64-bit integers, and we encode 128-bit operations in terms of 32-bit instructions. Using a generic 64-bit implementation in this way results in a 3x penalty. If performance on 32-bit machines is desired, we recommend writing custom 32-bit implementations for these algorithms. As an experiment, we wrote and verified a 32-bit implementation of Poly1305 and found that its performance was close to that of libsodium. We again note that even with the performance penalty, our code is faster than TweetNaCl.

CompCert Performance Finally, we evaluate the performance of our code when compiled with the new 64-bit CompCert compiler (version 3.0) for Intel platforms. Although CompCert supports 64-bit instructions, it still does not provide 128-bit integers. Consequently, our code again needs to encode 128-bit integers as pairs of 64-bit integers. Furthermore, CompCert only includes verified optimizations and hence does not compile code that is as fast as GCC. Table 5.3 depicts the performance of HACL*, libsodium, and TweetNaCl, all compiled with CompCert. As with 32-bit platforms, HACL* performs well for symmetric algorithms, and suffers a penalty for algorithms that rely on 128-bit integers. If CompCert supports 128-bit integers in the future, we expect this

Algorithm	HACL*	libsodium	TweetNaCl
SHA-256	25.71	30.87	-
SHA-512	16.15	26.08	97.80
Salsa20	13.63	43.75	99.07
ChaCha20 (ref)	10.28	17.69	-
Poly1305	13.89	10.79	111.42
Curve25519	980,692	458,561	4,866,233
Ed25519 sign	276.66	70.71	736.07
Ed25519 verify	272.39	58.37	1153.42
Chacha20Poly1305	23.28	28.21	-
NaCl SecretBox	27.51	54.31	206.36
NaCl Box	94.63	83.64	527.07

Table 5.3: Intel64-CompCert: Performance Comparison in cycles/byte on an Intel(R) Xeon(R) CPU E5-1630 v4 @ 3.70GHz running 64-bit Debian Linux 4.8.15. Code was compiled with CompCert 3.0.1 with a custom library for 128-bit integers.

penalty to disappear.

Conclusion

Formal methods are bow applicable to large scale projects. While the complete verification of historically large codebases such as OpenSSL is still out of reach, we were able to release HACL*, a standalone, full-fledged and entirely verified cryptographic library. Besides the scale of the project, it demonstrates that our approach is viable to produce industry-grade code, both in terms of code quality and performance. After integrating the F* verification and compilation process to Mozilla's continuous integration framework, HACL*'s implementations of Curve25519, ChaCha20 and Poly1305 have been integrated to NSS, Mozilla's cryptographic library, notably used by the Firefox web browser and Red Hat. In the integration process, the code has been manually reviewed and audited, contributing and validating our approach to make the output of the KreMLin compiler as clean as possible.

HACL* benefits from all the advances presented in the preceeding chapters: it relies entirely on F* for the proofs, a heavy use of code sharing among different cryptographic primitives to reduce the proof burden and the Low* compilation backend to generate verified and fast C code. Off course, it would also benefit from the improvement of those techniques. In particular, memory safety proofs require a significant proof effort, while tools like Rust can handle it completely automatically. Specifically improving the proof automation for the memory safety clauses is future work.

Chapter 6

Going Further: Building Secure Cryptographic Applications

Parts of the text are taken from [45], a paper that appeared in IEEE S&P 2017 and was co-authored by me along with Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jianyang Pan, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy and Santiago Zanella-Béguelin, as well as from [39], a paper that appeared in IEEE S&P 2015, and was co-authored by me along with Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti and Pierre-Yves Strub.

The development and verification methodology demonstrated in the HACL* library does not only apply to cryptographic primitives, but also extends to other fields. In particular, while cryptographic algorithms are an essential component of cryptographic applications, other parts of those applications are also security critical and would certainly benefit from the additional guaranties provided by formal methods.

A typical application built on top of cryptographic libraries is a secure communication application, aimed at transferring data between two authenticated parties, while ensuring confidentiality and integrity for instance. The security of the cryptographic protocols in such applications relies not only on the cryptographic library, but also :

- on the proper combination of the cryptographic algorithms together to get security guarantees;
- on the implementation of the state machine of the protocol;
- on the parsing and serializing of the incoming and outgoing messages before and after the internal processing.

Section 6.1 illustrates how to leverage on a verified cryptographic library such as HACL* to build cryptographic proofs for complex cryptographic pro-

tocols, taking the example of the record layer in TLS1.3. Section 6.2 describes how to implement a verified state-machine monitor which can track implementation flaws for complex protocols. Such flaws can have terrible consequences for security as described in previous work on a series of attacks on TLS popular implementations [39]. Eventually we discuss in Section 6.3 how to verify the correctness of parsing and serializing algorithms, for instance following the ASN.1 standard, using the F* proof capabilities.

6.1 Towards Cryptographic Proofs of Protocols

Part of the complexity regarding cryptographic protocol verification is having end-to-end verification with a minimal trusted computing base. In practice, many *verified* cryptographic protocols do make strong assumptions about the underlying cryptography. Often, cryptographic components will be treated as black boxes, assuming perfectly secure functions, regardless of the correctness of the algorithms, and their combination together. Functional correctness and memory safety and essential, but cryptographic guarantees are the very reason why we rely on cryptographic software. Incorrectly using cryptographic primitives often results in attacks.

In order to mitigate these potential attack vectors, it is possible to leverage on the proof methods developed in the preceding chapters : one can go from functionally verified and memory safe cryptographic primitives to basic cryptographic constructions such as *Message Authenticating Codes* (MAC) and *Pseudo Random Functions* (PRF) to more elaborate constructions (*Authenticated Encryption with Associated Data* (AEAD) for instance) to complex cryptographic protocols. The detail is highlighted in a previous contribution [45]. Chapter 5 presented the value of going from F* to verified C standalone primitives, which can readily be integrated into existing projects and libraries. In this section, on the contrary, we highlight the additional security value of maintaining a full modular F* development, from the low-level cryptographic primitives to the high-level protocol. This approach allows us to strengthen the functional correctness and memory safety guaranties with cryptographic security guaranties.

We illustrate it by showing how to implement and verify the AEAD construction of the TLS 1.3 record layer, using a compositional approach to functional correctness and cryptographic security based on F*.

6.1.1 Methodology: Compositional Verification by Typing

This section does not focus on the subtleties of cryptographic proofs in F*. For more details, [70] gives a general presentation of our approach and [22] gives a probabilistic semantics of F* and additional cryptographic examples. Rather, this section focuses on the methodology and the approach we follow.

The key idea is that we use F* not only to implement cryptographic primitives, but also as the formal syntax for their game-based security definitions. This is akin to the approach taken by [43] in their proof of a TLS 1.2 handshake implementation using F7, an ancestor of F*. However, in contrast to F7, F* supports an imperative programming style that is closer to pseudo-code used by cryptographers in code-based games [26].

The AEAD cryptographic construction is a generic encryption construction which simultaneously ensures confidentiality, integrity and authenticity. The scheme relies on a PRF, which is needed for the confidentiality part, and a MAC, needed for the integrity and the authenticity part. The first step of the compositional proof is thus to map verified cryptographic primitives in F*, such as those of HACL*, to MAC functions and PRFs.

A crypto game example Let us consider a simplified version of the authenticated encryption (AE) functionality found at the core of the TLS record layer. In F*, we may write an AE module with the following interface:

```
val ℓ_p: nat (* Length of the plaintext *)
val ℓ_c: nat (* Length of the ciphertext *)
type lbytes (ℓ:nat) = b:bytes{length b = ℓ}
type bbytes (ℓ:nat) = b:bytes{length b ≤ ℓ}
type plain = lbytes ℓ_p
type cipher = lbytes ℓ_c
abstract type key
val keygen: unit → ST key
val decrypt: key → cipher → Tot (option plain)
val encrypt: k:key → p:plain → ST (c:cipher{decrypt k c = Some p})
```

Plaintexts and ciphertexts are represented as immutable bytestrings of fixed lengths ℓ_p and ℓ_c. We rely on type abbreviations to statically enforce length checks for fixed-length bytestrings using lbytes ℓ, and for bounded-length bytestrings using bbytes ℓ. The excerpt uses immutable bytestrings for simplicity, the real record-layer implementation also uses mutable buffers of bytes.

Next, our interface defines an abstract type key; because of the abstraction, values of this type can only be generated via the keygen method and accessed via encrypt and decrypt. In this setting the internal representation of keys is hidden from all other modules to protect their integrity and secrecy. The function keygen needs to generate randomness by calling an effectful external function; so we give this function the ST effect to indicate that the computation is impure and stateful, even though it does not explicitly modify the memory. In particular, two calls to keygen may yield different results. The function encrypt would typically generate a nonce for use in the underlying AE construction, and hence is also marked as stateful. In contrast, decrypt is deterministic, so is marked with the Tot effect. Its result is an optional plain value: either Some p if decryption succeeds, or None (or $\perp$ in pseudo-code) otherwise.

This interface does not express any security guarantees yet, but it does require a functional correctness guarantee, namely that decryption undoes encryption.

Expressing Security Guaranties Given an AE scheme, one usually measures its concrete security as the advantage of an adversary $\mathcal{A}$ that attempts to guess the value of b in the following game:

Game $\mathsf{Ae}(\mathcal{A}, \mathsf{AE})$

$b \xleftarrow{\$} \{0,1\};\ L \leftarrow \varnothing;\ k \xleftarrow{\$} \mathsf{AE.keygen}()$
$b' \leftarrow \mathcal{A}^{\mathsf{Encrypt},\mathsf{Decrypt}}();\ \textbf{return}\ (b \stackrel{?}{=} b')$

Oracle $\mathsf{Encrypt}(p)$

if b **then** $c \xleftarrow{\$} \mathsf{byte}^{\ell_c};\ L[c] \leftarrow p$
else $c \leftarrow \mathsf{AE.encrypt}\ k\ p$
return c

Oracle $\mathsf{Decrypt}(c)$

if b **then** $p \leftarrow L[c]$
else $p \leftarrow \mathsf{AE.decrypt}\ k\ c$
return p

The adversary $\mathcal{A}$ is a program that can call the two oracle functions to encrypt and decrypt using a secret key k. In the real case ($b = 0$) they just call the real AE implementation. In the ideal case ($b = 1$), Encrypt returns a randomly sampled ciphertext and stores the associated plaintext in a log L, while Decrypt performs decryption by looking up the plaintext in the log, returning $\perp$ when there is no plaintext associated with the ciphertext. Ideal AE is perfectly secure, inasmuch as the ciphertext does not depend on the plaintext. Thus, we define AE security by saying that the attacker cannot easily distinguish between the real and ideal cases.

For such a game as is standard, we define $\mathcal{A}$'s advantage probabilistically as $|2\Pr[\mathsf{Ae}(\mathcal{A}, \mathsf{AE})] - 1|$, e.g. an adversary flipping a coin to guess b will succeed with probability $\frac{1}{2}$ and has 0 advantage.

Embedding games in F* modules Although we wrote the game Ae^b above in pseudo-code, each the game reflects a verified F* module, written e.g. AE^b, that uses a boolean flag b to select between real and ideal implementations of the underlying cryptographic module AE. For example, AE^b may define the key type and encrypt function as

```
abstract type key = {key: AE.key; log: encryption_log}
let encrypt (k:key) (p:plain) =
  if b then
    let c = random_bytes ℓc in
    k.log ← k.log ++ (c,p);
    c
  else AE.encrypt k.key p
```

where the (private) key representation now includes both the real key and the ideal encryption log. The encrypt function uses k.log to access the current log, and ++ to append a new entry, much as the Encrypt oracle.

Idealization Interfaces The idealized module AE^b can be shown to implement the following typed interface that reflects the security guarantee of the Ae^b game:

```
abstract type key
val log: memory → key → Ghost (seq (cipher × plain))
val keygen: unit → ST k:key
  (ensures b ⟹ log k′ = ∅)
val encrypt: k:key → p:plain → ST (c:cipher)
  (ensures b ⟹ log k′ = log k ++ (c,p))
val decrypt: k:key → c:cipher → ST (o:option plain)
  (ensures b ⟹ o = lookup c (log k) ∧ log k' = log k)
```

The interface declares keys as abstract, hiding both the real key value and the ideal log, and relies on the log to specify the effects of encryption and decryption. To this end, it provides a log function that reads the current content of the log—a sequence of ciphertexts and plaintexts. This function is marked as Ghost, indicating that it may be used *only in specification* and will be discarded by the compiler after typechecking.

Each of the 3 ensures clauses above uses this proof-only function to specify the state of the log before (log k) and after the call (log k′). Hence, the interface states that, in the ideal case, the function keygen creates a key with an empty log; encrypt k p returns a ciphertext c and extends the log for k with an entry mapping c to p; and decrypt k c returns exactly the result of looking up for c in the current log. This post-condition formally guarantees that decrypt succeeds if and only if it is passed a ciphertext that was generated by encrypt; in other words it guarantees both functional correctness and authentication (a notion similar to INT-CTXT).

AE^b is also parametrized by a module Plain^b that defines abstract plaintexts, with an interface that allows access to their concrete byte representation only when $b = 0$ (for real encryption). By typing AE^b, we verify that, when $b = 1$, our idealized functionality is independent (information-theoretically) from the values of the plaintexts it processes.

From the viewpoint of the application, the plaintext abstraction guarantees that AE^1 preserves the confidentiality and integrity of encrypted data (as in classic information flow type systems). An application can rely on this fact to prove application-level guarantees. For instance, an application may prove, as an invariant, that only well-formed messages are encrypted under a given key, and thus that parsing and processing a decrypted message always succeeds.

Probabilistic Semantics We model randomness (e.g. random_bytes) using primitive sampling functions. Two Boolean terminating F* programs A^0 and A^1 are equivalent, written $A^0 \approx A^1$, when they return true with the same probability. They are ϵ-equivalent, noted $A^0 \approx_\epsilon A^1$, when $|\Pr[A^1 \Downarrow \mathsf{true}] - \Pr[A^0 \Downarrow \mathsf{true}]| \leq \epsilon$ where $\Pr[A \Downarrow v]$ denotes the probability that program A evaluates to

value v according to the probabilistic semantics of F*. These definitions extend to program evaluation contexts, written $A^b[\_]$, in which case ϵ depends on the program plugged into the context, which intuitively stands for the adversary. From here on, we can develop code-based game-playing proofs following the well-established approach of Bellare and Rogaway [26] directly applied to F* programs rather than pseudo-code.

Security definitions will consist of a game and a notation for the adversary advantage, parameterized by a measure of oracle use (e.g. how many times an adversary calls an oracle is called). We can then provide concrete bounds on those advantages, as a function of their parameters for the record layer. To this end, in [45] for the full record composition, reduction theorems will relate the advantage for a given construction to the advantages of its building blocks.

Games vs Idealized Modules We conclude this presentation of our approach by discussing differences between the games on paper and the modules of the F* implementation.

Standard-compliant modules include many details elided in informal games; they also use lower level representations to yield more efficient code, and require additional type annotations to keep track of memory management.

These modules are part of the HACL* general-purpose verified cryptographic libraries, providing real functionality (when idealizations flags are off) so they always support multiple instances of their functionality. Here for instance, AE^b has a function to generate keys, passed as parameters to the encrypt function, whereas the game oracle uses a single, implicit key.

Modules rely on the F* type system to enforce the rules of the games. Hence, dynamic checks in games (say, to test whether a nonce has already been used) are often replaced with static pre-conditions on typed adversaries. Similarly, types enforce many important but trivial conditions, such as the length of oracle arguments, and are often kept implicit in the paper.

6.1.2 One-Time MACs

The AEAD construction uses fresh key materials for each message, so we consider authentication when keys are used to compute at most one MAC.

We treat one main MAC construction, Poly1305, although we could rely on others, such as GHASH. We suppose that the whole key is freshly generated for each MAC (as in ChaCha20-Poly1305).

One-time MAC functionality and security

The interface for the message authentication code (MAC) is outlined below:

```
val ℓ_{k_0}: nat (* static key length, may be 0 *)
val ℓ_k: n:nat {ℓ_{k_0} ≤ ℓ_k} (* total key length *)
```

```
val ℓ_t: nat (* tag length *)
val ℓ_m: nat (* maximal message length *)
type key_0 = lbytes ℓ_k0 (* static key shared between MACs *)
type key = lbytes ℓ_k (* one-time key (including static key) *)
type tag = lbytes ℓ_t (* authentication tag *)
type message = b:bbytes ℓ_b {wellformed b}
val keygen_0: unit → ST key_0
val keygen: key_0 → ST key
val verify: key → message → tag → Tot bool
val mac: k:key → m:message → Tot (t:tag{verify k m t})
```

This interface defines concrete byte formats for keys, tags, and messages. Authenticated messages are strings of at most ℓ_m bytes that comply with an implementation-specific well-formedness condition. We let m range over well-formed messages.

Key-generation functions are marked as stateful (ST) to reflect their use of random sampling. Static keys of type key_0 may be used to generate multiple one-time keys of type key. (For example, keygen may concatenate the static key with $\ell_k - \ell_{k_0}$ random bytes.) To begin with, we assume $\ell_{k_0} = 0$ so that k_0 is the empty string ε.

The two main functions produce and verify MACs. Their correctness is captured in the verify post-condition of mac: verification succeeds at least on the tags correctly produced using mac with matching key and message.

One-Time Security MAC security is usually defined using computational unforgeability, as in the following game:

Game UF-1CMA($\mathcal{A}$, MAC)	**Oracle** Mac(m)
$k \xleftarrow{\$} \mathsf{MAC.keygen}(\varepsilon)$; $log \leftarrow \bot$	**if** $log \neq \bot$ **return** $\bot$
$(m^\star, t^\star) \leftarrow \mathcal{A}^{\mathsf{Mac}}$	$t \leftarrow \mathsf{MAC.mac}(k, m)$
return $\mathsf{MAC.verify}(k, m^\star, t^\star)$	$log \leftarrow (m, t)$
$\quad \wedge\ log \neq (m^\star, t^\star)$	**return** t

The oracle permits the adversary a single chosen-message query (recorded in *log*) before trying to produce a forgery. The advantage of $\mathcal{A}$ playing the UF-1CMA game is defined as $\epsilon_{\mathsf{UF\text{-}1CMA}}(\mathcal{A}[\ell_m]) \triangleq \Pr[\mathsf{UF\text{-}1CMA}(\mathcal{A}, \mathsf{MAC}) = 1]$.

We seek a stronger property for AEAD—the whole ciphertext must be indistinguishable from random bytes—and we need a decisional game for type-based composition, so we introduce a variant of unforgeability that captures indistinguishability from a random tag (when r is set).

Definition 1 (IND-UF-1CMA). *Let $\epsilon_{\mathsf{Mac1}}(\mathcal{A}[\ell_m, q_v])$ be the advantage of an adversary $\mathcal{A}$ that makes q_v* Verify *queries on messages of length at most ℓ_m in the following game:*

Game $\mathsf{Mac1}^b(\mathsf{MAC})$

$k \xleftarrow{\$} \mathsf{MAC.keygen}(\varepsilon)$; $log \leftarrow \bot$
return {Mac, Verify}

Oracle $\mathsf{Verify}(m^\star, t^\star)$

if b **return** $log = (m^\star, t^\star)$
return $\mathsf{MAC.verify}(k, m^\star, t^\star)$

Oracle $\mathsf{Mac}(m)$

if $log \neq \bot$ **return** $\bot$
$t \leftarrow \mathsf{MAC.mac}(k, m)$
if $b \wedge r$
  $t \xleftarrow{\$} \mathsf{byte}^{\mathsf{MAC}.\ell_t}$
$log \leftarrow (m, t)$
return t

In this game, the MAC oracle is called at most once, on some chosen message m; it returns a tag t and logs (m, t). Conversely, Verify is called q_v times before and after calling MAC. When b is set, the game idealizes MAC in two ways: verification is replaced by a comparison with the log; and (when r is also set) the tag is replaced with random bytes.

This definition implies UF-1CMA (see [45]) when $q_v \geq 1$ and that random tags are neither necessary nor sufficient for unforgeability.

Verified Implementation m-IND-UF-1CMA security reflects the type-based security specification of our idealized module $\mathsf{MMac1}^b$, which has an interface of the form

```
val log: memory → key → Ghost (option (message × tag))
val mac: k:key → m:message → ST (t:tag)
  (requires log k = None)
  (ensures log k′ = Some(m,t))
val verify: k:key → m:message → t:tag → ST (v:bool)
  (ensures b ⟹ v = (log k′ = Some(m,t)))
```

The types of mac and verify express the gist of our security property: the specification function log gives access to the current content of the log associated with a one-time key; mac requires that the log be empty (None in F*) thereby enforcing our one-time MAC discipline; verify ensures that, when b is set, verification succeeds if and only if mac logged exactly the same message and tag. Their implementation is automatically verified by typing $\mathsf{MMac1}^b$. However, recall that typing says nothing about the security loss incurred by switching b—this is the subject of the next subsection.

Our verified implementation of $\mathsf{MMac1}^b$ supports the construction described next, including code and functional correctness proofs for the algorithm. It also provides a more efficient interface for computing MACs incrementally. Instead of actually concatenating all authenticated materials in a message, the user creates a stateful hash, then repeatedly appends 16-byte words to the hash, and finally calls mac or verify on this hash, with a type that binds the message to the final hash contents in their security specifications. Our code further relies on indexed abstract types to separate keys and hashes for different instances of the functionality, and to support static key compromise.

6.1.3 Wegman-Carter-Shoup (WCS) Constructions

Next, we set up notations so that our presentation applies to multiple constructions, including of course Poly1305; we factor out the encodings to have a core security assumption on sequences of field elements; we verify their injectivity; we finally prove concrete bounds in general, and in particular for Poly1305.

From bytes to polynomials and back In addition to fixed lengths for keys and tags, the construction is parameterized by

- a field $\mathbb{F}$;
- an encoding function $\overline{\cdot}$ from messages to polynomials in $\mathbb{F}$, represented as sequences of coefficients $\overline{m} \in \mathbb{F}^*$.
- a truncation function from $e \in \mathbb{F}$ to $\mathsf{tag}(e) \in \mathsf{byte}^{\ell_t}$;

The key consists of two parts: an element $r \in \mathbb{F}$ and a one-time pad $s \in \mathsf{byte}^{\ell_t}$. We assume that r and s are sampled uniformly at random, from some $R \subseteq \mathbb{F}$ and from byte^{ℓ_t}, respectively. We write $r \| s \leftarrow k$ for the parsing of key materials into r and s, including the encoding of r into R.

Generic Construction Given a message m encoded into the sequence of d coefficients $\overline{m}_0, \ldots, \overline{m}_{d-1}$ of a polynomial $\overline{m}(x) = \sum_{i=1..d} \overline{m}_{d-i} x^i$ in $\mathbb{F}$, the tag is computed as:

$$\begin{array}{rl} \mathsf{hash}_r(m) \leftarrow \mathsf{tag}(\overline{m}(r)) & \text{in } \mathbb{F} \text{ before truncation} \\ \mathsf{mac}(r\|s, m) \leftarrow \mathsf{hash}_r(m) \boxplus s & \text{in } \mathsf{byte}^{\ell_t} \end{array}$$

where the blinding operation $\boxplus$ is related to addition in $\mathbb{F}$ (see specific details below). We refer to $\mathsf{hash}_r(m)$, the part of the construction before blinding, as the hash.

Poly1305 uses the field $GF(p)$ for $p = 2^{130} - 5$, that is, the prime field of integer addition and multiplication modulo p, whose elements can all be represented as 130-bits integers. Its message encoding $\overline{\cdot}$ similarly splits the input message into 16-byte words, seen as integers in $0..2^{128} - 1$, then adds 2^ℓ to each of these integers, where ℓ is the word length in bits. (Hence, the encoding precisely keeps track of the length of the last word; this feature is unused for AEAD, which applies its own padding to ensure $\ell = 128$.) The truncation function is $\mathsf{tag}(e) = e \mod 2^{128}$. The blinding operation $\boxplus$ and its inverse $\boxminus$ are addition and subtraction modulo 2^{128}. For ChaCha20-Poly1305, both r and s are single-use ($\ell_{k_0} = 0$) but our proof also applies to the original Poly1305-AES construction [31] where r is shared.

Injectivity Properties We intend to authenticate messages, not just polynomial coefficients. To this end, we instantiate our wellformed predicate on messages and show (in F*) that

$$\forall(m0\text{: bytes})\ (m1\text{: bytes}).\ (\ \mathsf{wellformed}\ m0 \wedge \mathsf{wellformed}\ m1 \wedge \mathsf{Poly.equals}\ \overline{m0}\ \overline{m1}\ ) \Longrightarrow m0 = m1$$

where Poly.equals specifies the equality of two formal polynomials by comparing their sequences of coefficients, extending the shorter sequence with zero coefficients if necessary.

We verify that the property above suffices to prove that both encodings are secure, and also that it holds in particular once we define wellformed as the range of formatted messages for AEAD (which are 16-byte aligned and embed their own lengths). We also confirm by typing that, with Poly1305, there is no need to restrict messages: its encoding is injective for all bytestrings [31, Theorem 3.2].

From those properties, we can deduce a security theorem and concrete bounds for Poly1305.

6.1.4 Pseudo-Random Functions for AEAD

Let us now consider the use of symmetric ciphers in counter mode, both for keying one-time MACs and for generating one-time pads for encryption. We model ciphers as PRFs. From HACL*, we will use Chacha20. A pseudo-random function family PRF implements the following interface:

```
type key
val keygen: unit → ST key
val ℓ_d : nat (* fixed domain length *)
val ℓ_b : nat (* fixed block length *)
type domain = lbytes ℓ_d
type block = lbytes ℓ_b
val eval: key → domain → Tot block (* λctional specification *)
```

This interface specifies an abstract type for keys and a key-generation algorithm. (Type abstraction ensures that these keys are used only for PRF computations.) It also specifies concrete, fixed-length bytestrings for the domain and range of the PRF, and a function to compute the PRF. We refer to the PRF outputs as blocks. As usual, we define security as indistinguishability from a uniformly random function with lazy sampling.

Game $\mathsf{Prf}^b(\mathsf{PRF})$	**Oracle** $\mathsf{Eval}(m)$
$T \leftarrow \varnothing$	**if** $T[m] = \bot$
$k \xleftarrow{\$} \mathsf{PRF.keygen}()$	**if** b **then** $T[m] \xleftarrow{\$} \mathsf{byte}^{\ell_b}$
return $\{\mathsf{Eval}\}$	**else** $T[m] \leftarrow \mathsf{PRF.eval}(k, m)$
	return $T[m]$

Verified Implementation

We use an idealized PRF module parametrized by a Cipher module that implements real ChaCha20 and by a MAC module. The separation of the

PRF domain is enforced by typing: depending on alg, j_0, j, b, and b', its range includes keys, blocks, and pairs (p, c).

6.1.5 From MAC and PRF to AEAD

From the MAC and the PRF constructions, we can implement the two main AEAD constructions used by TLS 1.3 and modern ciphersuites of TLS 1.2. Their composition of a PRF and a one-time MAC yields a standard notion of AEAD security. The proof is generic and carefully designed to be modular and TLS-agnostic: the AEAD code is shared between TLS 1.2 and 1.3, and can be generalized for other protocols such as QUIC.

AEAD functionality Our F* authenticated encryption with associated data (AEAD) implementation has a real interface of the form

```
val ℓn: nat (* fixed nonce length *)
val ℓa: n:nat{n < 2^32} (* maximal AD length *)
val ℓp: n:nat{n < 2^32} (* maximal plaintext length *)
val cipherlen: n:nat{n ≤ ℓp} → Tot nat
type nonce = lbytes ℓn
type ad = bbytes ℓa
type plain = bbytes ℓp
type cipher = bytes

val decrypt: key → nonce → ad → c:cipher →
  ST (option (p:plain{length c = cipherlen (length p)}))
val encrypt: k:key → n:nonce → a:ad → p:plain →
  ST (c:cipher{length c = cipherlen (length p)})
```

with two functions to encrypt and decrypt messages with associated data of variable lengths, and types that specify the cipher length as a function of the plain length. We omit declarations for keys, similar to those for PRFs earlier in the section.

Definition 2 (Aead security)**.** *Let* $\epsilon_{\mathsf{Aead}}(\mathcal{A}[q_e, q_d, \ell_p, \ell_a])$ *be the advantage of an adversary that makes at most* q_e Encrypt *and* q_d Decrypt *queries on messages and associated data of lengths at most* ℓ_p *and* ℓ_a *in the game:*

Game $\mathsf{Aead}^b(\mathsf{AEAD})$

$C \leftarrow \varnothing$
$k \xleftarrow{\$} \mathsf{AEAD.keygen}()$
return {Encrypt, Decrypt}

Oracle $\mathsf{Encrypt}(n, a, p)$

if $C[n] \neq \perp$ **return** $\perp$
if b $c \xleftarrow{\$} \mathsf{byte}^{\mathsf{cipherlen}(|p|)}$
else $c \leftarrow \mathsf{AEAD.encrypt}(k, n, a, p)$
$C[n] \leftarrow (a, p, c)$
return c

Oracle $\mathsf{Decrypt}(n, a, c)$

if b
  if $C[n] = (a, p, c)$ *for some* p
    return p
  return $\perp$
else
  $p \leftarrow \mathsf{AEAD.decrypt}(k, n, a, c)$
  return p

Our definition generalizes the AE game presented earlier; it has a richer domain with plaintext and associated data of variable lengths; a function cipherlen from plaintext lengths to ciphertext lengths; and nonces n. It similarly maintains a log of encryptions, indexed by nonces. Crucially, Encrypt uses the log to ensure that each nonce is used at most once for encryption.

Generic AEAD Construction Given a PRF and a compatible MAC, AEAD splits plaintexts into blocks which are then blinded by pseudo-random one-time pads generated by calling PRF on increasing counter values. (Blocks for MAC keys and the last mask may require truncation.)

To authenticate the ciphertext and associated data, the construction formats them into a single 16-byte-aligned buffer (ready to be hashed as polynomial coefficients) using an encoding function declared as val encode: bbytes ℓ_p × bbytes ℓ_a → Tot bbytes $(\ell_p + \ell_a + 46)$ and implemented (in pseudo-code) as

Function $\mathsf{encode}(c, a)$	**Function** $\mathsf{pad}_{16}(b)$
return $\mathsf{pad}_{16}(a) \,\|\, \mathsf{pad}_{16}(c)$	$r, b_1, \ldots, b_r \leftarrow \mathsf{split}_{16}(b)$
$\quad \|\, \mathsf{length}_8(a) \,\|\, \mathsf{length}_8(c)$	**return** $b \,\|\, \mathsf{zeros}(16 - \lvert b_r \rvert)$

where the auxiliary function $\mathsf{split}_\ell(b)$ splits the bytestring b into a sequence of r non-empty bytestrings, all of size ℓ, except for the last one which may be shorter. (that is, if $r, b_1, \ldots b_r \leftarrow \mathsf{split}_b(\ell)$, then $b = b_1 \,\|\, \cdots \,\|\, b_r$.); where $\mathsf{zeros}(\ell)$ is the bytestring of ℓ zero bytes; and where $\mathsf{length}_8(n)$ is the 8-byte representation of the length of n. Thus, our encoding adds minimal zero-padding to a and c, so that they are both 16-bytes aligned, and appends a final 16-byte encoding of their lengths.

The rest of the AEAD construction is defined below, using an operator $otp \,\overline{\oplus}\, p$ that abbreviates the expression $\mathsf{truncate}(otp, |p|) \oplus p$, and a function untag_{16} that separates the ciphertext from the tag.

Function keygen()

$k \xleftarrow{\$} \mathsf{PRF.keygen}()$; $k_0 \leftarrow \varepsilon$
if j_0
  $o \leftarrow \mathsf{PRF.eval}(k, 0^{\ell_b})$
  $k_0 \leftarrow \mathsf{truncate}(o, \mathsf{MAC}.\ell_{k_0})$
return $k_0 \| k$

Function $\mathsf{encrypt}(K, n, a, p)$	**Function** $\mathsf{decrypt}(K, n, a, c)$
$(k_0, k) \leftarrow \mathsf{split}_{\ell_{k_0}}(K)$; $c \leftarrow \varepsilon$	$(k_0, k) \leftarrow \mathsf{split}_{\ell_{k_0}}(K)$; $p \leftarrow \varepsilon$
$k_1 \leftarrow \mathsf{PRF.eval}(k, j_0 \| n)$	$k_1 \leftarrow \mathsf{PRF.eval}(k, j_0 \| n)$
$k_m \leftarrow \mathsf{truncate}(k_0 \| k_1, \mathsf{MAC}.\ell_k)$	$k_m \leftarrow \mathsf{truncate}(k_0 \| k_1, \mathsf{MAC}.\ell_k)$
$r, p_1, \ldots, p_r \leftarrow \mathsf{split}_{\ell_b}(p)$;	$(c, t) \leftarrow \mathsf{untag}_{16}(c)$
for $j = 1..r$	$m \leftarrow \mathsf{encode}(c, a)$
$\quad otp \leftarrow \mathsf{PRF.eval}(k, j_0 + j \| n)$	**if** $\neg \mathsf{MAC.verify}(k_m, m, t)$
$\quad c \leftarrow c \,\|\, (otp \,\overline{\oplus}\, p_j)$	$\quad$ **return** $\perp$
$t \leftarrow \mathsf{MAC.mac}(k_m, \mathsf{encode}(c, a))$	$r, c_1, \ldots, c_r \leftarrow \mathsf{split}_{\ell_b}(c)$;
return $c \| t$	**for** $j = 1..r$
	$\quad otp \leftarrow \mathsf{PRF.eval}(k, j_0 + j \| n)$
	$\quad p \leftarrow p \,\|\, (otp \,\overline{\oplus}\, c_j)$
	return p

The main result is that it is Aead-secure when PRF is Prf-secure and MAC is MMac1-secure [45]:

Theorem 6.1.1 (AEAD construction)**.** *Given* $\mathcal{A}$ *against* Aead*, we construct* $\mathcal{B}$ *against* Prf *and* $\mathcal{C}$ *against* MMac1*, with:*

$$\epsilon_{\mathsf{Aead(AEAD)}}(\mathcal{A}[q_e, q_d, \ell_p, \ell_a]) \leq \epsilon_{\mathsf{Prf(PRF)}}(\mathcal{B}[q_b]) + \epsilon_{\mathsf{MMac1(MAC)}}(\mathcal{C}[\ell_p + \ell_a + 46, q_d, q_e + q_d])$$

where q_b *(the number of distinct queries to the PRF) satisfies:*

$$q_b \leq j_0 + q_e \left(1 + \left\lceil \frac{\ell_p}{\ell_b} \right\rceil\right) + q_d$$

Verified Implementation We outline below the idealized interface of our main AEAD^b module built on top of the idealized interfaces of PRF and MAC, the security guaranties of both taken as cryptographic assumption. For types for encryption and decryption:

```
abstract type key (* stateful key, now containing the log *)
val log: memory → key →
  Ghost (seq (nonce × ad × cipher × plain)
val keygen : unit → ST (k:key)
  (ensures b ⟹ log k = ∅)
val encrypt: k:key → n:nonce → a:ad → p:plain → ST (c:cipher)
  (requires b ⟹ lookup_nonce n (log k) = None)
  (ensures (b ⟹ log k' = log k ++ (n,a,c,p)))
val decrypt: k:key → n:nonce → a:ad → c:cipher →
  ST (o:option plain)
  (ensures b ⟹ o = lookup (n,a,c) (log k))
```

This is a multi-instance idealization, with a log for each instance stored within an abstract, stateful key; and we provide a proof-only function log to access its current contents in logical specifications. Hence, key generation allocates an empty log for the instance; encryption requires that the nonce be fresh and records its results; and decryption behaves exactly as a table lookup, returning a plaintext if, and only if, it was previously stored in the log by calling encryption with the same nonce and additional data.

This step of the construction is entirely verifiable by typing. To this end, we supplement its implementation with a precise invariant that relates the AEAD log to the underlying PRF table and MAC logs. For each entry in the log, we specify the corresponding entries in the PRF table (one for the one-time MAC key, and one for each block required for encryption) and, for each one-time MAC key entry, the contents of the MAC log (an encoded message and the tag at the end of the ciphertext in the AEAD log entry). By typing the AEAD code that implements the construction, we verify that the invariant is preserved as it completes its series of calls to the PRF and MAC idealized

interfaces. Hence, although our code for decryption does *not* actually decrypt by a log lookup, we prove that (when b holds) its results always matches the result of a lookup on the current log. As usual, by setting all idealization flags to false, the verified code yields our concrete TLS implementation.

Following this methodology, one can build and run a full, verified down to the cryptographic primitives, implementation of her protocol.

6.2 A Verified State Machine for OpenSSL

In complex protocols, such as TLS, properly implementing the cryptographic constructions is not the only challenging part. Implementing composite state machines for TLS has also proven to be hard and error-prone. Systematic state machine testing can be useful to uncover bugs but does not guarantee that all flaws have been found and eliminated. Instead, it would be valuable to formally prove that a given state machine implementation complies with the TLS standard. Since new ciphersuites and protocol versions are continuously added to TLS implementations, it would be even better if we could set up an automated verification framework that could be maintained and systematically used to prevent regressions.

The miTLS implementation [42] uses refinement types to verify that its handshake implementation is correct with respect to a logical state machine specification. The handshake protocol is one of the four inner protocols described in the TLS specification. It is responsible for negotiating the shared secret between the two communicating parties, on an insecure channel and with dynamically chosen parameters. However, in the miTLS implementation, it only covers `RSA` and `DHE` ciphersuites and only applies to carefully written F* code. In this section, we investigate how to achieve a similar, if less ambitious, proof for the state machine implemented in the OpenSSL open source library.

OpenSSL Clients and Servers In OpenSSL 1.0.1j, the client and server state machines for `SSLv3` and `TLSv1.0-TLSv1.2` are implemented in `ssl/s3_clnt.c` and `ssl/s3_srvr.c`, respectively. Both state machines maintain a data structure of type `SSL` that has almost 100 fields, including negotiation parameters like the version and ciphersuite, cryptographic material like session keys and certificates, running hashes of the handshake log, and other data specific to various TLS extensions.

Both state machines implement the message sequences depicted in Figure .1 in Appendix 7, structured as an infinite loop with a large switch statement, where each case corresponds to a different state, roughly one for each message in the protocol. Depending on the state, the switch statement either calls a `ssl3_send_*` function to construct and send a message or calls a `ssl3_get_*` function to receive and process a message.

For example, when the OpenSSL client is in the state `SSL3_ST_CR_KEY_EXCH_A`, it expects to receive a `ServerKeyExchange`, so it calls the function `ssl3_get_key_exchange(s)`.

This function in turn calls ssl3_get_message (in s3_both.c) and asks to receive *any* handshake message. If the received message is a `ServerKeyExchange`, it processes the message. Otherwise, it assumes that the message was optional and returns control to the state machine which transitions to the next state (to try and process the message as a `CertificateRequest`). If the `ServerKeyExchange` message was in fact not optional, this error may only be discovered later when the client tries to send the `ClientKeyExchange` message.

Due to its complex handling of optional messages, it is often difficult to understand whether an OpenSSL client or server correctly implements the intended state machine. The flaws discussed in [39] indicate that they do not. Furthermore, the message sequence needs to be consistent with the values stored in the SSL session structure (such as the handshake hashes), and this is easy to get wrong.

A new state machine We propose a new state machine structure for OpenSSL that makes the allowed message sequences more explicit and easier to verify.

In addition to the full SSL data structure that is maintained and updated by the OpenSSL messaging functions, we define a separate data structure that includes only those elements that we need to track the message sequences allowed by Figure .1 in Appendix 7:

```
typedef struct state {
  Role role; /* r ∈ \{Client,Server\} */
  PV version; /* v ∈ {SSLv3, TLSv1.0, TLSv1.1, TLSv1.2} */
  KEM kx; /* kx ∈ {DH*, ECDH*, RSA*} */
  Auth client_auth; /* (c_ask, c_offer) */
  int resumption; /* (r_id, r_tick) */
  int renegotiation; /* reneg = 1 if renegotiating */
  int ntick; /* n_tick */

  Msg_type last_message; /* previous message type */
  unsigned char* log; /* full handshake log */
  unsigned int log_length;
} STATE;
```

The STATE structure contains various negotiation parameters: a role that indicates whether the current state machine is being run in a client or a server, the protocol version (v in Figure .1), the key exchange method (kx), the client authentication mode (c_{ask}, c_{offer}), and flags that indicate whether the current handshake is a resumption or a renegotiation, and whether the server sends a `ServerNewSessionTicket`. Each field is represented by an enum that includes an UNDEFINED value to denote the initial state. The server sets all the fields except client_auth immediately after `ServerHello`. The client must wait until later in the handshake to discover the final values for resumption, client_auth and ntick.

The STATE structure keeps track of the last message received, to record the current position within a protocol message sequence. It also keeps the full

handshake log as a byte array. We use this array to specify and verify our invariants about the state machine, but in production environments it would probably be replaced by the running hashes of the handshake log already maintained by OpenSSL.

The core of our state machine is in one function:

```
int ssl3_next_message(SSL* ssl, STATE *st,
      unsigned char* msg, int msg_len,
      int direction, unsigned char content_type);
```

This function takes the current state (ssl,st), the next message to send or receive msg, the content type (handshake/CCS/alert/application data) and direction (outgoing/incoming) of the message. Whenever a message is received by the record layer, this function is called. It then executes one step of the state machine in Figure .1 to check whether the incoming message is allowed in the current state. If it is, it calls the corresponding message handler, which processes the message and may in turn want to send some messages by calling ssl3_next_message with an outgoing message. For an outgoing message, the function again checks whether it is allowed by the state machine before writing it out to the record layer. In other words, ssl3_next_message is called on all incoming and outgoing messages. It enforces the state machine and maintains the handshake log for the current message sequence.

To implement our verified state machine, while being inter-operable with OpenSSL's original code, we were able to reuse the OpenSSL message handlers (with small modifications). We wrote our own simple message parsing functions to extract the handshake message type, to extract the protocol version and key exchange method from the `ServerHello`, and to check for empty certificates.

Experimental Evaluation We tested our new state machine implementation in two ways.

First, we checked that our new state machine does not inhibit compliant message sequences for ciphersuites supported by OpenSSL. To this end, we implemented our state machine as an inline reference monitor. As before, the function ssl3_get_message is called whenever a message is to be sent or received. However, it does not itself call any message handlers; it simply returns success or failure based on whether the incoming or outgoing message is allowed. Other than this modification, messages are processed by the usual OpenSSL machine. In effect, our new state machine runs in parallel with OpenSSL on the same traces.

We ran this monitored version of OpenSSL against various implementations and against OpenSSL itself (using its inbuilt tests). We tested that our inline monitor does not flag any errors for these valid traces. In the process, we found and fixed some early bugs in our state machine.

Second, we checked that our new state machine does detect and prevent the deviant traces presented in [39]. We ran our monitored OpenSSL implementation against a FlexTLS peer running deviant traces and, in every case, our monitor flagged an error.

Logical Specification of the State Machine To gain further confidence in our new state machine, we formalized the allowed message traces of Figure .1 as a logical invariant to be maintained by ssl3_next_message. Our invariant is called isValidState and is depicted in Figure 6.1.

The predicate StateAfterInitialState specifies how the STATE structure is initialized at the beginning of a message sequence. The predicate isValidState says that the current STATE structure should be consistent with either the initial state or the expected state after receiving some message; it has a disjunction for every message handled by our state machine.

For example, after `ServerHelloDone` the current state st must satisfy the predicate StateAfterServerHelloDone. This predicate states that there has to exist a previous state prev and a new (message), such that the following holds:

- message must be a `ServerHelloDone`,
- st→last_message must be S_HD (a Msg_type denoting `ServerHelloDone`),
- st→log must be the concatenation of prev→log and the new message,
- and for each incoming edge in the state machine:
  - the previous state prev must an allowed predecessor (a valid state after an allowed previous message),
  - if the previous message was `CertificateRequest` then st→client_auth remains unchanged from prev→client_auth; in all other cases it must be set to AUTH_NONE
  - (plus other conditions to account for other ciphersuites.)

Predicates like StateAfterServerHelloDone can be directly encoded by looking at the state machine; they do not have to account for the particular details of any implementation. Indeed, our state predicates look remarkably similar to (and were inspired by) the *log predicates* used in the cryptographic verification of miTLS [42]. The properties they capture depend only on the TLS specification; except for syntactic differences, they are even independent of the programming language.

Verification with Frama-C To mechanically verify that our state machine implementation satisfies the isValidState specification, we used the C verification tool Frama-C [64]. Contrary to the HACL* approach where we implement code in the high-level F* language, verify that some properties hold and then compile it down to C code while retaining the original properties, here we start straight from C code. The Frama-C tool is built to work on native C

```
predicate isValidState(STATE *state) =
        StateAfterInitialState(state) || StateAfterClientHello(state) ||
        StateAfterServerHello(state) || StateAfterServerCertificate(state) ||
        StateAfterServerKeyExchange(state) || StateAfterServerCertificateRequest(state) ||
        StateAfterServerHelloDone(state) || StateAfterClientCertificate(state) ||
        StateAfterClientKeyExchange(state) || StateAfterClientCertificateVerify(state) ||
        StateAfterServerNewSessionTicket(state) || StateAfterServerCCS(state) ||
        StateAfterServerFin(state) || StateAfterClientCCS(state) ||
        StateAfterClientFin(state) || StateAfterClientCCSLastMsg(state) ||
        StateAfterClientFinLastMsg(state);

predicate StateAfterInitialState(STATE *state) =
   state→version == UNDEFINED_PV && state→role == UNDEFINED_ROLE &&
   state→kx == UNDEFINED_CS &&
   state→last_message == UNDEFINED_TYPE  &&
   state→log_length == 0 && state→client_auth == UNDEFINED_AUTH &&
   state→resumption == UNDEFINED_RES && state→ntick == UNDEFINED_TICK &&
   state→renegotiation == UNDEFINED_RENEG;

predicate StateAfterServerHelloDone(STATE *st) =
  ∃ STATE *prev, unsigned char *message,
    unsigned int len, int direction;
      isServerHelloDone(message,len,handshake) &&
      st→last_message == S_HD &&
      HaveSameStateValuesButClientAuth_E(st, prev) &&
      MessageAddedToLog_E(st, prev, message, len) &&
      ( (StateAfterServerCertificate(prev) &&
           st→kx == CS_RSA &&
           st→client_auth == NO_AUTH)
      || (StateAfterServerKeyExchange(prev) &&
            (st→kx == DHE || st→kx == ECDHE) &&
            st→client_auth == NO_AUTH)
      || (StateAfterServerCertificateRequest(prev) &&
            (st→kx == DHE || st→kx == ECDHE
              || st→kx = CS_RSA) &&
            st→client_auth == s→client_auth)
      || .... /* other ciphersuites */
      );
```

Figure 6.1: Logical Specification of State Machine (Excerpt)

code out of the box, hence we annotated our code with logical assertions and requirements in Frama-C's specification language, called ACSL.

For example, the logical contract on the inline monitor variant of our state machine is listed in Figure 6.2, embedded within a /*@ ... @*/ comment.

We read this contract bottom-up. The main pre-condition (requires) is that the state must be valid when the function is called (isValidState(st)). (The OpenSSL state SSL is not used by the monitor.) The post-condition (ensures) states that the function either rejects the message or returns a valid state.

```
/*@
 requires \valid(st);
 requires \valid(msg+(0..(len-1)));
 requires \valid(st->log+(0..(st->log_length+len-1)));

 requires \separated(msg+(0..(len-1)),
                     st+(0..(sizeof(st)-1)));
 requires \separated(msg+(0..(len-1)),
                     st->log+(0..(st->log_length + len-1)));
 requires \separated(st+(0..(sizeof(st)-1)),
                     st->log+(0..(st->log_length+len-1)));

 requires isValidState(st)
 ensures (isValidState(st) && \result == ACCEPT)
         || \result == REJECT;
 @*/
int ssl3_next_message(SSL* s, STATE *st,
      unsigned char* msg, int len,
      int direction, unsigned char content_type);
```

Figure 6.2: Logical contract on the inline monitor

That is, isValidState is an invariant for error-free runs.

Moving up, the next block of pre-conditions requires that the areas of memory pointed to by various variables do not intersect. In particular, the given msg, state st, and log st→log, must all be disjoint blocks of memory. This pre-condition is required for verification. In particular, when ssl3_next_message tries to copy msg over to the end of the log, it uses memcpy, which has a logical pre-condition in Frama-C (reflecting its input assumptions) that the two arrays are disjoint.

The first set of pre-conditions require that the pointers given to the function be valid, that is, they must be non-null and lie within validly allocated areas of memory that are owned by the current process. These annotations are required for Frama-C to prove memory safety for our code: that is, all our memory accesses are valid, and that our code does not accidentally overrun buffers or access null-pointers.

From the viewpoint of the code that uses our state machine (the OpenSSL client or server) the preconditions specified here require that the caller provide ssl3_next_message with validly allocated and separated data structures. Otherwise, we cannot give any functional guarantees.

Formal Evaluation Our state machine is written in about 750 lines of code, about 250 lines of which are message processing functions. This is about the same length as the current OpenSSL state machine.

The Frama-C specification is written in a separate file and takes about 460 lines of first-order-logic to describe the state machine. To verify the code, we

ran Frama-C which generates proof obligations for multiple SMT solvers. We used Alt-Ergo to verify some obligations and Z3 for others (the two solvers have different proficiencies). Verifying each function took about 2 minutes, resulting in a total verification time of about 30 minutes.

Technically, to verify the code in a reasonable amount of time, we had to provide many annotations (intermediate lemmas) to each function. The total number of annotations in the file amounts to 900 lines. Adding a single annotation often halves the verification time of a function.

One may question the value of a logical specification that is almost as long as the code being verified (460 lines is all we have to trust). What, besides being declarative, makes it a better specification than the code itself? And at that relative size, how can we be confident that the predicates themselves are not as buggy as the code?

We find our specification and its verification useful in several ways. First, in addition to our state invariant, we also prove memory safety for our code, a mundane but important goal for C programs. Second, our predicates provide an alternative specification of the state machine, and verifying that they agree with the code helped us find bugs, especially regressions due to the addition of new features to the machine. Third, our logical formulation of the state machine allows us to prove theorems about its precision. For example, we can use off-the-shelf interactive proof assistants for deriving more advanced properties.

To illustrate this point, using the Coq proof assistant, we formally establish that the valid logs are unambiguous, that is, equal logs imply equal states:

```
theorem UnambiguousValidity: ∀ STATE *s1, *s2;
  (isValidState(s1) && isValidState(s2)
   && LogEquality(s1,s2))
   ==> HaveSameStateValues_E(s1,s2);
```

This property is a key lemma for proving the security of TLS, inasmuch as the logs (not the states they encode) are authenticated in `Finished` messages at the end of the handshake. Its proof is similar to the one for the unambiguity of the logs in miTLS. However, the Frama-C predicates are more abstract, they better capture what makes the log unambiguous, and they cover a more complete set of ciphersuites.

Frama-C vs F*

This example shows the convenience and capabilities of other verification tools than F*. We could have followed the same approach as in HACL* to verify the state machine implementation, enforcing memory safety and checking C-compilable code against a short and clean version of the automaton. However, while taking that path would have also lead to clean and secure code, the produced C code would not have been as idiomatic as the one obtain with

Frama-C. Since the ACSL annotations are added directly in the comments, it allows for verification of existing code as is. Of course, there are limitations to doing that. In particular, C code which has not been written with verification in mind can prove to be extremely difficult to verify, leading to significant changes to that code, which would defeat the original purpose of using a pure C verification tool. But for this state machine example, in which the complexity of the code is low in terms of algorithmic complexity and data structures, Frama-C is perfectly fine, and thus we can immediately distribute the produced C code to OpenSSL library users for instance. Not only does it interfaces perfectly with the existing codebase, but also it is completely idiomatic and third party developers would have no problem understanding that code.

6.3 Parsing Protocol Messages

For large and complex programs, it is extremely difficult to get a complete coverage of all possible cases with unit tests. For that reason, *fuzzing* techniques are getting more and more attention, and people are trying hard to make them smarter in order to find bugs more quickly, and automatically adapt to black-box software.

And indeed, fuzzers have had quite some success in finding bugs in various kinds of programs [71, 25]. Among those, cryptographic protocols remain a target of choice. For certain kinds of protocol, such as the Transport Layer Security (TLS) protocol, the legacy and retrocompatibility constraints of the protocol have led to very complex messages and state machine structures. As outlined in the previous section, deciding whether messages of certain shapes are valid or not is challenging, as it depends on previously negotiated cryptographic parameters, protocol extensions etc.

In this context, fuzzers will help indeed, but they cannot be called the panacea. They remain heuristic tools, which rely on their ability to learn of course, but also on time, computational resources and luck. Therefore, we advocate the use of formal methods to statically verify the correctness of security critical components of software. *Parsers*, which process uncontrolled — and potentially malicious — data, are particularly suited candidates.

Parsing Parsers are essential components for software which accepts input data from the outside. From the programmer's point of view, the code of the application is trusted: although it may contain bugs, the application has been genuinely designed and implemented to serve its purpose, and typically tested for correctness on legitimate and expected inputs. A risk remains, which may trigger bugs and lead to potential vulnerabilities. It is the control an attacker may be given over the code through the only channel she controls: the input of the program. A proper parser is designed to graciously handle data for any

```
type pinverse_t (#a:Type) (#b:Type) ($f:(a →Tot b)) = b →Tot (result a)

unfold type lemma_inverse_g_f (#a:Type) (#b:Type) ($f:a →Tot b) ($g:b →
Tot (result a)) (x:a) =
  g (f x) == Correct x

unfold type lemma_pinverse_f_g (#a:Type) (#b:Type) (r:b →b →Type) ($f:a →
Tot b) ($g:b →Tot (result a)) (y:b) =
  Correct? (g y) ==> r (f (Correct?._0 (g y))) y
```

Figure 6.3: F* invertible parser type

```
type error = alertDescription * string
type result 'a = Platform.Error.optResult error 'a
```

Figure 6.4: F* error and optional result type examples

input value. But for complex and extensible protocols such as TLS, properly designing such a parser comes with several challenges. One is that the protocol itself is complex. It is difficult to implement the parsing functions so that they properly match the RFC specifications. Another one stems from the fact that those specifications are difficult to implement: common implementations tend to be over tolerant and accept messages which are not completely specification compliant in order to be more widely interoperable. Yet another challenge goes down to the very specification of the protocol. Indeed, the TLS protocol being widely used, it has to adapt to the new constraints of the web, and thus new versions of the protocol are standardized. A typical issue which appears during the standardization of a new version of the protocol, which will interoperate with all prior versions, is that none of the new message formats must present any kind of ambiguity with already existing ones. If that happened, the specifications would not be deterministic, leading to failures or vulnerabilities. Hence, enforcing the correctness of the parsing algorithms, including at the level of the reference textual specifications, is a dire necessity. Formal methods (through an injectivity proof of the parsing for instance) can provide great help to such needs.

Specifying Injective Parsers The approach we propose to verify parsers in F* relies on a systematic typing discipline for the parsing and serializing functions. To be correct, a parsing function needs to be at least injective. Indeed, this injectivity property is necessary to ensure an unambiguous exchange of messages between two parties. A non-injective parser implies that the same message could be interpreted in two different ways, which will inevitably lead to comprehension issues between the parties. The second property a parsing message should verify is that parsing a serialized data structure must return

the same data structure as the original one. It ensures that the messages are properly interpreted: the parsing function should at least be capable of processing the messages the serializing function produces.

The other way is more complex. Indeed, for large and complicated protocols such as TLS, some messages may not be supported by a particular implementation, because they are not mandatory part of the standard, but only extensions. Hence, in practice, protocol implementations will tend not to be able to serialize a parsed message back into exactly the original, because some parts of the original message are not supported, and thus the corresponding pieces of data were discarded. Nonetheless, the F* system allows for *ghost* witnesses, which allow us to account even for those more complicated cases and show that, in the event of properly formatted messages, a well written parser is the inverse of the corresponding serializer.

Figure 6.3 shows the types we use to describe mutually inverse functions. An object of type `pinverse_t #a #b f` is a function which takes a value of type `b` in argument and returns a value of type `Result a`. The `Result` type is shown on figure 6.4. It is an optional type which either contains an error modeled as a descriptor and a string message, or the actual value. Is it accompanied by two additional lemmas: `lemma_inverse_g_f` and `lemma_pinverse_f_g`. The first one specifies that the composition of `g` and `f` always yields a `Correct` result. Transposed to our goal, this means that `g` is the parser, `f` the serializing function, and that since the serializing function always produces a properly formatted result, we are guaranteed to get a result of the `Correct` type when parsing the output of this serializing `f` function.

The second lemma, `lemma_pinverse_f_g` states that, if we take `g` as a parsing and `f` as a serializing function, if the parser yields a `Correct` result (if the input data is well formatted), then the original data and the result of the serialization function on the parsed value from the original data satisfy the `r` relationship. `r` is typically meant to be the equality, but as discussed, it can be a more relaxed version of it. It could, for instance, ignore the unsupported part of the message. That way, one still gets the fact that except for the unsupported part of the protocol, the rest of the message is encoded and processed properly.

Unambiguous datastructures The input messages for the parsing functions are bytestrings. Because bytestrings are extremely simple and unconstrained data structures, protocol specifications typically use a first encoding to encapsulate the length of the underlying messages. It is made so that the parser can decide, based on the specification of the protocol, where to cut the stream of bytes and how to isolate a message as a unit of data. The *ASN.1* standard is typically used to describe how to interpret bytestrings into data structures. Its variants specify how to encode all types of data, either of fixed or variable length. We propose an F* approach to automatically verify the correctness of parsers with regard to the corresponding ASN.1 like specifica-

```
1  (** Transform and concatenate a natural number to bytes *)
2  val vlbytes: lSize:nat →b:bytes{repr_bytes (length b) <= lSize} →
   Tot (r:bytes{length r = lSize + length b})
3  let vlbytes lSize b = bytes_of_int lSize (length b) @| b
4
5  val vlsplit: lSize:nat{lSize <= 4}
6    →vlb:bytes{lSize <= length vlb}
7    →Tot (result (b:(bytes * bytes){
8                          repr_bytes (length (fst b)) <= lSize
9                        /\ Seq.equal vlb (vlbytes lSize (fst b) @| (snd b))}))
10 let vlsplit lSize vlb =
11   let (vl,b) = Platform.Bytes.split vlb lSize in
12   let l = int_of_bytes vl in
13   if l <= length b
14   then Correct(Platform.Bytes.split b l)
15   else Error(AD_decode_error, perror __SOURCE_FILE__ __LINE__ "")
16
17
18 val vlparse: lSize:nat{lSize <= 4} →vlb:bytes{lSize <= length vlb}
19   →Tot (result (b:bytes{repr_bytes (length b) <= lSize /\ Seq.equal vlb (vlbytes lSize b)}))
20 let vlparse lSize vlb =
21   let vl,b = split vlb lSize in
22   if int_of_bytes vl = length b
23   then Correct b
24   else Error(AD_decode_error, perror __SOURCE_FILE__ __LINE__ "")
```

Figure 6.5: F* vlbytes definition and main functions

tion.

To that intent, message parts which have variable length are encoded using vlbytes. The definition of vlbytes is shown in figure 6.5. The "vl" part of the vlbytes name stands for *variable length*. The vlbytes type itself carries a variable length field which contains the length of the piece of data which follows it. Because those variable length data blobs can go up to different values, the length on which the vlbytes encode the length also varies, from 1 bytes (for blobs which length is smaller than 256), up to 4 bytes in our case (for lengths of at most 4294967296). The actual data is then concatenated to its length encoding on n-bytes to form the vlbytes structure.

Because this data structure is so useful and so convenient, we also provide the programmer with two helper functions:

- vlsplit, which takes a vlbytes value and returns a tuple which contains its length, and the actual data;
- and vlparse, which takes a vlbytes value and returns only the actual data.

In both case, of course, in order to properly process the provided data structure, the programmer has to input the expected number of bytes the length of

```
1  (** Lemmas associated to bytes manipulations *)
2  val lemma_vlbytes_len : i:nat →b:bytes{repr_bytes (length b) <= i}
3    →Lemma (ensures (length (vlbytes i b) = i + length b))
4  let lemma_vlbytes_len i b = ()
5
6  val lemma_vlbytes_inj : i:nat
7    →b:bytes{repr_bytes (length b) <= i}
8    →b':bytes{repr_bytes (length b') <= i}
9    →Lemma (requires (Seq.equal (vlbytes i b) (vlbytes i b')))
10          (ensures (b == b'))
11 let lemma_vlbytes_inj i b b' =
12   let l = bytes_of_int i (length b) in
13   Seq.lemma_append_inj l b l b'
14
15 val vlbytes_length_lemma: n:nat →a:bytes{repr_bytes (length a) <= n} →
   b:bytes{repr_bytes (length b) <= n} →
16   Lemma (requires (Seq.equal (Seq.slice (vlbytes n a) 0 n) (Seq.slice (vlbytes n b) 0 n)))
17         (ensures (length a = length b))
18 let vlbytes_length_lemma n a b =
19   let lena = Seq.slice (vlbytes n a) 0 n in
20   let lenb = Seq.slice (vlbytes n b) 0 n in
21   assert(Seq.equal lena (bytes_of_int n (length a)));
22   assert(Seq.equal lenb (bytes_of_int n (length b)));
23   int_of_bytes_of_int n (length a); int_of_bytes_of_int n (length b)
```

Figure 6.6: vlbytes associated lemmas

the underlying data is encoded on, otherwise parsing become impossible. The result is of type Result, which is an optional type: if the provided buffer is too short for the length of the data it supposedly encodes, the parsing function fails.

The encoding of bytes into a vlbytes value is done by the vlbytes function which intuitively acts as a constructor here, while vlsplit and vlparse are instances of projectors.

The data structure has nice properties. Is it easy to retrieve the length of the data and the data itself, provided that the size the length is encoded on is known in advance. Since those properties are often needed for the correctness or the injectivity proofs of the more complex structures formed with vlbytes, those are equipped in our F* setting with dedicated lemmas. Those lemmas (some of them are shown in figure 6.6) are part of the library and can either be instantiated manually in the code, or automatically, using patterns.

F* also shines in such situations because of its *abstraction* feature. Since we consider that the vlbytes are the base components of arbitrary structures from RFC protocol messages, we precisely decide how much information is leaked from the data structure to the SMT solver. This way, proofs get faster and more easily automatable. If needed, the abstraction can be lifted manually by the programmer, who then controls which part of the encoding information is

```
1  (* Parsing function for PreSharedKey *)
2  val parsePreSharedKey: pinverse_t preSharedKeyBytes
3  let parsePreSharedKey b =
4    match vlparse 2 b with
5    | Correct b' →
6      begin
7        match vlparse 2 b with
8        | Correct b'' →(* Client case *)
9          begin
10         if length b >= 2 && length b < 65538 then
11           begin
12           match parseClientPreSharedKey b with
13           | Correct psks →Correct (ClientPreSharedKey psks)
14           | Error z →Error z
15           end
16         else Error(AD_decode_error, perror __SOURCE_FILE__ __LINE__
17                                      "Failed to parse psk")
18         end
19       | Error _ →(* Server case *)
20         begin
21         match parseServerPreSharedKey b with
22         | Correct psk →Correct (ServerPreSharedKey psk)
23         | Error z →Error z
24         end
25     end
26   | Error z →Error(AD_decode_error, perror __SOURCE_FILE__ __LINE__
27                                      "Failed to parse pre shared key")
```

Figure 6.7: Example of parser: the PreSharedKey parsing function

passed to the SMT solver. It lets us differentiate between the concrete data being processed by the parsing and serializing functions, and the *representation* of that data the automated solver works with.

Combining parsers The other reason why our F* verification methodology is a great for writing parsers is the modular structure of the code, which lets it scale easily. Indeed, however complex a protocol message is, it can be broken down into smaller and smaller parts, until we get to the original unit of data.

Therefore, the corresponding parsing function just needs to multiplex between the different *sub*-parsing functions which handle the different blobs of data. The programmer then relies on the already proven properties of those subsequent functions to ensure the correctness of the code. We follow a bottom - top approach, where the library provides functionalities such a vlparse to process the base components. We give an example in Figure 6.7. It represents the parsing of the pre-shared key part of a message. There are actually two different possibilities: either the piece of data is the imbrication of two vlbytes which lengths are encoded on two bytes. It then corresponds to a *client* pre-

shared key, and the corresponding function can be called. Otherwise, it has to be a single vlbytes encoding, in which case it corresponds to a server pre-shared key and again the corresponding function is called. Of course, if it is none of the two, then an error is raised as the message is ill-formatted.

The verification of these functions relies on the injectivity of the underlying parseClientPreSharedKey and parseServerPreSharedKey, but the F* verification system is also able to automatically prove that the two encodings (for the client, and for the server) are mutually distinct and unambiguous.

This lets us ensure by construction that the parsing function is injective, and that the serializing function is the inverse of the parsing function, and the small parsing / serializing functions compose together gracefully in the F* verification system. And because the system is rich, it is not limited or restricted to simple structures: recursive data structures can also be handled just the same way, relying on the fact that length of the data being parsed strictly decreases to ensure termination and correction.

This scales from to core vlbytes handling to full TLS 1.2 message parsing. The use of the F* type system, its lemma triggers and the abstraction mechanism reduce the proof burden on the programmer significantly. As of today this mechanism allows an F* programmer to ensure the correctness of a parser with regard to an F* specification, the in spirit of what we do in HACL*. Potential improvements could consist of extracting the F* specification automatically from textual RFC specification. Going further, we could even generate the parsing code automatically, as it code generation is directed completely from the specification, and contrary to HACL*, parsing code is not performance critical. Even if it were, one could easily write function equivalence proofs between two parts of the parsing functions, and replace on with the other to gain additional performance. The F* modular system has the advantage that, provided that two pieces of code expose the same contract, one can be swapped for the other for free.

Conclusion

This work was heavily focused on formal methods applied to cryptographic primitives, as these are the foundation of all cryptographic software. This does not preclude that F* and the techniques presented in the previous chapters can be applied outside low-level cryptographic code. On the contrary, F*, as a functional language relying on SMT solvers to discharge verification conditions is not *a priori* a good candidate to verify imperative code, which correctness relies on complex non-linear arithmetic properties. However, since even those properties are achievable in F*, it opens to way to a single, unified verification framework based on F* where all the different components can share the specified properties.

Above cryptographic primitives, cryptographic protocols have a wide range

of security critical components and we highlighted in this Chapter how F* could be used to verify some of them, namely cryptographic constructions, state machine implementations and message parsing. Building incrementally, one can used F* to verify entire stacks of software. The Everest project which aims to verify the full HTTPS stack is a good illustration of it, and throughout this work we showed how to contribute to the verification of a number of those stack layers.

Chapter 7

Conclusion

This work presented how the recent advances in formal methods allow us to raise the level of trust in software. It is now feasible to verify large scale applications, with components of various complexities and of different security and performance criticality. With the end goal to make formal methods usable in real world software, we presented throughout this work the several steps which enabled us to implement a full-fledged verified and efficient cryptographic library.

Any formal verification work revolves around a set of tools and languages. Among the different approaches and solutions which exist, we chose the F* verification language for its expressiveness, its proof automation capabilities and the active community around it. Rather than verifying existing large scale projects written without verification in mind, we chose to reimplement everything in F*. The fact that we were quickly able to provide a verified and extensible cryptographic library for elliptic curves validated this choice. Indeed, thanks to the modularity and the abstraction mechanisms of F*, we were able to reuse more than 50% of the code and the proofs in our bignum library, which let us cover specific bignum code for three different curves — Curve25519, Curve448 and P-256 — certainly faster than if we had tried to tackle existing code.

Yet, as vanilla F* extracts to OCaml and we wished to generate reference code in a widely known language, without garbage collector, we proposed to use Low* to implement our library. Being a shallow embedding of C in F*, Low* benefits from a compilation correctness proof which ensures functional correctness and memory safety to the generated code. Furthermore, the compilation process guaranties secret-independence for the security sensitive inputs. Low* allowed us to effectively write C code while using the full F* verification system to prove security and safety properties about that code. A specific emphasis was made on ensuring that the resulting code was clean and human readable, so that people willing to incorporate Low* generated code into their projects could do so without having to trust the compiler.

We validated our code generation and compilation to C approach by implementing HACL*. HACL* is a standalone, full-fledged and fully verified cryptographic library. We chose C code has a backend as it provides a great compromise between performance and portability, and is the most widely used language for cryptographic software reference implementations. Originally inspired from the reference C implementations of the cryptographic primitives, HACL* compiled and verified code performs as fast if not faster than the originals. The code of several primitives of HACL* has readily been integrated into NSS, Mozilla's cryptographic library and is now used by millions of users of the Firefox web browsers. In addition to the integration of the verification and compilation process of HACL* into Mozilla's continuous integration toolchain, the fact that the generated code was readable and thus audited by Mozilla's developers helped a lot in their adoption of the code.

Yet, our methodology does not only apply to cryptographic code. We started from cryptographic primitives because they are notoriously hard to implement properly, have good mathematical specifications and constitute the foundations of any cryptographic software. However, many other critical components can benefit from this technology. We illustrated that with cryptographic constructions, state-machines and message parsing. Still, because a language such as F* is so expressive, the programmer is free to model, implement and verify almost anything. Furthermore, F*'s module system allows for a clean separation of components, sharing specifications through a common interface, but isolating the proof and code complexity.

Off course, a lot remains to be done in order to make formal methods more widespread. If the proof is, in contrast with other solutions, a strength of F*, a lot remains to be done in that area, with two main paths to explore. The first one is to rely more heavily on F*'s normalizer for the proofs. SMT solvers are particularly bad at non-linear arithmetic. Hence, it would make sense to discharge such verification conditions to another backend. In future work, we should add symbolic execution capabilities to the F* normalizer in order to discharge certain assertions without querying the solver. Other possible proof backends include computer algebraic systems such as SAGE, or proof-assistants such as Coq or Isabelle/HOL. The second automation weakness in our approach concerns memory safety. The HACL* library is entirely stack-based, and the memory management is quite simple. Yet, proving memory safety in a Low* setting still requires significant proof and annotation effort from the programmer. In future work, we will work at automating it better, with the end goal to reach a level of automation such as the one of Rust. We will also focus on reducing the trusted computing base in F*. The F* compiler and the Low* compiler could be verified with a mechanized proof, while they only have pencil and paper proofs for now. From the SMT solver, we will extract proof certificates which can be processed by third party verified tools to ensure that, although the SMT solver itself is not trusted, the proofs produced are indeed correct.

Overall, although a lot remains to be done, we have made a step towards "industrializing" formal methods. The verification projects are getting larger and larger, the Everest Project which aims to verify the full HTTPS stack is one example, and knowing that leading companies such as Google or Mozilla are not only interested but also integrating formally verified code into their products is great news. The process can only become faster as we improve the tooling around formal methods.

Bibliography

[1] The Rust programming language, 2010–2017. URL `https://www.rust-lang.org`. 16

[2] ChaCha20 and Poly1305 for IETF Protocols. IETF RFC 7539, 2015. 128

[3] Elliptic Curves for Security. IETF RFC 7748, 2016. 10, 111, 124

[4] Edwards-Curve Digital Signature Algorithm (EdDSA) . IETF RFC 8032, 2017. 125

[5] The Transport Layer Security (TLS) Protocol Version 1.3. IETF Internet Draft 20, 2017. 128

[6] Andreas Abel. foetus – termination checker for simple functional programs. Programming Lab Report 474, LMU München, 1998. 14

[7] Andreas Abel. *Type-based termination: a polymorphic lambda-calculus with sized higher-order types*. PhD thesis, LMU München, 2007. 14

[8] Danel Ahman, Cédric Fournet, Cătălin Hriţcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. Recalling a witness: foundations and applications of monotonic state. *Proceedings of the ACM on Programming Languages*, 2(POPL):65, 2017. 14

[9] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 515–529. ACM, January 2017. doi: 10.1145/3009837.3009878. URL `https://www.fstar-lang.org/papers/dm4free/`. 14, 22, 35, 95, 96

[10] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC. *IACR Cryptology ePrint Archive*, 2015: 1241, 2015. URL `http://eprint.iacr.org/2015/1241`. 17

[11] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium, USENIX Security 16*, pages 53–70, 2016. 17, 103

[12] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pereira Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. 2017. 17

[13] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, et al. COGENT: Verifying high-assurance file system implementations. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–188. ACM, 2016. 16

[14] Nada Amin and Tiark Rompf. LMS-Verify: Abstraction without regret for verified systems programming. To appear in 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'17), 2017. URL `https://www.cs.purdue.edu/homes/rompf/papers/amin-draft2016b.pdf`. 16

[15] Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7, 2015. 15

[16] Andrew W Appel. Verification of a cryptographic primitive: Sha-256. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(2):7, 2015. 16

[17] Lennart Augustsson. Cayenne—a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 239–250, 1998. 14

[18] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA, May 1997. URL `https://hal.inria.fr/inria-00069968`. Projet COQ. 20

[19] Gilles Barthe, Maria João Frade, E. Giménez, Luis Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004. doi: 10.1017/S0960129503004122. URL `http://dx.doi.org/10.1017/S0960129503004122`. 14

[20] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013. ISBN 978-3-319-10081-4. doi: 10.1007/978-3-319-10082-1_6. URL `http://dx.doi.org/10.1007/978-3-319-10082-1_6`. 17

[21] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *2014 ACM SIGSAC Conference on Computer and Communications Security, CCS 2014*, pages 1267–1279, 2014. 17, 97, 103

[22] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *41st Annual ACM Symposium on Principles of Programming Languages, POPL 2014*, pages 193–206, 2014. 138

[23] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 457–485. Springer, 2015. 17

[24] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 116–129. ACM, 2016. 17

[25] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. Finding software vulnerabilities by smart fuzzing. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 427–430. IEEE, 2011. 157

[26] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology – EUROCRYPT 2006*, pages 409–426, 2006. 139, 142

[27] David Benjamin. poly1305-x86.pl produces incorrect output. `https://mta.openssl.org/pipermail/openssl-dev/2016-March/006161`, 2016. 124

[28] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2002*, pages 1–12. ACM, 2002. 87

[29] Lennart Beringer, Adam Petcher, Q Ye Katherine, and Andrew W Appel. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 207–221, 2015. 15

[30] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of openssl HMAC. In *USENIX Security Symposium*, pages 207–221, 2015. 16

[31] Daniel J. Bernstein. The Poly1305-AES message-authentication code. In *12th International Workshopo on Fast Software Encryption, FSE 2005*, pages 32–49, 2005. 145, 146

[32] Daniel J Bernstein. Cache-timing attacks on aes, 2005. 84

[33] Daniel J. Bernstein. The poly1305-aes message-authentication code. In *Fast Software Encryption (FSE)*, pages 32–49, 2005. 59

[34] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *Public Key Cryptography-PKC 2006*, pages 207–228. Springer, 2006. 9, 57, 111, 124

[35] Daniel J Bernstein. ChaCha, a variant of Salsa20. 2008. 128

[36] Daniel J Bernstein and Peter Schwabe. NEON crypto. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 320–339. Springer, 2012. 129

[37] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, pages 1–13. 125

[38] Daniel J Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. Tweetnacl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 64–83. Springer, 2014. 124

[39] 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 *Security and Privacy (SP),*

2015 IEEE Symposium on, pages 535–552. IEEE, 2015. 84, 137, 138, 151, 153

[40] Karthikeyan Bhargavan, Cédric Fournet, Andrew D Gordon, and Riccardo Pucella. Tulafale: A security tool for web services. In *International Symposium on Formal Methods for Components and Objects*, pages 197–222. Springer, 2003. 17

[41] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Modular verification of security protocol code by typing. In *37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*, pages 445–456, 2010. 21

[42] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing tls with verified cryptographic security. In *IEEE Symposium on Security & Privacy (Oakland)*, pages 445–462, 2013. 16, 17, 21, 150, 153

[43] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella-Béguelin. Proving the TLS handshake secure (as it is). Cryptology ePrint Archive, Report 2014/182, 2014. `http://eprint.iacr.org/2014/182/`. 139

[44] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella-Béguelin. Proving the tls handshake secure (as it is). Cryptology ePrint Archive, Report 2014/182, 2014. `https://eprint.iacr.org/2014/182`. 17, 21

[45] Karthikeyan Bhargavan, Antoine Delignat-Lavaud Cédric Fournet, Markulf Kohlweiss, Jianyang Pan, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, and Santiago Zanella-Béguelin andJean Karim Zinzindohoué. Implementing and proving the tls 1.3 record layer. Cryptology ePrint Archive, Report 2016/1178, IEEE S&P 2017, 2016. URL `https://eprint.iacr.org/2016/1178`. 21, 137, 138, 142, 144, 149

[46] Bruno Blanchet. Cryptoverif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied*, page 117, 2007. 17

[47] Bruno Blanchet. Security protocol verification: Symbolic and computational models. In *Proceedings of the First international conference on Principles of Security and Trust*, pages 3–29. Springer-Verlag, 2012. 17

[48] Bruno Blanchet, V Cheval, X Allamigeon, and B Smyth. Proverif: Cryptographic protocol verifier in the formal model. *URL http://prosecco. gforge. inria. fr/personal/bblanche/proverif*, 2010. 17

[49] Bruno Blanchet et al. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016. 17

[50] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. 101

[51] Hanno Böck. Wrong results with Poly1305 functions. `https://mta.openssl.org/pipermail/openssl-dev/2016-March/006413`, 2016. 124

[52] 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*, August 2017. 17, 55, 123

[53] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 133–144, 2013. URL `https://eb.host.cs.st-andrews.ac.uk/drafts/effects.pdf`. 14

[54] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 33–46, 2014. URL `http://repository.upenn.edu/cgi/viewcontent.cgi?article=2031&context=cis_reports`. 14

[55] Chiyan Chen and Hongwei Xi. Combining Programming with Theorem Proving. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 66–77, Tallinn, Estonia, September 2005. 14

[56] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible OS kernels and device drivers. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*, pages 431–447, 2016. 15

[57] 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 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 299–309. ACM, 2014. 16, 123, 125

[58] Adam Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *ACM SIGPLAN Notices*, volume 48, pages 391–402. ACM, 2013. 16

[59] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 79–90, 2009. URL `http://ynot.cs.harvard.edu/papers/icfp09.pdf`. 14

[60] 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 *International Conference on Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009. 15

[61] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C Necula. Dependent types for low-level programming. In *European Symposium on Programming*, pages 520–535. Springer, 2007. 16

[62] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *Journal of Automated Reasoning*, 46(3-4):225–259, 2011. 17

[63] Véronique Cortier, Steve Kremer, et al. Formal models and techniques for analyzing security protocols: A tutorial. *Foundations and Trends® in Programming Languages*, 1(3):151–267, 2014. 17

[64] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *International Conference on Software Engineering and Formal Methods*, pages 233–247. Springer, 2012. 153

[65] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975. ISSN 0001-0782. doi: 10.1145/360933.360975. URL `http://doi.acm.org/10.1145/360933.360975`. 22

[66] Robert W. Dockins. *Operational Refinement for Compiler Correctness*. PhD thesis, Princeton University, 2012. 102

[67] François Dupressoir, Andrew D Gordon, Jan Jürjens, and David A Naumann. Guiding a general-purpose c verifier to prove cryptographic protocols. *Journal of Computer Security*, 22(5):823–866, 2014. 17

[68] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Systematic synthesis of elliptic curve cryptography implementations. 17

[69] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013. URL `https://hal.inria.fr/hal-00789533/document`. 15

[70] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *18th ACM Conference on Computer and Communications Security, CCS 2011*, pages 341–350, 2011. 138

[71] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008. 157

[72] Martin Goll and Shay Gueron. Vectorization on chacha stream cipher. In *Information Technology: New Generations (ITNG)*, pages 612–615, 2014. 129

[73] Anthony Green. The libffi home page. 2014. URL `http://sourceware.org/libffi`. 106

[74] David Greenaway, June Andronick, and Gerwin Klein. Bridging the gap: Automatic verified abstraction of C. In *3rd International Conference on Interactive Theorem Proving, ITP 2012*, volume 7406 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2012. 15

[75] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: formal verification of C code without the pain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014*, pages 429–439. ACM, 2014. 15

[76] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, 2014. 16, 53

[77] Martín Abadi (invited speaker), Bruno Blanchet, and Hubert Comon-Lundh. Models and proofs of protocol security: A progress report. In Ahmed Bouajjani and Oded Maler, editors, *21st International Conference on Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 35–49, Grenoble, France, June 2009. Springer. 17

[78] Bart Jacobs, Jan Smans, and Frank Piessens. The VeriFast program verifier: A tutorial. iMinds-DistriNet, Department of Computer Science, KU Leuven - University of Leuven, Belgium, 2014. URL `https://people.cs.kuleuven.be/~bart.jacobs/verifast/tutorial.pdf`. 15

[79] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002. 16

[80] Cameron F. Kerry, Acting Secretary, and Charles Romine Director. Fips pub 186-4 federal information processing standards publication digital signature standard (dss), 2013. 57

[81] Garrin Kimmell, Aaron Stump, Harley D. Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins, and Ki Yung Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. *Progress in Informatics*, 2013. URL `http://www.tyconmismatch.com/papers/pi13.pdf`. 14

[82] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015. 15

[83] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009. 15

[84] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009. 53

[85] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009. 16

[86] Alexander Krauss, Christian Sternagel, René Thiemann, Carsten Fuhs, and Jürgen Giesl. Termination of Isabelle functions via termination of rewriting. In *Second International Conference on Interactive Theorem Proving*, volume 6898 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2011. ISBN 978-3-642-22862-9. doi: 10.1007/

978-3-642-22863-6_13. URL http://verify.rwth-aachen.de/giesl/papers/ITP2011-distribute.pdf. 15

[87] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. *SIGPLAN Not.*, 12 (2):1–79, February 1977. ISSN 0362-1340. doi: 10.1145/954666.971189. URL http://doi.acm.org/10.1145/954666.971189. 15

[88] Adam Langley. ctgrind : Checking that functions are constant time with valgrind, 2017. URL https://github.com/agl/ctgrind. 84

[89] Nate Lawson. Side-channel attacks on cryptographic software. *IEEE Security & Privacy*, 7(6), 2009. 82

[90] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-17510-4, 978-3-642-17510-7. URL http://dl.acm.org/citation.cfm?id=1939141.1939161. 15

[91] Xavier Leroy. The CompCert C verified compiler. http://compcert.inria.fr/, 2004–2016. 101

[92] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. 16, 90

[93] Xavier Leroy et al. The compcert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012. 53

[94] Pierre Letouzey. Extraction in Coq: An overview. In *4th Conference on Computability in Europe*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2008. ISBN 978-3-540-69405-2. doi: 10.1007/978-3-540-69407-6_39. URL http://www.pps.univ-paris-diderot.fr/~letouzey/download/letouzey_extr_cie08.pdf. 103

[95] Nicholas D Matsakis and Felix S Klock II. The Rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014. 12, 16

[96] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962. 86

[97] Catherine A Meadows. Formal verification of cryptographic protocols: A survey. In *International Conference on the Theory and Application of Cryptology*, pages 133–150. Springer, 1994. 17

[98] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *International Conference on Computer Aided Verification*, pages 696–701. Springer, 2013. 17

[99] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *8th International Conference on Information Security and Cryptology, ICISC 2005*, pages 156–168. Springer, 2006. 97

[100] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6): 865–911, 2008. URL `http://ynot.cs.harvard.edu/papers/jfpsep07.pdf`. 14

[101] Erick Nascimento, Łukasz Chmielewski, David Oswald, and Peter Schwabe. Attacking embedded ECC implementations through cmov side channels. In *Selected Areas in Cryptology – SAC 2016*, Lecture Notes in Computer Science, 2016. 126

[102] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002. 15

[103] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007. 14

[104] Liam O'Connor, Christine Rizkallah, Zilin Chen, Sidney Amani, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Alex Hixon, Gabriele Keller, Toby Murray, et al. Cogent: certified compilation for a functional systems language. *arXiv preprint arXiv:1601.05520*, 2016. 16

[105] Colin Percival. Cache missing for fun and profit, 2005. 84

[106] Davide Pozza, Riccardo Sisto, and Luca Durante. Spi2java: Automatic cryptographic protocol java code generation from spi calculus. In *Advanced Information Networking and Applications, 2004. AINA 2004. 18th International Conference on*, volume 1, pages 400–405. IEEE, 2004. 17

[107] 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*. *PACMPL*, 1(ICFP):17:1–17:29, September 2017. doi: 10.1145/3110261. URL `http://arxiv.org/abs/1703.00053`. 81, 90

[108] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, PLDI '08, pages 159–169, 2008. 14

[109] John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24:709–720, 1998. 26

[110] Norbert Schirmer. *Verification of sequential imperative programs in Isabelle-HOL*. PhD thesis, Technical University Munich, 2006. 15

[111] Jeff Seibert, Hamed Okhravi, and Eric Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 54–65. ACM, 2014. 82

[112] Matthieu Sozeau. Subset Coercions in Coq. In Thorsten Altenkirch and Conor McBride, editors, *TYPES'06*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2007. doi: http://dx.doi.org/10.1007/978-3-540-74464-1_16. 26

[113] Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing stateful authorization and information flow policies in Fine. In *In Proceedings of the European Symposium on Programming (ESOP)*, 2010. 21

[114] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013. URL `http://prosecco.gforge.inria.fr/personal/karthik/pubs/secure_distributed_programming_fstar_jfp15.pdf`. 14

[115] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the Dijkstra monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, pages 387–398, 2013. URL `http://research-srv.microsoft.com/pubs/189686/paper-pldi13.pdf`. 14, 22, 35

[116] Nikhil Swamy, Cătălin Hriţcu, 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 *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, 2016. URL `https://www.fstar-lang.org/papers/mumon/`. 14, 19, 22, 35, 87

[117] David Tarditi. Extending C with bounds safety. Checked C Technical Report, Version 0.6, November 2016. URL `https://github.com/Microsoft/checkedc`. 16

[118] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and computation*, 132(2):109–176, 1997. 86

[119] A. Tomb. Automated verification of real-world cryptographic implementations. *IEEE Security Privacy*, 14(6):26–33, 2016. 17, 55

[120] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010. 84

[121] 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 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1973–1987. ACM, 2017. 17

[122] Mark Utting. Reasoning about aliasing. In *The Fourth Australasian Refinement Workshop*, pages 195–211, 1996. 15

[123] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming (ICFP'14)*, pages 269–282, 2014. 14

[124] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. Mind the gap. In *22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 500–515. Springer, 2009. 15

[125] 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 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2007–2020, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3133974. URL `http://doi.acm.org/10.1145/3133956.3133974`. 16

[126] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 427–440, 2012. 103

[127] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of ssa-based optimizations for LLVM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 175–186, 2013. 103

[128] YongBin Zhou and DengGuo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptology ePrint Archive*, 2005:388, 2005. 84

[129] Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. A verified extensible library of elliptic curves. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 296–309, 2016. 16, 53, 123, 125

[130] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1789–1806, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3134043. URL `http://doi.acm.org/10.1145/3133956.3134043`. 107

Appendix A : Bignum code excerpts

Listing 1: F* Modulo module interface, to be implemented per prime

```
let op_Bar_Amp x y = log_and_limb x y
let op_Bar_Greater_Greater x y = shift_left_wide x y
let op_Bar_Plus x y = add_wide x y
let op_Bar_Star x y = mul_wide x y

(* Set of constraints to satisfy, necessary to call the 'freduce_degree' and 'carry'
     functions
   consecutively *)
val satisfies_modulo_constraints: h:FStar.Heap.heap →b:bigint_wide{live h b} →GTot bool

val freduce_degree: b:bigint_wide →ST unit
  (requires (fun h →live h b /\ satisfies_modulo_constraints h b))
  (ensures (fun h0 _ h1 →live h0 b /\ live h1 b /\ satisfies_modulo_constraints h0 b
    /\ getLength h0 b >= 2*norm_length - 1
    /\ getLength h1 b = getLength h0 b /\ modifies (getRef b) h0 h1 /\ getLength h1 b >=
         norm_length+1
    /\ (forall (i:nat). {:pattern (v (getValue h1 b i))} i <= norm_length ==>
        v (getValue h1 b i) < pow2 (platform_wide - 1))
    /\ eval h1 b norm_length % reveal prime = eval h0 b (2*norm_length-1) % reveal prime
        ))

val freduce_coefficients: b:bigint_wide{getTemplate b = templ} →ST unit
  (requires (fun h →live h b /\ getLength h b >= 2*norm_length-1 /\ getLength h b >=
       norm_length + 1
     /\ (forall (i:nat). {:pattern (v (getValue h b i))} i <= norm_length ==>
         v (getValue h b i) < pow2 (platform_wide -1))))
  (ensures (fun h0 _ h1 →live h0 b /\ getLength h0 b >= 2*norm_length-1
    /\ Normalized h1 b /\ modifies (getRef b) h0 h1
    /\ eval h1 b norm_length % reveal prime = eval h0 b norm_length % reveal prime))

val carry:
    b:bigint_wide{getTemplate b = templ} →ctr:nat →ST unit
    (requires (fun h →live h b /\ getLength h b >= norm_length+1
      /\ (forall (i:nat). {:pattern (v (getValue h b i))} i <= norm_length ==>
          v (getValue h b i) < pow2 (platform_wide - 1)) ))
    (ensures (fun h0 _ h1 →live h0 b /\ getLength h0 b >= norm_length +1
      /\ Normalized h1 b /\ modifies (getRef b) h0 h1 /\ getLength h1 b = getLength h0 b
      /\ eval h1 b (norm_length+1) % reveal prime = eval h0 b (norm_length+1) % reveal
          prime))

val carry_top_to_0:
  b:bigint_wide →ST unit (requires (fun h →True)) (ensures (fun h0 _ h1 →True))

val normalize:
  output:bigint →ST unit
  (requires (fun h →Normalized h output))
  (ensures (fun h0 _ h1 →Normalized h0 output /\ Normalized h1 output
    /\ eval h0 output norm_length % reveal prime = eval h1 output norm_length % reveal
        prime
    /\ eval h1 output norm_length < reveal prime
```

```
45     /\ modifies (getRef output) h0 h1))
46
47 opaque type Larger (h:FStar.Heap.heap) (b:bigint) (n:pos) =
48   live h b /\ getLength h b >= norm_length /\ getTemplate b = templ
49   /\ (forall (i:nat). {:pattern (v (getValue h b i))} i < norm_length ==>
50       (v (getValue h b i) < pow2 n) /\ v (getValue h b i) >= pow2 (getTemplate b i))
51
52 (* Same as in parameters.fsti *)
53 assume val n1: n:pos{n = Parameters.ndiff /\ n <= platform_size}
54 assume val n0: n:pos{n = Parameters.ndiff'}
55
56 val add_big_zero:
57   output:bigint →ST unit
58     (requires (fun h →Normalized h output))
59     (ensures (fun h0 _ h1 →Normalized h0 output
60       /\ Filled h1 output n0 n1
61       /\ eval h0 output norm_length % reveal prime = eval h1 output
             norm_length % reveal prime
62       /\ modifies (getRef output) h0 h1))
63
64 val sum_satisfies_constraints: h0:heap →h1:heap →cpy:bigint_wide{getTemplate
       cpy = templ} →a:bigint →b:bigint →
65   Lemma
66     (requires (Normalized h0 a /\ Normalized h0 b /\ live h1 cpy /\ getLength
         h1 cpy >= 2*norm_length-1
67               /\ (forall (i:nat). i < norm_length ==> v (getValue h1 cpy i) =
                     v (getValue h0 a i)
68                                                     + v (getValue h0 b i))
69               /\ (forall (i:nat). (i >= norm_length /\ i < getLength h1 cpy)
                     ==>
70                   v (getValue h1 cpy i) = 0)))
71     (ensures (live h1 cpy /\ satisfies_modulo_constraints h1 cpy))
72
73 val difference_satisfies_constraints: h0:heap →h1:heap →cpy:bigint_wide{
       getTemplate cpy = templ} →a:bigint →b:bigint →
74   Lemma
75     (requires (Filled h0 a n0 n1 /\ Normalized h0 b /\ live h1 cpy
76       /\ getLength h1 cpy >= 2*norm_length-1
77       /\ (forall (i:nat). i < norm_length ==> v (getValue h1 cpy i) = v (
             getValue h0 a i) - v (getValue h0 b i))
78       /\ (forall (i:nat). (i >= norm_length /\ i < getLength h1 cpy) ==> v (
             getValue h1 cpy i) = 0) ))
79     (ensures (live h1 cpy /\ satisfies_modulo_constraints h1 cpy))
80
81 val mul_satisfies_constraints: h0:heap →h1:heap →cpy:bigint_wide{getTemplate
       cpy = templ} →a:bigint →b:bigint →
82   Lemma
83     (requires (Normalized h0 a /\ Normalized h0 b /\ live h1 cpy /\ getLength
         h1 cpy >= 2*norm_length-1
84       /\ maxValue h1 cpy <= norm_length * maxValueNorm h0 a * maxValueNorm h0
             b))
85     (ensures (live h1 cpy /\ satisfies_modulo_constraints h1 cpy))
```

Listing 2: Top-level generic bignum API

```
val fsum:
  a:bigint{getTemplate a = templ} →b:bigint{Similar a b} →
  ST unit
    (requires (fun h →(Norm h a) /\ (Norm h b) ))
    (ensures (fun h0 _ h1 →
      (Norm h0 a) /\ (Norm h1 a) /\ (Norm h0 b)
      /\ (valueOf h1 a = (valueOf h0 a ^+ valueOf h0 b))
      /\ (modifies (getRef a) h0 h1) ))

val fdifference:
  a:bigint{getTemplate a = templ} →b:bigint{Similar a b} →
  ST unit
    (requires (fun h →(Norm h a) /\ (Norm h b)))
    (ensures (fun h0 _ h1 →
      (Norm h0 a) /\ (Norm h0 b) /\ (Norm h1 a)
      /\ (valueOf h1 a = (valueOf h0 b ^- valueOf h0 a))
      /\ (modifies (getRef a) h0 h1)
    ))

val fscalar:
  res:bigint{getTemplate res = templ} →b:bigint{Similar res b} →#n:nat{n <= ndiff'} →s:
      limb{bitsize (v s) n} →ST unit
  (requires (fun h →(Live h res) /\ (Norm h b)))
  (ensures (fun h0 _ h1 →
    (Norm h0 b) /\ (Live h0 b) /\ (Norm h1 res)
    /\ (valueOf h1 res = (v s +* valueOf h0 b))
    /\ (modifies (getRef res) h0 h1)
  ))

val fmul:
  res:bigint{getTemplate res = templ} →
  a:bigint{Similar res a} →
  b:bigint{Similar res b} →
  ST unit
    (requires (fun h →(Live h res) /\ (Norm h a) /\ (Norm h b)))
    (ensures (fun h0 _ h1 →
      (Norm h0 a) /\ (Norm h0 b) /\ (Norm h1 res)
      /\ (valueOf h1 res = (valueOf h0 a ^* valueOf h0 b))
      /\ (modifies (getRef res) h0 h1)
    ))

val fsquare:
  res:bigint →a:bigint{Similar res a} →
  ST unit
    (requires (fun h →(Live h res) /\ (Norm h a)))
    (ensures (fun h0 _ h1 →
      (Norm h0 a) /\ (Live h0 res) /\ (Norm h1 res)
      /\ (valueOf h1 res = (valueOf h0 a ^* valueOf h0 a))
      /\ (modifies (getRef res) h0 h1)
    ))
```

Appendix B : HACL* specification and code

Listing 3: Curve25519 F* specification

```
(* Field types and parameters *)
let prime = pow2 255 - 19
type elem : Type0 = e:int{e >= 0 /\ e < prime}
let fadd e1 e2 = (e1 + e2) % prime
let fsub e1 e2 = (e1 - e2) % prime
let fmul e1 e2 = (e1 * e2) % prime
let zero : elem = 0
let one : elem = 1
let ( +@ ) = fadd
let ( *@ ) = fmul
(** Exponentiation *)
let rec ( ** ) (e:elem) (n:pos) : Tot elem (decreases n) =
  if n = 1 then e
  else
    if n % 2 = 0 then op_Star_Star (e `fmul` e) (n / 2)
    else e `fmul` (op_Star_Star (e `fmul` e) ((n-1)/2))

(* Type aliases *)
type scalar = lbytes 32
type serialized_point = lbytes 32
type proj_point = | Proj: x:elem ->z:elem ->proj_point

let decodeScalar25519 (k:scalar) =
  let k = k.[0] <- (k.[0] &^ 248uy) in
  let k = k.[31] <- ((k.[31] &^ 127uy) |^ 64uy) in k

let decodePoint (u:serialized_point) =
  (little_endian u % pow2 255) % prime

let add_and_double qx nq nqp1 =
  let x_1 = qx in
  let x_2, z_2 = nq.x, nq.z in
  let x_3, z_3 = nqp1.x, nqp1.z in
  let a = x_2 `fadd` z_2 in
  let aa = a**2 in
  let b = x_2 `fsub` z_2 in
  let bb = b**2 in
  let e = aa `fsub` bb in
  let c = x_3 `fadd` z_3 in
  let d = x_3 `fsub` z_3 in
  let da = d `fmul` a in
  let cb = c `fmul` b in
  let x_3 = (da `fadd` cb)**2 in
  let z_3 = x_1 `fmul` ((da `fsub` cb)**2) in
  let x_2 = aa `fmul` bb in
```

```
  let z_2 = e `fmul` (aa `fadd` (121665 `fmul` e)) in
  Proj x_2 z_2, Proj x_3 z_3

let ith_bit (k:scalar) (i:nat{i < 256}) =
  let q = i / 8 in let r = i % 8 in
  (v (k.[q]) / pow2 r) % 2

let rec montgomery_ladder_ (init:elem) x xp1 (k:scalar) (ctr:nat{ctr<=256})
  : Tot proj_point (decreases ctr) =
  if ctr = 0 then x
  else (
    let ctr' = ctr - 1 in
    let (x', xp1') =
      if ith_bit k ctr' = 1 then (
        let nqp2, nqp1 = add_and_double init xp1 x in
        nqp1, nqp2
      ) else add_and_double init x xp1 in
    montgomery_ladder_ init x' xp1' k ctr'
  )

let montgomery_ladder (init:elem) (k:scalar) : Tot proj_point =
  montgomery_ladder_ init (Proj one zero) (Proj init one) k 256

let encodePoint (p:proj_point) : Tot serialized_point =
  let p = p.x `fmul` (p.z ** (prime - 2)) in
  little_bytes 32ul p

let scalarmult (k:scalar) (u:serialized_point) : Tot serialized_point =
  let k = decodeScalar25519 k in
  let u = decodePoint u in
  let res = montgomery_ladder u k in
  encodePoint res
```

Listing 4: Curve25519 RFC extracts

```
[...] The "X25519" and "X448" functions perform scalar multiplication on
the Montgomery form of the above curves. (This is used when
implementing Diffie-Hellman.) The functions take a scalar and a
u-coordinate as inputs and produce a u-coordinate as output.
Although the functions work internally with integers, the inputs and
outputs are 32-byte strings (for X25519) or 56-byte strings (for
X448) and this specification defines their encoding.

def decodeLittleEndian(b, bits):
    return sum([b[i] << 8*i for i in range((bits+7)/8)])

def decodeUCoordinate(u, bits):
    u_list = [ord(b) for b in u]
    # Ignore any unused bits.
    if bits % 8:
        u_list[-1] &= (1<<(bits%8))-1
```

```
    return decodeLittleEndian(u_list, bits)

def encodeUCoordinate(u, bits):
    u = u % p
    return ''.join([chr((u >> 8*i) & 0xff)
                    for i in range((bits+7)/8)])
def decodeScalar25519(k):
    k_list = [ord(b) for b in k]
    k_list[0] &= 248
    k_list[31] &= 127
    k_list[31] |= 64
    return decodeLittleEndian(k_list, 255)

x_1 = u
x_2 = 1
z_2 = 0
x_3 = u
z_3 = 1
swap = 0

For t = bits-1 down to 0:
    k_t = (k >> t) & 1
    swap ^= k_t
    // Conditional swap; see text below.
    (x_2, x_3) = cswap(swap, x_2, x_3)
    (z_2, z_3) = cswap(swap, z_2, z_3)
    swap = k_t

    A = x_2 + z_2
    AA = A^2
    B = x_2 - z_2
    BB = B^2
    E = AA - BB
    C = x_3 + z_3
    D = x_3 - z_3
    DA = D * A
    CB = C * B
    x_3 = (DA + CB)^2
    z_3 = x_1 * (DA - CB)^2
    x_2 = AA * BB
    z_2 = E * (AA + a24 * E)

// Conditional swap; see text below.
(x_2, x_3) = cswap(swap, x_2, x_3)
(z_2, z_3) = cswap(swap, z_2, z_3)
Return x_2 * (z_2^(p - 2))

cswap(swap, x_2, x_3):
      dummy = mask(swap) AND (x_2 XOR x_3)
      x_2 = x_2 XOR dummy
      x_3 = x_3 XOR dummy
      Return (x_2, x_3)
```

Appendix C : HACL* Performance Benchmarks

Algorithm	Implementation	Language	Architecture	Cycles
ChaCha20	moon/avx2/64	assembly	AVX2	1908
	dolbeau/amd64-avx2	C	AVX2	2000
	goll_guerin	C	AVX2	2224
	krovetz/avx2	C	AVX2	2500
	moon/avx/64	assembly	AVX	3584
	moon/ssse3/64	assembly	SSSE3	3644
	krovetz/vec128	C	SSSE3	4340
	hacl-star/vec128	C	SSSE3	4364
	moon/sse2/64	assembly	SSE2	4528
	e/amd64-xmm6	assembly	SSE	4896
	e/x86-xmm6	assembly	SSE	5656
	hacl-star/ref	C	x86_64	9248
	e/amd64-3	assembly	x86_64	9280
	e/ref	C	x86	9596
Poly1305	moon/avx2/64	assembly	AVX2	2508
	moon/avx/64	assembly	AVX	4052
	moon/sse2/64	assembly	SSE2	4232
	hacl-star	C	x86_64	5936
	amd64	assembly	x86_64	8128
	x86	assembly	x86	8160
	53	C	x86	11356
	avx	assembly	AVX	13480
	ref	C	x86	111212
Curve25519	amd-64-64	assembly	x86_64	580132
	sandy2x	assembly	AVX	595272
	amd-64-51	assembly	x86_64	617244
	hacl-star	C	x86_64	632544
	donna_c64	C	x86_64	635620
	donna	assembly	x86	1026040
	ref10	C	x86	1453308
	athlon	assembly	x86	1645992
	ref	C	x86	17169436
SHA-512	openssl	assembly	x86	9028
	ref	C	x86	12620
	sphlib	C	x86	13396
	hacl-star	C	x86	15844
Ed25519	amd64-64-24k	assembly	x86_64	235932
	ref10	C	x86	580232
	hacl-star	C	x86_64	1353932
	ref	C	x86	5234724

Table .1: Intel64 SUPERCOP Benchmarks: ranked list of best performing implementations on an Intel(R) Xeon(R) CPU E5-1630 v4 @ 3.70GHz running 64-bit Debian Linux 4.8.15. All numbers are estimated CPU cycles. Curve25519 is measured for two variable-base and two fixed-base scalar multiplications. All other primitives are measured for an input of 1536 bytes: Chacha20 is measured for a single encryption; Poly1305 is measured for one MAC plus one verify; SHA-512 is measured for a single hash computation; Ed25519 is measured for one sign plus one verify.

Algorithm	Operation	HACL*	OpenSSL (C)	libsodium (C)	TweetNaCl	OpenSSL (asm)
SHA-256	Hash	45.83	40.94	37.00	-	14.02
SHA-512	Hash	34.76	20.58	27.26	37.70	15.65
Salsa20	Encrypt	13.50	-	27.24	40.19	-
ChaCha20	Encrypt	17.85 (ref) 14.45 (vec)	30.73	19.60	-	9.61
Poly1305	MAC	11.09	7.05	10.47	310.84	3.00
Curve25519	ECDH	833,177	890,283	810,893	5,873,655	-
Ed25519	Sign	310.07	-	84.39	1157.73	-
Ed25519	Verify	283.86	-	105.27	2227.41	-
Chacha20Poly1305	AEAD	29.32	26.48	30.40	-	13.05
NaCl SecretBox	Encrypt	24.56	-	38.23	349.96	-
NaCl Box	Encrypt	85.62	-	97.80	779.91	-

Table .2: AARCH64-GCC: Performance Comparison in cycles/byte on an ARMv7 Cortex A53 Processor @ 1GHz running 64-bit OpenSuse Linux 4.4.62. All code was compiled with GCC 6.2.

Algorithm	HACL*	OpenSSL	libsodium	TweetNaCl	OpenSSL (asm)
SHA-256	25.70	30.41	25.72	-	14.02
SHA-512	70.45	96.20	101.97	100.05	15.65
Salsa20	14.10	-	19.47	21.42	-
ChaCha20	15.21 (ref) 7.66 (vec)	18.81	15.59	-	5.2
Poly1305	42.7	17.41	7.41	140.26	1.65
Curve25519	5,191,847	1,812,780	1,766,122	11,181,384	-
Ed25519	1092.83	-	244.75	1393.16	-
Ed25519	1064.75	-	220.92	2493.59	-
Chacha20Poly1305	62.40	33.43	23.35	-	7.17
NaCl SecretBox	56.79	-	27.47	161.94	-
NaCl Box	371.67	-	135.80	862.58	-

Table .3: ARM32-GCC: Performance Comparison in cycles/byte on an ARMv7 Cortex A53 Processor @ 1GHz running 32-bit Raspbian Linux 4.4.50. All code was compiled with GCC 6.3 with a custom library providing 128-bit integers.

Algorithm	Implementation	Language	Architecture	Cycles
ChaCha20	moon/neon/32	assembly	NEON	9694
	hacl-star/vec128	C	NEON	12602
	dolbeau/arm-neon	C	NEON	13345
	hacl-star/ref	C	NEON	17691
	moon/armv6/32	assembly	ARM	18438
	e/ref	C	ARM	22264
Poly1305	moon/neon/32	assembly	NEON	10475
	neon2	assembly	NEON	11403
	moon/armv6/32	assembly	ARM	18676
	53	C	ARM	20346
	hacl-star	C	ARM	127134
	ref	C	ARM	395722
Curve25519	neon2	assembly	NEON	1935283
	ref10	C	ARM	4969185
	hacl-star	C	ARM	13352774
	ref	C	ARM	60874070
SHA-512	sphlib	C	ARM	82589
	ref	C	ARM	118118
	hacl-star	C	ARM	121327
Ed25519	ref10	C	ARM	2,093,238
	ref	C	ARM	18,763,464
	hacl-star	C	ARM	29,345,891

Table .4: ARM32 SUPERCOP Benchmarks: ranked list of best performing implementations on an ARMv7 Cortex A53 Processor @ 1GHz running 32-bit Raspbian Linux 4.4.50.

Algorithm	Implementation	16by	64by	256by	1024by	8192by	16384by
ChaCha20	HACL*	90381.10k	353297.74k	377317.29k	380701.70k	386591.17k	385418.53
	HACL* vec	115770.29k	486701.81k	728594.24k	860998.38k	910695.60k	924024.72
	OpenSSL C	204657.84k	318616.27k	342565.63k	346045.80k	371442.81k	370262.02
	OpenSSL ASM	285974.37k	526845.47k	1165745.92k	2382449.36k	2452002.59k	2470173.90
ChachaPoly	HACL*	39405.99k	143626.18k	238075.98k	277331.74k	292995.07k	302145.07
	OpenSSL C	169799.71k	262761.53k	285738.89k	304376.49k	300509.41k	290193.41
	OpenSSL ASM	217872.74k	399483.59k	848875.62k	1518847.66k	1632862.87k	1638246.57
SHA-256	HACL*	20331.67k	54075.54k	106500.44k	141369.19k	158401.50k	153695.16
	OpenSSL C	18121.99k	49251.87k	104402.28k	144965.29k	161028.97k	166327.74
	OpenSSL ASM	25321.67k	78481.92k	201910.03k	310514.47k	375845.67k	389046.03
SHA-512	HACL*	16513.59k	65673.72k	127720.99k	201159.46k	234087.09k	236592.63
	OpenSSL C	17280.47k	68173.85k	135549.35k	213524.48k	263108.41k	264705.37
	OpenSSL ASM	20556.52k	82447.35k	194595.05k	368933.21k	519731.71k	546442.02
Poly1305	HACL*	33945.66k	125367.98k	382090.15k	817432.47k	1204432.92k	1246641.57
	OpenSSL C	35947.80k	134963.35k	421210.62k	928101.54k	1355694.08k	1418755.77
	OpenSSL ASM	33354.96k	125854.18k	433647.19k	1383256.87k	3630256.03k	4032672.28
Curve25519	HACL*	144895					
	OpenSSL C	68107					

Table .5: OpenSSL speed comparison for our algorithms. Each algorithm is run repeatedly for three seconds on different input sizes, and we measure the number of bytes per second via the `openssl speed` command. The experiment is performed on an Intel Core i7 @ 2.2Ghz running OSX 10.12.4. For Curve25519, we measure the number of ECDH computations per second.

Appendix D : TLS charts

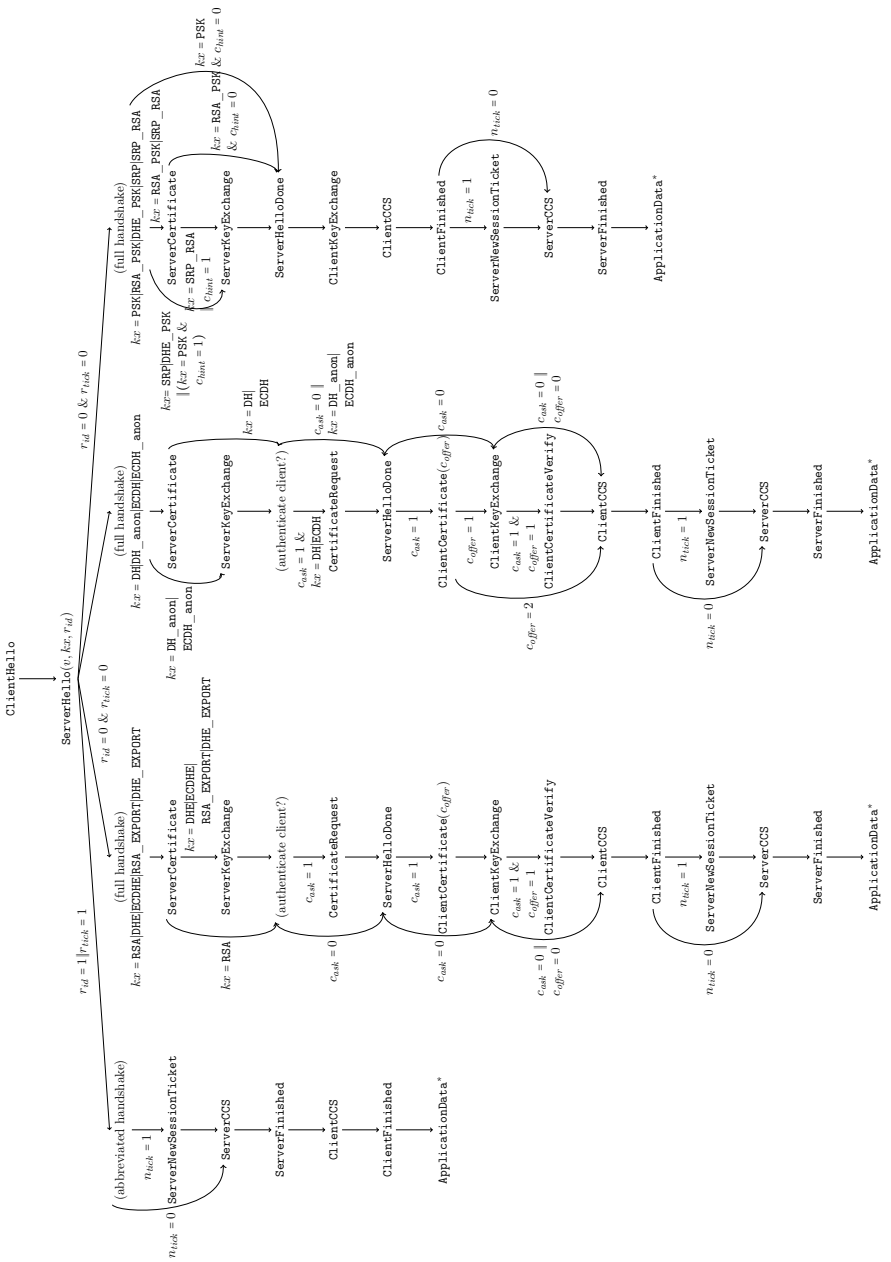


Figure .1: Message sequences for the ciphersuites commonly enabled in OpenSSL

Résumé

La sécurité des applications sur le web est totalement dépendante de leur design et de la robustesse de l'implémentation des algorithmes et protocoles cryptographiques sur lesquels elles s'appuient. Cette thèse présente une nouvelle approche, applicable à de larges projets, pour vérifier l'état de l'art des algorithmes de calculs sur les grands nombres, tel que rencontrés dans les implémentations de référence. Le code et les preuves sont réalisés en F*, un langage orienté preuve et qui offre un système de types riche et expressif. L'implémentation et la vérification dans un langage d'ordre supérieur permet de maximiser le partage de code mais nuit aux performances. Nous proposons donc un nouveau langage, Low*, qui encapsule un sous ensemble de C en F* et qui compile vers C de façon sûre. Low* conserve toute l'expressivité de F* pour les spécifications et les preuves et nous l'utilisons pour implémenter de la cryptographie, en y intégrant les optimisations des implémentations de référence. Nous vérifions ce code en termes de sûreté mémoire, de correction fonctionnelle et d'indépendance des traces d'exécution vis à vis des données sensibles. Ainsi, nous présentons HACL*, une bibliothèque cryptographique autonome et entièrement vérifiée, dont les performances sont comparables sinon meilleures que celles du code C de référence. Plusieurs algorithmes de HACL* font maintenant partie de la bibliothèque NSS de Mozilla, utilisée notamment dans Firefox et dans RedHat. Nous appliquons les mêmes concepts sur miTLS, une implémentation de TLS vérifiée et montrons comment étendre cette méthodologie à des preuves cryptographiques, du parsing de message et une machine à état.

Mots Clés

Méthodes formelles, informatique, preuves, cryptographie, compilation, correction fonctionnelle, sûreté mémoire, canaux auxiliaires.

Abstract

The security of Internet applications relies crucially on the secure design and robust implementations of cryptographic algorithms and protocols. This thesis presents a new, scalable and extensible approach for verifying state-of-the-art bignum algorithms, found in popular cryptographic implementations. Our code and proofs are written in F$*$, a proof-oriented language which offers a very rich and expressive type system. The natural way of writing and verifying higher-order functional code in F$*$ prioritizes code sharing and proof composition, but this results in low performance for cryptographic code. We propose a new language, Low$*$, a fragment of F$*$ which can be seen as a shallow embedding of C in F$*$ and safely compiled to C code. Nonetheless, Low$*$ retains the full expressiveness and verification power of the F$*$ system, at the specification and proof level. We use Low$*$ to implement cryptographic code, incorporating state-of-the-art optimizations from existing C libraries. We use F$*$ to verify this code for functional correctness, memory safety and secret independence. We present HACL$*$, a full-fledged and fully verified cryptographic library which boasts performance on par, if not better, with the reference C code. Several algorithms from HACL$*$ are now part of NSS, Mozilla's cryptographic library, notably used in the Firefox web browser and the Red Hat operating system. Eventually, we apply our techniques to miTLS, a verified implementation of the Transport Layer Security protocol. We show how they extend to cryptographic proofs, state-machine implementations and message parsing verification.

Keywords

Formal methods, computer science, proofs, cryptography, compilation, functional correctness, memory safety, side channels.