Foundational Verification of Cryptographic Primitives

Benjamin Salling Hvass

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Foundational Verification of Cryptographic Primitives

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Benjamin Salling Hvass
December 31, 2022

Abstract

Cryptographic software is difficult to get right. Not only does cryptographic protocols have complex specifications, but since the use of cryptography is ubiquitous in modern software, implementations must also be heavily optimized and programmed in low-level languages. Furthermore, the abstraction level at which cryptographic pen-and-paper proofs are conducted, is usually very far from the gritty implementation details.

For these reasons, applying mechanized reasoning tools to get formal guarantees about cryptographic software has within the last decade become increasingly popular. However, so far the focus has been primarily on the most common cryptographic primitives and protocols, such as TLS, SHA-3, Curve 25519, etc.

In this thesis, we explore how some of the tools developed so far can be applied to *pairing-based cryptography* or *elliptic curve cryptography* in general. We do this first by extending an existing tool with a finite field inversion algorithm which is particularly desirable in the pairing-based case. Secondly, we extend the same algorithm to additionally compute the Kronecker symbol and benchmark its performance in hashing to elliptic curves. In both cases, we provide an extensive analysis of the algorithms in the Coq Proof Assistant.

Lastly we formally connect the Jasmin language to the Coq framework SSProve, allowing for end-to-end verification of efficient primitives implemented using Jasmin. Connecting to Coq opens up for the opportunity to utilize the existing mathematical libraries formalized in Coq in the verification of cryptographic implementations; e.g. elliptic curves. We however leave this for later work, but do exemplify the connection with an encryption scheme using AES.

Resumé

Kryptografisk software er svært at implementere korrekt. Kryptografiske protokoller har komplekse specifikationer, og da kryptografi er overalt i moderne software, så skal implementationer også være superoptimerede og programmerede i maskinnære sprog. Desuden er abstraktionsniveauet for kryptografiske beviser ofte langt fra konkrete implementationer.

Pga. disse udfordringer er det blevet mere og mere populært at anvende automatiseret software til at få formelle garantier for sikkerheden af kryptografiske implementationer. Indtil nu har fokus dog primært været på de mest udbredte kryptografisk protokoller og primitiver, f.eks TLS, SHA-3 og Curve25519.

I denne afhandling undersøger vi hvordan nogle af de hidtil udviklede værktøjer kan finde anvendelse inden for *"pairing"-baseret kryptografi* eller *elliptisk kurve-kryptografi* mere generelt. Vi gør dette ved at udvide et eksisterende værktøj med en algoritme til invertering i endelige legemer, som er specielt anvendelig i legemer anvendt til "pairing"-baseret kryptografi. Dernæst generaliserer vi denne algoritme til også at beregne Kronecker-symbolet og tester en implementation ved at anvende den til "hashing" til elliptiske kurver. I begge tilfælde giver vi en omfattende analyse af algoritmerne i bevisassistenten Coq.

Til sidst forbinder vi programmeringssproget Jasmin med Coq-frameworket SSProve. En sådan forbindelse giver mulighed for at verificerer implementationer fra abstrakt specifikation til effektiv implementation. Forbindelsen til Coq åbner desuden op for muligheden for at anvende den enorme mængde matematik der er formaliseret i bevisassistenten. F.eks. elliptiske kurver. Vi efterlader dog dette til senere arbejde, men eksemplificerer Jasmin-Coq-forbindelsen med en krypteringsprotocol baseret på AES.

Acknowledgments

First I would like to thank my advisors Diego de Freitas Aranha and Bas Spitters. Working together with them have given me a unique perspective on verified cryptography from two very different angles – the side of foundational formalization and the side of aggressive optimization. Thanks to Diego for teaching me the many details of cryptographic implementations and to Bas for teaching me formal methods, in particular the Coq Proof Assistant. Thanks to Peter Schwabe for hosting me at the MPI-SP in Bochum, I greatly enjoyed my time in Germany.

I'd also like to thank all the people I have worked together with during my PhD and earlier studies. In particular, many thanks to Philipp and Théo for our collaboration during my stay in Bochum.

Special thanks to all my friends and family for keeping me sane for the last three and a half years, I could not have done it without you. Getting beaten at board games is the perfect distraction from research, so thanks to Emil and Kim (and several others) for always being up for another round.

Finally a very sincere thank you to my partner Frederikke who is the primary reason this dissertation ever got written. Thanks for the never-ending support.

Benjamin Salling Hvass,
Aarhus, December 31, 2022.

Contents

Part I

Overview

Chapter 1

Introduction

In this chapter we give a brief review of the history and state-of-the-art of *high-assurance cryptography.* We also give background details for some of the tools used in Part II and lastly give an overview of the structure and contents of Part II.

1.1 Verification in cryptography

Verification of cryptographic implementations, formal or otherwise, is an arduous task for two major reasons:

(1) Security proofs have become increasingly mathematically complex

(2) Cryptographic implementations are highly specialized and optimized

The increase in complexity of cryptographic arguments led to the development of more formal frameworks for exercising cryptographic reasoning. Examples include formal theories for using *game-playing proofs* [29, 66] and for analyzing *cryptographic protocols* [53]. These developments led to more proofs being verifiable by experts and have paved the way for yet more specialized frameworks (e.g. [52]) for proving security properties of cryptographic algorithms.

Meanwhile, cryptographic implementations have also become increasingly complicated. Even though algorithms usually have *reference implementations*, implementations in a high-level language written to resemble the mathematical specification as much as possible, the actual implementations running in the world are usually written in low-level assembly language and often in an architecture-dependent way. Even for similar architectures different optimizations might or might not be employed, such as using specialized intrinsics or SIMD instructions.

The payoff for this increased complexity is increased performance. In the modern age, cryptography is pervasive and all-encompassing – slight increases in performance can mean massive reductions in computational cost. This need for high-speed implementations has led to the development of specialized tools [33].

Yet another aspect of cryptographic implementations contributing to complexity is that they have to take the execution environment into account. In particular, the implementations have to be *side-channel resistant*, that is, not leak secret information through variable execution time, power consumption, or other detectable inconsistencies. Guarding against such leakage is again heavily dependent on platform and compiler characteristics.

The combination of these two developments – increase in proof and implementation complexity – has made current cryptographic implementations almost unverifiable and relying on experts to be able to audit code bases is both fragile and extremely resource-consuming. As a possible solution to this problem, the field of *high-assurance cryptography* or *computer-aided cryptography* [16] has become increasingly popular. Here, computers are utilized to model and verify desirable cryptographic and security properties of cryptographic protocols or implementations.

Current formalization projects, which go all the way to the implementation level, have managed to cover a wide variety of algorithms and primitives (see e.g. [94]). However, not much has been explored in the fields of *elliptic curve cryptography* (ECC) (except for Curve25519) or *pairing-based cryptography* (PBC). PBC has recently seen applications in blockchain implementations used for *zero-knowledge proofs*, i.e. proofs of knowledge which does not leak any information about the prover's secret input.

In this thesis, we achieve progress towards more formal verification in this field of cryptography. In particular, we will focus on the functional correctness of efficient primitives.

1.2 High-assurance cryptography

Automated protocol analysis Several tools have had success in verifying complex protocols and also found severe bugs in the specification of the protocols. Examples include ProVerif [43] and Tamarin [26] which have been used to verify TLS 1.3 [41] and many other protocols; see e.g. [16] for an overview. In this thesis, we will not be looking at this aspect of high-assurance cryptography, which models the communication between parties. Due to the complexity of the protocols, these tools often favor *automation* and *scalability* over *expressivity.* If one wants to do a more granular analysis of cryptographic implementations, more *interactive* theorem provers are often necessary.

Theorem provers One tool from formal verification which has seen quite a lot of application to cryptographic protocols and implementations is *proof assistants* or *interactive theorem provers.* Using such tools one no longer has to trust each individual proof: If the *proof checker* validates a proof, then only the implementation of the checker has to be trusted. The set of implementations which have to be trusted to have faith in the checked proofs is usually referred to the *trusted computing base*, or TCB for short. A major goal for many such proof assistants is to contain the TCB as much as possible.

Apart from isolating the need for trust, proof-assistants also allow to varying degrees identifying and *automating* recurring patterns of arguments. One advantage of this is that trivial or boilerplate parts of complex arguments can be abstracted away which allows experts to more easily evaluate the validity of proposed proofs of security.

Proof assistants for use in cryptographic proofs usually have a syntax for defining *cryptographic games*, i.e. programs which an adversary can interact with and then usually is challenged to distinguish between, and support reasoning about probabilistic properties of such games.

CertiCrypt [22] and its successor EasyCrypt [24] are two such proof assistants which have been used to formally prove several cryptographic schemes secure. It has, e.g. , been used to prove the security of an implementation of the cryptographic hash function SHA-3 [6]. The proof reduces the security of the implementation to the differentiability of the underlying permutation from a random oracle (see also [37]). To get an efficient implementation they use the Jasmin language, a low-level language that has an extraction mechanism to EasyCrypt. This implementation is also proven secure against *timing attacks*, i.e. side-channel attacks that exploit variations in execution time to get information about secret input. This is possible since the model in EasyCrypt is capable of modeling *leakage*. EasyCrypt has also been used to formally verify one of the NIST post-quantum finalists Saber [72].

F* [103] is a general-purpose functional programming language and proof assistant. Within high-assurance cryptography, F* has been used to implement and verify all necessary cryptographic primitives for the TLS protocol [40]. F* supports compilation to both C [93] and assembly [61] for efficient implementation. Because of this, F* implementations can achieve performance matching that of non-verified code; an F* implementation of Curve25519 achieve the best performance in an experiment conducted in [16].

CryptoVerif [42] is another proof assistant for doing game-based proofs. CryptoVerif has been shown to scale to complex protocols, e.g. by verifying TLS 1.3. CryptoVerif has recently [79] also been connected to verified and efficient implementations in F*. This allows the user to conduct probabilistic reasoning in CryptoVerif and then extract the model to F* where an implementation can be compiled and where implementation security can be verified.

Foundations None of the previously mentioned tools are *foundational* in that they do *not* reduce all logical steps to a core foundational logic. By using a foundational tool, one does not have to trust the implementation of the analysis but merely the implementation of the proof checker. A common non-foundational tool used by both EasyCrypt and F* are SMT solvers – proof obligations are verified by an external SMT solver. One issue here is that such SMT solvers might have bugs. This decision to use a non-foundational tool is usually that it scales a lot better, since foundational proofs tend to be much longer and tend to be more difficult to automate.

The Foundational Cryptography Framework (FCF) [88] is a foundational tool based on the the Coq Proof Assistant [106]. It has been used to verify an implementation of hash-based message authentication [32] and connected with the CompCert [76] compiler, achieving end-to-end verification from source code to C.

Another recent foundational tool is SSProve [1], which implements the State Separation Proofs (SSP) methodology [52] for proving cryptographic reductions in Coq. This is the tool we will use in Chapter 4 to prove functional correctness and security properties of Jasmin implementations. While SSProve and FCF share the same underlying type theory, they differ in quite substantial ways: SSProve offers an extensible semantic model of the logic and SSProve allows one to reason using the SSP method.

Yet another high-assurance cryptography tool using Coq is Fiat Cryptography [58] or just Fiat-Crypto which is a framework for synthesis of finite field arithmetic.

Specification language Another direction in high-assurance cryptography is that of devising a *formal specification language*. Considering the several tools that exist for reasoning about cryptographic protocols, a common language for writing formal specifications would save a lot time. Simultaneously, it would increase the connection between the formalization of the same primitives in different frameworks.

To design such a language is exactly the goal of Hacspec [85], a specification language which is a strict subset of the Rust programming language. Then, since the language has a formal semantic, protocol specifications can be translated into different verification backends. This would require each backend (EasyCrypt, CryptoVerif, FCF, etc.) to supply its own translation.

In Chapter 4 we describe and implement such a backend for the SSProve framework and connect Hacspec specifications to Jasmin implementations.

1.3 The Coq Proof Assistant

The Coq Proof Assistant [106] is a general-purpose proof assistant built around a core language based on the type theory Calculus of Inductive Constructions. By constructing terms in this type theory, one can prove theorems using the Curry-Howard Correspondence [71], while also being able to implement programs in the same core language.

Coq has native support for *inductive types*, and the use of these is pervasive in Coq developments. The canonical example of such a type is the *Peano natural numbers*, representing the naturals as either zero or a successor.

```
Inductive nat :=
  | O : nat
  | S : nat → nat.
```

To use inductive types, Coq features *fixpoints* and *pattern matching*, allowing us e.g. to define addition over the naturals.

```
Fixpoint add n m :=
  match n with
  | O ⇒ m
  | S n' ⇒ S (add n' m)
  end.
```

It is worth noting here that such fixpoints are quite restrictive; Coq's *termination checker* has to be able to determine that the definition will not loop forever. Indeed, if we were allowed to define a non-terminating function, e.g. `fix bad (n : nat) := bad n`, then we could produce a term of *any type*. If we want to use the Curry-Howard Correspondence and encode proofs/theorems as terms/types then this is bad news: We would get a proof of any proposition.

Coq comes with a *tactic language* Ltac, which is used to programmatically construct terms (e.g. proofs) in the underlying type theory. This alleviates the programmer from writing terms directly by hand. In addition to the standard tactics for manipulating contexts and goals, several advanced high-level tactics have been developed for Coq including tactics for

- solving linear integer arithmetic: `lia` [39] (also non-linear and real-arithmetic variants)
- solving equations over abstract or concrete rings: `ring` [64]
- solving equations modulo associativity and commutativity: `aac-tactics` [49]

All of these proved very useful in the development described in Chapter 2. `lia` is particularly useful, basically discharging all arithmetical subgoals. Furthermore, Coq supports *generalized rewriting* [100], i.e. rewriting terms up relations different from strict equality. This is also used in Chapter 2, e.g. for the types `Q` of rationals and `mat Q` of rational matrices. Indeed, $a/b = c/d$ if just $ad = bc$, i.e. reflexivity $(a/b = a/b)$ is not the only relation generating equality of rationals. This *setoid equality* (i.e. equivalence relation) extends to matrices over `Q` in the obvious way.

Coq features *extraction* to OCaml [77]. Using this, one can implement and verify programs in Coq and then get an equivalent implementation in OCaml. This can yield better performance and it also allows inclusion into larger developments.

It is important to note that Coq's underlying logic is inherently *constructive*: The law of excluded middle does not hold *a priori*, though it is consistent with Coq's logic. That does not mean one should assume it without reason, though, since it might be inconsistent with other axioms one might need.

1.4 Fiat Cryptography

Fiat Cryptography [59] (or just Fiat-Crypto) is a framework for generating verified finite field arithmetic which is correct by design. It provides a simple CLI which takes a prime and a machine word size and generates C source files implementing most finite field operations necessary to implement e.g. elliptic curve cryptography. Go, Rust and Zig are also supported. There are separate binaries for each style of multi-precision arithmetic: Montgomery, saturated and unsaturated.

Fiat-Crypto is implemented in Coq and consists of a verified compiler from a subset of Gallina to a simple language embedded in Coq consisting only of bitwise and machine-integer operation. From here, the generated terms can be pretty-printed to C or other languages (currently Java, Rust and Go are supported). Code generated by Fiat-Crypto is currently being used in production in Firefox[1], BoringSSL[2] and the WireGuard VPN[3].

To implement new algorithms in Fiat-Crypto, one writes implementation directly in Coq. At this level *bignums*, i.e. integers larger than the word size of the underlying architecture, are modeled as lists of integers (i.e. `list Z` in Coq). Implementations using such bignums are allowed to be parametric in their *limbwidth*, i.e. the length of the inputs of type `list Z`. This allows the implementation to be instantiated with different prime moduli, further down the compilation pipeline.

The implementation can then be *partially evaluated* by instantiating certain parameters, e.g. the limbwidth of inputs and the prime wrt. numbers should be reduced. If the implementation only used suitable function, then it should reduce such that it basically resembles straightline C code, only with Coq **`let`**-bindings instead of variable assignments and arbitrary precision integers instead of bounded integers.

By specifying the range of each input and output limb, this representation can be reified into an internal AST containing architecture specific types and

[1]`https://blog.mozilla.org/security/2020/07/06/performance-improvements-via-formally-verified-cryptography-in-firefox/`

[2]`https://boringssl.googlesource.com/boringssl/+/master/third_party/fiat/`

[3]`https://www.wireguard.com/formal-verification/`

operations. This can finally be compiled into a resembling straightline C code, which can readily be pretty-printed to an efficient implementation. Note that the compilation passes are *certified*, i.e. proven correct. This resulting C implementation can the be benchmarked against other implementation of the same primitives.

Proofs about the implementation can be carried out at the source (parametric Gallina) level. This simplifies the proofs substantially and the certified compiler ensures that proven properties are carried to the target level. Note that the pretty-printing pass is not verified; this would require a formal semantics of C (or other target language) in Coq.

1.5 Jasmin

Jasmin [5] is a low-level language and compiler designed for implementing efficient implementations of cryptographic primitives. The compiler is partially implemented in Coq and verified to preserve the semantics of the source code to the target. The parts of the compiler which are not implemented in Coq are implemented in OCaml and checked using *translation validation*. It currently supports x86 assembly, but development to make the compiler more generic is ongoing work[4].

Jasmin has been used to implement several cryptographic primitives, including SHA-3 [6] and ChaCha20-Poly1305 [8]. The implementations have been proven to be comparable in performance to unverified, hand-crafted assembly implementations.

Jasmin features extraction to EasyCrypt. This way, implementations can be connected to a formal model in EasyCrypt, allowing one to reason about the implementation. Since EasyCrypt also supports probabilistic reasoning, this allows users to prove security properties of cryptographic implementations. In Chapter 4, we develop another "extraction" mechanism for Jasmin, by implementing a translation from Jasmin to SSProve. We can the use the fact that both languages have formal semantics defined in Coq, to prove that the translation is correct, i.e. preserves the semantics of programs.

1.6 Contents of this thesis

In this thesis, we try to answer the question: Has cryptographic verification tools matured to a point where we can get certified implementations of PBC?

We do this in three manuscripts that all try to shrink the gap between implementation and formal verification. The first two are more tightly coupled and should be read in order. Chapter 4 can be read independently.

[4]See e.g. the arm branch `https://github.com/jasmin-lang/jasmin/wiki/Branches`

Chapter 2 is based on the paper [73] which extends and supersedes [74, 75]. The paper appears as in the literature, except for some minor formatting.

> **High-assurance field inversion for curve-based cryptography**
> *Benjamin S. Hvass, Diego F. Aranha, Bas Spitters*
> *2023 IEEE 36th Computer Security Foundations Symposium*

In this paper we explore the *Bernstein-Yang inversion algorithm* [35] (BY-INVERSION) for computing inverses in finite fields. We detail the development of a formal verification of the algorithm in Coq and connect it to a verified implementation in Fiat-Crypto.

The formalization of the algorithm has several aspects. The correctness-proof of the algorithm reduces to checking a finite set of matrices satisfying a particular predicate. This reduction hinges on the fact that the norm of all matrices in this finite set of matrices is bounded by a sufficiently decreasing sequence of numbers. This theorem is checked *by exhaustion*, i.e. the authors of [35] implement a procedure that recursively checks the set of matrices and asserts that each member satisfy the inequality. The termination of this procedure is then proof that both the set is finite and that all elements satisfies the assertion. To verify this theorem we implement the procedure in Coq and extract to OCaml to achieve the necessary performance. Note that this increases the TCB of the formalization.

For the verified implementation, we extend the Fiat-Crypto framework with several primitives, including signed bignum arithmetic. We choose Fiat-Crypto as a tool, since it is the only one that already synthesizes arithmetic for curves we are interested in – verified finite field arithmetic in other high-assurance cryptography frameworks are limited to the underlying field of Curve25519.

My own contributions to this paper include all the formalizations and the implementations, except for the benchmarking suites. Similarly for writing the paper, I wrote the majority of all sections except the section detailing the benchmarking.

Chapter 3 is based on the manuscript:

> **Faster constant-time evaluation of the Kronecker symbol with application to elliptic curve hashing**
> *Diego F. Aranha, Benjamin S. Hvass, Bas Spitters, Mehdi Tibouchi*
> *To be submitted*

In this paper, we extend BY-INVERSION to compute the Kronecker symbol of two integers and implement an application to elliptic curve hashing. We extend the formal development of [73] to formally verify the independently developed extension of BY-INVERSION in [67]. We also implement the version from [67] and do extensive benchmarking against other implementations of the Kronecker symbol.

My contributions to the paper were the formulation of the "full width" Divstep, i.e. Algorithm 7, and a preliminary reference implementation in python. In writing, I contributed with the preliminaries, formalization details, and the proof of correctness of Algorithm 7.

Chapter 4 is based on the manuscript:

> **The last yard: formal specification and security proofs for high-speed cryptography**
> *Benjamin S. Hvass, Lasse Letager Hansen, Philipp Haselwarter, Theo Winterhalter, Bas Spitters*
> *To be submitted*

Here we utilize that Jasmin has a formal semantic in Coq to provide a verified translation from Jasmin to the probabilistic language of SSProve. We are heavily inspired by the extraction mechanism to EasyCrypt [6], except we wish to be able to *verify* the translation. Additionally, when doing theorem proving in Coq, there are many established libraries of mathematical theory one can use (e.g. [81]).

The correctness theorem of the translation states that *if* the source program has a correct semantic *then* the translated program in SSProve has the same semantic. We argue that this is the best you can hope for. Indeed, we are not interested in ill-defined source programs and we also cannot hope for a bidirectional translation: The source program contains a lot of information for the compiler which we do not have a way of representing in SSProve. We also argue that this is *adequate*. We simply wish to know about the I/O behavior of the source code and this is fully determined by its semantic, so proving things about a program with the *same semantic* is sufficient to get guarantees about the source code. Note that we do not consider side-channel leakages in this paper, however it could possibly be added to SSProve similarly to how it is modeled in EasyCrypt.

We also manage to connect the translated Jasmin code with security proofs in SSProve. We exemplify this with an implementation of an encryption scheme based on AES, assuming AES implements a pseudo-random function. Our methodology prove sufficiently modular to not require us to redo any security proofs. We only need to prove that the concrete implementation of AES is indistinguishable from the abstract one in Coq.

In addition to the Jasmin-SSProve connection, we also provide a SSProve backend for Hacspec. This way we can draw cryptographic implementations from the Hacspec library and immediately reason about them in the SSProve framework.

We also generally expand on the SSProve framework, supporting more types and implementing *unary* reasoning on top of the existing relational programming logic.

My contributions to this project include a proportional amount of the translation from Jasmin to SSProve. I also implemented the representation of local memory in SSProve and verified that the translation respects local memory as part of the correctness of the entire translation. Additionally, I carried out all the examples of verification of translated Jasmin code in SSProve.

1.7 Further development

The work presented in this thesis explores how current tools can be applied to get verified implementations of PBC. There is, however, still plenty more to be done in the field of high-assurance elliptic curve cryptography.

One conclusion of the work presented in Chapter 2, is that Fiat-Crypto is not suited for more complex algorithms, since it does not support loops or function calls. To get an implementation of actual elliptic curve arithmetic, the support for function calls is necessary. Some work [70] have been done in connecting the correctness proofs from Fiat-Crypto with another proof backend, bedrock2[5], where more involved algorithms could be implemented.

The formalization of BY inversion in Chapter 2 is independent of the Fiat-Crypto framework. Therefore, it would be interesting to see if one could get a verified implementation from some other verification framework, e.g. the Verified Software Toolchain [9] or SSProve.

It would also be interesting to stress-test the development presented in Chapter 4 even more. If finite field arithmetic could be verified, then a connection to elliptic curve operations should not be too difficult, considering formalizations of elliptic curves already exist in Coq (see e.g. [25]).

[5]`https://github.com/mit-plv/bedrock2`

Part II

Manuscripts

Chapter 2

High-assurance field inversion for curve-based cryptography

Benjamin Salling Hvass, Aarhus University
Diego F. Aranha, Aarhus University
Bas Spitters, Aarhus University

Abstract

The security of modern cryptography depends on multiple factors, from sound hardness assumptions to correct implementations that resist side-channel cryptanalysis. Curve-based cryptography is not different in this regard, and substantial progress in the last few decades has been achieved in both selecting parameters and devising secure implementation strategies. In this context, the security of implementations of field inversion is sometimes overlooked in the research literature, because (i) the approach based on Fermat's Little Theorem (FLT) suffices performance-wise for many parameters used in practice; (ii) it is typically invoked only at the very end of a cryptographic computation, with a small impact on performance; (iii) it is challenging to implement securely for general parameters without a significant performance penalty. However, field inversion can process sensitive information and must be protected with side-channel countermeasures like any other cryptographic operation, as illustrated by recent attacks [2–4]. In this work, we focus on implementing field inversion for primes of cryptographic interest with security against timing attacks, irrespective of whether the FLT-based inversion can be efficiently implemented. We extend the Fiat-Crypto framework, which synthesizes provably correct-by-construction implementations, to implement the Bernstein-Yang inversion algorithm as a step towards this goal. This allows a correct implementation of prime field inversion to be synthesized for any prime. We benchmark the implementations across a range of primes for curve-based cryptography and they outperform traditional FLT-based approaches in most cases, with observed speedups up to 2 for the largest parameters. Our work is already used in production

in the MirageOS unikernel operating system, `zig` programming language, and the ECCKiila framework [30].

2.1 Introduction

Finite field arithmetic is pervasive in number-theoretic public-key cryptography, and an essential ingredient of Elliptic Curve Cryptography (ECC) and Pairing-based Cryptography (PBC). In many cases, its implementation dictates how efficiently and securely the overall cryptosystem behaves in practice. The field inversion operation is a peculiar case, since it is rarely among the performance-critical portions of the implementation, and most efficient algorithms for the general case are hard to implement securely without a high performance penalty [48]. For this reason, field inversion is often implemented using Fermat's Little Theorem (FLT) approach of exponentiating by $p-2$ in $\mathbb{F}_p$ for prime p. This is efficient for ECC implementations relying on special primes with fast modular reduction, especially when the exponent allows a short addition chain as in Curve25519 [34]. When performance is more pressing or parameters are not friendly to FLT inversion, implementers typically resort to an aggressively optimized version of the Extended Euclidean Algorithm (EEA). However, bugs and side-channel leakage in the EEA implementation can lead to attacks against RSA [3] and ECC [2, 4]. These are not just threats of research interest, as illustrated by a vulnerability recently discovered in the EEA implementation in Windows that could be exploited to mount denial of service attacks[1].

Field arithmetic in Montgomery representation, as commonly found in PBC, is a particularly challenging case for field inversion. The FLT approach is not favored by the *dense* prime moduli in popular families of pairing-friendly of curves [18, 20] that employ the slower modular reduction in Montgomery arithmetic [86]. In the context of PBC, the performance of field inversion matters during exponentiation in pairing groups, and it also unlocks an optimization called compressed squarings in the final exponentiation of the pairing [12]. With pairings being increasingly deployed as a fundamental building block for zero-knowledge proofs and privacy-preserving cryptocurrencies (for example in short signature schemes [47] and zkSNARKs [31]), the threat of implementation bugs becomes more important, as they can allow attacks which may compromise the security and privacy guarantees of these cryptographic systems [51][2]. A survey of implementation bugs in cryptographic libraries is collected in [44, 59].

In order to satisfy performance constraints, current efficient software implementations of ECC and PBC rely on hand-optimized architecture-specific assembly code for the underlying field arithmetic and a great deal of manual

[1]`https://bugs.chromium.org/p/project-zero/issues/detail?id=1804`

[2]`https://web.getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-cryptonote-based-currencies.html`

tuning to unlock the best performance across a range of architectures [13, 14]. This introduces low-level code which is both hard to audit and to verify as correct. Moreover, implementations need at least to be *constant-time*, in the sense that execution time does not depend on input, and protection against timing attacks is provided given some performance penalty. As an illustrating case using the popular BLS12-381 curve for motivation, the cost of one scalar multiplication required for computing short Boneh-Lynn-Schacham signatures [47] is reported to be around 400,000 cycles on Intel Skylake [10, 15] using variable-time inversion. According to our benchmarking in Table 2.3, a constant-time field inversion using publicly available code would add at least 200,000 cycles to that figure for the two required conversions to affine coordinates (one for the table of precomputed points, the other for the result). This impact is significant and motivates the need for our more efficient alternatives.

Recent progress in the literature allows this problem to be solved elegantly. Bernstein and Yang proposed in 2019 a constant-time Euclidean algorithm based on *division steps* that can be generalized for polynomial arithmetic, comes with a mathematical proof and is surprisingly efficient for field inversion [35]. In that same year, an alternative path for implementing cryptographic libraries was demonstrated as viable in the Fiat-Crypto framework [59]. By combining correct-by-construction optimized low-level code with automatically generated and formally verified high-level code, it became possible to develop libraries which are both efficient and formally verified. Simultaneously, the generated code stays within a small imperative language, thus avoiding notorious memory safety issues which cause many vulnerabilities [44]. Unfortunately, Fiat-Crypto does not provide an inversion operation and the implementer must build its own approach based on the other field operations, creating the same risk of insufficient *post-hoc* analysis.

Our contributions. We extend the Fiat-Crypto framework with a constant-time implementation of field inversion based on the Bernstein-Yang approach of iterating division steps. We implement the original version of the algorithm (with the jumpdivstep optimization) and the "half-delta" variant, recently developed to optimize inversion within ECDSA signing over the curve `secp256k1` adopted in Bitcoin [109]. This variant requires a lower number of division steps to be evaluated, which immediately translates to better performance.

Our work completes the set of finite field operations which Fiat-Crypto supports, and consists in the first efficient verified implementation of field inversion for several primes, including those needed for PBC. Moreover, it allows to conveniently synthesize a correct and portable implementation of the algorithm for *any* prime using the two main representations supported in Fiat-Crypto (unsaturated Solinas and Montgomery). This comes in contrast with previous work, which consisted of implementing the FLT approach on top of a verified multiplier, instead of a dedicated specialized inversion algorithm [111]. Our formulation of the algorithm maximally relies on what is provided by Fiat-Crypto, taking advantage of the field operations provided by the framework whenever possible instead of introducing new ones. In the context of Montgomery arithmetic, this introduces some expensive multiplications to update the algorithm's matrix coefficients, the effects of which we mitigate by employing the lazy reduction optimization and adjusting the precomputed constant.

According to our benchmarks, we achieve a performance penalty of up to 5.3 in comparison to our own unverified constant-time assembly-accelerated implementations of inversion for a range of parameters in both ECC and PBC settings from 254 to 575 bits. The slowdown is tolerable if correctness is of critical importance or if inversion performance is less critical. For the PBC primes, our implementation consistently outperforms the FLT approach accelerated with finite field arithmetic in unverified assembly, with speedups ranging from 1.6 to 2 for different sizes. For the ECC primes, we outperform the FLT approaches in the two largest parameters and improve performance up to 40% against an implementation based on Fiat-Crypto and 14% against handwritten assembly.

Our slowest implementation is already used in production the MirageOS unikernel[3] and `zig` language [4] and the ECCKiila framework [30], showing that it is fast enough for engineering projects with a focus on correctness.

Outline of the paper. We briefly explain the necessary preliminaries of Fiat-Crypto and the inversion algorithm in Sections 2.2 and 2.3. Sections 2.4 to 2.6 describe our implementation/formalization of the algorithm and our formalization of the correctness proof, respectively. The two final sections conclude with related and future work.

2.2 The Fiat-Crypto Framework

Fiat Cryptography[59] (or just Fiat-Crypto) is a framework for generating verified finite field arithmetic which is correct by design. The approach was illustrated through the implementation of field arithmetic for several standardized elliptic curves using an extensible code generation framework, capable of producing code competitive in performance with popular hand-optimized multi-precision libraries. It provides a simple CLI which takes a prime and a machine word size and generates C source files implementing most finite field operations necessary to implement e.g. elliptic curve cryptography. Java, Go and Rust are also supported. Code generated by Fiat-Crypto is currently being used in production in Firefox[5], BoringSSL[6] and the WireGuard VPN[7].

[3]`https://github.com/mirage/mirage-crypto/tree/main/ec/native`

[4]https://github.com/ziglang/zig/blob/master/lib/std/crypto/pcurves/common.zig

[5]`https://blog.mozilla.org/security/2020/07/06/performance-improvements-via-formally-verified-cryptography-in-firefox/`

[6]`https://boringssl.googlesource.com/boringssl/+/master/third_party/fiat/`

[7]`https://www.wireguard.com/formal-verification/`

In Fiat-Crypto there are separate binaries to generate code for each style of multi-precision arithmetic: Montgomery, saturated and unsaturated representations. Although, there is no formal proof that the code is of constant time, only "straight-line code" is generated, i.e. code without branching that should run in constant-time after it is processed by an optimizing compiler. Fiat-Crypto consists of a verified compiler written in the Coq proof assistant [106]. It compiles from a subset of Coq to a simple language embedded in Coq containing only bitwise and machine-integer operations. From here, the generated terms can be pretty-printed to the programming languages mentioned above.

Correct-by-construction vs. verifying existing code. Fiat-Crypto differs significantly from other verification projects: Instead of verifying an existing implementation against a specification, it provides a pipeline for generating verified implementations. This has the advantage of only requiring a single formalization effort. Verification of complex software is a laborious procedure, so in many cases it will not be deemed important enough. Having auto-generated code allows verified code to be used in such cases. Another advantage is the multi-language support: in general each implementation in a different language would present a separate formalization effort.

Coq. Coq is a state-of-the-art interactive proof assistant based on dependent type theory [105]. Coq reduces all proofs to a small *kernel* — it is thus *foundational* in that it reduces everything to the axioms of mathematics. Next to its built-in functional programming language, Coq also has a more ad-hoc scripting language for *tactics*. Users write tactics to direct Coq to construct, or search for, proofs. When, after a list of such tactic instructions, the proof is fully completed and it is finally checked by the kernel for correctness.

We will use the Coq standard library throughout this paper. In particular, we use the standard implementations of (unary) naturals `nat` and (binary and infinite precision) integers `Z`.

Multi-precision arithmetic. In cryptography, it is common to compute on numbers much larger than a single machine word. These are usually represented using arrays of digits and interpreted as a number in some large radix size (e.g. a full word size). We will refer to the entries of these arrays as *limbs* and numbers represented as such as *multi-limb numbers*.

In Fiat-Crypto multi-limb integers are represented as lists of integers, i.e. as the type `list Z`. Such a list of numbers, say `[1;12;123]`, corresponds to the sum of its elements up to some weighing of the indices, e.g. $1 \cdot 2^{\text{weight } 0} + 12 \cdot 2^{\text{weight } 1} + 123 \cdot 2^{\text{weight } 2}$, where `weight` is some map from `nat` to `Z`. Note that the representation is little-endian. When reasoning about multi-limb numbers, one uses the function `eval` to evaluate the number as an integer by adding together its limbs (multiplied by their respective weights).

We will refer to representations using a full-word radix as *saturated*. When computing on such a representation, one has to take care of propagating carries, as additions do not fit within one register. Conversely, we will refer to a radix smaller than a full word size as *unsaturated*. We will refer to arithmetic on these numbers as *multi-precision arithmetic*, as opposed to *single-precision arithmetic*, which we will assume is implemented natively in the platform.

There are a variety of optimizations and algorithms for multi-precision arithmetic, and more precisely for multi-precision *modular* arithmetic modulo some large number, as used in cryptography. One of the more expensive operations in modular arithmetic is *reduction*, as it generally requires a multi-precision division. Reduction is necessary after a multi-precision multiplication or squaring. We will briefly describe two specialized approaches, both used in our implementations.

For integers a, b and c we write $a \equiv b \pmod{c}$ when c divides the difference between a and b. For integers a, c we write $a \bmod c$ for the unique integer b between 0 and c satisfying $a \equiv b \pmod{c}$.

Generalized Mersenne Reduction. If the modulus M is of the form $2^k + c_1 2^{k-1} + \cdots + c_k$ for some integers k and c_i (which satisfies some constraints [99]), then M is said to be a *generalized Mersenne number* (or *Solinas number*). In that case there is an improved algorithm for reduction which replaces division with a linear number of additions and shifting operations. The efficiency depends on the coefficients and exponents of the integral polynomial. A notable example of a generalized Mersenne number which is used in cryptographic implementations is the prime $2^{255} - 19$ over which the elliptic curve Curve25519 is defined [34].

Montgomery Reduction. In addition to Generalized Mersenne Reduction, Fiat-Crypto supports Montgomery arithmetic [86]. If R is a number coprime to the modulus M, then the *Montgomery reduction* modulo M of a number a is the number $aR^{-1} \bmod M$. Montgomery reduction can be computed more efficiently than generic reduction when R is chosen appropriately. The algorithm performs divisions by R instead of M, so R can be chosen as a power of 2 such that divisions become cheap and simple shifts.

The factor R^{-1} might look out of place, but Montgomery reduction can be used when computing multiplications by working in the "Montgomery domain", which simply means operations are performed on numbers multiplied by R. That is, to compute $ab \bmod M$ we instead compute $(aR \bmod M)(bR \bmod M)$ and compute a Montgomery reduction. We obtain $(aR \bmod M)(bR \bmod M)R^{-1} \bmod M = abR \bmod M$, the product in the Montgomery domain. This achieves modular multiplication without divisions.

Multiplying with $R \bmod M$ every time might seem expensive, but if multiple arithmetic operations can be performed before converting back again, then this cost becomes negligible. One can also add naturally in the Montgomery domain:

$$(aR \bmod M + bR \bmod M) \bmod M = (a + b)R \bmod M.$$

Because Montgomery reduction has the same complexity as a multi-precision multiplication, another popular optimization in Montgomery arithmetic is *lazy* *reduction*, which adds unreduced multiplication results (up to $M \times R$) before a full reduction is needed.

2.3 Bernstein-Yang inversion

The Bernstein-Yang (BY) inversion algorithm [35] is a new and efficient constant-time algorithm for inverting in finite fields. In this paper we will only be using the algorithm over a field $\mathbb{F}_p$, for prime p. The algorithm is a constant-time variant of the classical Extended Euclidean Algorithm (EEA). We implement the BY algorithm in Fiat-Crypto (Section 2.4), and formalize its proof of correctness (Section 2.5).

Specification and correctness

The algorithm uses a division step (divstep), which we define for all integers δ, g and odd integers f as

$$\text{divstep}(\delta, f, g) = \begin{cases} \left(1-\delta, g, \frac{g-f}{2}\right) & \text{if } \delta > 0 \text{ and } g \text{ odd} \\ \left(1+\delta, f, \frac{g+(g \bmod 2)f}{2}\right) & \text{otherwise.} \end{cases}$$

The requirement that f is odd makes divstep an endofunction on $\mathbb{Z}\times\mathbb{Z}\times(2\mathbb{Z}+1)$. The branch can be implemented in constant time and thus so can the divstep function.

We will also use the following *transition matrices*

$$\mathcal{T}(\delta, f, g) = \begin{cases} \begin{pmatrix} 0 & 2 \\ -1 & 1 \end{pmatrix} & \text{if } \delta > 0 \text{ and } g \text{ odd} \\ \begin{pmatrix} 2 & 0 \\ g \bmod 2 & 1 \end{pmatrix} & \text{otherwise.} \end{cases}$$

These are transition matrices in the sense that multiplication corresponds to applying divstep once (up to a factor; see also Theorem 9.1 in [35]). Note that this definition differs slightly from the one in [35] (it is scaled by a factor).

To compute the inverse of g modulo f we will need to iterate the divstep, compute the transition matrix of the resulting values and sequentially multiply these matrices. This procedure is depicted in Algorithm 4.

Note that while Algorithm 4 as described is not constant time, the branch can be implemented as a conditional swap (which can be implemented in constant time). This is also how it is implemented in [35].

For integers δ, f and g we write $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g)$ and $\mathcal{T}_n = \mathcal{T}(\delta_n, f_n, g_n)$.

```
Algorithm 1: Divsteps
    Input   : Integers $n, \delta, f$ and $g$ such that $f$ is odd
    Output : The integers $\delta_n, f_n$ and $g_n$ and the matrix product
             $\mathcal{T}_n \mathcal{T}_{n-1} \cdots \mathcal{T}_0$
1  $u \leftarrow 1$, $v \leftarrow 0$, $q \leftarrow 0$, $r \leftarrow 1$ ;
2  for $i \leftarrow 1$ to $n$ do
3      if $0 < \delta$ and $g$ odd then
4          $\delta \leftarrow -\delta$, $f \leftarrow g$, $g \leftarrow -f$, $u \leftarrow q$, $v \leftarrow r$, $q \leftarrow -u$, $r \leftarrow -v$ ;
5      $g_0 \leftarrow g \bmod 2$ ;
6      $\delta \leftarrow \delta + 1$ ;
7      $g \leftarrow \frac{g + g_0 f}{2}$ ;
8      $u \leftarrow 2u$ ;
9      $v \leftarrow 2v$ ;
10     $q \leftarrow q + g_0 u$ ;
11     $r \leftarrow r + g_0 v$ ;
12 return $\delta, f, g, \left(\begin{smallmatrix} u & v \\ q & r \end{smallmatrix}\right)$
```

The Divsteps procedure can then be used to implement modular inversion as described in Algorithm 2. To implement field inversion for a fixed modulus, we can precompute d, m and e in the algorithm. The algorithm does precomputations (lines 1-7), iterates division steps a constant number of times (line 8) and combines the two (line 9); where $\mathrm{sgn}(\cdot)$ computes the sign of an integer.

```
Algorithm 2: BY-inversion
    Input   : Integers $f$ and $g$ such that $f$ is odd and $\gcd(f, g) = 1$
    Output : Integer $g^{-1}$ such that $gg^{-1} = 1 \pmod f$
1  $d \leftarrow \max(\log_2 f, \log_2 g)$ ;
2  if $d < 46$ then
3      $m \leftarrow \lfloor (49d + 80)/17 \rfloor$ ;
4  else
5      $m \leftarrow \lfloor (49d + 57)/17 \rfloor$ ;
6  $e \leftarrow ((f + 1)/2)^m \bmod f$ ;
7  $\delta \leftarrow 1$ ;
8  $\delta, f, g, \left(\begin{smallmatrix} u & v \\ q & r \end{smallmatrix}\right) \leftarrow$ Divsteps$(m, \delta, f, g)$ ;
9  $g^{-1} \leftarrow e \cdot v \cdot \mathrm{sgn}(f)$ ;
```

The correctness of this algorithm is summarized in the following theorem:

Theorem 1 (Theorem 11.2 in [35]). *Let f and g be integers with f odd. Let d be a real number such that $f^2 + 4g^2 \leq 5 \cdot 2^{2d}$. Let m be an integer such that $m \geq \lfloor (49d + 80)/17 \rfloor$ if $d < 46$ and $m \geq \lfloor (49d + 57)/17 \rfloor$ if $d \geq 46$.*

For $i = 1, 2, \ldots, m$, let $(\delta_i, f_i, g_i) = \operatorname{divstep}^i(1, f, g)$ and $\mathcal{T}_i = \mathcal{T}(\delta_i, f_i, g_i)$ and $\begin{pmatrix} u_i & v_i \\ q_i & r_i \end{pmatrix} = \mathcal{T}_{i-1}\mathcal{T}_{i-2}\cdots\mathcal{T}_0$. Then $g_m = 0$, $f_m = \pm\gcd(f, g)$ and $v_m g = 2^m f_m \pmod{f}$.

The correctness of Algorithm 2 follows from Theorem 2 since f and g are assumed to be coprime, the final values of f and v are respectively f_m and v_m, and p is the inverse of 2^m modulo f, so the following holds:

$$p \cdot v \cdot \operatorname{sgn}(f) \cdot g = (2^{-m})v(\pm 1)g = (\pm 1)(\pm 1) = 1 \pmod{f}.$$

The theorem as stated here differs slightly from the one in [35] since our definition of $\mathcal{T}$ is scaled by a factor to avoid having to reason about rational numbers.

Outline of proof

The proof of Theorem 2, as given in [35], is in 4 parts:

- Specification of a related algorithm for computing the gcd of two numbers.
- Complexity analysis of the related algorithm; in particular giving a worst-case bound.
- Establishing the relation between divsteps and the related algorithm.
- Proving that reaching a fixed point of divstep yields the modular inverse.

These are described in Appendix E, F, G and Section 11 in [35], respectively.

We will expand on how each part was formalized in Section 2.5. For the proofs we need the definition of *2-adic valuation.* If g is an integer and p is a prime, then the *p-adic valuation* of g is the highest power of p which divides g. We will denote it by $\operatorname{ord}_p g$ or $\operatorname{val}_p g$ (in the literature ν_p is also common). We will also write $\operatorname{split}_p g$ for g divided by this maximal power of p, i.e. $\operatorname{split}_p g = g/p^{\operatorname{ord}_p g}$.

While the proof in the paper uses 2-adic integers, we only use the corresponding statements for integers. This facilitates the formalization and suffices to prove Theorem 2.

The jumpdivstep optimization

Algorithm 4 can be optimized by observing that computing the k first iterations of DIVSTEPS only depends on the k first bits in f and g. This allows working on smaller numbers and "jumping" through the DIVSTEPS computations in larger steps. This optimized version is depicted in Algorithm 3 (see section 10 in [35] for details).

Algorithm 3: JumpDivsteps

```
Input   : Integers $n, k, b, \delta, f$ and $g$ such that $f$ is odd and $k \mid n$ and
          $k \leq b$
Output  : The integers $\delta_n, f_n$ and $g_n$ and the matrix product
          $\mathcal{T}_n\mathcal{T}_{n-1}\cdots\mathcal{T}_0$
1 $\mathcal{T} \leftarrow \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)$ ;
2 for $i \leftarrow 1$ to $n/k$ do
3     $f' \leftarrow f \bmod 2^b, g' \leftarrow g \bmod 2^b$ ;
4     $\delta, f', g', \mathcal{T}' \leftarrow \textsc{Divsteps}(k, \delta, f', g')$ ;
5     $\left(\begin{smallmatrix}f\\g\end{smallmatrix}\right) \leftarrow \mathcal{T}\left(\begin{smallmatrix}f\\g\end{smallmatrix}\right)/2^k$ ;
6     $\mathcal{T} \leftarrow \mathcal{T}' \cdot \mathcal{T}$ ;
7 return $\delta, f, g, \mathcal{T}$
```

One way to see that Algorithm 3 is correct is to note that one run through the loop corresponds to k runs through the loop in Algorithm 4 (where the matrix $\mathcal{T}$ corresponds to the four variables u, v, q and r). Indeed, which branch is chosen in Divsteps for the first k iterations only depends on the first k bits of f and g, since the $k-1$ first bits in $(g-f)/2$ and $(g+(g \bmod 2)f)/2$ (the two possibilities for subsequent g-values) only depend on the k first bits of g and f.

The concrete values of f and g have no influence on u, v, q and r, so the matrix we get in line 4 of Algorithm 3 is indeed correct, and we multiply with the current product in line 6 as required. Now, to see that the updated values of f and g are correct, simply note that for integers $i \leq j$,

$$2^{j-i}\begin{pmatrix}f_j\\g_j\end{pmatrix} = \mathcal{T}_{j-1}\mathcal{T}_{j-2}\cdots\mathcal{T}_i\begin{pmatrix}f_i\\g_i\end{pmatrix}$$

This follows from the fact that $2\begin{pmatrix}f_{i+1}\\g_{i+1}\end{pmatrix} = \mathcal{T}_i\begin{pmatrix}f_i\\g_i\end{pmatrix}$ (by definition) and induction (see also theorem 9.1 in [35]). We already established that $\mathcal{T}'$ is equal to the intermediate matrix product, so f' and g' are equal to f_{ik} and g_{ik} (in the ith iteration).

2.4 Verified and efficient field inversion in Fiat-Crypto

We will now shift gears and focus on the technical details of our formalization. Our contributions can be found in our fork of Fiat-Crypto at `https://github.com/bshvass/fiat-crypto/tree/main`. The generated code for the programming language `lang` can be found in the folder `fiat-lang`, and standalone testing/benchmarking programs for illustration can be found in the folder `inversion/c` together with a `Makefile`. All file paths in this section will be relative to this folder.

To implement the Bernstein-Yang algorithm in Fiat-Crypto we needed to add several primitives to the framework. The implementation is verified by relating it to the algorithm formalized in Section 2.5. The majority of our contribution to Fiat-Crypto is in the `src/Arithmetic/BYInv` folder and in the present section all file paths will be relative to this root.

A major part of specifying and implementing the algorithm was implementing and formalizing *signed* multi-precision arithmetic for the types of f and g in Algorithm 4, since this was absent from the framework. In the following, `machine_wordsize : Z` or `mw : Z` will refer to the machine word size for which the implementation is parameterized. We will use `m : Z` to refer to the prime underlying the modular arithmetic and wrt. which we are trying to invert; `f g : list Z` will usually refer to bound multi-limb numbers and `a b : Z` to word-sized integers. The function `eval : list Z → Z` will refer to evaluation of multi-limb numbers wrt. some weight function. In the context of Montgomery arithmetic this will be a uniform (or saturated) weight function and in the context of unsaturated Solinas it will be unsaturated.

When programming in Fiat-Crypto, one has to use the supported low-level language, i.e. the language whose terms can be compiled into the embedded C-like language and consequently generate C code. Notable supported operations are bitwise operations on integers: `>>` for right shifts, `|'` for bitwise OR and `&'` for bitwise AND. Furthermore, there is `Z.lnot_modulo n` which interprets a number to be of bit-length n and then flips all bits in its binary representation (bitwise negation).

Representing signed word-sized integers

We use the following definition to represent the numbers from -2^{mw} to $2^{\mathrm{mw}} - 1$

```
Definition twos_complement (mw a : Z) :=
  if (a mod 2 ^ mw) <? 2 ^ (mw - 1)
  then a mod 2 ^ mw
  else a mod 2 ^ mw - 2 ^ mw.
```

Then, e.g. `twos_complement mw (2 ^ mw - 1) = -1` as usual for a two's-complement representation. We will usually shorten `twos_complement` to `tc` and omit the bitwidth if it is implicit. We implement several operations working on these representations. They are all found at the bottom of the file `src/Util/ZUtil/Definitions.v` and have separate files in the same folder for their properties. In particular, we provide implementations of arithmetic shifts, addition in two's complement and multiplication in two's complement. These were all needed for the implementation, for the Divsteps over words in JumpDivsteps. The correctness statements about these implementations prove how interpreting their results in two's complement corresponds to operations on their inputs in two's complement. For example, the correctness of shifting can be expressed by the lemma below.

```
Lemma arithmetic_shiftr_spec (mw a k : Z)
      (Hm : 0 < mw)
      (Ha : 0 ≤ a < 2 ^ mw)
      (Hk : 0 ≤ k) :
  tc mw (arithmetic_shiftr mw a k)
    = (tc mw a) / 2 ^ k.
```

Representing signed multi-limb integers

We extend Fiat-Crypto with signed multi-limb arithmetic. These numbers will be represented using lists of integers and evaluated as a multi-limb number in saturated representation interpreted in two's complement. In Coq this is defined as

```
Definition tc_eval (n : nat) (f : list Z) :=
  tc (mw * n) (eval n f).
```

where `eval` is evaluation of multi-limb numbeers in saturated representation (wrt. `mw`). The variable `n` is the amount of limbs needed, i.e. the length of f, and the evaluation is interpreted in two's complement at bitwidth `bw * n`. We will usually omit the `mw` and `n` parameters for brevity. All operations we support for multi-limb signed arithmetic can be found in `Definitions.v`. We will summarize a few of them here.

Arithmetic right shift. As for word-sized integers, we will need to divide multi-limb numbers by powers of two. We do this in the obvious way by shifting each limb and xor'ing the underflow with the shift of the next limb. Additionally, the most significant limb has to be arithmetically shifted to preserve the sign (here we can reuse our implementation for word-sized integers). The function `arithmetic_shiftr` does not fit Fiat-Crypto's DSL (since shifting by a non-constant is not supported), so we cannot synthesize its implementation for general k. However, we can do so for each specific k; we only need to instantiate k before generating. In the case of jumpdivstep, we instantiate k to $b - 2$, where b is the machine word length. The correctness theorem is as follows

```
Lemma tc_eval_arithmetic_shiftr (f : list Z) (k : Z) (...) :
  tc_eval (arithmetic_shiftr f k)
    = tc_eval f / (2 ^ k).
```

```
Definition divstep_aux data :=
  let '(d,f,g,v,r) := data in
  let cond := land (pos d) (mod2 g) in
  let d' := zselect cond d (opp d) in
  let f' := select cond f g in
  let g' := select cond g (tc_opp f) in
  let v' := select cond v r in
  let v'':= addmod v' v' in
  let r' := select cond r (oppmod v) in
  let g0 := mod2 g' in
  let d'' := (d' + 1) mod 2 ^ mw in
  let f'' := select g0 tc_zero f' in
  let g'' := arithmetic_shiftr1 (tc_add g' f'') in
  let v''' := select g0 mont_zero v' in
  let r'' := addmod r' v''' in
(d'',f',g'',v'',r'').

Definition divstep (d : Z) (f g v r : list Z) :=
  divstep_aux (d, f, g, v, r).
```

Listing 1: Implementation of a divstep in Fiat-Crypto

where k is the shift amount and f is a multi-limb integer.

Addition, negation and subtraction. Addition is already implemented in Fiat-Crypto (signed addition is the same as unsigned addition in two's complement), so we simply wrap this implementation in the function `tc_add`. To negate we flip all bits of all limbs in the list and then use `tc_add` to add one. Subtraction is defined by combining addition and negation.

Implementing Divsteps

We provide implementations of a single divstep, i.e. the body of the for loop in Algorithm 4. The implementations are in the file `Definitions.v` and properties about the implementations are proven in `Divstep.v`. There are implementations for Montgomery arithmetic, unsaturated Solinas and for word-sized divsteps (needed for JumpDivsteps). We have included the implementation using Montgomery arithmetic in Listing 1.

The implementation uses modular arithmetic for variables u, v, q and r since these would otherwise grow much larger than necessary (by Theorem 2 we only need these numbers modulo f), regular signed multiple-precision arithmetic for f and g and word-sized arithmetic for δ.

We utilize the `zselect` and `select` functions provided by the library, which implement constant-time selection of values depending on a condition. The functions `addmod` and `oppmod` are multi-limb modular addition and negation, respectively; these were also implemented in Fiat-Crypto. The functions which we have implemented and verified are `pos`, which checks positivity of a word-sized integer in two's complement, `mod2`, which checks the parity of a multi-limb integer, and `arithmetic_shiftr1` which is a singular arithmetic right shift of multi-limb numbers. We did not have to implement `tc_add`, but we had to specify how it is computed in two's complement. Note also that the implementation differs slightly from the specification in Algorithm 4, in that we do not compute the u and q quantities. This is simply because they are not necessary in Algorithm 2 (they are however used in the proof of correctness of Algorithm 2).

We prove that executing `divstep` and then evaluating corresponds to the reference Coq function

```
Definition divstep_vr_mod m '(d, f, g, v, r) :=
  if Z.odd g
  then if 0 <? d
       then (1 - d, g, (g - f) / 2,
             (2 * r) mod m, (r - v) mod m)
       else (1 + d, f, (g + f) / 2,
             (2 * v) mod m, (r + v) mod m)
  else (1 + d, f, g / 2,
       (2 * v) mod m, r mod m).
```

which is simply defined over the `Z` type of integers in Coq, i.e. it does not compute over lists of numbers. Concretely, we prove the theorem

```
Lemma divstep_correct (d : Z) (f g v r : list Z) (...) :
let '(d1,f1,g1,v1,r1) := (divstep_aux (d, f, g, v, r)) in
(tc d1, tc_eval f1, tc_eval g1,
        eval v1 mod m, eval r1 mod m) =
  divstep_vr_mod m (tc d, tc_eval f, tc_eval g,
                          eval v mod m, eval r mod m).
```

which states that for a given input d, f, g, v, r to divstep computing the Fiat-Crypto implementation Listing 1 and interpreting the output in two's complement, corresponds to interpreting d in two's complement, f, g as multi-limb numbers in two's complement, and r, v as multi-limb numbers modulo m, and computing `divstep_vr_mod_m` of the interpretations.

The proof is straightforward, in that we simply have to propagate the correctness theorems of each function that is called (`addmod`, `tc_add`, etc.). The correctness of the modular operations are provided by Fiat-Crypto, but others we have to prove ourselves. Most functions have a separate file dedicated to their correctness and properties, e.g. `TCAdd.v`, `ArithmeticShiftr.v`, `Mod2.v`, etc.

Note that the theorem has several assumptions not depicted here. Most notably, f and g must be less than half the maximum integer, since otherwise the first `tc_add` might overflow. Also, f have to be assumed odd, but recall that this is simply by the definition of divstep (see Section 2.3).

Implementing BY-inversion using Divsteps

We provide implementations of Algorithm 2 using Divsteps. The implementations are in the file `Definitions.v` and properties about the implementations are proven in `BYInv.v`. There are implementations for Montgomery arithmetic and unsaturated Solinas. We have included the implementation using Montgomery arithmetic here in Listing 3.

In the implementation, we use the function `partition`, which simply takes an integer and represents it as a multi-limb number (as a list) according to some weight function. Obviously, the arguments to this has to be known a compile time and is a way to get Coq to compute constants and translate them to multi-limb numbers. We use it to get the representation of m, the prime wrt. which we are inverting, and `divstep_precomp`, which is the e from Algorithm 2. The fold in Listing 3 computes the for loop from Algorithm 2 and the function `iterations` computes

```
Definition iterations (b : Z) :=
  if b <? 46 then (49 * b + 80) / 17
    else (49 * b + 57) / 17.
```

as expected by Algorithm 2. Note that `iterations` is in `Ref.v`.

We prove that our implementation is equivalent to a reference Coq implementation, which we will prove correct in Section 2.5. The reference, `by_inv_ref`, is in `Ref.v`. The correctness theorem (for the Unsaturated Solinas case) is depicted in Listing 2. The assumptions are rather natural: `g_length` requires that g has the appropriate amount of limbs, `g_in_bounded` requires that each limb of g is within a word size and `g_bounds` requires that g is less than half the absolute max value representable in two's complement. We ensure that this is true when g is a number mod m, by choosing the quantity `tc_limbs` such that $m < 2^{\text{machine\_wordsize}\cdot\text{tc\_limbs}} - 2$.

The proof of theorem required us to prove several invariants of `divstep`. Fiat-Crypto provided sufficient machinery to make this fairly simple; see e.g. `divstep_iter_bounds` in `Divstep.v`. The only potential "overflow" we have to worry about, is d, since it can in practice grow unboundedly if one iterates divstep sufficiently many times; this shows up in the lemma `divstep_iter_correct` where

```
Theorem eval_by_inv (g : list Z)
  (g_length : length g = tc_limbs)
  (g_in_bounded : in_bounded g)
  (g_bounds :
    - 2 ^ (mw * tc_limbs - 2) <
      tc_eval g <  2 ^ (mw * tc_limbs - 2)) :
  eval (by_inv g) mod m = by_inv_ref m (tc_eval g).
```

Listing 2: Correctness of `by_inv`

you have to bound d depending on how many times you iterate. The other quantities cannot overflow, since f, g are decreasing for each divstep and v, r are computed using modular arithmetic. Only in the very first iteration, when you add f and g, do you risk to overflow: this is exactly why we need to have 2 spare bits when representing the prime in two's complement (1 for this potential overflow and 1 for the sign).

Montgomery inversion. Inverting in Montgomery arithmetic needs additional care to compute the inverse wrt. Montgomery multiplication, not regular modular multiplication. This amounts to computing the regular inverse multiplied by R^2. This works since,

$$\begin{aligned}(aR \bmod M)((aR)^{-1} \cdot R^2 \bmod M)R^{-1} \bmod M \\ = 1 \cdot R \bmod M\end{aligned}$$

which indeed is 1 in the Montgomery domain. Accordingly, the theorem `eval_by_inv_jump` contains this factor in the Montgomery case.

Implementing JumpDivsteps

We provide implementations of a single jumpdivstep, i.e. the body of the for loop in Algorithm 3. The implementations are in the file `Definitions.v` and properties about them are proven in `JumpDivstep.v`. There are implementations for both the Montgomery arithmetic and unsaturated Solinas. We have included the implementation in Montgomery arithmetic here in Listing 4.

For JumpDivsteps we need a couple of additional methods. The idea of JumpDivsteps is computing Divsteps (line 4) on word-sized integers (we use $k = \mathrm{mw} - 2$ such that all intermediate values in divstep fit in a word). This however also means that the entries of the result matrix $\mathcal{T}'$ are word-sized integers and thus we have to multiply word-sized and multi-limb numbers when computing the matrix-vector product in line 5. This functionality was already implemented in Fiat-Crypto, but we wrapped it in `word_sat_mul`.

Also, the numbers in $\mathcal{T}$ have to be modular reduced (otherwise they grow too large), so when we have to compute the matrix product in line 6, we have to reduce the entries of $\mathcal{T}$ modulo f (they might for instance be negative).

```
Definition by_inv (g : list Z) :=
  let bits := (Z.log2 m) + 1 in
  let msat := partition m in
  let its := iterations bits in
  let pc := partition divstep_precomp in
  let '(_, fm, _, vm, _) :=
    fold_left (fun data i ⇒ divstep_aux data)
              (seq 0 (Z.to_nat its))
              (1,msat,g,zero,one) in
  let sign := tc_sign_bit fm in
  let inv := mulmod pc vm in
  let inv := select sign inv (oppmod inv) in
  inv.
```

Listing 3: Implementation of a BY-inversion in Fiat-Crypto using Montgomery arithmetic

This is what `twosc_word_mod_m` does (the corresponding function for Solinas is `word_to_solina`), by computing the negation and then choosing based on sign (recall that it has to compute the negation always otherwise a branch is introduced).

Proving `jump_divstep` correct is similar to proving `by_inv`, in that we have to prove invariants about (word-sized) divsteps. As we did for the other algorithms, we prove it correct wrt. a reference, namely the function

```
Definition jump_divstep_vr
  (n : nat) (mw m : Z) '(d, f, g, v, r) :=
  let '(d1, f1, g1, u1, v1, q1, r1) :=
    iter n divstep_uvqr
      (d, f mod 2 ^ mw, g mod 2 ^ mw, 1, 0, 0, 1) in
  let f1' := (u1 * f + v1 * g) / 2 ^ n in
  let g1' := (q1 * f + r1 * g) / 2 ^ n in
  let v1' := (u1 * v + v1 * r) mod m in
  let r1' := (q1 * v + r1 * r) mod m in
  (d1, f1', g1', v1', r1').
```

Here, `iter` is just an iterator applying the second argument to itself `n` times, initialized with the third argument to `iter`. The function `divstep_uvqr` is similar to `divstep_vr_mod` from earlier, but it also computes the q and r quantities; these are needed in JumpDivsteps to update the values f, g, v and r.

The correctness of `jump_divstep` states that

```
Definition jump_divstep_aux '(d, f, g, v, r) :=
  let '(d1,f1,g1,u1,v1,q1,r1) :=
    fold_right word_divstep
      (d,nth_default 0 f 0,nth_default 0 g 0,1,0,0,1)
      (seq 0 (machine_wordsize - 2)) in
  let f2 := word_tc_mul u1 f in
  let f3 := word_tc_mul v1 g in
  let g2 := word_tc_mul q1 f in
  let g3 := word_tc_mul r1 g in
  let f4 := tc_add word_tc_mul_limbs f2 f3 in
  let g4 := tc_add word_tc_mul_limbs g2 g3 in
  let f5 := arithmetic_shiftr f4 (machine_wordsize - 2) in
  let g5 := arithmetic_shiftr g4 (machine_wordsize - 2) in
  let f6 := firstn tc_limbs f5 in
  let g6 := firstn tc_limbs g5 in
  let u2 := twosc_word_mod_m u1 in
  let v02 := twosc_word_mod_m v1 in
  let q2 := twosc_word_mod_m q1 in
  let r02 := twosc_word_mod_m r1 in
  let v2 := mulmod u2 v in
  let v3 := mulmod v02 r in
  let r2 := mulmod q2 v in
  let r3 := mulmod r02 r in
  let v4 := addmod v2 v3 in
  let r4 := addmod r2 r3 in
(d1,f6, g6, v4, r4).
```

Listing 4: Implementation of jumpdivstep using Montgomery arithmetic.

```
Theorem jump_divstep_correct d f g v r (...) :
let '(d1, f1,g1,v1,r1) :=
  jump_divstep_aux (d, f, g, v, r) in
(tc d1, tc_eval f1, tc_eval g1,
        eval v1 mod m, eval r1 mod m)
  = jump_divstep_vr (mw - 2) mw m
     (tc d, tc_eval f, tc_eval g,
           eval v mod m, eval r mod m).
```

Note that the argument `mw - 2` corresponds to k in Algorithm 3, i.e. how many times we iterate divstep on word-sized integers, and the argument `mw` corresponds to where we truncate (truncating at `mw` is easy: simply take the first limb). This correctness theorem is for unsaturated Solinas arithmetic.

```
Theorem eval_by_inv_jump g
  (g_length : length g = tc_limbs)
  (g_in_bounded : in_bounded g)
  (g_bounds :
    - 2 ^ (mw * tc_limbs - 2) <
      tc_eval g < 2 ^ (mw * tc_limbs - 2)) :
  eval (by_inv_jump g) mod m = by_inv_jump_ref (tc_eval g).
```

Listing 5: Correctness of `by_inv_jump`

We have omitted the preconditions to this theorem here for clarity, but they are all natural (and necessary). E.g., d has to have a distance of `mw - 2` to the lowest and highest value representable in two's complement (otherwise iterating divsteps `mw - 2` times might overflow).

Lazy Montgomery reduction When we translate words to multi-limb numbers in the Montgomery arithmetic setting, we ought to multiply with R, such that we get the representations in the Montgomery domain; we can however just propagate these factors through the execution and include them in the final recomputed constant. Note that this means that these factors show up in the correctness theorem in the Montgomery setting (see the `WordByWordMontgomery` section in `JumpDivstep.v`).

Implementing BY-inversion using JumpDivsteps

We provide implementations of Algorithm 2 using JumpDivsteps. The implementations are in the file `Definitions.v` and properties about the implementations are proven in `BYInvJump.v`. There are implementations for Montgomery arithmetic and Unsaturated Solinas. We have included the implementation using Montgomery arithmetic here in Listing 6.

The implementation is very close to BY-inversion using Divsteps – the only changes are the number of iterations and (consequently) the precomputed value to multiply at the end.

To prove correctness of `by_inv_jump` as specified in Listing 5 we need to prove that `jump_divstep` preserves several invariants. Here we utilize that we have the reference Coq implementation and instead prove the invariants about this function and then transport them from the Coq function defined over `Z` to the one defined over `list Z` (`jump_divstep`).

We do this e.g. in `jump_divstep_invariants2` in `JumpDivstep.v` and in the Montgomery version of `jump_divstep_iter_correct` in `JumpDvstep`. In the first one we use `jump_divstep_vr_invariants`, proven in `Ref.v`, which asserts how the bounds of the outputs of an iterated jumpdivstep depend on the bounds of the inputs. In the second one, we use that a multiplication on inputs propagates through and

```
Definition by_inv_jump g :=
  let bits := (Z.log2 m) + 1 in
  let msat := partition m in
  let jump_its := jump_iterations bits in
  let pc := partition jumpdivstep_precomp in
  let '(_, fm, _, vm, _) :=
    fold_left (fun data i ⇒ jump_divstep_aux data)
              (seq 0 (Z.to_nat jump_its))
              (1,msat,g,zero,one) in
  let sign := tc_sign_bit fm in
  let inv := mulmod pc vm in
  let inv := select sign inv (oppmod inv) in
  inv.
```

Listing 6: Implementation of BY-INVERSION using JUMPDIVSTEPS and Montgomery arithmetic

corresponds to a multiplication on the outputs (used to propagate R-factors through the computation). This is lemma `nat_iter_jump_divstep_vr_mul` proven at `Ref.v`.

Differences from [35]. In Section 12 of [35], the authors compute the matrix product in Algorithm 3 by recursively dividing it into halves, resulting in a total of $n-1$ matrix multiplications. This allows them to keep the precision of the entries as low as possible.

We compute the product iteratively, because we attempt to minimize the new code introduced to Fiat-Crypto. This requires $4n$ modular multiplications; and since only the top right entry of the final matrix is needed, it suffices to do matrix-vector multiplications (note that this is not possible when recursively dividing the product). However, by using this method one cannot keep the precision low for as many multiplications. This was fine for our implementation, since keeping track of different precision (and using appropriate multiplication implementations) in Fiat-Crypto would be difficult. Our unverified implementation of the jumpdivstep approach keeps track of how these coefficients grow (one limb with every iteration of the outer loop), making it possible to delay the expensive modular reduction until it is strictly necessary (lazy reduction).

Generating Fiat-Crypto code

Thus far we have not explained how to turn our Fiat-Crypto implementation into efficient low-level code, which is not so straightforward. Even though the implementations `by_inv` (Listing 3) and `by_inv_jump` (Listing 6) are reifiable into Fiat-Crypto's internal language, they are both too large for this to be feasible. The reason is that (1) the folds used are unrolled by Fiat-Crypto, and there is currently no way to get Fiat-Crypto to generate a proper loop, and (2) each function is fully inlined, since Fiat-Crypto does support function calls. As a result, the generated code would be many thousands of lines long and the code generation would slow down prohibitively.

It is unclear if these limitations will be fixed in Fiat-Crypto, in particular supporting function calls would require a richer internal language. On the other hand, these are also not functions which Fiat-Crypto claims to be able to generate, so we are probably pushing the limit for how large programs should be synthesized. One way to fix this, would be to export Fiat-Crypto implementations along with their proofs of correctness to a richer language. Work in this direction has started using bedrock2 [70].

We manage to generate code by splitting our implementation into two parts: The body of fold (i.e., a single divstep and a single jumpdivstep respectively) and the functions outside the loop. For the jumpdivstep version, we unfortunately also had to split the body of the loop into three parts to make reification and code generation succeed.As a result, one has to reassemble the code manually in C; as depicted in Listing 7 and Listing 8. When doing this, one should be very careful to replicate the structure of Fiat-Crypto implementation; as depicted in Listing 3 and Listing 6. Then the correctness theorem about the assembled program should still hold, though you cannot prove this formally. That this has been done correctly should be manually verified.

Implementing BY-inversion using half-delta JumpDivsteps

We also provide an implementation of a faster variant of BY-INVERSION proposed by Wuille *et al.* [109]. It is the same as `by_inv_jump` but with slightly different constants, see `jump_divstep_hd` and `jump_divstep_precomp_hd` in `Definitions.v`. This variant starts with the value $\delta = 1/2$ and runs for around 18% fewer iterations, as given by the closed formula $\lfloor (45907 \log_2(M) + 26313)/19929 \rfloor$ for inversion modulo M. While the authors provide a formal correctness proof in the latest version of the repository for the result, we understand this has neither been peer-reviewed nor formalized. So, we take the extra precaution of validating the lower number of iterations. We adapted their 256-bit Coq proofs for our various parameter sizes and executed them with two optimizations: using the `native_compute` reduction machine in Coq, which cut execution time to 32 hours from the initially reported 2.5 days; and extracted the proofs using Coq's built-in extraction mechanism [78] to OCaml native binaries for another 2-factor reduction in time. Table 2.1 reports on the time for running all proofs. At the moment, there is no connection between our implementation in Fiat-Crypto and this formalized proof. We would merely get an equivalence between the Fiat-Crypto implementation and a reference Coq implementation.

We do not prove properties about the half-delta BY-INVERSION implementation, though one could easily adapt the proofs of correctness of the other

```
void inverse(WORD out[LIMBS], WORD g[SAT_LIMBS]) {

  WORD precomp[LIMBS], h[LIMBS];
  WORD f1[SAT_LIMBS], f[SAT_LIMBS], g1[SAT_LIMBS];
  WORD v1[LIMBS], v[LIMBS];
  WORD r1[LIMBS], r[LIMBS];
  WORD d, d1, its;
  uint8_t s;

  MSAT(f);
  ITERATIONS(&its);
  PRECOMP(precomp);

  for (int j = 0; j < LIMBS; j++) {
    v[j] = 0;
    r[j] = 0;
  }
  r[0] = 1;
  d = 1;

  for (int i = 0; i < its - (its % 2); i += 2) {
    DIVSTEP(&d1, f1, g1, v1, r1, d, f, g, v, r);
    DIVSTEP(&d, f, g, v, r, d1, f1, g1, v1, r1);
  }
  if (its % 2) {
    DIVSTEP(&d1, f1, g1, v1, r1, d, f, g, v, r);
    for (int k = 0; k < LIMBS; k++)
      v[k] = v1[k];
    for (int k = 0; k < SAT_LIMBS; k++)
      f[k] = f1[k];
  }

  SIGN(&s, f);
  MUL(out, v, precomp);

  OPP(h, out);
  SZNZ(out, s, out, h);

  return;
}
```

Listing 7: handwritten reassembly of the Fiat-Crypto implementation of BY-INVERSION in Listing 3

```
void inverse(WORD out[LIMBS], WORD g[SAT_LIMBS]) {

  WORD precomp[LIMBS], h[LIMBS];
  WORD f1[SAT_LIMBS], f[SAT_LIMBS], g1[SAT_LIMBS];
  WORD v1[LIMBS], v[LIMBS];
  WORD r1[LIMBS], r[LIMBS];
  WORD d, d1, un, vn, qn, rn, its;
  uint8_t s;

  MSAT(f);
  ITERATIONS(&its);
  PRECOMP(precomp);

  for (int j = 0; j < LIMBS; j++) {
    v[j] = 0;
    r[j] = 0;
  }
  r[0] = 1;
  d = 1;

  for (int i = 0; i < its - (its % 2); i += 2) {
    FN_INNER_LOOP(&d1, &un, &vn, &qn, &rn, d, f, g);
    UPDATE_FG(f1, g1, f, g, un, vn, qn, rn);
    UPDATE_VR(v1, r1, v, r, un, vn, qn, rn);

    FN_INNER_LOOP(&d, &un, &vn, &qn, &rn, d1, f1, g1);
    UPDATE_FG(f, g, f1, g1, un, vn, qn, rn);
    UPDATE_VR(v, r, v1, r1, un, vn, qn, rn);
  }
  if (its % 2) {
    FN_INNER_LOOP(&d1, &un, &vn, &qn, &rn, d, f, g);
    UPDATE_FG(f1, g1, f, g, un, vn, qn, rn);
    UPDATE_VR(v1, r1, v, r, un, vn, qn, rn);

    for (int k = 0; k < LIMBS; k++)
      v[k] = v1[k];
    for (int k = 0; k < SAT_LIMBS; k++)
      f[k] = f1[k];
  }

  SIGN(&s, f);
  MUL(out, v, precomp);

  OPP(h, out);
  SZNZ(out, s, out, h);

  return;
}
```

Listing 8: handwritten reassembly of the Fiat-Crypto implementation of BY-INVERSION in Listing 6

implementations to this one. The reason we did not do this, is that in the end we would not get an end-to-end proof, since the method we use to formalize the reference implementation in Section 2.5 is not capable of proving this lower bound of iterations, see [35] section 8.

Table 2.1: Time taken to run computational proofs to validate the number of iterations for various prime moduli sizes in the half-delta optimization using different strategies.

Size (bits)	Iterations	Coq-`native`	Coq-`ExtOCaml`
256	590	32.1 hours	14.7 hours
381	878	213.0 hours	100.5 hours
448	1033	634.3 hours	281.1 hours
521	1201	1226.7 hours	557.6 hours
575	1325	2671.5 hours	906.7 hours

Experimental results

We have generated and benchmarked field inversion in C for primes commonly used in both ECC and PBC settings of curve-based cryptography. For ECC, we chose the well-known primes $2^{255} - 19$, $2^{448} - 2^{224} - 1$ and $2^{521} - 1$ labeled by their named curves Curve25519, Curve448 and NIST-P521 at respectively the 128-, 224- and 256-bit security levels. For PBC, we took the base fields for Barreto-Naehrig (BN) [20] and Barreto-Lynn-Scott (BLS) curves [21] at three different security levels. These are the 254-bit prime used in the now legacy 110-bit secure BN curves [12, 84], the 381-bit prime for BLS curves with embedding degree 12 undergoing standardization at 128-bit security [84], and the 575-bit prime for BLS curves with embedding degree 48 proposed for 256-bit security [82].

The generated code was integrated in the RELIC toolkit [11], a cryptographic library containing several state-of-the-art implementations of pairings. RELIC uses a combination of handwritten assembly (ASM) with higher-level C-code and has set speed records for several of the PBC parameters. Integrating the code within RELIC allowed convenient comparison between the efficiency of our approach and other field inversion algorithms already implemented in the library. We benchmarked the implementations on an Intel Skylake Core i7-6700K CPU running at 4.00GHz, using GCC version 12.1 and `clang` from LLVM 13. Numbers were obtained by computing the average of 10^4 consecutive executions measured using the cycle counter. TurboBoost and HyperThreading were disabled for benchmarking stability.

We present our results in Table 2.2 and Table 2.3. In both tables, the first part has baseline implementations from the GMP 6.2.1 library [63] used for reference. With the exception of *Variable-time GMP*, all operations are implemented in constant-time. These timings set a lower bound (aggressively optimized variable-time code) and upper bound (generic constant-time approach) that illustrate how challenging implementing efficient field inversion in constant-time can be for general fields. The next part has timings for the FLT approaches using exponentiation by $p - 2$. For the ECC primes, we took state-of-the-art timings from the literature in the ASM case [60, 87] (FLT+ASM) and benchmarked the same addition chains over field arithmetic generated by Fiat-Crypto (FLT+Fiat). For PBC, we built and benchmarked RELIC using both the existing ASM backends and field arithmetic code generated by Fiat-Crypto. Since the same number of multiplications are executed in FLT, the timings illustrate the penalty of going from handwritten ASM to Fiat-Crypto for the different parameters: an approximate slowdown of 1.2–3.0 when compiling using either `clang` or GCC.

The remaining rows in the tables show the performance of our various implementations of BY-INVERSION. The most interesting entries performance-wise are jumpdivstep and hdjumpdivstep, respectively the jumpdivstep implementation that we generate automatically from Fiat-Crypto; and the half-deta variant proposed later. These are also benchmarked in RELIC in the PBC case using the provided ASM backends.

We compare performance against FLT due to its generality, and acknowledge that performance speedups are due in part to choice of algorithms. For ECC, the hdjumpdivstep implementations outperform the FLT implementations in the two largest primes, showing that FLT approaches do not scale well for larger parameters, with speedups over unverified assembly (FLT+ASM) of 14% for Curve448 and 13% for P521. For PBC, the speedup over FLT+ASM is visible in all fields and grows to the range between 39%–49%. We also outperform FLT over a verified multiplier (FLT+Fiat) by up to a 2-factor in all cases, except Curve25519. When comparing the fastest verified implementations of BY-INVERSION with our implementation within RELIC using its unverified ASM backends (RELIC+ASM), the performance difference ranges from 2 to 5.3, a tolerable trade-off considering the correctness guarantees provided by the Fiat-Crypto version.

Timings for Curve25519. We report detailed timings for the prime $2^{255} - 19$ generated in the unsaturated representation. There are many applications for such an implementation, due to the widespread adoption of Curve25519 and Ed25519 as key exchange and digital signature algorithms [34, 36]. The Bernstein-Yang paper reports 8,778 Skylake cycles for inversion, later improved to 3,900 cycles[8]. An alternative approach by Thomas Pornin [91] was benchmarked at 6,200 cycles in our Skylake processor. In comparison, the performance degradation of our best implementation is around 3.8 in comparison to those results, but we note that these faster implementations are not

[8]`https://gcd.cr.yp.to/software.html`

verified beyond exhaustive testing and/or they employ handwritten assembly optimizations including vector instructions. We benchmarked inversion from the C+ASM verified implementation of Curve25519 in EverCrypt [94] at 12,728 cycles in the same machine, which gives us a small 15% difference in latency. For reference, Fiat-Crypto is 21% slower than Evercrypt according to the original benchmarks.

Table 2.2: Benchmarks of different approaches for field inversion over ECC fields. Numbers in bold are the fastest for group of implementations in this work or related work among the different compilers for a certain choice of prime.

		Curve25519		Curve448		NIST-P521	
	Verified	`clang`	`gcc`	`clang`	`gcc`	`clang`	`gcc`
Variable-time GMP	No	3,098	3,314	4,724	5,799	6,814	7,128
Constant-time GMP	No	75,895	76,300	186,637	186,186	257,935	270,085
FLT+ASM	No	–	**9,301**	****41,400**	–	–	**53,828**
FLT+Fiat	Partially	13,638	11,778	49,867	46,103	78,565	55,139
This work+Fiat (divstep)	Yes	69,942	72,583	189,542	248,685	230,221	308,478
This work+Fiat (jumpdivstep)	Yes	17,797	20,394	43,529	53,186	57,894	67,928
This work+Fiat (hdjumpdivstep)	Yes	**14,652**	17,166	**35,549**	43,401	**46,747**	54,708

$^{(*)}$ The authors also report 35,000 for an AVX2 implementation, but we consider the 64-bit ASM implementation more fair for comparison.

Table 2.3: Benchmarks of different approaches for field inversion over PBC fields. Numbers in bold are the fastest for group of implementations in this work or related work among the different compilers for a certain choice of prime. The last three rows represent this work using Fiat-Crypto. The three next to last rows uses arithmetic from the RELIC library.

		BN-254		BLS12-381		BLS48-575	
	Verified	`clang`	`gcc`	`clang`	`gcc`	`clang`	`gcc`
Variable-time GMP	No	3,291	3,270	4,724	4,716	7,495	7,504
Constant-time GMP	No	75,639	76,168	146,157	146,083	270,631	271,168
FLT+ASM	No	**31,452**	31,492	104,513	**103,361**	288,719	**288,109**
FLT+Fiat	Partially	57,748	75,283	182,825	262,919	577,626	856,437
RELIC+ASM (divstep)	No	87,456	87,584	167,724	164,459	335,864	337,641
RELIC+ASM (jumpdivstep)	No	14,382	14,383	23,820	23,810	43,941	43,989
RELIC+ASM (hdjumpdivstep)	No	**9,777**	9,873	16,377	**16,183**	31,963	**27,911**
This work+Fiat (divstep)	Yes	80,018	120,486	172,497	296,065	390,773	671,370
This work+Fiat (jumpdivstep)	Yes	23,406	30,572	62,555	75,412	180,023	220,852
This work+Fiat (hdjumpdivstep)	Yes	**19,733**	25,836	**52,628**	63,237	**147,402**	180,530

```
Definition iterations b :=
  if b <? 46 then (49 * b + 80) / 17
    else (49 * b + 57) / 17.
Definition jump_iterations b mw :=
  ((iterations b) / (mw - 2)) + 1.

Definition by_inv_ref (f g : Z) :=
  let bits := Z.log2 f + 1 in
  let i := iterations bits in
  let k := (f + 1) / 2 in
  let pc := (k ^ i) mod f in
  let '(_, fm, _, vm, _) :=
    iter its (divstep_vr_mod f) (1, f, g, 0, 1)  in
  let sign := if fm <? 0 then (-1) else 1 in
  sign * pc * vm mod f.

Definition by_inv_jump_ref mw f g :=
  let bits := (log2 f) + 1 in
  let its := jump_iterations bits mw in
  let total_iterations := its * (mw - 2) in
  let k := (f + 1) / 2 in
  let pc := (k ^ total_iterations) mod f in
  let '(_, fm, _, vm, _) := iter its (jump_divstep n mw f) (1, f, g, 0, 1)  in
  let sign := if fm <? 0 then (-1) else 1 in
  sign * pc * vm mod f.
```

Listing 9: Implementation of BY-INVERSION in Coq

2.5 Formalization of Bernstein-Yang inversion

This section presents the formalization of Theorem 2, which proves the correctness of the BY-INVERSION algorithm (Algorithm 2). The development is available at `https://github.com/bshvass/by-inversion` and paths in this section will be relative to `by-inversion/src/`.

The two Coq versions of the algorithm are in Listing 9 (using DIVSTEPS and JUMPDIVSTEPS respectively) and the main theorems of the development, which state that the Coq algorithms are correct, are in Listing 10.

p-adic valuations

Since some theory of p-adic valuations was required for the proof, we developed a small library for the basics of this theory. This is in the file `PadicVal.v`. We implemented the p-adic valuation by simply counting the number of times a

```
Theorem by_inv_spec f g
  (f_bound : (21 < log2 f))
  (g_bound : 0 < g ≤ f)
  (fg_rel_prime : gcd f g = 1)
  (f_odd : odd f = true) :
by_inv_ref f g * g mod f = 1.

Theorem by_inv_jump_spec mw f g
  (f_bound : (21 < log2 f))
  (g_bound : 0 < g ≤ f)
  (mw_bound : 2 < mw)
  (fg_rel_prime : gcd f g = 1)
  (f_odd : odd f = true) :
by_inv_jump_ref mw f g * g mod f = 1.
```

Listing 10: Correctness of BY-INVERSION in Coq

number is divisible by p and proved the following specification

```
Lemma pval_spec p a: a <> 0 -> 1 < p ->
  (p ^+ (pval p a) | a) /\ ~ (p ^+ (S (pval p a)) | a).
```

i.e. that `pval p a` is the maximal power of p which divides a. We also prove uniqueness such that the full specification becomes

```
Lemma pval_full_spec p a v: a <> 0 -> 1 < p ->
  pval p a = v <-> (p ^+ v | a) /\ ~ (p ^+ (S v) | a).
```

We also define split p, which divides a number by the maximal power of p which divides it evenly. We prove the specification

```
Lemma psplit_spec p a: a <> 0 -> 1 < p ->
  a = (p ^+ (pval p a)) * psplit p a /\ ~ (p | psplit p a).
```

The gcd algorithm

The formalization of the gcd algorithm using divsteps (described in Appendix E in [35]) is in `AppendixE.v`. We prove two main results:

1. The existence and specification of $q(f, g)$ (Theorem E.1 in [35])

2. The correctness of the gcd algorithm assuming termination (Theorem E.3 in [35])

We implemented a function computing $q(f, g)$ instead of proving its existence abstractly. This allows us to use it in the recursive definition of R_i (see Theorem E.2 in [35]). One minor issue here is that to construct $q(f, g)$ one needs to compute the inverse of $\mathrm{split}_2\, g$ as a 2-adic number. Recall that in the paper most theorems are stated over this larger ring. This inverse is not necessarily an integer (e.g. 2 does not have a multiplicative inverse in $\mathbb{Z}$), but when constructing $q(f, g)$ one only needs the inverse to a finite precision. This is done by computing the inverse of $\mathrm{split}_2\, g$ modulo 2^i. We recall that Coq's default logic and type theory are constructive. This has the advantage that all definable functions are actually computer programs, and allows us to carry out proofs by computation. It is possible to consistently add the axiom of excluded middle, but as a result the computation may get stuck. We have decided to use such non-computable real numbers. This allows us to carry out all operations in the same type (`R`), as opposed to silently coercing (embedding) rational numbers into the real numbers. For example, the expression $f\, \mathrm{div}_2\, g$ is rational, but we give it type `R`:

```
Definition div_2 f g : R := IZR (q f g) / IZR (2 ^+ ord2 g).
```

where `IZR` is the embedding from the `Z` to `R` (see also Theorem E.1 in [35]).

A constructive solution could have been to use a library of algebraic, or constructive, real numbers.

Complexity analysis

The formalization of the termination proof of the gcd algorithm is in `AppendixF.v`. This part of the proof is the content of Appendix F in [35]. Here, their proof becomes more complicated and one theorem from [35] depends on the termination of a SAGE program (see theorem F.22 and figure F.23 in [35]).

The mathematical community is ambivalent about such 'computer proofs'; see, for example, the discussions around the 4-color theorem [62] and Kepler's conjecture [65]. A popular solution is to carry out the computation inside a proof assistant. Although Coq has become much faster in recent years, e.g. due to addition of native compilation [45], this computation is still beyond the scope of what can be done in Coq in reasonable time. Instead, we use Coq's extraction mechanism [78] to translate the Coq program to a related program in OCaml. This slightly extends the *trusted computing base*. That is, to trust formalization of the proof is correct, one also has to trust the unverified extraction mechanism. However, possible bugs in extraction are likely to be orthogonal to possible issues in [35].

The operator norm. To prove termination of the gcd algorithm, we need to prove that the operator norm of products of transition matrices is bounded by an exponentially decreasing sequence of numbers. In particular, we have to introduce the operator norm of matrices; we only define it for 2 by 2 matrices since this suffices for the proof. To this end, we use the following formula

```
Definition mat_norm (m : mat) :=
  let '(m11, m12, m21, m22) := m in
  let a := (m11 ^ 2 + m12 ^ 2)%R in
  let b := (m11 * m21 + m12 * m22)%R in
  let c := (m11 * m21 + m12 * m22)%R in
  let d := (m21 ^ 2 + m22 ^ 2)%R in
  sqrt ((a + d + sqrt ((a - d) ^ 2 + 4 * b ^ 2)) / 2).
```

To prove that this definition enjoys properties such as $|Mv|_2 \leq |M||v|_2$ and $|MN|_2 \leq |M|_2|N|_2$, we prove and use the spectral theorem for 2 by 2 real matrices. This is the content of `Spectral.v`. See also theorem F.11 in [35] (note that by taking this formula as our definition, we do not need to prove theorem F.11).

The file `Spectral.v` only contains lemmas pertaining to the spectral theorem and the norm of matrices over reals. Our formalization of 2 by 2 matrices over general rings is in `Matrix.v`. This theory is built on top of a small theory of algebraic structures, which is the content of `Hierarchy/`. This library is in the style of math-classes [101], although in the interest of simplicity we do not depend on it. For similar reasons, we did not reuse [80]. We used automation in this development to avoid tedious algebraic proofs. The tactics `auto_mat` and `inversion_mat` use the decision procedure `ring` over types declared as algebraic rings to solve most equational proofs about matrix operations.

Bounding the operator norm. Next, the proof proceeds by computing a bound on the operator norm of products of matrices of a particular form, namely

```
Definition M (e : nat) (q : Z) :=
  [ 0 , 1 / (2 ^ e) ; - 1 / (2 ^ e) , q / (2 ^ (2 * e)) ].
```

The bound is given by theorem F.16 in [35], which is formalized in `AppendixF.v`. The formalized proof is a little laborious, since we did not find any simple tactics for manipulating expressions involving square roots (`sqrt`). Our general strategy is to reduce an expression to an expression without square roots by isolating and squaring appropriately. These two methods do not, however, suffice in general (consider e.g. $\sqrt{5} \leq 1+\sqrt{2}+\sqrt{3}$). A general solution would be to construct the ring of integers extended with square roots as a subring of the real closed field extending the rationals and use advanced decision procedures there [55].

The bounding sequence. Next, we define the number sequence α_n which will bound the operator norms. Using this we prove three main facts

- If a particular subset of the matrices $M(e, q)$ are bounded by α_n, then all such matrices are bounded by α_n (Theorem F.21 in [35]).
- That all matrices in this particular subset are bounded by α_n (Theorem F.22 in [35]).
- That the gcd algorithm terminates (Theorem F.26 in [35]).

Now we use Coq's extraction mechanism to prove Theorem F.22, as discussed at the beginning of this section. The development of this computational proof is the contents of `Comp1`. The file `Comp1/Mem.v` contains the program to be extracted (`depth_first_verify`) which utilizes memoization to achieve performance. Memoization complicated the formalization quite a bit, and we split the proof as follows:

1. Proving that the memoized program terminates (using extraction).
2. Proving the memoized program equivalent to a non-memoized program (this is in `Comp1/Mem.v`).
3. Proving that if the non-memoized program terminates, then theorem F.22 follows (this is the content of `Comp1/NoMemNew.v`).

Running the extracted program terminates[9] and outputs the same result as reported in [35]. The axiom which we add is `comp1_theorem` in `Comp1/Mem.v`.

The other proofs in this section were more straightforward as they mostly combine previously established theorems (about matrices and about the R_j sequence from `AppendixE.v`).

We developed a small "big operator" library [38] to reason about the big multiplications ($\prod$) and big additions ($\sum$), as required in theorem F.21. Like our theory of matrices, it is built on top of `Hierarchy.v`. The same reason for developing our *own* matrix library applies here, we wanted to have a greater degree of control over the implementation.

Relating the gcd algorithm and divstep

We finally prove Theorem 2 in `Section11.v` by proving and utilizing the connection between the gcd algorithm and iterating divstep in `AppendixG.v`. We have also included an implementation of the inversion algorithm, Algorithm 2, in `BYInv.v` with an accompanying correctness proof. Note that we prove a slightly

[9]In 332 minutes using OCaml 4.11.1 on a Intel Coffee Lake Core i7-9750H.

specialized version of Theorem 2: We require that the maximum bitwidth of f and g to be at least 21. This suffices for all cryptographically interesting primes and, in particular, the primes which we have benchmarked in Section 3.4.

A formal proof of correctness for inputs of bounded size exists [109]. Their approach is different from ours, in that they generate bounds and proofs of correctness from bounds on the inputs (which the caller provides). They achieve this by using a new proof strategy using convex hulls of all possible branches in a sequence of divsteps. Their proofs, like ours, use very heavy computation based on a program written in Coq. Unlike ours, the time it takes for their algorithm to finish is (barely) feasible within Coq.

Assumptions and trusted computing base (TCB)

Fiat-Crypto is axiom free but the printer from the intermediate language to C is not verified, and thus it has to be trusted. The formalization of the proof of BY-INVERSION depends on a few axioms; these are printed at end of building the development. The first four are standard axioms from the classical reals in the standard library, which are needed because Coq's logic is constructive: `sig_not_dec` (axiom of limited omniscience), `sig_forall_dec`, `functional_extensionality_dep` (functional extensionality) and `classic` (law of excluded middle). The last axiom `comp1_theorem` is the assertion that the computation of the term `depth_first_verify` terminates and yields `Some` number (as described in Section 2.5). We verify this last axiom using extraction to OCaml, adding the extraction mechanism of Coq to the TCB. This is a small, but standard, extension of the TCB, similar to Coq's `native_compute` [45], which also uses the ocaml compiler after a translation of a Coq term to an OCaml program. The precise term is

```
comp1_theorem : depth_first_verify = Some 3787975117
```

2.6 Connecting the formalization of BY-inversion and Fiat-Crypto

Having proven both Listing 2 and Listing 10 we can combine them and prove that Fiat-Crypto does in fact compute the inverse (wrt. the modular operations in Fiat-Crypto).

However, since the theorems are proven under different developments, we have to copy the correctness theorem Listing 10 to the Fiat-Crypto development, and include it as an assumption in our assertions. We hesitate to merge the two libraries, as it would quite a large dependency to the already substantial Fiat-Crypto library – however the propositions are identical as can be inspected (compare `Ref.v` in the Fiat-Crypto repository to `BYInv.v` in the by-inversion repository).

Converting representations. There is a technical caveat when comparing the input of `by_inv` to its output: The function `by_inv` assumes that its input g is represented as a multi-limb number in two's complement, and it ensures that its output is in either Montgomery or unsaturated Solinas representation (depending on the implementation). Usually, you will want both input and output to be in the appropriate representation for modular arithmetic (Montgomery/Solinas). Thus, to use the function, we have to convert between representations first. We will write `to_tc` for this conversion (to two's complement).

Word by word Montgomery. For multi-limb numbers in Montgomery form this function is simply zero-extending by a single limb if necessary, i.e.

```
Definition to_tc := extend_to_length mont_limbs tc_limbs.
```

where `extend_to_length` is provided by Fiat-Crypto, `mont_limbs` is the amount of limbs in Montgomery representation and `tc_limbs` is amount of limbs needed for representing in two's complement. Recall that we need 2 more bits to represent m in two's complement, so this will just be the identity, unless m's is within 2 bits of a multiple of the machine word size. Using this, we can prove the theorem

```
Lemma by_inv_correct g
  (g_length : length g = mont_limbs)
  (g_nonzero : eval g <> 0)
  (g_valid : valid g)
  (spec : by_inv_spec m (eval g)) :
eval (mulmod (by_inv (to_tc g)) g) mod m
  = 2 ^ (machine_wordsize * mont_limbs) mod m.
```

by applying the general lemmas about `mulmod` and the correctness theorem of `by_inv` (Listing 2). The `valid` predicate on g asserts that the evaluation of g is less than m and that it is in the unique saturated representation. This is required for all Montgomery operations and is the responsibility of the caller. Note that g is not invertible if g is zero and that `2 ^ (machine_wordsize * mont_limbs) mod m` *is* indeed the identity in the Montgomery domain.

We prove an identical theorem about `by_inv_jump` in `BYInvJump.v`.

Unsaturated Solinas. For the unsaturated representation, we have to be a bit more careful, since the representation is not unique. If we just naively convert to two's complement, we might get an "overflow" error, where a positive integer incorrectly becomes interpreted as negative. We therefore utilize the `freeze` function provided by Fiat-Crypto to get a canonical representation, which can safely be converted to two's complement. The definition becomes

```
Definition to_sat mw num den n tc_limbs m g :=
  let g := freeze (weight num den) n (ones mw) m g in
  convert_bases num den mw 1 n tc_limbs g.
```

Here num/den is the base of the unsaturated representation and n is the amount of limbs, such that `convert_bases` converts from the unsaturated representation to two's complement, which has base mw/1 and limbwidth `tc_limbs`, as expected. We are then able to prove the theorem

```
Lemma by_inv_correct g
  (g_length : length g = n)
  (g_nonzero : eval g mod m <> 0)
  (g_bounds : 0 < eval g < 2 * m)
  (spec : by_inv_spec m (eval g mod m)) :
eval (carry_mulmod (by_inv (to_tc g)) g) mod m = 1.
```

again by combining previous results. The bound requirements on g are needed for the correctness of `freeze`. They are the responsibility of the caller, but can always be met by calling the unsaturated Solinas function `carrymod`, which ensures that its output is within these bounds.

Additional constraints on m. Note that we need some assumptions about the prime m in addition to the ones already required by Fiat-Crypto. As mentioned in Section 2.4, we need that `m < 2 ^ (machine_wordsize * tc_limbs - 2)` to ensure that we can interpret m correctly in two's complement. Secondly, we need that $21 < \log_2 m$, which is true for all cryptographic use cases. Finally we need to prove that the amount of times we iterate in the implementations of BY-INVERSION do not result in overflows. The concrete bounds are in `BYInv.v` and `BYInvJump.v`, named `iterations_bounds`. These constraints are all checked before code generation (see `check_args` in `UnsaturatedSolinas.v` and `WordByWordMontgomery.v`, in `src/PushButtonSynthesis`).

2.7 Related work

The verified *synthesis* approach adopted by Fiat-Crypto is not the only possibility for verifying implementations of cryptographic algorithms. An alternate approach consists of writing optimized code by hand in a low-level language embedded in a proof assistant.

EverCrypt [94] is one example of this approach that provides a formally verified cryptographic provider, i.e., a collection of verified cryptographic implementation together with an API. It builds on two other projects HACL* [111], which is a collection of cryptographic protocols implemented, specified and verified in a subset of the F* language (Low*) and compiled to C, and ValeCrypt which is a collection of cryptographic primitives implemented in an assembly language and verified using the Vale tool [46]. EverCrypt does not generate code for new primitives, but it supports a large amount of cryptographic primitives including AES, SHA-3, MD5 and implementations of elliptic curves as well as signature and key exchange protocols on top of these. Notably, EverCrypt does not support curves for pairing-based cryptography. Yet another approach is followed by Jasmin [5], which provides a higher level assembly language and a verified compiler to Intel assembly. Jasmin has been used to verify a high-performance SHA3 implementation [6], among many other primitives [7].

There are differences in the guarantees of the tools mentioned. Coq is *foundational* in that it reduces all proofs to the axioms of mathematics. Neither Easycrypt nor F* are foundational, even though they were carefully designed, they depend on an unverified reduction to unverified SMT-solvers. Moreover, Easycrypt does not include evaluation of functions. We are not aware that F* would be able to complete the computation that Coq cannot.

2.8 Future work

We can extend our work in several directions. In one angle, we can target embedded platforms running over ARM or RISC-V and study the performance trade-offs in those systems. In another angle, we can extend the scope to other arithmetic layers employed in many other cryptographic protocols based on pairings. Fiat-Crypto can be extended with the general construction of field extension, by implementing polynomial arithmetic. Since these polynomials would have coefficients in the finite fields currently generated by Fiat-Crypto, one would have to be able to generate representations of these, e.g. as arrays of integers. It is unclear whether or not Fiat-Crypto is geared for this, but if it is, then the implementation should be no more difficult than what is presented in Section 2.4.

Going even further, one could extend Fiat-Crypto to generate elliptic curve arithmetic directly. However, Fiat-Crypto's low-level language does not include function calls and these are necessary to implement elliptic curve operations. A solution could be to the use bedrock2, a low-level language embedded in Coq with links to the Fiat-Crypto library. This has been pursued in [70].

After generating elliptic curve arithmetic, we can go yet another step up the abstraction layer and implement bilinear pairings over these curves. Bilinear pairings are maps from a product of curve groups of prime order to the multiplicative subgroup of an extension field. At this point we would have all necessary primitives to implement pairing-based protocols. This has the added complexity of requiring a formalization of bilinear pairings in Coq. Formalizations of elliptic curves in Coq already exist, e.g. in [25] and in the

Fiat-Crypto library, but no formalized implementation of a pairing has been publicly developed.

2.9 Acknowledgements

We would like to thank Pieter Wuille and Tim Ruffing for interesting discussions about their related work and the anonymous reviewers for the suggested improvements. Part of this research was supported by the Concordium Foundation.

Chapter 3

Faster constant-time evaluation of the Kronecker symbol with application to elliptic curve hashing

Diego F. Aranha, Aarhus University
Benjamin Salling Hvass, Aarhus University
Bas Spitters, Aarhus University
Mehdi Tibouchi, NTT Corporation

Abstract

We generalize the Bernstein-Yang (BY) algorithm [35] for constant-time modular inversion to compute the Kronecker symbol, of which the Jacobi and Legendre symbols are special cases. We start by developing a basic and easy-to-implement DIVSTEP version of the algorithm defined in terms of full-precision division steps. We then describe an optimized version due to [67] over word-sized inputs, similar to JUMPDIVSTEP version of the BY algorithm, and formally verify its correctness. We introduce a number of optimizations for implementing both versions in constant time and at high-speed. The resulting algorithms are particularly suitable for the special case of computing the Legendre symbol with dense prime p, where no efficient addition chain exists for the conventional approach by exponentiation to $\frac{p-1}{2}$. This is often the case for the base field of popular pairing-friendly elliptic curves. Our high-speed implementation for a range of parameters shows that the new algorithm is up to 14 times faster than the conventional exponentiation approach, and up to 25.7% faster than the previous state of the art. We illustrate the performance of the algorithm with an application for hashing to elliptic curves, where the observed savings amount to approximately 40% when used for testing quadratic residuosity within the SWIFTEC hashing algorithm [54].

3.1 Introduction

Many algorithms of cryptographic interest in Number Theory can be expressed as variants of the Euclidean algorithm. Natural examples employed in many cryptosystems are algorithms for computing modular inversion, and testing if integers have a square root with respect to a modulus (quadratic residuosity). Respectively, these algorithms are used to generate keys in factorization-based cryptosystems such as RSA and Paillier, or to convert elliptic curve points to affine coordinates; and for testing validity of the x-coordinate of an elliptic curve point when hashing.

In its original form, the Euclidean algorithm executes a variable number of iterations computed over monotonically decreasing inputs until a certain condition is met. This aspect of the algorithm, however, poses a challenge for their implementation when both security and performance are priorities. This is true under a threat model where the adversary is capable of measuring characteristics of the running implementation, such as execution time or power consumption, mounting a so-called *side-channel attack*. In such a setting, an adversary can observe irregular patterns in the implementation and collect leakage that reveals sensitive information, for example bits of private inputs [2–4]. A particular class of side-channel attacks that is considered easier to exploit consists of *timing attacks*, in which an adversary collects leakage correlated with private inputs by measuring timing differences locally or remotely. Common countermeasures against such attacks are employing *constant-time* implementation or employing additional randomness to make inputs or execution less predictable.

The literature has many hardened variations of Euclidean algorithms that behave in a more regular way and reduce the amount of leakage. The simplest approach is to evaluate the functions computed by these algorithms using one or more exponentiations by a fixed power depending on the modulus, for example $(p-2)$ for inversion modulo p. When parameters permit, this can be evaluated efficiently in constant-time using a short addition chain. Alternatively, a branchless implementation could evaluate all targets of branches and conditionally select only the correct one at each iteration [48]. This would effectively eliminate timing attacks based on branch prediction, but at high performance penalty. Another common trick is to introduce a blinding factor that randomizes the execution, which might keep the algorithm in variable-time but decorrelates the leakage from the actual inputs [2]. In many cases, these protections impose a non-trivial performance penalty that greatly reduce the efficiency of implementations in comparison to the unprotected versions; or improve security at most heuristically by modestly increasing attack complexity, while not addressing the root causes for side-channel leakage.

Related work. The Bernstein-Yang (BY) algorithm for modular inversion [35] is a recent development that elegantly improved solutions available to this problem. It consists of a fast algorithm that can be easily implemented in constant time, since it is formulated in terms of *division steps* that can be efficiently evaluated in a regular manner. The algorithm is quite flexible, and can also be generalized for polynomial arithmetic or to solve other problems related to the Euclidean algorithm. Recent work by Hvass et al. [73] has formally verified the theory underlying the BY algorithm using a foundational approach, and synthesized efficient and formally verified implementations for inclusion in the Fiat-Crypto framework [59].

In a recent independent preprint [67], Hamburg continues the Bernstein-Yang line of work and proposes a variation of the BY algorithm to compute the Jacobi symbol $\left(\frac{a}{b}\right)$ for $a, b \in \mathbb{Z}$, in constant time. The Jacobi symbol is a generalization of the Legendre symbol and can be computed by factoring $b = \prod_i p_i^{e_i}$ and multiplying the Legendre symbols $\left(\frac{a}{p_i}\right)^{e_i}$ for each prime factor p_i and multiplicity e_i. In turn, the Legendre symbol represents the quadratic residuosity of a modulo p_i and can be computed as $\left(\frac{a}{p_i}\right) \equiv a^{\frac{p_i-1}{2}} \pmod{p_i}$.

From another research angle, Pornin recently introduced a constant-time algorithm to compute modular inversion in constant time by unrolling the inner loop in a variation of the binary Euclidean algorithm [89]. He later adapted it for the Legendre symbol as well, and presented timings for an implementation targeting ARM Cortex-M0 microcontrollers [90].

Contributions. In this paper, we generalize the full-precision version of the BY algorithm to compute the Kronecker symbol that generalizes both the aforementioned Legendre and Jacobi symbols. When restricted to the Legendre symbol, which is the case of higher interest in cryptography, we obtain a significant speedup over its computation using exponentiation for dense prime moduli, where a short addition chain to compute exponentiation by $\frac{p-1}{2}$ is not known and modular multiplication is more expensive due to Montgomery arithmetic. The case of dense prime moduli frequently occur in pairing-based cryptography, where the finite fields are defined in terms of prime numbers parameterized by polynomials that quickly grow in density when evaluated over sparse integers. Moreover, we optimize Hamburg's faster word-sized version of the algorithm and obtain an additional speedup over the exponentiation approach. The correctness of our algorithms is backed by a formal verification of the underlying theory using the framework developed in [73].

Our efficient implementation and experimental evaluation point out that the basic and faster versions of the algorithm are 2 and 14 times faster than the exponentiation approach when implemented in constant time for several prime moduli underlying popular pairing-friendly curves, and up to 25.7% over the previous state of the art established in [90]. We also illustrate the algorithm when applied for testing quadratic residuosity within the SwiftEC approach to hash arbitrary strings to points on an elliptic curve, obtaining an approximate speedup of 40%. Our version of the algorithm can be easily modified to compute the Kronecker symbol instead, which is a generalization of the Jacobi symbol supporting negative integers as input.

Organization. The paper is organized as follows. Section 3.2 gives the mathematical background about the various symbols and the original BY algorithm. Section 3.3 develops the new algorithms and optimizations, both the full and word-sized precision versions. Section 3.4 presents our benchmarking approach and experimental results, and Section 3.5 concludes.

3.2 Preliminaries

In this section, we review definitions for the Legendre, Jacobi and Kronecker symbols and their main properties. These are functions applied to integer parameters taken from monotonically larger sets. In textbooks, all three use the same notation $\left(\frac{a}{b}\right)$, but we will denote them with L, J and K subscripts to make explicit what symbol we are referring to. We note that this does not change their definitions in any way, but it will prove important later on when we introduce the algorithmic parts and move across different parameter ranges.

The Legendre symbol of a over p, written as $\left(\frac{a}{p}\right)_L$ for integer a and odd prime p, is defined as:

$$\left(\frac{a}{p}\right)_L = \begin{cases} 0 & \text{if } a \equiv 0 \pmod{p} \\ 1 & \text{if } a \not\equiv 0 \pmod{p} \text{ and } \exists x \in \mathbb{Z} : a \equiv x^2 \pmod{p} \\ -1 & \text{otherwise.} \end{cases}$$

In other words, the Legendre symbol encodes the information about a being a quadratic residue modulo p, and thus can be evaluated through Euler's criterion by computing $\left(\frac{a}{p}\right)_L \equiv a^{\frac{p-1}{2}} \pmod{p}$.

The Legendre symbol satisfies several properties:

- *Periodicity in the first argument (numerator):* if $a \equiv b \pmod{p}$ then $\left(\frac{a}{p}\right)_L = \left(\frac{b}{p}\right)_L$
- *Complete multiplicativity:* $\left(\frac{ab}{p}\right)_L = \left(\frac{a}{p}\right)_L \left(\frac{b}{p}\right)_L$.

In particular, it allows to state the *law of quadratic reciprocity* for odd primes p and q:

$$\left(\frac{p}{q}\right)_L \left(\frac{q}{p}\right)_L = (-1)^{\frac{p-1}{2}\frac{q-1}{2}},$$

and its two supplements:

$$\left(\frac{-1}{p}\right)_L = (-1)^{\frac{p-1}{2}} = \begin{cases} 1 & \text{if } p \equiv 1 \pmod 4 \\ -1 & \text{if } p \equiv 3 \pmod 4. \end{cases}$$

$$\left(\frac{2}{p}\right)_L = (-1)^{\frac{p^2-1}{8}} = \begin{cases} 1 & \text{if } p \equiv \pm 1 \pmod 8 \\ -1 & \text{if } p \equiv \pm 3 \pmod 8. \end{cases}$$

The Jacobi symbol generalizes the Legendre symbol to odd positive integers b, given the factorization of $b = \prod_i p_i^{e_i}$ for $p_i > 2$:

$$\left(\frac{a}{b}\right)_J = \prod_i \left(\frac{a}{p_i}\right)_L^{e_i}.$$

It also satisfies many of the same properties of the Legendre symbol, for example periodicity of the first argument, and complete multiplicativity in one argument when the other is fixed. Like the Legendre symbol, if $\left(\frac{a}{b}\right)_J = -1$ then a is a quadratic nonresidue modulo b. The converse $\left(\frac{a}{b}\right)_J = 1$ for quadratic residue a modulo b is only true if a and b are coprime. Hence it also satisfies a more general law of quadratic reciprocity with supplements, expressed in the same way but for odd positive coprime integers.

The Kronecker symbol further generalizes the Jacobi symbol to all remaining cases, where $b = u \cdot \prod_i p_i^{e_i}$ for $u \in \{-1, 0, 1\}$ and $p_i \geq 2$, defined as:

$$\left(\frac{a}{0}\right)_K = \begin{cases} 1 & \text{if } a = \pm 1 \\ -1 & \text{otherwise.} \end{cases} \qquad \left(\frac{a}{-1}\right)_K = \begin{cases} 1 & \text{if } a \geq 0 \\ -1 & \text{otherwise.} \end{cases}$$

$$\left(\frac{a}{2}\right)_K = \begin{cases} 0 & \text{if } a \text{ is even,} \\ 1 & \text{if } a \equiv \pm 1 \pmod 8 \\ -1 & \text{otherwise.} \end{cases} \qquad \left(\frac{a}{b}\right)_K = \left(\frac{a}{u}\right)_K \prod_i \left(\frac{a}{p_i}\right)_L^{e_i}.$$

The Kronecker symbol shares many of the basic properties of the Jacobi symbol, but under more restrictions, and does not have the same connection to quadratic residuosity as the previous two. In this work, we will consider the most general definition of the Kronecker symbol, but note that some textbooks restrict the definition to $b > 0$ for simplicity [96].

The Bernstein-Yang algorithm

The BY algorithm for modular inversion relies on the definition of a *division step* that updates the operands as the algorithm executes for a fixed number of iterations [35]. A division step (divstep) is defined for all integers δ, g and odd integers f as:

$$\text{divstep}(\delta, f, g) = \begin{cases} \left(1 - \delta, g, \frac{g-f}{2}\right) & \text{if } \delta > 0 \text{ and } g \text{ odd} \\ \left(1 + \delta, f, \frac{g + (g \bmod 2) f}{2}\right) & \text{otherwise.} \end{cases}$$

The algorithm also computes *transition matrices*:

$$\mathcal{T}(\delta, f, g) = \begin{cases} \begin{pmatrix} 0 & 2 \\ -1 & 1 \end{pmatrix} & \text{if } \delta > 0 \text{ and } g \text{ odd} \\ \begin{pmatrix} 2 & 0 \\ g \bmod 2 & 1 \end{pmatrix} & \text{otherwise.} \end{cases}$$

Note that we use the definition of $\mathcal{T}$ from [73] which differs slightly from the presentation in [35].

For integers δ, f and g we write $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g)$ and $\mathcal{T}_n = \mathcal{T}(\delta_n, f_n, g_n)$.

Algorithm 4 iterates the divstep function, computing and sequentially multiplying the transition matrix of the resulting values. For a constant-time implementation, the branch can be implemented by just looking at individual bits of δ and g, the sign flips by converting the corresponding variables in two's-complement representation, and the assignments by conditional swaps to exchange the values when the branch is taken (lines 4-5 in the algorithm).

Algorith 5 implements modular inversion by computing the number of iterations required (lines 1-5), setting up the constants (lines 6-7), iterating the division steps the required number of times (line 8) and combining the results at the end (line 9). In the algorithm, $\text{sgn}(\cdot)$ computes the sign of an integer. For a fixed modulus f and assuming $1 < g < f$, which is the case commonly occurring for modular arithmetic in cryptography, we can precompute the values d, m and e in advance. The correctness of the algorithm is given by the main theorem in [35], reproduced below.

Theorem 2 (Theorem 11.2 in [35])**.** *Let f and g be integers with f odd. Let d be a real number such that $f^2 + 4g^2 \leq 5 \cdot 2^{2d}$. Let m be an integer such that $m \geq \lfloor (49d + 80)/17 \rfloor$ if $d < 46$ and $m \geq \lfloor (49d + 57)/17 \rfloor$ if $d \geq 46$.*

For $i = 1, 2, \ldots, m$, let $(\delta_i, f_i, g_i) = \text{divstep}^i(1, f, g)$ and $\mathcal{T}_i = \mathcal{T}(\delta_i, f_i, g_i)$ and $\begin{pmatrix} u_i & v_i \\ q_i & r_i \end{pmatrix} = \mathcal{T}_{i-1}\mathcal{T}_{i-2} \cdots \mathcal{T}_0$. Then $g_m = 0$, $f_m = \pm \gcd(f, g)$ and $v_m g = 2^m f_m \pmod{f}$.

Since f and g are assumed to be coprime, the final values of f and v are respectively f_m and v_m, and p is the inverse of 2^m modulo f, so the following holds:

$$p \cdot v \cdot \text{sgn}(f) \cdot g = (2^{-m})v(\pm 1)g = (\pm 1)(\pm 1) = 1 \pmod{f}.$$

Again we use the presentation from [73] which differs slightly from the one in [35] because of our definition of $\mathcal{T}$.

Algorithm 4: Divsteps for inversion

```
    Input  : Integers $n, \delta, f$ and $g$ such that $f$ is odd
    Output : The integers $\delta_n, f_n$ and $g_n$ and the matrix product
             $\mathcal{T}_n\mathcal{T}_{n-1}\cdots\mathcal{T}_0$
1   $k \leftarrow 0$, $u \leftarrow 1$, $v \leftarrow 0$, $q \leftarrow 0$, $r \leftarrow 1$
2   for $i \leftarrow 1$ to $n$ do
3       if $0 < \delta$ and $g$ odd then
4           $\delta \leftarrow -\delta$, $f \leftarrow g$, $g \leftarrow -f$,
5           $u \leftarrow q$, $v \leftarrow r$, $q \leftarrow -u$, $r \leftarrow -v$
6       $g_0 \leftarrow g \bmod 2$, $\delta \leftarrow \delta + 1$
7       $g \leftarrow \frac{g+g_0 f}{2}$, $u \leftarrow 2u$, $v \leftarrow 2v$
8       $q \leftarrow q + g_0 u$, $r \leftarrow r + g_0 v$
9   return $\delta, f, g, \left(\begin{smallmatrix} u & v \\ q & r \end{smallmatrix}\right)$
```

Algorithm 5: Bernstein-Yang modular inversion algorithm.

```
    Input  : Integers $f$ and $g$ such that $f$ is odd and $\gcd(f, g) = 1$
    Output : Integer $g^{-1}$ such that $gg^{-1} = 1 \pmod f$
1   $d \leftarrow \max(\log_2 f, \log_2 g)$ if $d < 46$ then
2       $n \leftarrow \lfloor (49d + 80)/17 \rfloor$
3   else
4       $n \leftarrow \lfloor (49d + 57)/17 \rfloor$
5   $e \leftarrow ((f + 1)/2)^n \bmod f$
6   $\delta \leftarrow 1$
7   $\delta, f, g, \left(\begin{smallmatrix} u & v \\ q & r \end{smallmatrix}\right) \leftarrow$ Divsteps$(m, \delta, f, g)$
8   return $e \cdot v \cdot \mathrm{sgn}(f)$
```

3.3 Algorithms for Kronecker symbol

We describe two versions of the algorithm: an easier-to-implement full-precision version based on divstep, described in Section 3.3; and a faster word-sized version described in Section 27. Like in modular inversion, we will assume that the first argument to Divsteps is public. It corresponds to the modulus in inversion and the bottom argument in the Kronecker symbol. Hence the algorithms can leak the values of specific bits of f, and the lengths of both arguments f and g, since the number of iterations depend directly on those. Since f is known in advance and $0 \leq g < f$ in the cases of interest within cryptography, there is no impact in security.

Full-precision Divstep version

We start by defining a symbol that extends the Jacobi symbol to include negative numbers in the denominator as $\left(\frac{a}{b}\right) := \left(\frac{a}{|b|}\right)_J$, for integers a, b. Even numbers are also handled by factoring out powers of $\left(\frac{a}{2}\right)_K = \left(\frac{2}{a}\right)_K$ under multiplicativity. This is well-defined, so we have the following version of reciprocity for coprime a and b:

$$\left(\frac{a}{b}\right)\left(\frac{b}{a}\right) = \varepsilon(a,b)(-1)^{\frac{a-1}{2}\frac{b-1}{2}}, \tag{3.1}$$

where $\varepsilon(a,b) = -1$ if both a and b are negative and 1 otherwise. All other properties of the usual Jacobi symbol are preserved (such as multiplicativity in both arguments and periodicity in the numerator).

Now we extend the divstep function to also record information about the value of the Jacobi symbol. For $k \in \{\pm 1\}$ we define:

$$\operatorname{divstep}_k(\delta, f, g, k) =$$
$$\begin{cases} \left(1-\delta, g, \frac{g-f}{2}, (-1)^{\frac{g^2-1}{8}}\varepsilon(g,-f)(-1)^{\frac{g-1}{2}\frac{-f-1}{2}}k\right) & \text{if } \delta > 0 \text{ and } g \text{ odd} \\ \left(1+\delta, f, \frac{g+(g \bmod 2)f}{2}, (-1)^{\frac{f^2-1}{8}}k\right) & \text{otherwise.} \end{cases}$$

In this first case, we swap (f, g) with $(g, -f)$ and employ the laws of quadratic reciprocity to handle even f and multiplicatively accumulate the result into the intermediate value k. In the second case, we apply the second supplement of the law of quadratic reciprocity to handle even g and accumulate the result. Now we can prove that the recurrence computes the extended symbol correctly.

Lemma 1. *For all $n \in \mathbb{N}$ write $(\delta_n, f_n, g_n, k_n) = \operatorname{divstep}_k^n(\delta, f, g, k)$. If we take $k = 1$ and assume that $\gcd(f, g) = 1$, then we have*

$$k_n\left(\frac{g_n}{f_n}\right) = \left(\frac{g}{f}\right).$$

In particular, if $f_n = \pm 1$, then $k_n = \left(\frac{g}{f}\right)$.

Proof. We proceed by induction in n. By assumption $k_0 = k = 1$, so the formula is true when $n = 0$.

If $\delta_{n+1} > 0$ and g_{n+1} is odd, then

$$\begin{aligned} k_{n+1}\left(\frac{2g_{n+1}}{f_{n+1}}\right) &= (-1)^{\frac{g_n^2-1}{8}}\varepsilon(g_n,-f_n)(-1)^{\frac{g_n-1}{2}\frac{-f_n-1}{2}}k_n\left(\frac{g_n-f_n}{g_n}\right) \\ &= \left(\frac{2}{f_{n+1}}\right)k_n\left(\frac{g_n}{-f_n}\right) \\ &= \left(\frac{2}{f_{n+1}}\right)\left(\frac{g}{f}\right) \end{aligned}$$

and the result follows by dividing by $\left(\frac{2}{f_{n+1}}\right)$. Note that we use that f_{n+1} is odd and that f_n and g_n are coprime (since f and g are assumed to be so).

If $\delta_{n+1} \leq 0$ or g_{n+1} is even, then

$$\begin{aligned} k_{n+1}\left(\frac{2g_{n+1}}{f_{n+1}}\right) &= (-1)^{\frac{f_n^2-1}{8}}k_n\left(\frac{g_n+(g_n \bmod 2)f_n}{f_n}\right) \\ &= \left(\frac{2}{f_{n+1}}\right)k_n\left(\frac{g_n}{f_n}\right) \\ &= \left(\frac{2}{f_{n+1}}\right)\left(\frac{g}{f}\right). \end{aligned}$$

and the result follows as before. □

By the main theorem of [35], there is an N such that $(f_N, g_N) = (\pm\gcd(f,g), 0)$ so we get the following corollary.

Lemma 2. *Let f and g be integers with f odd. We have that*

$$\left(\frac{g}{f}\right) = \begin{cases} k_N & \text{if} f_N = \pm 1 \\ 0 & \text{otherwise.} \end{cases}$$

This means that if we iterate divstep$_k$ as many times as is needed for divstep to converge, then we also get an algorithm for computing the extended symbol $\left(\frac{g}{f}\right)$ defined previously.

For simplicity, we will split the resulting algorithm for arbitrary integers f, g in two subroutines. The inner routine will evaluate the extended symbol $\left(\frac{g}{|f|}\right)$ by iterating the divstep$_k$ the required number of times. Algorithm 6 does exactly that, but it also computes the transition matrix. Algorithm 7 is a modification to evaluate the branches in constant time explicitly and to avoid computing the transition matrix that is not needed for the actual evaluation of the Kronecker symbol. We define an operator msb$(\cdot)$ to evaluate the most-significant bit in two's complement representation that coincides exactly with the sign evaluation. The other general cases corresponding to the

Kronecker symbol will be handled in the outer Algorithm 8, which can also be trivially modified to be executed in constant-time with respect to the value of g. In particular, we handle the case $f = 0$ right after counting the number of iterations, and use a temporary variable u to correct the sign when both f and g are negative and when f is even. We argue that Algorithm 8 does not leak information about g when the branches are evaluated in constant time, other than its length when iterating DIVSTEPS$_k$ for the corresponding number of iterations. In the cases of interest to cryptography (Legendre or Jacobi symbol), g is assumed to be reduced modulo f, so the number of iterations is determined by f alone and does not leak information about the length of g.

Algorithm 6: DIVSTEPS$_k$ for Kronecker symbol

```
Input  : Integers $n, \delta, f$ and $g$ such that $f$ is odd
Output : The integers $\delta_n, f_n$ and $g_n$
1  $k \leftarrow 0$, $u \leftarrow 1$, $v \leftarrow 0$, $q \leftarrow 0$, $r \leftarrow 1$
2  for $i \leftarrow 1$ to $n$ do
3      if $0 < \delta$ and $g$ odd then
4          $\delta \leftarrow -\delta$, $f \leftarrow g$, $g \leftarrow -f$,
5          $k = k + ((\lfloor \frac{f}{2} \rfloor \bmod 2) \cdot (\lfloor \frac{g}{2} \rfloor \bmod 2) + 1) \bmod 2$
6          if $f < 0 \wedge g < 0$ then
7              $k \leftarrow k + 1 \bmod 2$
8      $g_0 \leftarrow g \bmod 2$, $\delta \leftarrow \delta + 1$
9      $k = k + ((\lfloor \frac{f}{2} \rfloor \bmod 2) \cdot (\lfloor \frac{f}{4} \rfloor \bmod 2)) \bmod 2$
10     $g \leftarrow \frac{g + g_0 f}{2}$ $u \leftarrow 2u$, $v \leftarrow 2v$
11     $q \leftarrow q + g_0 u$, $r \leftarrow r + g_0 v$
12 return $\delta, f, g, k, \left(\begin{smallmatrix} u & v \\ q & r \end{smallmatrix}\right)$
```

Word-oriented JumpDivstep version

Algorithm 4 can be optimized by observing that computing the ℓ first iterations of DIVSTEPS only depends on the ℓ first bits in f and g. This allows working on smaller numbers and "jumping" through the computations of DIVSTEPS in larger steps (see section 10 in [35]).

This optimization, however, can unfortunately not be naively applied to Algorithm 7. The issue is that going to a lower bit length by truncating, one loses information about the most significant bits of f and g. Fortunately, a trick discovered Hamburg in [67] can be used to circumvent this issue. In [67] it is noted that to compute the contribution of $\operatorname{msb}(f) \wedge \operatorname{msb}(g)$ to k output by Algorithm 7, one only needs to compute the amount of times f changes sign. Now this in itself is still not enough, since f is truncated. Instead it

Algorithm 7: Optimized OptDivsteps$_k$ in constant time

```
   Input  : Integers $n, \delta, f$ and $g$ such that $f$ is odd
   Output : The integers $\delta_n, f_n$ and $g_n$
1  $k \leftarrow 0$
2  for $i \leftarrow 1$ to $n$ do
3      $d_0 = (g \bmod 2) \wedge (\delta > 0)$
4      $\delta = -\delta$ if $(\delta < 0)$
5      $k = k \oplus (1 \oplus (f \gg 1) \wedge (g \gg 1)) \oplus (\mathrm{msb}(f) \wedge \mathrm{msb}(g))$
6      $f \leftarrow g$ if $d_0 > 0$, $g = -g$ if $(d_0 > 0)$
7      $g_0 \leftarrow g \bmod 2$, $\delta \leftarrow \delta + 1$
8      $k = k \oplus (((f \gg 1) \bmod 2) \wedge ((f \gg 2) \bmod 2))$
9      $g \leftarrow (g + g_0 f) \gg 1$
10 return $\delta, f, g, k$
```

turns out that counting the amount of times q changes sign in Algorithm 6 is sufficient for computing the contribution of f's sign changes.

Hamburg defines a 2 by 2 number matrix to be *ratchet*, if it has strictly positive determinant and its second row is positive. Then he proceeds to prove the two theorems:

Theorem 3 (Theorem 1, [67]). *Let M_i be a sequence of ratchet matrices and let $T_i = \begin{pmatrix} a_i & b_i \\ c_i & d_i \end{pmatrix}$ be the sequence of matrices recursively defined by $T_0 = I$ and $T_i = M_i \cdot T_{i-1}$. Let $f, g \in \mathbb{Z}$ with either $f \neq 0$ or $g > 0$ and write $\begin{pmatrix} f_i \\ g_i \end{pmatrix} = T_i \cdot \begin{pmatrix} f \\ g \end{pmatrix}$. If t_j, u_j denote the number of times that f_i, c_i change signs for $i \leq j$, respectively, then $t_j - u_j \in \{0, 1\}$.*

Theorem 4. *For all $n \in \mathbb{N}$, $\mathcal{T}_n$ is ratchet.*

Using these two theorems, one can apply the jumpdivstep optimization to Algorithm 6. To this defines a variant of Algorithm 7 which does not use the most significant bits of f or g, but rather counts the sign changes of q to compute the same contribution. Let us refer to this version as JumpDivsteps $_k$.

For a processor with word size w, an optimal value for ℓ would be the largest integer smaller than $w-2$ that divides the required number of iterations. We also define Algorithm 9 as an adaptation of Algorithm 8 that uses the modified JumpDivsteps $_k$. Algorithm 9 computes the number of iterations using the half-delta optimization [109], that reduces the number of iterations by approximately 18% on average, then handles the corner cases just like Algorithm 8. It finishes by invoking JumpDivsteps$_k$ as a subroutine.

Algorithm 8: Kronecker symbol based on Divsteps

```
Input  : Integers $f$ and $g$
Output : Kronecker symbol $\left(\frac{g}{f}\right)_K$
1  $d \leftarrow \max(\log_2 f, \log_2 g)$
2  if $d < 46$ then
3      $n \leftarrow \lfloor (49d + 80)/17 \rfloor$
4  else
5      $n \leftarrow \lfloor (49d + 57)/17 \rfloor$
6  if $f = 0$ then
7      if $g = \pm 1$ then
8          return 1
9      else
10         return 0
11 $u \leftarrow 1$
12 if $(f < 0)$ and $(g < 0)$ then
13     $u \leftarrow -1$                              // handle negative operands
14 while $f \bmod 2 = 0$ do
15     $f = f/2$
16     if $g \bmod 2 = 0$ then
17         $u \leftarrow 0$                           // return 0 no matter k below
18     if $g \bmod 8 = \pm 3$ then
19         $u \leftarrow -u$                          // factor out (g|2) and flip u
20 $\delta \leftarrow 1$
21 $\delta', f', g', k \leftarrow \text{OptDivsteps}_k(n, \delta, |f|, g)$
22 if $|f'| \neq 1$ then
23     return 0
24 if $k = 1$ then
25     return $-u$
26 else
27     return $u$
```

Formal Verification of Hamburg's Paper.

The BY inversion algorithm has been formalized in the Coq Proof Assistant in related work [73]. We build on top of that framework to give formal guarantees about the Kronecker computation. The contributions of this development are publicly available[1].

[1] `https://github.com/bshvass/by-inversion/blob/jacobi2/src/Ratchet.v`

Algorithm 9: Kronecker symbol based on JumpDivsteps

```
Input   : Integers $f$ and $g$
Output  : Kronecker symbol $\left(\frac{g}{f}\right)_K$
$d \leftarrow \max(\log_2 f, \log_2 g)$
$n = \lfloor (45907d + 26313)/19929 \rfloor$
if $f = 0$ then
    if $g = \pm 1$ then
        return 1
    else
        return 0
$u \leftarrow 1$
if $(f < 0)$ and $(g < 0)$ then
    $u \leftarrow -1$                          // handle negative operands
while $f \bmod 2 = 0$ do
    $f = f/2$
    if $g \bmod 2 = 0$ then
        $u \leftarrow 0$                       // return 0 no matter k below
    if $g \bmod 8 = \pm 3$ then
        $u \leftarrow -u$                      // factor out (g|2) and flip u
$\delta \leftarrow 0$
$\delta', f', g', k \leftarrow \text{JumpDivsteps}_k(n, \delta, |f|, g)$
if $|f'| \neq 1$ then
    return 0
else
    return $u \cdot k$
```

Concretely, we formalize the results in Hamburg's paper [67], which are necessary for the correctness of the JumpDivsteps implementation of the Kronecker computation.

To formalize Theorem 3 we need some auxiliary functions. We define $\mu : \mathbf{Z} \times \mathbf{Z} \to \{0, 1\}$ by $\mu(a, b) = 1$ if a and b have different signs and 0 otherwise; this is used to count sign changes. We also define ε as in Section 3.3. Using these we can rephrase Theorem 3 as follows.

Theorem 5. *Let $i \in \mathbb{N}$ and c_j and f_j be defined as in Theorem 3 for all $j \in \{0, \ldots, i\}$. Then*

$$\sum_{j=0}^{i-1} \mu(f_{j+1}, f_j) = \sum_{j=0}^{i-1} \mu(c_{j+1}, c_j) + (\mu(f_i, f_0) + \mu(c_i, c_0) \bmod 2).$$

The formalization of this proposition in Coq is the following:

```
Definition ratchet_spec M i :=
  κ M i =
    (κ' M i) + ((μ (f M i) (f M 0)) + (μ (c M i) (c M 0))) mod 2.
```

where κ and κ' correspond to the sums in Theorem 5.

The proof proceeds much like in [67]. We split each matrix M_i into a product of three matrices, each of which are either upper triangular or rotation matrices. We implement this as a map t from sequences of matrices to sequences of matrices, defined in Coq as

```
Definition t (M : nat → mat Q) :=
  fun (i : nat) ⇒
    let '(a, b, c, d) := M (i / 3)%nat in
    if (decide (b = 0))
    then match mod3dec i with
         | inright _ ⇒ I
         | inleft pf ⇒ match pf with
                       | right _ ⇒ I
                       | left _ ⇒ (a, b, c, d)
                       end
         end
    else match mod3dec i with
         | inright _ ⇒ (b, 0, d, 1)
         | inleft pf ⇒ match pf with
                       | right _ ⇒ (0, 1, -1, 0)
                       | left _ ⇒ (a * d / b - c, 0, a / b, 1)
                       end
         end.
```

where mod3dec decides whether a number is $0, 1$ or 2 modulo 3. Note that all the matrices in the sequence $t(M)$ are either upper triangular or rotation matrices. In Coq we have that:

```
Lemma t_lt_or_rot M : forall i, lt_or_rot (t M i).
```

We prove that the sequence $t(M)$ actually decomposes the sequence M, such that multiplying three consecutive matrices in $t(M)$ (starting at a multiple of 3) gives a corresponding matrix in M:

$$(t(M))_{3\cdot i+2} \cdot (t(M))_{3\cdot i+1} \cdot (t(M))_{3\cdot i} = M_i.$$

We then prove that Theorem 5 is satisfied *if* M is a sequence of matrices which all are upper triangular or rotation matrices. This is the theorem

```
Theorem aux M :
  f0 ≠ 0 \/ g0 > 0 →
  (forall i, ratchet (M i)) →
  (forall i, lt_or_rot (M i)) →
  (forall i, ratchet_spec M i).
```

and by applying the decomposition function t, we get Theorem 5 in Coq

```
Theorem thm M :
  f0 ≠ 0 \/ g0 > 0 →
  (forall i, ratchet (M i)) →
  (forall i, ratchet_spec M i).
```

3.4 Implementation and experimental results

Our collection of experimental results is split in two parts. In the first subsection we study the performance of the Kronecker symbol in isolation, by benchmarking the Legendre symbol across multiple parameters from curve-based cryptography. In the second, we observe the performance of the algorithm in the context of elliptic curve hashing.

Legendre symbol

For benchmarking, we specialize for the Legendre symbol, which is the most interesting case of cryptographic interest. We have implemented our algorithms for a range of parameters commonly used in both elliptic curve cryptography (ECC) and pairing-based cryptography (PBC) settings. For the elliptic curve cryptography setting, we chose the 256-bit primes $2^{255} - 19$ and $2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$ labeled by their respective named curves Curve25519, and `secp256k1` at the 128-bit security level. We note however that the specific prime shape does not affect the performance of our algorithms. For pairing-based cryptography, we selected three different primes to define the base field for Barreto-Lynn-Scott (BLS) curves [21] at multiple security levels. In particular, those are respectively the prime moduli underlying the BLS curves with embedding degree 12 undergoing standardization at 128-bit security [84], the 509-bit prime for BLS curves with embedding degree 24 proposed for 192-bit security [13, 19], and the 575-bit prime for BLS curves with embedding degree 48 proposed for 256-bit security [82].

The code was developed on a fork of the RELIC toolkit [11] in the C programming language, due to its strong support for curve-based cryptography. For reference, RELIC has a number of pairing-friendly curves implemented efficiently, having set multiple speed records for their computation. There is support for advanced elliptic curve hashing algorithms [108] and handwritten Assembly acceleration for the field arithmetic. This approach allowed comparisons between our algorithms for Legendre symbol computation and the other algorithms already implemented in the library.

In the implementation, we followed the library standard and implemented arithmetic using saturated arithmetic, with 64-bit limbs. We also implemented support for Pornin's algorithm [90] by incorporating it from the optimized implementation found in the `blst` [102] multi-lingual library implementing Boneh-Lynn-Shacham short signatures [47] over the BLS12-381 elliptic curve. For fairness of comparison, we performed minor tweaks in this implementation to enjoy the same low-level field arithmetic primitives from RELIC as our algorithms.

Benchmarking measurements were taken by computing the average latency of running the code for 10^4 consecutive executions on an Intel Kaby Lake Core i7-7700 CPU at 3.60GHz. The compilers used were GCC version 12.2 and `clang` 14.0.6, with optimization flags `-O3 -funroll-loops -march=native -mtune=native`. Following benchmarking conventions [2], TurboBoost and Hyper-Threading were disabled for higher stability.

Our results are presented in Table 3.1. In the table, the first part sets the baseline for comparison. We included the highly-optimized variable-time implementation of the Jacobi symbol from the GNU MP library version 6.2.1 [63], and the exponentiation approach by Euler's criterion using a generic constant-time algorithm. The particular choice configured for the latter was a sliding-window exponentiation with precomputation table of 32 elements, as per the default RELIC configuration. These two algorithms represent the conventional approaches for variable-time and constant-time implementation, and respectively set both a lower bound for aggressively optimized variable-time code, and an upper bound for generic constant-time approach. The next part brings what is arguably the current state-of-the-art-algorithms for computing the Legendre symbol in constant time. The first line for related work labeled with (C+ASM) contains timings for Pornin's algorithm [90] using RELIC's Assembly acceleration for the various fields. The line labeled with (pure ASM) constains pure Assembly implementations for Pornin's algorithm found in the `blst` library for BLS12-381, or an implementation that we built ourselves. In the case of Curve25519 and secp256k1, the pure ASM implementation was built by computing shorter addition chains for $\frac{p-1}{2}$ using [83], combined with Assembly code for squarings/multiplications from [87]. The last part of the table show the performance of our algorithms, both the basic DIVSTEP approach and the

[2] `https://bench.cr.yp.to/supercop.html`

JumpDivstep optimization.

From the baseline entries in the first part of the table, we can observe the massive difference in performance between the two approaches, ranging from a 4- to 45-factor. These illustrate how poorly the generic exponentiation-based approach performs across the various parameters. It is at best competitive with the basic Divstep implementation, but only for the shorter parameters. Starting with the ECC primes, the sparse Curve25519 prime benefits a lot both the variable-time GMP implementation and the addition-chain based exponentiation approach, exactly as expected. For the constant-time implementations, the related work in pure ASM is the fastest, with a 11% speedup over our JumpDivstep algorithm (8,846 instead of 9,891 cycles, respectively). When we increase the prime length by 1 bit but move to the secp256k1 field, where the modulus is not as sparse and arithmetic is not as efficient, the addition-chain approach is penalized and the numbers are reversed: now JumpDivsteps becomes 11% faster (9,756 instead of 10,971 cycles, respectively). Now moving to the dense pairing-friendly primes, the generic constant-time exponentiation based approach does not scale well and cannot be easily optimized with shorter addition chains, so the basic Divsteps approach becomes competitive by providing a speedup between 1.5 and 2.5. The real competition is between Pornin's algorithm in (C+ASM) versus JumpDivstep, in which we obtain a consistent improvement of 17.8-25.7% across all fields. In the particular case of the BLS12-381 prime which is heavily optimized in `blst`, the fully unrolled pure ASM implementation of the same algorithm gives a substantial 42% improvement over the (C+ASM) version, and becomes 23% faster than our JumpDivstep approach. However, we believe that comparison is not entirely fair and our JumpDivstep approach would benefit similarly from the same-level of optimization, to the point of providing a similar improvement we obtained in comparison to the (C+ASM) version.

Table 3.1: Benchmarks of different approaches for Legendre computation over different fields. Numbers in bold are the fastest for group of implementations in this work or related work among the different compilers for a certain choice of prime. With exception of variable-time GMP, all implementations run in constant time.

	Curve25519		secp256k1		BLS12-381		BLS24-509		BLS48-575	
	`gcc`	`clang`	`gcc`	`clang`	`gcc`	`clang`	`gcc`	`clang`	`gcc`	`clang`
Variable-time GMP	2,702	2,692	7,777	7,820	12,007	11,930	16,120	16,222	18,353	18,314
Generic constant-time exp.	37,200	36,778	34,264	34,779	110,286	112,987	228,315	226,817	305,544	308,035
Related work (C+ASM)	11,336	11,322	11,271	**11,217**	17,323	17,460	24,357	**24,046**	27,565	**27,419**
Related work (pure ASM)	8,847	**8,846**	**10,971**	10,979	**10,000**	10,003	–	–	–	–
This work (Divsteps)	35,727	50,321	35,773	49,631	69,164	73,148	88,873	105,857	129,074	129,060
This work (JumpDivsteps)	**9,891**	10,277	**9,756**	10,182	**13,044**	**14,074**	**17,865**	19,101	**22,643**	23,760

Application to elliptic curve hashing

Hashing arbitrary strings to elliptic curve points is a fundamental operation in many cryptographic protocols, for example the pairing-based short signature scheme [47]. Various techniques have been proposed to perform this operation efficiently for an elliptic curve in short Weierstrass form $E : y^2 = x^3 + ax + b$ over a field of characteristic p, starting from the basic try-and-increment. With this approach, a cryptographic hash function is used to hash the string to the x-coordinate of a point, and then the value $z = x^3 + ax + b$ is tested for quadratic residuosity. In the positive case, the y-coordinate is computed as $\sqrt{z}$ in $\mathbb{F}_p$; otherwise x is incremented and another quadratic residuosity test performed. However, there are several problems with this approach: the distribution of outputs is uncertain; and the algorithm is intrinsically variable-time, which might leak information about the string being hashed.

A more principled alternative approach was proposed by Brier et al. [50]. This approach is composed in two stages, first the message m is hashed to a field element in $\mathbb{F}_p$, and then the Shallue-van de Woestijne-Ulas (SWU) encoding $f : \mathbb{F}_p \to E(\mathbb{F}_p)$ is used to map the hash output to a point in the elliptic curve. Given two cryptographic hash functions h_1 and h_2 mapping from arbitrary strings to $\mathbb{F}_p$, the complete process constructs the elliptic curve hash function $H : \{0,1\}^* \to E(\mathbb{F}_p)$ as simply $H(m) = f(h_1(m)) + f(h_2(m))$, and then multiplying the result by a cofactor to obtain a point in the right prime-order subgroup. From the security point of view, it is known that the map H will satisfy the standard security notion of random oracle indifferentiability when both h_1 and h_2 are indifferentiable from random oracles as well. Unfortunately, this approach is quite expensive in which it needs two evaluations of the encoding map f (plus a minor elliptic curve point addition) to perform its role. Most follow-up work has focused on specialized and optimizing this approach to different classes of elliptic curves, for example BLS12 [108], by accelerating the computation of the map f and proposing variants easier to implement in constant time.

The performance drawback of previous approaches was recently improved in the research literature with the SwiftEC [54] algorithm. In this new work, Chávez-Saab et al. reformulate the encoding map f as a parameterized family of functions, such that $f_{h_2(m)}(h_1(m))$ is indifferentiable from a random oracle if h_1 and h_2 are indifferentiable as well. The algorithm constructs a conic S admitting a two-parameterization over $\mathbb{F}_p$, and uses it to obtain three candidate coordinates x_1, x_2, x_3 for the x-coordinate of a point in E, such that at least one of them will be such that $(x_i^3 + ax_i + b)$ is a square. The latter will be exactly the x-coordinate, and a matching y-coordinate is computed by taking a square-root and choosing the sign based on an additional bit. This idea saves the evaluation of one encoding function, which in turn saves on square-root extractions and quadratic residuosity tests. The algorithm is also considerably easier to implement in constant-time than previous work, providing additional benefits beyond just the performance improvement.

We implemented the SwiftEC construction for the parameters in Table 3.1. We restricted the implementation for the cases where the JumpDivstep approach for computing the Legendre symbol improved on the previous state-of-the-art, which means all primes except Curve25519. Coincidentally, those were the exact parameters where the corresponding elliptic curve was such that $a = 0$, which represents a simpler and faster special case of SwiftEC. Algorithm 10 presents the algorithm, which executes one inversion, two Legendre symbol computations, one square root and other minor operations in the finite field (additions, squarings and multiplications). For a fully constant-time implementation, we would need to remove the branch to handle the case in line 8, which occurs with at most negligible probability. This can be easily handled by setting P to the point at infinity, conditionally randomizing w and making the assignment in line 22 conditional as well. We omit these details to simplify the description, with a remark that most protocols would reject the point at infinity as an output anyways, requiring another hashing attempt.

Table 3.2: Benchmarks of different approaches for hashing to elliptic curves defined over different fields. Numbers in bold are the fastest for group of implementations in this work or related work among the different compilers for a certain choice of prime. With exception of variable-time SWU, all implementations run in constant time.

	secp256k1		BLS12-381		BLS24-509		BLS48-575	
	`gcc`	`clang`	`gcc`	`clang`	`gcc`	`clang`	`gcc`	`clang`
Variable-time *SWU + SQRT*	361,170	362,420	629,625	624,955	1,071,457	1,093,585	1,641,249	1,595,068
SwiftEC+Exponentiation	260,834	261,456	560,359	563,114	928,453	928,714	1,372,557	1,375,696
SwiftEC+JumpDivsteps	**141,181**	142,893	**365,370**	370,817	515,451	**514,849**	**806,602**	810,699

Table 3.2 has the resulting timings, using the same benchmarking machine and iterations as described in the previous subsection. In the table, the first line represents the baseline for comparison established by RELIC's implementation of the SWU approach [108] in variable-time to save square root computations. This implementation computes modular square roots using exponentiation by $\frac{p+1}{4}$, relying on the fact that all benchmarked parameters have $p \equiv 3 \pmod 4$. In the second line, we have a naive implementation of the SwiftEC algorithm using Legendre symbols computed through Euler's criterion implemented in constant time using exponentiation. In the last line, we present numbers for the same implementation, but now using our algorithm for faster evaluation of the Legendre symbol using JumpDivsteps. In comparison to the variable-time baseline in the first line, SwiftEC+JumpDivstep in constant-time improves timings for hashing by 41% to 61%, depending on the parameter. The highest speedup is for the `secp256k1` prime modulus. In comparison to the SwiftEC algorithm implemented with baseline Legendre symbols, the speedups are

reduced to the range between 34% and 46%.

Algorithm 10: Special case of SwiftEC with $a = 0$

```
Input  : Elliptic curve $E(\mathbb{F}_p) : y^2 = x^3 + b$ with order $n = hr$, for
         prime $r$ and cofactor $h$, integer $\lambda$ such that $\lambda^2 \equiv -3 \pmod{p}$,
         hash functions $h_1, h_2 : \{0,1\}^* \to \mathbb{F}_p$, $h_s : \{0,1\}^* \to \{0,1\}$
Output : Point $P \in E(\mathbb{F}_p)$ of order $r$
1  $t \leftarrow h_1(m), u \leftarrow h_2(m), s \leftarrow h_s(m)$
2  $x_1 \leftarrow (u^3 + b - t^2)$
3  $y_1 \leftarrow 2t^2 + x_1$
4  $z_1 \leftarrow 2tu\lambda$
5  $x_1 \leftarrow u \cdot \lambda \cdot x_1$
6  $v \leftarrow z_1(x_1 - uy_1)$, $w \leftarrow 2y_1z_1$
7  if $w = 0$ then
8      $P \leftarrow \infty$                                   // point at infinity
9  else
10     $w \leftarrow w^{-1}$
11     $x_1 \leftarrow vw$
12     $x_2 \leftarrow -(u + x_1)$
13     $x_3 \leftarrow (4y_1^2w)^2 + u$
14     $y_1 \leftarrow x_1^3 + b$, $y_2 \leftarrow x_2^3 + b$, $y_3 \leftarrow x_3^3 + b$
15     $c_2 \leftarrow \left(\frac{u}{p}\right)_L$, $c_3 \leftarrow \left(\frac{v}{p}\right)_L$
16     $x_1 \leftarrow x_2$ if $c_2 = 1$                       // conditional copies
17     $y_1 \leftarrow y_2$ if $c_2 = 1$
18     $x_1 \leftarrow x_3$ if $c_3 = 1$
19     $y_1 \leftarrow y_3$ if $c_3 = 1$
20     $y_1 \leftarrow \sqrt{y_1}$
21     $y_1 \leftrightarrow -y_1$ if $w + s \bmod 2 = 1$        // choose y-coordinate
22     $P \leftarrow h \cdot E(x_1, w)$                        // multiply by cofactor
23 return $P$
```

The experimental results show that our proposed algorithms for evaluating the Legendre symbol as a special case of the Kronecker symbol can substantially accelerate computations in curve-based cryptography. While hashing to elliptic curves is used for illustration, any other evaluation of the symbol in a context involving a secret numerator operand can perform more efficiently by using our techniques.

3.5 Conclusion

In this paper, we generalized the efficient and constant-time BY algorithm for modular inversion to compute the Kronecker symbol. We defined two versions of the algorithm, with different trade-offs: a full-precision version that is easier to implement and a faster word-oriented variant. After proving correctness of the introduced algorithms using a formalization of the BY theory within the Coq proof assistant, we produced an optimized implementation of the algorithms inside the RELIC library. Benchmarking our implementation revealed that the new algorithm is up to 14 times faster than the conventional exponentiation approach, and 25.7% faster than the previous state of the art; and improved a recent approach for hashing to elliptic curves by approximately 40%.

Chapter 4

The last yard: formal specification and security proofs for high-speed cryptography

Benjamin S. Hvass, Aarhus University
Lasse Letager Hansen, Aarhus University
Philipp Haselwarter, Aarhus University
Theo Winterhalter, Inria Saclay
Bas Spitters, Aarhus University

Abstract

While the field of high-assurance cryptography matures, a unified framework for end-to-end verification without gaps is still missing. To address this, we connect three tools: (1) the emergent specification language Hacspec; (2) the Jasmin compiler to produce high-assurance efficient assembly code; (3) the SSProve foundational framework for end-to-end security and correctness verification of cryptographic implementations. SSProve has been used in the past for the modular verification of the security of cryptographic protocols. We provide both an imperative and a functional embedding from Hacspec to SSProve and automate equivalence proofs between those two translations. We further extend SSProve with a semantics-preserving embedding of Jasmin programs, allowing one to reason about high-performance cryptographic code. To evaluate our toolchain, we connect an existing AES Jasmin implementation to a security proof in SSProve and also partially relate it to a Hacspec specification.

This case study demonstrates how to leverage the modularity of SSProve's language by breaking the game-hopping refinement proofs into composable modules.

4.1 Introduction

In recent years, in the field of high-assurance cryptographic software, there has been an effort to close the gaps in the verification toolchains [8, 32]. However, a few important gaps in verification remain.

First, there is the specification gap. When considering formal proofs, one should consider whether one has proved the correct statement. Hacspec [85] addresses this problem by identifying a simple subset of the Rust programming language which is both understandable for the ordinary developer and cryptographer, while at the same time providing a precise (functional) semantics.

Second, high assurance implementations often want to use Jasmin, since unverified C compilers cannot always be trusted [98], while CompCert [76] (only) provides code comparable to code generated by GCC at optimization[1] level 1.

In the fundamental 'last mile' paper [8], the verification of Jasmin programs uses a mixture of Coq and EasyCrypt, with no precise formalized connection between the two. In this paper, we provide a toolchain that closes such verification gaps by replacing EasyCrypt by SSProve, and moreover, connecting to Hacspec. Importantly, this facilitates using large existing mathematical libraries in Coq, which are not available in EasyCrypt. Moreover, the verification of jasmin programs often starts by proving the equivalence of a functional specification with an imperative reference implementation. We automate this process by using Hacspec.

Third, cryptographic primitives do not live in isolation but are part of larger protocols. So, one needs some way to connect Jasmin code with a higher level language, such as Rust. Jasminify is a script that replaces Rust code by Jasmin code. However, it comes with no correctness guarantees. Our toolchain facilitates proving source-level correctness of such a transformation for cryptographic protocols written in Hacspec.

Contributions

In this paper, we contribute the following:

- The first translation of Hacspec into an imperative programming language in a proof assistant, including automatic proofs of program equivalence of a functional and imperative program embedding (Section 4.4 and 4.4)
- Connect SSProve to the verified Jasmin compiler via a verified translation from the Jasmin source code to SSProve.
- We evaluate our toolchain by proving security of the Hacspec specifications of OTP and AES in SSProve. Then we prove that the Jasmin implementations satisfy these specifications.

[1] `https://www.absint.com/factsheets/factsheet_compcert_c_web.pdf`

4.2 From specification to verified, efficient implementation

Given an efficient, low-level implementation in Jasmin and a reference implementation in Hacspec, we translate both to SSProve, an imperative language embedded in Coq. Once translated, the programs can be compared and we can prove probabilistic properties about them using probabilistic relational Hoare logic in Coq.

We begin by illustrating our methodology using a simple example: *exclusive or* (XOR), which we will use in the one-time-pad (OTP) in 4.2. A Jasmin implementation of this function could look as follows:

```
export fn xor(reg u64 x, reg u64 y)
  -> reg u64 {
  reg u64 r;
  r = x;
  r ^= y;
  return r;
}
```

which reads two variables x and y stored in registers (as specified by the `reg` keyword) and writes the XOR of x and y to another register.

Translation to Coq

The first step in our tool chain is to get the Coq AST of the jasmin source. We obtain the function's AST in Coq by using a pretty printer from the internal extracted AST generated by the Jasmin compiler to a proper Coq term; this boils down to basically de-extracting the extracted Coq term from OCaml. Then we use our translation from Jasmin functions to SSProve codes and get the following SSProve function.

```
Definition JXOR id0 w1 w2 :=
  #put x := w1 ;;
  #put y := w2 ;;
  #put r := w1 ⊕ w2 ;;
  r0 ← get r ;;
  ret [('word U64 ; r0)].
```

Note that this is not the literal output of the translation, but rather the result of some careful (but semi-automated) unfolding and rewriting. Note also that the code takes an id, `id0`, as input: This determines which locations on the heap it will use for its local memory.

Equivalence to reference

The reference implementation is given in the specification language Hacspec and is in this case quite similar to our ‘efficient’ implementation.

```
fn xor(x : u64, y : u64) -> u64 {
     x ^ y
}
```

This is similarly translated to a SSProve function `hacspec_xor` and now that we have both translations, we can state and prove the theorem, that they should be equivalent in our program logic.

```
Lemma xor_equiv : ∀ id0 w1 w2,
  ⊢ ⦃ λ '(h₀, h₁), ⊤ ⦄
    res ← JXOR id0 w1 w2 ;;
    ret (hdtc res)
    ≈
    hacspec_xor w1 w2
  ⦃ λ '(v₀, h₀) '(v₁, h₁), v₀ = v₁ ⦄.
```

Here, `hdtc` simply takes the head of the returned list of results. The pre- and postconditions are predicates over pairs of heaps and pairs of value-heap pairs (resp.) since we wish to reason about stateful programs. This particular equivalence only relates the return values of the two programs, so, we do not provide guarantees about how the two program uses memory. In particular, they might use different locations to store their intermediate values.

Securely implement OTP

We now prove security properties of our jasmin program. The property we are interested in is perfect security of a OTP scheme using the Jasmin implementation of XOR.

To this end, we first need to define some terminology. In SSProve a *package* is a finite set of codes which might contain calls to uninstantiated external codes. The set it implements is called its *exports* and the set on which it depends its *imports*. A *game* is a package with no imports and a *game pair* is a pair of games which export the same procedures. These can be used to model cryptographic games, e.g. a game pair might consist of a real encryption scheme and an oracle: These have the same interfaces but different implementations.

Returning to the OTP example, we need to define the game pair consisting of an implementation of OTP using the Jasmin code and an implementation which we know to be secure.

The Jasmin game is the package `JOTP_real` exporting the single code:

```
Definition JOTP id0 m :
  k_val ← sample uniform (word n) ;;
  JXOR id0 m k_val.
```

The implementation for which we already have a security proof is the package `OTP_real` exporting the single code:

```
Definition OTP m :
  k_val ← sample uniform (word n) ;;
  ret m ⊕ k_val.
```

This game is already proven to be indistinguishable under chosen plaintext attack (IND-CPA) in the SSProve library. This is done by proving that the advantage of an attacker in distinguishing between `OTP_real` and a game `OTP_ideal`, where the input is disregarded and a random message is encrypted, is zero.

If we can prove that `JOTP_real` is perfectly indistinguishable from `OTP_real`, then we can combine the two results using the triangle inequality for advantages of games (Lemma 1 in [1]) and prove that an adversary also cannot distinguish between `JOTP_real` and `OTP_ideal`, i.e. that the Jasmin implementation is IND-CPA.

That is, we only need to prove the following theorem:

```
Lemma JOTP_OTP_perf_ind id0 :
  JOTP_real id0 ≈₀ OTP_real.
```

where $\approx_0$ means that the advantage of an adversary trying to distinguish between the two games is zero. To prove this lemma we utilize Theorem 1 from [1], which allows us to conclude if we can prove the following code equivalence for all m and some *stable invariant* `inv`:

```
  ⊢ ⦃ λ '(s₀, s₁), inv (s₀, s₁) ⦄
  JOTP id0 m ≈ OTP m
  ⦃ λ '(b₀, s₀) '(b₁, s₁), b₀ = b₁ ∧ inv(s₀,s₁) ⦄.
```

For the precise definition of stable invariant see 4.2 in [1]. In our case we can use the invariant `heap_ignore`, which asserts that both heaps are preserved during execution, if the locations used by `JXOR` are ignored.

Combining this result with the already established security of `OTP_real` we get security of `JOTP_real`.

```
Theorem unconditional_secrecy_jas :
  forall LA A,
    fdisjoint LA xor_locs →
    ValidPackage LA
      [interface #val #[i1] : 'word → 'word ] A_export A →
    Advantage IND_CPA_jasmin A = 0.
```

That is, for all adversaries `A` and regions of adversarial memory `LA`, if the adversary cannot use the same locations as `JXOR` then their advantage in distinguishing between `JOTP_real` and `OTP_ideal` is zero.

4.3 Background

Hacspec

Hacspec is a **H**igh **A**ssurance **C**ryptography **SPEC**ification language [85] aiming to be the connection between programmers, cryptographers and proof engineers. It is a way to make internet standards, such as IETF and NIST, machine-readable. Hacspec is constructed as a subset of Rust, a programming language familiar to cryptographic engineers. The Hacspec language was carefully crafted to have a functional semantics in which assignments are translated to let-expressions. The Hacspec tool comes with translations to several proof assistants, currently F*, Coq and EasyCrypt. As such it is a convenient tool to share specifications across provers. This allows one to safely combine code generated from these proof assistants. As these backends have been used before, over time this gives us confidence in the translation.

The Hacspec language Unlike Rust, Hacspec comes with a precise operational semantics. The soundness of the translations with respect to the operational semantics is 'obvious'. However, this has not been proven formally. Because the Rust language is currently not specified, Hacspec can be seen as one proposed precise semantics for this subset of Rust. Other such proposals include those of Denis et al. [57], Ho and Protzenko [69].

Because Hacspec has translations to multiple backends/tools/libraries, all the translations of specifications/reference implementations in Hacspec can be used to either prove properties of the protocol, or to prove equivalence with an optimized implementation.

The Hacspec library Hacspec is aimed towards specifying cryptography, and therefore provides a builtin library that implements common functionalities needed by cryptographers. This includes:

- Modular Arithmetic Integers;

- Machine Integers;
- Fixed-length arrays and vectors.

Hacspec adds some type constructors for commonly used types in cryptographic specifications, e.g. fixed-length arrays, modular natural integers and sequences. These types have a special semantic in the backends. Sequences and arrays replace vectors in the subset of Hacspec, as we want types with known size at compile time, to get a safe and fast implementation of the Copy trait.

Extending Hacspec All additions to the Hacspec library will be immediately available to all the backends. We envision some small extensions to the language in the future.

Currently, all Hacspec backends use a functional semantics. However, both in EasyCrypt and in Coq, one could also choose to use a translation to an embedded imperative language. We will explain how to do so in Section 4.4.

Jasmin

Jasmin is a low-level language designed for implementing high-speed cryptography, with a verified compiler supporting arm and x86 output. The language has a formal semantics in The Coq Proof Assistant, and this is leveraged by the jasmin compiler, which is also implemented in Coq. The compiler is verified, in the sense that it preserves the semantics of its Jasmin source to its target (currently x86).

Jasmin allows the programmer to have control over the compiled assembly, while still providing some high-level mechanisms. For instance, the programmer can specify whether local variables should be stored in registers (using the `reg` keyword) or on the stack (using the `stack` keyword). To allow this fine grained control, Jasmin requires the programmer to do spilling by hand.

Jasmin offers high-level mechanisms such as for- and while-loops, and both inlined and non-inlined function calls.

SSProve

As we already hinted at, Abate et al. [1] introduced SSProve as a Coq library for specifying and proving security properties of probabilistic stateful programs. We will introduce the concepts needed to understand the current paper. An extensive overview of the tool can be found in the journal version of the SSProve paper [68].

Code In this paper we will de-emphasize the probabilistic capabilities of SSProve as they are not currently reflected in Jasmin. They are only needed when connecting implementations to security proofs. Thus, for our purposes, SSProve essentially embeds a stateful language inside Coq using a monad we call `raw_code`. In `raw_code A` one can (1) embed any pure value `x` of type `A` using `ret x`; (2) read from a memory location `ℓ` and use the read value in a continuation `k`, written `x ← get ℓ ;; k x`; (3) write a value `v` to a memory location `ℓ` and then continue with `k`, written `#put ℓ ;; k`; (4) combine two programs using a bind operator that we write `x ← u ;; k x`.

Memory model Memory locations consist in a natural number and a type that together serve as an index in a global shared memory. This global state is valid when all locations point to values of the same type. Note that to be able to use the type in the key of the memory, we must in fact use codes of types; since SSProve is built for probabilistic code, these codes represent types on which one may build (discrete) distributions. Technically, they are `choice_type` in the sence of [81]. For our purposes, it is sufficient to note that this type includes[2] all the types needed to represent Jasmin programs. Memory is simulated using a structure we call `heap`, essentially a map from locations to values. We would like to stress the fact that the memory is *global* rather than local to each program. This means that when generating programs one must take care not to generate overlaps by ensuring disjoint locations. We will address this point when talking about the translation from Jasmin to SSProve (Section 4.5).

Packages Another defining feature of SSProve is that of packages. Since we currently do not use them extensively, we will only introduce them briefly. Packages are collections of programs that might all refer to the same set of locations and invoke certain procedures that are part of an *import interface*. The signature of this collection defines the *export interface* of the package. Packages can thus be combined modularly to create bigger programs. For instance, one package can be linked to another one that implements its import interface or they can be composed in parallel to export the union of their respective export interfaces. This is very useful in proofs of composed protocols as described by Brzuska et al. [52].

Relational Hoare logic Finally, SSProve features a (probabilistic) relational Hoare logic that lets us prove relational properties of programs. Once again, we will focus on the stateful but deterministic fragment. In this program logic we prove judgments of the form

$$\vdash \ \{\phi\} \ c_0 \ \sim \ c_1 \ \{\psi\}$$

where c_0 and c_1 are two programs we wish to compare and ϕ and ψ are respectively a pre- and a postcondition relating (1) the initial heaps (for ϕ);

[2]In fact we extended the `choice_type` universe with sums, lists and words. Jasmin's types are words and arrays of words. Our translated Jasmin functions return a list of values.

(2) the final heaps and final return values of both programs (for ψ). In the case of deterministic programs, this is equivalent to: for all initial memory states m_0 and m_1 such that $\phi(m_0, m_1)$ holds, running c_i in state m_i will yield final state m'_i and final value v_i such that $\psi(m'_0, v_0)(m'_1, v_1)$ holds.

SSProve proves a number of rules for this logic and provides tactics to facilitate writing proofs. Moreover, one can also fall back on the semantics above to prove judgments; see Haselwarter et al. [68].

4.4 Hacspec & SSProve

Hacspec facilitates writing a specification (reference implementation) which one can then prove to be equivalent to an efficient implementation. We further this goal by adding a translation from Hacspec to the cryptography verification library SSProve. This translation is done in two parts: (1) a pure translation, which wraps the existing Coq translation to SSProve; (2) a stateful translation, which uses the SSProve imperative language. We also generate a proof of functional equality between the two translations. In Section 4.5, we show how to prove functional equivalence between a Jasmin and a Hacspec implementation.

In EasyCrypt [8], a correctness proof of an efficient implementation often starts by proving the equivalent between a functional implementation and an unoptimized imperative implementation. In Section 4.4 we show how Hacspec allows us to automate this step.

The pure Translation

Hacspec already comes with a Coq backend [95]. We make a minor change to this backend to facilitate connecting to Jasmin. Coq does not provide a standard library for machine integers, so, the existing backend chose the CompCert library to model machine integers. Jasmin uses its own word library. In the long run we would hope for a unified word library in the Coq ecosystem. Meanwhile, we have chosen to change the backend to use Jasmin words.

In the future, one may either want to parameterize the backend by a module type for machine integers, or use parametricity [56] or univalence [28, 104, 107] to translate between these data structures.

Another issue in the library implementation is the use of coercions to cast between types (e.g. embedding $u8$ into $\mathbb{N}$). To allow casting for the types used in SSProve, we needed to introduce a new type, which represents an equality between the Coq types, and the types used in code. We can then coerce between the Coq types, and cast to this equivalence type, to still be able to use the coercion mechanism of Coq.

We translate for-loops as a fixed-point over the difference in the bounds of the loop. So, for each loop iteration we decrement the counter of the loop to a monadic do, and bind the result of the code block applied to the current accumulated values. This value is then used as the accumulated value in the next iteration, until the iteration counter hits zero, where we return the value of all the mutable variables with assignments in the loop. In this way we prove that the code block is equalivalent to a `fold` in pure Coq.

The Stateful Translation

Since we provide the first translation from Hacspec to an imperative programming language, we need to extend Hacspec's analysis. SSProve needs information about what memory locations and functions are used in a given scope. To compute this, we add static dependency analysis to the Hacspec pipeline. This is done by walking the AST for every block of code and adding a unique memory location for each mutable variable. In a second pass, we add external dependencies by inspecting all the local (non-external) function calls, for which we already know the scoping information.

Functional Equality of Translation

We prove functional equality between the two translations by constructing primitives with both a pure and a stateful version. Then instead of translating assignments to a pure and stateful assignment, we translate to the assignment primitive, which has the proof of equality. This way the translation remains close to the original specification, made more readable by the notation engine in Coq, while having projections to the stateful and pure translation as well as the functional equality between them.

4.5 Jasmin & SSProve

We implement a translation of Jasmin programs of type `_prog`, to SSProve programs of type `raw_code`. To prove this translation correct, it proved useful to extend SSProve with the notion of *unary* judgments.

Unary deterministic judgements

SSProve originally supported only relational judgments of the form $\vdash \; \{\phi\} \; c_0 \sim c_1 \; \{\psi\}$, as presented in Section 4.3. We build on top of it a new unary judgment that deals with the special case where we relate a program with a return value: $\vdash \; \{\phi\} \; c \Downarrow v \; \{\psi\}$. This time ϕ is a precondition on the initial state of c while ψ is a postcondition on the final state after running c. There is no longer any mention of another state or of a final value in the postcondition, instead the final value v is part of the judgment. The reason is that we consider c to be a deterministic program, i.e. one that does not sample.

We define $\vdash\ \{\phi\}\ c\ \Downarrow\ v\ \{\psi\}$ as the following judgment relating c to `return` v:

$$\begin{array}{ll}\vdash & \{(m_0, m_1).\ \phi\ m_0\} \\ & c \\ \sim & \texttt{return}\ v \\ & \{(m_0', a_0), (m_1', a_1).\ \psi\ m_0' \wedge a_0 = a_1 \wedge a_1 = v\}\end{array}$$

The precondition only considers the memory of the left-hand side, while the postcondition also states that both sides must produce the same value and that this value must be equal to v.

Here we define a simple concept using a complex semantic interpretation. The advantage is that we can reuse the existing theory. Moreover, we obtain a precise connection between the two logics by proving that whenever c is free of sampling operations, the judgment above is equivalent to saying that running c on any initial state m such that $\phi\ m$ will yield return value v and final state m' such that $\psi\ m'$. An advantage of this approach is that we can easily leverage the rules of the relational program logic to prove unary judgments.

We show the two main rules of the unary program logic derived from the relational one. They are (thankfully) not surprising.

$$\frac{\forall m.\ \phi\ m \Longrightarrow \psi\ m \wedge v = v'}{\vdash\ \{\phi\}\ \texttt{return}\ v\ \Downarrow\ v'\ \{\psi\}} \qquad \frac{\vdash\ \{\phi\}\ c\ \Downarrow\ u\ \{\xi\} \qquad \vdash\ \{\xi\}\ k\ u\ \Downarrow\ v\ \{\psi\}}{\vdash\ \{\phi\}\ x \leftarrow c\,;\ k\ x\ \Downarrow\ v\ \{\psi\}}$$

We also derive rules for reasoning about state.

Memory

A major difference between the Jasmin and SSProve semantics, is how memory is handled. In particular, Jasmin supports both global and local variables: global variables corresponding to pointers into memory and local variables representing (and compiling to) variables stored on the stack frame of a single function call; SSProve only has a notion of global memory. We emulate the behavior of local variables in SSProve by parameterizing all translated programs over a process ID, which reserves a region of the global memory to local variables. Then instantiating a program with a process ID, correctly assigns new process IDs to all its called functions. We choose to store the global memory in a map (from integers to bytes), though this could also just have been region of memory.

Program translations

Types The only base types missing from SSProve's `choice_type` (the restricted set of types which a `raw_code` can return) were word and array types. Mimicking Jasmin, we used the `coqword` library for a type of words, which is based on the Mathematical Components library. We represent arrays as maps from integers to bytes. The only minor difference is our implementation of maps differs from Jasmin. The benefit of using similar types made it easy to embed Jasmin values into SSProve values (via the identity) for all except array values.

Expressions For the translation of expressions we had to be careful to do the right casts and truncations, as dictated by the semantics of Jasmin. e.g. when looking up in an array, the index is always cast to an integer type. For the translation of function applications in expressions (additions, subtractions, etc.), we reused the semantics from Jasmin expressions, by transporting values back to Jasmin types, applying the operations, and then transporting back to SSProve types. Note that this transport is only non-trivial for arrays. This simplifies the proof significantly, only requiring us to prove that all operations are invariant under this transportation. This is again related to the transport problem, we mentioned in 4.4.

Instructions The main difficulty in translating instruction was translating function calls; for operation applications we could mostly use the same solution as for expressions and for for-loops we simply iterate the translated body. To be able to call functions, we choose to let our translation keep track of previously translated functions, and only allow these to be called; this avoids cyclic function calls and recursion. Furthermore we make sure to call these translated function with a fresh process ID, such that there are no collisions between local variables in separate function calls.

Note that we currently do not translate while loops or system calls, as we have did not give them semantics in SSProve yet.

Correctness theorem

We prove that our translation preserves the semantics of well defined programs. To do this we define a relation between Jasmin memory-states and SSProve memory-states. First we relate the global Jasmin memory of type `mem` to the “global memory map” stored on the heap in SSProve. The relation is the natural one, i.e. if one can successfully read a single byte at an address from the Jasmin memory, then one can look up the corresponding value on the SSProve heap.

```
Definition rel_mem (m : mem) (h : heap) :=
  ∀ (ptr : pointer) (v : u8),
    read m ptr U8 = ok v → (get_heap h mem_loc) ptr = Some v.
```

Here `mem_loc` is the static `Location` on the heap where the global memory map is stored.

To relate the local memory of Jasmin of type `vmap` and our implementation of local memory in SSProve, we define a relation between `vmap` and a pair of a `p_id` (process ID) and a heap. The relation states that if one can successfully look up a variable in the `vmap`, then looking up that variable on the heap *relative to the process ID* yields the same value.

```
Definition rel_vmap (vm : vmap) (p : p_id) (h : heap) :=
  ∀ i v,
    vm.[i] = ok v → get_heap h (translate_var p i) = embed v.
```

Here `embed` here is the function embedding Jasmin values in SSProve values. This is usually the identity as discussed earlier.

Now, the relation between Jasmin memory of type `estate` (which is just a record containing a `emem : mem` and current `evm :   vmap`) and SSProve memory of type `heap` is not just the conjunction of `rel_mem` and `rel_vmap`, since we need to know that a certain process can spawn arbitrarily many sub-processes and not run out of space on the heap. To state this we need some terminology. We will say that a process ID `m_id` (*main* ID) is *fresh* wrt. a heap `h` when `rel_vmap vmap0   m_id h` holds, where `vmap0` is the empty variable map. We will assume that we have a prefix order, $\preceq$, on process IDs and say that a process ID is `valid` when all its strict successors wrt. this order are fresh.

Then we will define the notion of a *stack-frame*

```
Definition stack_frame :=
  vmap * p_id * p_id * list p_id.
```

which consists of a `vmap`, two process IDs and list of process IDs. The intuition for a term `(vm, m_id, s_id, s_st)` of this type, is that the `vm` should be related (via `rel_vmap`) to `m_id` and `s_id` (*sub*-ID) should be a valid process ID (from which the process can spawn new processes with fresh memory). In that case we say that the stack-frame is valid (given some auxiliary conditions on `s_st`). The list `s_st` (*sub-stack*) is kept around to remember which processes has been spawned and how they relate to current process IDs (namely that the memory they point to should be disjoint). Note that this list is only need for the proof of correctness, and is not actually used in the translation of a given program.

We then define a *stack* simply as a list of stack frames

```
Definition stack := list stack_frame.
```

and define an inductive relation `valid_stack` between stacks and heaps. We define it such that the following key lemmas are satisfied:

(1) A single valid `stack_frame` constitutes a valid `stack`.

```
Lemma valid_stack_single vm m_id s_id s_st h :
  valid_stack_frame (vm, m_id, s_id, s_st) h →
  valid_stack [::(vm, m_id, s_id, s_st)] h.
```

(2) Popping from a valid `stack` yields a valid `stack`.

```
Lemma valid_stack_pop stf st :
  ∀ h, valid_stack (stf :: st) h →
  valid_stack st h.
```

(3) Given a valid `stack`, you can push an empty `stack_frame` and preserve validity, if you update the sub-ID of the previous top.

```
Lemma valid_stack_push vm m_id s_id s_st st :
  ∀ h, valid_stack ((vm, m_id, s_id, s_st) :: st) h →
  valid_stack ((vmap0, s_id~1, s_id~1, [::])
    :: ((vm, m_id, s_id~0, s_st) :: st)) h.
```

Here `s_id~0` and `s_id~1` are some process IDs which point to disjoint regions of the heap and are both strict successors of `s_id`.

The final relation between Jasmin and SSProve memory is then parameterized by a (current) stack-frame and stack as

```
Definition rel_estate (s : estate)
  (m_id : p_id) (s_id : p_id)
  (s_st : list p_id) (st : stack) (h : heap) :=
  rel_mem s.(emem) h ∧
  valid_stack ((s.(evm), m_id, s_id, s_st) :: st) h.
```

Using this relation, we can prove how our translation of Jasmin code relates to its source. For example, if we consider the function `translate_pexpr`, which translates Jasmin expressions to `raw_code`, we get the following correctness theorem[3].

```
Lemma translate_pexpr_correct :
  ∀ (e : pexpr) (s : estate) (v : value) m_id s_id s_st st,
    sem_pexpr gd s e = ok v →
    ⊢ ⦃ rel_estate s m_id s_id s_st st ⦄
    translate_pexpr m_id e
```

[3]Note that we have omitted some boilerplate code for conciseness.

```
    ⇓
    translate_value v
    ⦃ rel_estate s m_id s_id s_st st ⦄.
```

This theorem states that if a Jasmin expression `e` has the value `v` in the jasmin memory-state `s` and `s` is related to an SSProve heap `e`, then we conclude two things: (1) evaluating the translated expression (in the heap `h`) gives the same value `v`; (2) after evaluating the translated expression `s` and `h` are still related. This is the expected results, since evaluating expressions should not have any side-effects on memory.

The main theorem we prove, which establishes the connection between *function calls* in Jasmin and in SSProve is the following.

```
Theorem translate_prog_correct P m vargs m' vres :
  ∀ fn,
    sem_call P m fn vargs m' vres →
    handled_program P →
    ∀ vm m_id s_id s_st st,
    ⊢ ⦃ rel_estate {| emem := m; evm := vm |} m_id s_id s_st st ⦄
      get_translated_fun P fn s_id~1 [seq totce (translate_value v) | v ← vargs]
      ⇓
      [seq totce (translate_value v) | v ← vres]
    ⦃ rel_estate {| emem := m' ; evm := vm |} m_id s_id~0 s_st st ⦄.
```

The theorem states that if calling the function `fn` in the Jasmin program (basically, a list of Jasmin functions) `P` and global memory `m` with arguments `vargs` results in the new global memory `m'` and returns the values `vres`, and the global memory `m` together with some local memory `vm` is related to a heap `h`, then we can conclude two things:

1. Translating the function at a fresh ID (`s_id~1`) and calling it with the translation of `vargs` as arguments evaluates to the translation of `vres`
2. After calling the translated function, the global memory `m'` is related to heap where we have updated the sub-ID to a fresh one (from `s_id` to `s_id~0`)

Again this is the expected behavior: Calling a function should be able to change the global but not the local state. We have to update our sub-ID because the previous one is no longer fresh, since we might have stored local state inside the function call.

Note the `handled_program` simply ensures that the program `P` does not contain any functions using while-loops (as these are not yet supported by our translation).

4.6 AES example

As a larger test case of our framework, we verify the security of a Jasmin implementation of an encryption scheme using AES. The Jasmin implementation and the general methodology are similar to the presentation inEasyCrypt [17], except we use SSProve instead of EasyCrypt.

Concretely, we prove indistinguishability under chosen plaintext attack of an PRF-based encryption scheme using an Intel AES-NI Jasmin implementation of AES.

As was the case in Section 4.2 we do not actually have to provide a security proof of the abstract encryption scheme, since such a proof, using a generic function as pseudo-random function (PRF) instead of AES, is already present in the SSProve library.

The PRF-based encryption scheme is given by the code:

```
Definition PRF_ENC f m :=
k_val ← kgen ;;
enc m k_val.
```

where `kgen` is a key generation code (by uniform sampling) and `enc` is given by the code:

```
Definition enc m k :=
r ← sample uniform N ;;
let pad := f r k in
let c := m ⊕ pad in
ret c.
```

Here f is the function which we assume to be a PRF and which we will instantiate with AES in our example. For all functions `f : word → word → word` we denote the game consisting of the single export `PRF_ENC f` by `PRF_real f`.

To connect to the existing proof, we have to prove that `PRF_real aes` is perfectly indistinguishable from the same scheme where `enc` has been substituted with the translated Jasmin code. For details on the security proof for the PRF-based encryption scheme see section 2.3 in [1].

The high-level structure of the security analysis of the implementation is as follows:

1. Implement program in Jasmin
2. Extract Coq AST from the Jasmin source
3. Translate the Coq AST to imperative SSProve code (using the translation from Section 4.5)

4. Write the intermediate imperative implementation directly in SSProve code

5. Write the functional implementation directly in Coq (Gallina)

6. Prove the equivalence between the intermediate implementation and the functional implementation

7. Prove the equivalence between the translated implementation and the intermediate implementation

8. Connect the equivalences to the existing security proof of the abstract encryption scheme

We can skip step (1) by using the implementation from [17]. Steps (3) and (4) can also be copied almost verbatim from the EasyCrypt development: The syntactic similarities of the two EasyCrypt and SSProve makes translation very straightforward. For the proofs in steps (5) and (6) we can reuse some parts, e.g. the loop invariants, but in general the differences in the operation semantics and the underlying proof assistants require new proofs.

Translation

As mentioned in Section 4.2, we start by printing the Coq AST's of all the involved functions during Jasmin compilation. Then we use the translation described in Section 4.5 to obtain SSProve codes of each function used in the implementation.

Specification

Next, we write intermediate specifications for the Jasmin functions. When comparing to the example in Section 4.2, these correspond to the pure Coq XOR function. As mentioned, we can reuse the specifications from [17], which simplifies this step considerably.

The reasoning behind having the intermediate specification between the translated and the functional one, is that it is usually easier to get rid of the artifacts from the translation first (generated memory locations, generated operation specs) and then afterwards worry about the underlying mathematical logic.

Equivalences for intermediate code

Here we prove that our intermediate implementations are equivalent to functional (stateless) Coq functions. The statements we prove are generally of the form:

$$\begin{array}{ll} \vdash & \{(m_0, m_1).\ \phi\ (m_0, m_1)\} \\ & c\ i \\ \sim & \texttt{return}\ (f\ i) \\ & \{(m_0', a_0), (m_1', a_1).\ \phi\ (m_0', m_1') \land a_0 = a_1\} \end{array}$$

where i is arbitrary input, c is the intermediate SSProve code and f is the Coq function. Note that we also prove that these equivalences preserve the precondition ϕ; for the equivalences to hold we usually have to assume that ϕ is stable wrt. memory locations used by c.

Even though f is usually stateless, we have to keep the heap of the right hand side in mind, since it might be relevant in a context where we wish to apply the program equivalence. Note that we could have used the unary judgments described in Section 4.5 if we could ignore the heap of the right hand side.

Equivalences for translated code

Now we have to reason about the code generated by the Jasmin compiler and passed through our translation to SSProve. The general form of these equivalences is:

$$\begin{array}{ll} \vdash & \{(m_0, m_1).\ \phi\ (m_0, m_1)\} \\ & o\ \leftarrow \texttt{translate\_call}\ id\ F\ i\ ;; \\ & \texttt{return}\ o \\ \sim & c\ i \\ & \{(m_0', a_0), (m_1', a_1).\ \phi\ (m_0', m_1')\ \land\ a_0 = a_1\} \end{array}$$

where i is an arbitrary input, id is an arbitrary process ID (determining the locations used by the function), F is the name of a function in the jasmin program and c is the intermediate code.

The function `translate_call` is a subprocedure of our translation function that handles calls to other functions in the program; by proving an equivalence of this form, we can reuse it in proofs where F appears as a called function. Note that it is therefore important that the equivalences are parametric in the id, since functions can call other functions at arbitrary ids.

Here we also want to preserve the precondition ϕ and again we have to assume that ϕ is stable wrt. the locations of F and c. However, there is one issue here: the set locations of F is not straightforward to compute and might also be rather large. Instead we require that ϕ is stable wrt. to *all possible* locations used by ϕ, i.e. , locations stored using an id' with prefix id ($id \preceq id'$). This turns out to be a sufficient and reasonably manageable invariant to preserve.

Connecting to security proof

The encryption function of which we want to prove the security can be implemented in Jasmin as:

```
fn enc(reg u128 n, reg u128 k, reg u128 p) -> reg u128 {
  reg u128 mask,c;
  mask = aes(n,k);
  c = xor(mask,p);
  return(c);
}
```

We translate this into SSProve as `JENC` and use it in the following encryption scheme:

```
Definition JPRF_ENC id0 m :=
  k_val ← kgen ;;
  r ← sample uniform N ;;
  JENC id0 k_val r m
```

We will denote the game consisting of just `JPRF_ENC` by `JPRF_real`. Then we prove perfect indistinguishability between this scheme and a similar scheme `CPRF_real`, which simply uses an intermediate SSProve encryption function, `ENC`, in place of `JENC`.

To do this, we again use Theorem 1 from [1]. We thus have to find a stable invariant which is preserved by a run of each of these schemes and prove that their return values are equal. Here we prove a slight generalization of the version of Theorem 1 previously implemented in Coq. In the previous version of the theorem the invariant was required to be stable wrt. the *finite sets* of locations used by the program. Moreover, these sets were assumed to be disjoint from the state of the adversary. We generalize this and only require the invariant to be stable wrt. some *arbitrary* sets of locations assumed to be disjoint from the state of the adversary. In particular, the sets are no longer required to be finite.

This generalization facilitates applying the theorem to the case where one of the programs is the output of our translation, since we do not have to provide the concrete set of locations used by the program, but instead we can just give an infinite over approximation of locations used by the program. We obtain following theorem.

```
Theorem JPRF_perf_indist id0 :
  JPRF_real id0 ≈₀ CPRF_real.
```

Now we prove that `CPRF_real` is perfectly indistinguishable from `PRF_real aes` where `aes` is the functional Coq specification of AES, i.e. we prove the theorem:

```
Theorem CPRF_perf_indist id0 :
  CPRF_real ≈₀ PRF_real aes.
```

Here we can apply the original version of Theorem 1, since we have better control over which locations are used.

Now we can use the triangle equality for advantages similar to how we used them in Section 4.2 and derive that `JPRF_real` is IND-CPA with the same bounds as in [1, Section 2.3].

4.7 Future work

Jasminify[4] is a python tool that simplifies the process of calling Jasmin code from Rust. After the compilation of a program, the Rust object file is replaced with the Jasmin object file. However, Jasminify does not come with any correctness guarantees. Above we have shown how to prove the equivalence of a Rust (Hacspec) implementation for AES with a Jasmin program. Hacspec is expressive enough to implement high-level cryptographic protocols. For such protocols, we now have a safe way to replace its cryptographic primitives by optimized Jasmin ones, as we know that their source-level semantics agrees. For future work, one could try to test this toolchain, by using Jasminify, proving equivalence between the Hacspec and Jasmin implementations and then benchmarking to see what kinds of performance gains one can achieve.

4.8 Related work

Formal verification of cryptographic software has been intensely investigated; see [16] for an overview. More narrowly, work related to SSProve can be found in [68]. In this section, we survey the closest related work in this space.

CertiCrypt [22] is the earliest framework for reasoning about cryptographic code in Coq. It is currently unmaintained. FCF [88] is a more recent foundational Coq framework for cryptographic proofs. It was used to verify the HMAC implementations in OpenSSL [32] and mbedTLS [110].

A detailed comparison with those works can be found in the journal version of the SSProve paper [68]. Briefly, SSProve is in active development, uses the well-developed math-comp library and supports modular proofs.

EasyCrypt [23, 24] is a proof assistant and verification tool specifically designed and built from scratch for game-based cryptographic proofs. EasyCrypt's good integration with automatic theorem provers (e.g. SMT solvers) is

[4]`https://gitlab.com/Jur/jasminify`

helpful for such large proofs, even if it does come at a cost in terms of trusted computing base.

CryptHOL [27] is a foundational framework for game-based proofs that uses the theory of relational parametricity to achieve automation in the Isabelle/HOL proof assistant. However, unlike EasyCrypt, CryptHOL has not been used for the verification of efficient programs.

The last mile paper [8] is an important point of comparison. It proves the security and correctness of a Jasmin implementation of SHA3.

The verification of the HMAC C-implementation in OpenSSL [32] uses FCF and VST. Our work is similar in that we prove the security and correctness of the Jasmin implementation of AES.

Schwabe et al. [97] proves correctness of the C-implementation of X25519 in TweetNaCl using VST. Protzenko and Parno [92] verifies an impressive library of cryptographic code in F*. Fiat-cryptography [58] can be used to *generate* verified efficient implementations of finite field arithmetic. None of these works considers cryptographic security.

Acknowledgements

This work was in part supported by the Concordium Blockchain Research Center at Aarhus University.

Bibliography

[1] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. SSProve: A foundational framework for modular cryptographic proofs in Coq. 2021. URL `https://eprint.iacr.org/2021/397`. 6, 77, 79, 88, 91, 92

[2] Alejandro Cabrera Aldaya, Alejandro Cabrera Sarmiento, and Santiago Sánchez-Solano. SPA vulnerabilities of the binary extended euclidean algorithm. *J. Cryptogr. Eng.*, 7(4):273–285, 2017. 15, 16, 52

[3] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-timing attacks on RSA key generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):213–242, 2019. 16

[4] Alejandro Cabrera Aldaya, Cesar Pereida García, and Billy Bob Brumley. From A to Z: projective coordinates leakage in the wild. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):428–453, 2020. 15, 16, 52

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *CCS*, pages 1807–1823. ACM, 2017. 9, 49

[6] José Bacelar Almeida, Cecile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of SHA-3. In *CCS*, pages 1607–1622. ACM, 2019. 5, 9, 11, 49

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *IEEE S & P*, pages 965–982. S & P, 2020. 49

[8] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 965–982. IEEE, 2020. 9, 74, 81, 93

[9] Andrew W. Appel. Verified software toolchain - (invited talk). In *ESOP*, volume 6602 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2011. doi: 10.1007/978-3-642-19718-5_1. URL `https://doi.org/10.1007/978-3-642-19718-5_1`. 12

[10] D. F. Aranha. Pairings are not dead, just resting. `https://ecc2017.cs.ru.nl/slides/ecc2017-aranha.pdf`, 2017. 17

[11] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`, 2022. 38, 66

[12] Diego F. Aranha, Koray Karabina, Patrick Longa, Catherine H. Gebotys, and Julio López. Faster Explicit Formulas for Computing Pairings over Ordinary Curves. In *EUROCRYPT*, volume 6632 of *LNCS*, pages 48–68. Springer, 2011. 16, 38

[13] Diego F. Aranha, Laura Fuentes-Castañeda, Edward Knapp, Alfred Menezes, and Francisco Rodríguez-Henríquez. Implementing Pairings at the 192-Bit Security Level. In *Pairing*, volume 7708 of *LNCS*, pages 177–195. Springer, 2012. 17, 65

[14] Diego F. Aranha, Paulo S. L. M. Barreto, Patrick Longa, and Jefferson E. Ricardini. The Realm of the Pairings. In *Selected Areas in Cryptography*, volume 8282 of *LNCS*, pages 3–25. Springer, 2013. 17

[15] Diego F. Aranha, Elena Pagnin, and Francisco Rodríguez-Henríquez. LOVE a pairing. In *LATINCRYPT*, volume 12912 of *LNCS*, pages 320–340. Springer, 2021. 17

[16] Manuel Barbosa, Gilles Barthe, Karthikeyan Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. *IACR Cryptol. ePrint Arch.*, 2019:1393, 2019. URL `https://eprint.iacr.org/2019/1393`. 4, 5, 92

[17] Manuel Barbossa, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Pierre-Yves Strub, and Tiago Oliveira. Easycrypt and jasmin tutorial, june 2022. URL `https://formosa-crypto.gitlab.io/news/2022-06-07/sibenik`. 88, 89

[18] Razvan Barbulescu and Sylvain Duquesne. Updating key size estimations for pairings. *J. Cryptol.*, 32(4):1298–1336, 2019. 16

[19] Razvan Barbulescu and Sylvain Duquesne. Updating key size estimations for pairings. *J. Cryptol.*, 32(4):1298–1336, 2019. 65

[20] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. In *SAC*, volume 3897 of *LNCS*, pages 319–331. Springer, 2005. 16, 38

[21] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *SCN*, volume 2576 of *LNCS*, pages 257–267. Springer, 2002. 38, 65

[22] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *POPL*, pages 90–101, 2009. 5, 92

[23] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *CRYPTO*, volume 6841 of *LNCS*, pages 71–90. Springer, 2011. 92

[24] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In *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. 5, 92

[25] Evmorfia-Iro Bartzia and Pierre-Yves Strub. A formal library for elliptic curves in the Coq proof assistant. In *ITP*, volume 8558 of *LNCS*, pages 77–92. Springer, 2014. 12, 49

[26] David Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. Tamarin: Verification of Large-Scale, Real World, Cryptographic Protocols. *IEEE Security and Privacy Magazine*, 2022. doi: 10.1109/msec.2022.3154689. URL https://hal.archives-ouvertes.fr/hal-03586826. 4

[27] David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *J. Cryptol.*, 33(2):494–566, 2020. doi: 10.1007/s00145-019-09341-z. URL https://doi.org/10.1007/s00145-019-09341-z. 93

[28] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The hott library: a formalization of homotopy type theory in coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 164–172, 2017. 81

[29] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. *IACR Cryptol. ePrint Arch.*, page 331, 2004. URL `http://eprint.iacr.org/2004/331`. 3

[30] Dmitry Belyavsky, Billy Bob Brumley, Jesús-Javier Chi-Domínguez, Luis Rivera-Zamarripa, and Igor Ustinov. Set it and forget it! turnkey ECC for instant integration. In *ACSAC*, pages 760–771. ACM, 2020. 16, 18

[31] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *S&P'14*, pages 459–474. IEEE Computer Society, 2014. 16

[32] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium*, pages 207–221. USENIX Association, 2015. URL `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer`. 6, 74, 92, 93

[33] Daniel J. Bernstein. qhasm: tools to help write high-speed software. URL `https://cr.yp.to/qhasm.html`. 4

[34] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *PKC*, volume 3958 of *LNCS*, pages 207–228. Springer, 2006. 16, 20, 39

[35] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):340–398, 2019. 10, 17, 21, 22, 23, 24, 34, 38, 42, 43, 44, 45, 51, 52, 55, 56, 59, 60

[36] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *J. Cryptogr. Eng.*, 2(2):77–89, 2012. 39

[37] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In Nigel Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, pages 181–197, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78967-3. 5

[38] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In *TPHOLs*, volume 5170 of *LNCS*, pages 86–101. Springer, 2008. 45

[39] Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture*

Notes in Computer Science, pages 48–62. Springer, 2006. ISBN 978-3-540-74463-4. doi: 10.1007/978-3-540-74464-1_4. URL `https://doi.org/10.1007/978-3-540-74464-1_4`. 7

[40] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and P Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security and Privacy*, pages 445–459, 2013. 5

[41] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symposium on Security and Privacy (S&P'17)*, pages 483–503, San Jose, CA, May 2017. IEEE. Distinguished paper award. 4

[42] Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE S&P*, pages 140–154. IEEE Computer Society, 2006. doi: 10.1109/SP.2006.1. URL `https://doi.org/10.1109/SP.2006.1`. 5

[43] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, October 2016. 4

[44] Jenny Blessing, Michael A. Specter, and Daniel J. Weitzner. You really shouldn't roll your own crypto: An empirical study of vulnerabilities in cryptographic libraries, 2021. 16, 17

[45] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In *CPP*, volume 7086 of *LNCS*, pages 362–377. Springer, 2011. 43, 46

[46] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *USENIX Security Symposium*, pages 917–934, 2017. 49

[47] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptol.*, 17(4):297–319, 2004. doi: 10.1007/s00145-004-0314-9. URL `https://doi.org/10.1007/s00145-004-0314-9`. 16, 17, 66, 68

[48] Joppe W. Bos. Constant time modular inversion. *J. Cryptogr. Eng.*, 4 (4):275–281, 2014. 16, 52

[49] Thomas Braibant and Damien Pous. Tactics for reasoning modulo ac in coq. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs*, pages 167–182, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-25379-9. 7

[50] Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2010. 68

[51] Billy Bob Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. Practical Realisation and Elimination of an ECC-Related Software Bug Attack. In *CT-RSA*, volume 7178 of *LNCS*, pages 171–186. Springer, 2012. 16

[52] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. In *ASIACRYPT*, pages 222–249, Cham, 2018. Springer International Publishing. ISBN 978-3-030-03332-3. URL `https://eprint.iacr.org/2018/306`. 3, 6, 80

[53] Ran Canetti. Universally composable security. *J. ACM*, 67(5):28:1–28:94, 2020. doi: 10.1145/3402457. URL `https://doi.org/10.1145/3402457`. 3

[54] Jorge Chávez-Saab, Francisco Rodríguez-Henríquez, and Mehdi Tibouchi. Swiftec: Shallue-van de woestijne indifferentiable function to elliptic curves. *IACR Cryptol. ePrint Arch.*, page 759, 2022. URL `https://eprint.iacr.org/2022/759`. 51, 68

[55] Cyril Cohen and Assia Mahboubi. Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. *Log. Methods Comput. Sci.*, 8(1), 2012. 45

[56] Maxime Dénès, Anders Mörtberg, and Vincent Siles. A refinement-based approach to computational algebra in coq. In *International Conference on Interactive Theorem Proving*, pages 83–98. Springer, 2012. 81

[57] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A foundry for the deductive verification of rust programs. In Adrian Riesco and Min Zhang, editors, *Formal Methods and Software Engineering*, pages 90–105. Springer, 2022. ISBN 978-3-031-17244-1. 78

[58] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *IEEE S&P*, 2019. doi: 10.1109/SP.2019.00005. URL `http://adam.chlipala.net/papers/FiatCryptoSP19/`. 6, 93

[59] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *S& P*, pages 1202–1219. IEEE, 2019. 8, 16, 17, 18, 53

[60] Armando Faz-Hernández, Julio César López-Hernández, and Ricardo Dahab. High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Trans. Math. Softw.*, 45(3):25:1–25:35, 2019. 39

[61] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A verified, efficient embedding of a verifiable assembly language. *PACMPL*, (POPL), 2019. URL `https://github.com/project-everest/project-everest.github.io/raw/master/assets/vale-popl.pdf`. 5

[62] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008. 43

[63] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.2.1 edition, 2012. `https://gmplib.org/`. 38, 66

[64] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2005. ISBN 3-540-28372-2. doi: 10.1007/11541868_7. URL `https://doi.org/10.1007/11541868_7`. 7

[65] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. A formal proof of the kepler conjecture. In *Forum of Mathematics, Pi*, volume 5. CUP, 2017. 43

[66] Shai Halevi. A plausible approach to computer-aided cryptographic proofs. *IACR Cryptol. ePrint Arch.*, page 181, 2005. URL `http://eprint.iacr.org/2005/181`. 3

[67] Mike Hamburg. Computing the Jacobi symbol using Bernstein-Yang. Cryptology ePrint Archive, Paper 2021/1271, 2021. URL `https://eprint.iacr.org/2021/1271`. `https://eprint.iacr.org/2021/1271`. 10, 11, 51, 53, 60, 61, 63, 64

[68] Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. SSProve: A foundational framework for modular cryptographic proofs in Coq. Cryptology ePrint Archive, Paper 2021/397, 2021. URL `https://eprint.iacr.org/2021/397`. Journal version `https://eprint.iacr.org/2021/397`. 79, 81, 92

[69] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.*, 6(ICFP), 2022. doi: 10.1145/3547647. URL `https://doi.org/10.1145/3547647`. 78

[70] Rasmus Holdsbjerg-Larsen, Mikkel Milo, and Bas Spitters. A verified pipeline from a specification language to optimized, safe rust. In *CoqPL 2022*, 2022. URL `https://popl22.sigplan.org/details/CoqPL-2022-papers/5/`. 12, 35, 49

[71] William Alvin Howard. The formulae-as-types notion of construction. In Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980. 6

[72] Andreas Hülsing, Matthias Meijers, and Pierre-Yves Strub. Formal verification of saber's public-key encryption scheme in easycrypt. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO*, volume 13507 of *Lecture Notes in Computer Science*, pages 622–653. Springer, 2022. doi: 10.1007/978-3-031-15802-5_22. URL `https://doi.org/10.1007/978-3-031-15802-5_22`. 5

[73] B. Salling Hvass, D. F. Aranha, and B. Spitters. High-assurance field inversion for curve-based cryptography. In *2023 2023 IEEE 36th Computer Security Foundations Symposium (CSF) (CSF)*, pages 111–126, Los Alamitos, CA, USA, jul 2023. IEEE Computer Society. doi: 10.1109/CSF57540.2023.00008. URL `https://doi.ieeecomputersociety.org/10.1109/CSF57540.2023.00008`. 10, 53, 56, 62

[74] Benjamin Salling Hvass, Diego F. Aranha, and Bas Spitters. High-assurance field inversion for pairing-friendly primes. July 2020. URL `https://fmbc.gitlab.io/2020/`. 2nd Workshop on Formal Methods for Blockchains, FMBC ; Conference date: 20-07-2020 Through 21-07-2020. 10

[75] Benjamin Salling Hvass, Diego F. Aranha, and Bas Spitters. High-assurance field inversion for pairing-friendly primes. July 2020. The Coq Workshop 2020 ; Conference date: 05-07-2020 Through 06-07-2020. 10

[76] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016. 6, 74

[77] P. Letouzey. Coq extraction, an overview. In *LTA '08*, volume 5028 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008. 8

[78] Pierre Letouzey. A new extraction for Coq. In *TYPES*, volume 2646 of *LNCS*, pages 200–219. Springer, 2002. 35, 43

[79] Benjamin Lipp. *Mechanized Cryptographic Proofs of Protocols and their Link with Verified Implementations*. PhD thesis, Inria Paris, 2022. 5

[80] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, 2020. doi: 10.5281/zenodo.4282710. URL `https://doi.org/10.5281/zenodo.4282710`. 44

[81] Assia Mahboubi and Enrico Tassi. Mathematical components. Online book, 2021. URL `https://math-comp.github.io/mcb/`. 11, 80

[82] Narcisse Bang Mbang, Diego F. Aranha, and Emmanuel Fouotsa. Computing the optimal ate pairing over elliptic curves with embedding degrees 54 and 48 at the 256-bit security level. *Int. J. Appl. Cryptogr.*, 4(1): 45–59, 2020. 38, 65

[83] Michael B. McLoughlin. addchain: Cryptographic addition chain generation in go. Repository `https://github.com/mmcloughlin/addchain`, October 2021. URL `https://doi.org/10.5281/zenodo.5622943`. 66

[84] Alfred Menezes, Palash Sarkar, and Shashank Singh. Challenges with assessing the impact of NFS advances on the security of pairing-based cryptography. In *Mycrypt*, volume 10311 of *LNCS*, pages 83–108. Springer, 2016. 38, 65

[85] Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan. hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust. Technical report, Inria, March 2021. URL `https://hal.inria.fr/hal-03176482`. 6, 74, 78

[86] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985. 16, 20

[87] Kaushik Nath and Palash Sarkar. Efficient arithmetic in (pseudo-)mersenne prime order fields. *Adv. Math. Commun.*, 16(2):303–348, 2022. 39, 66

[88] Adam Petcher and Greg Morrisett. The foundational cryptography framework. In Riccardo Focardi and Andrew C. Myers, editors, *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, volume 9036 of *Lecture Notes in Computer Science*, pages 53–72. Springer, 2015. ISBN 978-3-662-46665-0. doi: 10.1007/978-3-662-46666-7_4. URL `http://adam.petcher.net/papers/FCF.pdf`. 6, 92

[89] Thomas Pornin. Optimized binary gcd for modular inversion. Cryptology ePrint Archive, Paper 2020/972, 2020. URL `https://eprint.iacr.org/2020/972`. `https://eprint.iacr.org/2020/972`. 53

[90] Thomas Pornin. Faster modular inversion and legendre symbol, and an x25519 speed record. `https://research.nccgroup.com/2020/09/28/faster-modular-inversion-and-legendre-symbol-and-an-x25519-speed-record/`, 2020. 53, 66

[91] Thomas Pornin. Optimized binary gcd for modular inversion. Cryptology ePrint 2020/972, 2020. 39

[92] Jonathan Protzenko and Bryan Parno. EverCrypt cryptographic provider offers developers greater security assurances. Microsoft Research Blog, April 2019. URL `https://www.microsoft.com/en-us/research/blog/evercrypt-cryptographic-provider-offers-developers-greater-security-assurances/`. 93

[93] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, 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`. 5

[94] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella Béguelin. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *S& P*, pages 983–1002. IEEE, 2020. 4, 40, 48

[95] Mikkel Milo Rasmus Holdsbjerg-Larsen, Bas Spitters. A verified pipeline from a specification language to optimized, safe rust. CoqPL, 2022. URL `https://cs.au.dk/~spitters/CoqPL22.pdf`. 81

[96] Harvey E Rose. *A course in number theory*. Oxford University Press, 1995. 55

[97] Peter Schwabe, Benoît Viguier, Timmy Weerwag, and Freek Wiedijk. A coq proof of the correctness of x25519 in tweetnacl. In *2021 34th CSF*, pages 1–16, 2021. doi: 10.1109/CSF51468.2021.00023. 93

[98] Laurent Simon, David Chisnall, and Ross Anderson. What you get is what you c: Controlling side effects in mainstream c compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–15. IEEE, 2018. 74

[99] Jerome A. Solinas. Generalized Mersenne numbers. Technical report, CACR, 1999. URL `https://cacr.uwaterloo.ca/techreports/1999/corr99-46.pdf`. 20

[100] Matthieu Sozeau. Generalized rewriting. URL `https://coq.inria.fr/refman/addendum/generalized-rewriting.html`. 7

[101] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *MSCS, special issue on 'Interactive theorem proving and the formalization of mathematics'*, 21:1–31, 2011. doi: 10.1017/S0960129511000119. 44

[102] Supranational. The `blst` multilingual bls12-381 signature library. `https://github.com/supranational/blst`. 66

[103] 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, January 2016. ISBN 978-1-4503-3549-2. URL `https://www.fstar-lang.org/papers/mumon/`. 5

[104] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. The marriage of univalence and parametricity. *Journal of the ACM (JACM)*, 68(1):1–44, 2021. 81

[105] The Coq Development Team. The Coq proof assistant, version 8.12.0, July 2020. URL `https://doi.org/10.5281/zenodo.4021912`. 19

[106] The Coq development team. *The Coq proof assistant.* URL `http://coq.inria.fr`. 6, 19

[107] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Institute for Advanced Study, 2013. 81

[108] Riad S. Wahby and Dan Boneh. Fast and simple constant-time hashing to the BLS12-381 elliptic curve. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):154–179, 2019. 66, 68, 69

[109] Peter Wuille, Gregory Maxwell, and Russell O'Connor. Bounds on divsteps iterations in safegcd. `https://github.com/sipa/safegcd-bounds`, 2021. 17, 35, 46, 61

[110] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedTLS HMAC-DRBG. In *CCS'17*, pages 2007–2020. ACM, 2017. doi:

10.1145/3133956.3133974. URL `https://doi.org/10.1145/3133956.3133974`. 92

[111] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *CCS*, pages 1789–1806. ACM, 2017. 17, 48