PSL UNIVERSITÉ PARIS

THÈSE DE DOCTORAT DE L'UNIVERSITÉ PSL

Préparée à l'ENS et INRIA Paris

Construction d'une bibliothèque cryptographique multi-plateformes formellement vérifiée à haute performance en F*

Soutenue par
Marina POLUBELOVA
Le 17/01/2022

École doctorale n°386
Sciences Mathématiques de Paris Centre

Spécialité
Informatique

Composition du jury :

Véronique CORTIER	LORIA	*Présidente*
Peter SCHWABE	MPI-SP	*Rapporteur*
Paul ZIMMERMANN	INRIA	*Rapporteur*
Manuel BARBOSA	INESC TEC	*Examinateur*
Jonathan PROTZENKO	Microsoft Research	*Examinateur*
Karthikeyan BHARGAVAN	INRIA	*Directeur de thèse*

ENS | PSL

Ph.D. Thesis

Building a Formally Verified High-Performance Multi-Platform Cryptographic Library in F*

Marina POLUBELOVA

Thesis supervised by:
Karthikeyan BHARGAVAN

November 22, 2021

Institut National de Recherche en Informatique,
Automatique et mathématiques appliquées

Abstract

Many security-critical applications need efficient and secure implementations of cryptographic algorithms. To address this demand, general-purpose cryptographic libraries like OpenSSL include dozens of mixed assembly-C implementations for each primitive, highly optimized for multiple popular platforms. Most of these optimizations exploit single-instruction, multiple-data (SIMD) parallelism that significantly changes the structure of the code, making it no longer resemble the original scalar algorithm.

However, despite years of careful design and implementation, bugs and attacks continue to be found in such optimized code. These include memory safety errors, timing leaks, and functional correctness bugs. The probability of catching such bugs by testing is extremely low, but they still can be exploited to conduct an attack. We advocate the use of formal verification to mathematically prove the absence of such implementation bugs. Unlike other approaches, such as testing and auditing, it provides strong guarantees that code is memory safe, functionally correct against its high-level specification, and secret independent ("constant-time") to protect against certain timing side-channel attacks. The challenge is to verify hundreds of thousands of lines of highly-optimized code, and, hence, a more systematic approach is needed.

This thesis presents a new approach towards building a formally verified high-performance multi-platform cryptographic library that compiles generic verified code written in F* to optimized C code for different platforms, composable with verified assembly. Our approach reduces programming and verification effort by sharing code between implementations of an algorithm for different platforms and between implementations of different cryptographic primitives.

Our library incorporates the results of three projects, each described in its own chapter. First, we show how to write verified portable C and mixed assembly-C implementations for the Curve25519 elliptic curve. Second, we show how to write verified vectorized implementations for platforms that support 128-bit, 256-bit, and 512-bit SIMD vector instructions for the ChaCha20 encryption algorithm, the Poly1305 one-time MAC, and the SHA-2 and Blake2 families of hash algorithms. Third, we show how to write verified portable C implementations for the RSA-PSS and Ed25519 signature schemes and for the Finite-Field Diffie-Hellman key exchange over standard groups. Hence, we developed a new version of the open-source cryptographic library HACL*, which now includes the first verified implementations of, for example, RSA-PSS and the first verified vectorized implementations on ARM Neon and AVX512.

Our work closes the performance gap between verified and high-performance unverified cryptographic implementations. Various algorithms from our verified cryptographic library are already deployed in Mozilla's NSS cryptographic library, the Wireguard VPN, the Zinc crypto library for the Linux Kernel, the Tezos blockchain, ElectionGuard, and msQuic. The library can

be further improved with respect to side-channels protections and performance. Future work can build upon our results to provide high-assurance implementations of newly designed constructions for post-quantum and lightweight cryptography and coming standards for multi-party threshold cryptography.

Publications

[107] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy*, pages 983–1002. IEEE Computer Society Press, May 2020

[103] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. HACLxN: Verified generic SIMD crypto (for all your favourite platforms). In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 899–918. ACM Press, November 2020

Table of Contents

Chapter 1

Introduction

Modern cryptography plays a crucial role in many applications, from military or governance to daily use, as it supports many security properties, such as confidentiality, data integrity, authentication, and non-repudiation. One of the cryptographic protocols widely used today is Transport Layer Security (TLS), which has been deployed for secure communications over the Internet (HTTPS), emails, messaging systems, and media streams. The most recent version of the protocol is TLS 1.3, standardized in IETF RFC 8446 [109]. It includes several cryptographic primitives, such as the ChaCha20-Poly1305 and AES-GCM authenticated encryption with associated data (AEAD), Elliptic-Curve Diffie-Hellman (ECDH), Elliptic-curve and Edwards-curve Digital Signature Algorithms (ECDSA and EdDSA) with elliptic curves P-256, P-384, P-521, Curve25519, Curve448, along with Finite-Field Diffie-Hellman (FFDH) key exchange over standard groups, and the RSA-PSS signature scheme.

A typical cryptographic library like OpenSSL includes dozens of implementations for each of the algorithms listed above, written primarily in assembly and hand-optimized for various popular hardware platforms, while still providing a fallback C implementation that works on any platform. OpenSSL assembly implementations [1] exploit many platform-specific instruction set extensions, such as SHA-EXT [65] for the SHA-224 and SHA-256 hash functions, AES-NI [62] for the AES encryption algorithm and single-instruction multiple data (SIMD) parallelism for various primitives, such as the ChaCha20-Poly1305 AEAD for instance.

In practice, the security of cryptographic algorithms relies not only on computational hardness assumptions (e.g., the integer factorization and discrete logarithm problems) but also on their implementation, which must be both efficient and secure. In particular, the implementation must be resistant against side-channel attacks, avoid memory safety errors, and efficiently compute a functionally correct result. However, it is notoriously hard to write cryptographic code that is fast, secure and functionally correct. For instance, OpenSSL's `libcrypto` reported 24 vulnerabilities between January 1, 2016, and May 1, 2019 (Table C.1), including memory safety errors (e.g., buffer overflow and dereferencing a NULL pointer), timing leaks and integer arithmetic bugs (e.g., integer overflow and carry propagation bugs). The probability of catching such bugs by testing is extremely low, but they can be exploited to permit an attack [39].

Cryptographic Software Verification. The strongest way of providing the guarantees of

[1] `https://www.openssl.org/docs/manmaster/man3/OPENSSL_ia32cap.html`

functional correctness and memory safety for cryptographic code is to employ formal verification. As opposed to testing or auditing, this approach allows mathematically proving the absence of a certain class of bugs.

Multiple research projects have developed various approaches towards formally verifying hand-optimized assembly (Vale [36, 59], Jasmin [14, 15]) and portable C implementations (HACL* [127], Fiat-Crypto [56]). The former does not need to sacrifice any performance, but it is notoriously difficult to add support for a new platform or algorithm, and it becomes a challenge to scale it to large cryptographic constructions and libraries. The latter uses high-level verification-oriented programming languages to share code and verification effort between the algorithms and produces readable and portable C code, but it is not as efficient as the state-of-the-art hand-optimized assembly.

Contributions. Our goal is to build a formally verified high-performance multi-platform cryptographic library that compiles into portable and platform-specific C implementations that are close in performance to hand-optimized assembly implementations. Towards achieving this goal, we enhance the open-source cryptographic library HACL*, which we present in Chapter 2, with the following three projects.

— **EverCrypt: a Verified Cryptographic Provider.** Chapter 3 presents a new approach based on dependently-typed generic programming [17] to prove the safety and correctness of fine-grained interoperation between C and assembly. We also demonstrate how the use of verified zero-cost generic programming techniques can enhance developer productivity and streamline the verification of cryptographic algorithms without sacrificing performance. We apply this methodology to obtain verified portable C and mixed assembly-C implementations for Curve25519 and SHA-2 that rely on the Intel ADX [101] and SHA-EXT instruction set extensions, respectively.

— **HACL×N: Verified Generic SIMD Crypto.** Chapter 4 presents a new hybrid approach towards a high-performance multi-platform library of vectorized cryptographic algorithms, compiling it to platform-specific C implementations that rely on compiler intrinsics for SIMD vector instructions. We apply the methodology to obtain verified vectorized implementations for ARM NEON and Intel AVX/AVX2/AVX512 platforms for the ChaCha20 encryption algorithm, the Poly1305 one-time MAC, and the SHA-2 and Blake2 families of hash algorithms.

— **A Verified Bignum Library.** Chapter 5 presents a new multi-precision arithmetic library that is generic, portable, and relatively fast, i.e., does not depend on a modulus form and works on any platform without sacrificing performance. We use the library to obtain verified portable C implementations for both the Ed25519 and the RSA-PSS signature schemes and the FFDH key exchange.

We conclude the thesis and discuss the limitations and future directions of our work in Chapter 6.

All our code is publicly available at

`https://github.com/project-everest/hacl-star` (commit 50047e3)

Chapter 2

Methodology

2.1 Background on HACL*, F*, Low*

The HACL* cryptographic library [127] contains verified implementations for many popular cryptographic algorithms. The source code for each algorithm is written in the F* programming language [118] and verified using the F* type-checker for memory safety, for functional correctness against a high-level specification also written in F*, and for a side-channel guarantee called *secret independence*, which states that secret data cannot be used in either branching or to compute memory addresses. The F* code is then compiled into fast C code that can be easily integrated into existing software. We briefly review the languages and tools used in HACL*.

F* [118] is a state-of-the-art verification-oriented programming language. It is a functional programming language with dependent types and an effect system, and it relies on SMT-based automation to prove properties about programs using a weakest-precondition calculus [12].

Low* [108] is a stateful subset of F*, which models the C memory layout of the heap and stack. Using Low*, the programmer manipulates arrays, reasoning about their liveness, disjointness and location in memory. Low* uses machine integers (instead of mathematical integers), which forces the programmer to reason about their overflow semantics and to choose low-level data representations for abstract values.

KreMLin compiles Low* code, once type-checked, to *auditable, readable* C, using a series of many small, composable passes. The Low* preservation theorem [108] states that the translated C code exhibits the same execution traces as the original verified Low* program. Hence, we obtain C code that is functionally correct, memory safe, and secret independent.

Although Low* is a C-like first-order language, a programmer can use the higher-order features of F* to write *generic* code parameterized by constants, types, and functions. At compile time, such functions (or *meta-programs*) are inlined at each call site and aggressively simplified (or *specialized*) to yield first-order Low* code that can be translated to C.

Online materials. The following tutorials provide various examples of how to write code in F* and Low*, starting from simple to more advanced projects, and are publicly available at

F* tutorial: `https://www.fstar-lang.org/tutorial`

Low* tutorial: `https://fstarlang.github.io/lowstar/html/index.html`

Example. Let us consider the increment function as an example to show the basics of the F*

programming language, which resembles OCaml syntax.

Syntax for pure functions. Unbounded mathematical integers. We define a function incr that takes x of type int as input and returns an int. In F*, the type int represents unbounded mathematical integers, and + is an addition operator defined over all pairs of integers.

```
let incr (x:int) : Tot int = x + 1
```

The incr function is annotated with the F* Tot effect, meaning that it is a pure mathematical function with no side effects. In other words, it is a total function that always returns a result.

Refinement types. Lemmas. We might want to prove, for instance, that if incr takes x of type nat, it returns a positive number. The type nat is defined using a *refinement type* as a subset of the unbounded integer type int.

```
let nat = x:int{x ≥ 0}
```

We can express pre-conditions and post-conditions of functions using refinement types. In F*, we can state this property about incr using one of two styles (or their combination). The first is called *intrinsic*, where we enrich the type of the function:

```
let incr (x:int{x ≥ 0}) : Tot (res:int{res > 0}) = x + 1
```

The other is called *extrinsic*, where we write a separate lemma, i.e., a total function that always returns a unit.

```
let incr_nat_lemma (x:nat) : _:unit{incr x > 0} = ()
```

F* provides syntactic sugar for writing lemmas, which allows us to omit a pre-condition (requires) if it is trivial (⊤):

```
let incr_nat_lemma (x:int) : Lemma (requires x ≥ 0) (ensures incr x > 0) = ()
let incr_nat_lemma (x:nat) : Lemma (requires ⊤) (ensures incr x > 0) = ()
let incr_nat_lemma (x:nat) : Lemma (ensures incr x > 0) = ()
(* or simply *)
let incr_nat_lemma (x:nat) : Lemma (incr x > 0) = ()
```

F* automatically proves all the above variants of the lemma incr_nat_lemma, so our proof is simply ().

The intrinsic style may be preferred for stating simple properties about a function, while the extrinsic style separates proofs from a specification, making them more robust and modular.

Machine integers. Functional correctness. KreMLin cannot extract the incr function into C, as it uses unbounded mathematical integers, which have no built-in support in C. For simplicity, we can decide that our incr function works with 32-bit unsigned machine integers, for which F* provides operators of *strict* and *modular* addition, add and add_mod, respectively.

```
let u32_incr (x:U32.t{U.size (incr (U32.v x)) 32})
  : Tot (res:U32.t{U32.v res = incr (U32.v x)}) = U32.add x 1ul

let u32_incr_mod (x:U32.t)
  : Tot (res:U32.t{U32.v res = incr (U32.v x) % pow2 32}) = U32.add_mod x 1ul
```

The u32_incr function requires that the result of invoking incr with the mathematical value of x to be in the type's representable range, i.e., between 0 and $(2^{32} - 1)$. The u32_incr_mod function has no requirements on the result, but it returns the result modulo 2^{32}, which does not match the result of calling incr on $(2^{32} - 1)$. Therefore, only u32_incr is *functionally correct* with respect to the specification of incr.

Stateful code. Memory safety and functional correctness. Let us now define a stateful function for incr, u32_incr_st, which takes as input a 32-bit array x of size one, increments its first element and writes the result in x.

```
let ind0 (h:mem) (x:buffer U32.t{U32.v (len x) = 1}) =
  U32.v (Seq.index (as_seq h x) 0)

val u32_incr_st: x:buffer U32.t{U32.v (len x) = 1} → Stack unit
  (requires λ h → live h x ∧ U.size (incr (ind0 h x)) 32)
  (ensures λ h0 _ h1 → modifies (loc_buffer x) h0 h1 ∧
    ind0 h1 x = incr (ind0 h0 x))

let u32_incr_st x =
  let x0 = index x 0ul in
  let res = u32_incr x0 in
  upd x 0ul res
```

The u32_incr_st function is annotated with the Low* Stack effect, ensuring there are no memory leaks as only memory allocations on the stack are allowed. The pre-condition (requires) of u32_incr_st talks about the *liveness* of the input array (i.e., still allocated), needed for *memory safety*. The post-condition (ensures) has a modifies clause: only x is modified, leaving other disjoint objects in memory unchanged. It also has a *functional correctness* guarantee relating the output to the specification of incr. That is, the mathematical value of the first element of x in a final memory h1 is equal to the result of invoking incr with the mathematical value of the first element of x in an initial memory h0 (if the result is between 0 and $(2^{32} - 1)$). as_seq h x returns the contents of a given array x in a given memory h.

F* verifies that the implementation of u32_incr_st meets this type signature, where index x i returns the i-th element of array x, and upd x i res updates the i-th element of array x with a value res. In both cases, the length of array x must be greater than i.

Extracted C code. We now can extract our verified Low* implementation for incr to C. The extracted code will not contain any proofs, ghost computations, unused variables, etc. *Ghost computations* are functions or parameters annotated in F* with the GTot effect. Similar to the F* Tot effect, such functions do not have side effects, but they will not be extracted in the target language (C or OCaml).

Some functions and constructions also receive special treatment from KreMLin. For example, machine integers and arrays are replaced with calls to native C constructions. The shape of the generated code can be controlled by annotating functions and variables with F* keywords, e.g., inline_for_extraction inlines the body of a definition (which can be stateful), or inline_let inlines the body of a pure local definition.

The resulting C code for the u32_incr_st function is depicted below, where u32_incr was

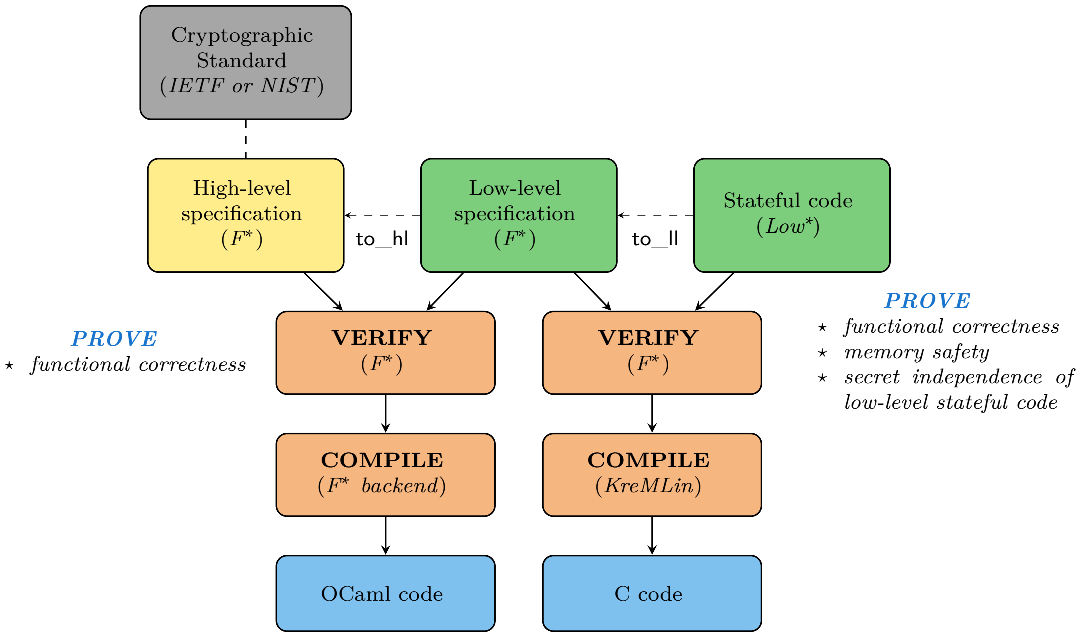

Cryptographic Standard (*IETF or NIST*)
High-level specification (*F**)
Low-level specification (*F**)
Stateful code (*Low**)
to_hl
to_ll
PROVE
⋆ *functional correctness*
VERIFY (*F**)
VERIFY (*F**)
PROVE
⋆ *functional correctness*
⋆ *memory safety*
⋆ *secret independence of low-level stateful code*
COMPILE (*F* backend*)
COMPILE (*KreMLin*)
OCaml code
C code

Figure 2.1 – HACL* programming and verification workflow.

marked as inline_for_extraction.

```
void u32_incr_st(uint32_t *x) {
  uint32_t x0 = x[0U];
  uint32_t res = x0 + (uint32_t)1U;
  x[0U] = res; }
```

HACL* programming and verification workflow. Figure 2.1 depicts the high-level methodology of the HACL* cryptographic library.

High-level specification. For each cryptographic algorithm, we write a *succinct* formal specification in Pure F*, a subset of F* with no side effects. It can use high-level concepts, such as mathematical (unbounded) integers, sequences, lists, and loop combinators. This specification is *trusted* as we adopt it from the IETF or NIST cryptographic standard, where the algorithm description is given in a mixture of English, pseudocode, and mathematical formulas. The specification is also *executable*, so we can extract it to OCaml code and test it against multiple test vectors.

Low-level specification. In order to separate the reasoning between memory safety and functional correctness, we introduce a low-level specification, where we prove *functional correctness* against a high-level specification for the chosen low-level data representations for abstract values. This specification is optional and can be omitted.

Stateful code. We then write an optimized implementation of the algorithm, sometimes porting it in Low* from the state-of-the-art C implementations. The Low* code is then verified for *memory safety*, *functional correctness* against a low-level specification (and, hence, a high-level specification) and *secret independence* (constant-time). Finally, we can compile our Low* implementation to C code.

Therefore, code components shown in green in Figure 2.1 are formally verified; those in yellow are trusted and carefully tested. Each of the green and yellow boxes is hand-written. The to_hl function ("to high-level") reflects the chosen low-level data representation from low-level to high-level specification. Similarly, the to_ll function ("to low-level") reflects it from stateful code to low-level specification. For example, we can represent unbounded mathematical integers as a sequence of machine integers in a low-level specification and as an array of machine integers in stateful code.

Trusted Computing Base (TCB). As with any verification project, our proofs rely on the soundness of our verification toolchain (F* and Z3). To produce executable code, we rely on F*'s backend for extracting Low* to C code [108], which can be compiled by using a verified compiler (e.g., CompCert [79]) or by trusting a faster, unverified C compiler. We also rely on an untrusted assembler and linker to produce our final executable. These trusted tools are comparable to those found in other verification efforts; e.g., implementations verified in Coq [119] trust Coq, the Coq extraction to OCaml, and the OCaml compiler and runtime. To provide higher assurance, multiple research results show that each element of these toolchains can itself be verified [118, 79, 91, 35, 76]. Furthermore, several studies have confirmed that, despite the use of such complex verification toolchains, the empirical result is qualitatively better code compared with traditional software development practices [124, 58, 57]. These studies found numerous defects and vulnerabilities in traditional software, but failed to find any in the verified software itself; the only defects found were in the specifications.

2.2 Verified Libraries for Generic Integers

As we have previously seen, Low* cannot extract unbounded mathematical integers as they have no built-in support in C. Instead, it provides types and operators for machine integers (from 8 to 128 bits, signed and unsigned) that KreMLin will replace with a call to native C constructions. However, if we wanted to define a new operator or function that works generically for all machine integers, such as rotate-left or constant-time comparison, we would have to implement (and verify) it for each size of machine integer, which would be tedious and unproductive. In a language like C++, we would usually use a template to define such functions and in C we would use an untyped macro. To recover the convenience of templates within the strong type system and semantics of F*, we define a generic machine integer module Lib.IntTypes for use in cryptographic code:

```
type inttype = | U8 | U16 | U32 | U64 | U128 | S8 | S16 | S32 | S64 | S128
type secrecy_level = | SEC | PUB
(* the type of secret integers is abstract *)
val sec_int_t: inttype → Type

let int_t (t:inttype) (l:secrecy_level) : Type = match l, t with
  | SEC, _ → sec_int_t t
  | PUB, U8 → LowStar.UInt8.t
  | PUB, U16 → LowStar.UInt16.t | ...
```

The type int_t is parametrized by two indices: inttype enumerates all supported variants of integer types, while secrecy_level distinguishes public data from secret data. The former allows hiding the proliferation of integer models under a single type. The latter enforces the secret-independent coding discipline of HACL* [127] and unifies secret and public integers under a single abstraction, thus relieving the programmer from having to deal with yet another set of operators for secret data. The prefix U is used for unsigned integers, whereas the prefix S is used for signed integers. The type int_t hence has 20 variants. For convenience, our library defines several abbreviations such as let uint32 = int_t U32 SEC or let uint8 = int_t U8 SEC.

To express the semantics of operators, we introduce a function v which maps a machine integer to the mathematical integer of type range_t t, defined as a subset of integers between the minimum and maximum representable value of a given type t (the # symbol denotes implicit arguments, inferred automatically by F*).

```
let range (n:int) (t:inttype) : Type0 = minint t ≤ n ∧ n ≤ maxint t
type range_t (t:inttype) = x:int{range x t}

val sec_int_v: #t:inttype → sec_int_t t → range_t t

let v #t #l (u:int_t t l) : range_t t = match l, t with
  | SEC, _ → sec_int_v #t u
  | PUB, U8 → LowStar.UInt8.v u
  | PUB, U16 → LowStar.UInt16.v u | . . .
```

The function v can be used for specification and proof purposes (i.e., for ghost computations) and cannot be invoked in cryptographic code.

Arithmetic operators and integer overflow. When the result of an arithmetic operator (e.g., addition, multiplication, bitwise left shift) is not within the type's representable range, it results in an integer overflow. According to the C standard, for signed integers, an overflow leads to undefined behaviour. For unsigned integers, it is defined as a wraparound, i.e., all arithmetic operations for unsigned integers are implicitly computed modulo $2^{\mathsf{bits\ t}}$, where bits t is the number of bits in a given type t. Our library provides two types of functions for such operators. For example, we define a *strict* addition operator add for both signed and unsigned integers when the result is in the type's representable range to prevent integer overflow.

```
val add: #t:inttype → #l:secrecy_level
  → a:int_t t l → b:int_t t l{range (v a + v b) t}
  → c:int_t t l{v c == v a + v b}
```

We also introduce a *modular* addition operator add_mod for unsigned integers, with no requirements on the result.

```
let unsigned (t:inttype) = match t with
  | U8 | U16 | U32 | U64 | U128 → true
  | _ → false

val add_mod: #t:inttype{unsigned t} → #l:secrecy_level
  → a:int_t t l → b:int_t t l
  → c:int_t t l{v c == (v a + v b) % pow2 (bits t)}
```

The advantage of providing the function add for unsigned integers is to simplify the proofs, as there is no need to invoke the lemma proving that (v a + v b) % pow2 (bits t) = (v a + v b) when range (v a + v b) t holds.

Under the hood, operators such as add_mod perform a case analysis on the integer type t and call the appropriate built-in machine integer operation. Thanks to inlining and partial evaluation, whenever add_mod is applied to, say, a public (PUB) 32-bit unsigned integer (U32), F* will simplify away all the non-matching cases and replace Lib.IntTypes.add_mod #U32 #PUB by a call to native 32-bit addition (LowStar.UInt32.add_mod). Our library also defines convenient abbreviations for operators such as let (+!) = add and let (+.) = add_mod.

To add a new integer type to Lib.IntTypes, we need to extend inttype and define a new case for each operator that this integer type supports. Adding new operators is similar. This style of defining parametrized types with selectively enabled overloaded operators is a lightweight form of (bounded) type classes [115]. Full-fledged type classes were not available in F* when we began this work, but we plan to experiment with their use in future work.

Secret integers. The type of secret integers sec_int_t is abstract, hence nothing is revealed about the nature of secret integers, and the only operations over them are those offered by Lib.IntTypes. In particular, operations that are known to be non-constant-time (e.g., division, modulo, comparison) have a precondition that the arguments be public. We also use public integers for loop counter, array indices, and lengths. Any attempt to use a secret integer for branching or memory access is a type error.

For example, we define two kinds of functions to compare machine integers. The first eq is a boolean comparison operator that takes two public integers as input and is extracted as the "equal to" operator (==) in C. The other eq_mask is a masked comparison operator that expects two secret integers and returns a secret integer, equal to either 0xff..f (all bits are set) or zero, and is implemented in a constant-time way. When one of the arguments is a public value, we can cast it (explicitly) into a secret integer, but not the other way around, as it would break secret independence.

```
val eq: #t:inttype → a:int_t t PUB → b:int_t t PUB → r:bool{r == (v a = v b)}

val eq_mask: #t:inttype → a:int_t t SEC → b:int_t t SEC
 → m:int_t t SEC{if v a = v b then v m == ones_v t else v m == 0}
```

Although it is known that integer multiplication is variable-time on some platforms, e.g., PowerPC or ARM Cortex-M3 [1], we do not provide a special implementation for such cases and leave that for future work.

2.3 Packed and Unpacked Bignum Representation

Many cryptographic algorithms work with large numbers that do not fit within a machine word. These include elliptic curve cryptography, where we need to implement arithmetic operations over field elements of hundreds of bits in size. Another typical example is the RSA

[1] https://www.bearssl.org/ctmul.html

algorithm or the Finite-Field Diffie-Hellman key exchange, where the main arithmetic operation is an exponentiation modulo a large number of size 2048 bits and larger.

Bignum representation. A non-negative mathematical integer A can be encoded as a *fixed-length* sequence of unsigned machine integers $a = [a_0; a_1; \ldots; a_{len-1}]$ such that

$$A = \sum_{i=0}^{len-1} a_i \cdot \beta^i, \text{where}$$

— a radix β is a positive integer,

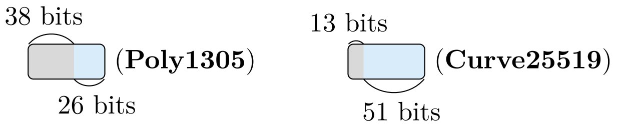
— a positive integer len is the length of the sequence and $A < \beta^{len}$,

— $\forall\ 0 \leq i < len$, a limb a_i is the i-th element of the sequence and $0 \leq a_i < \beta$.

Such a representation is *unique* (as len is fixed) and is called a *little-endian representation in radix-β*, i.e., the least significant limb is at index 0.

Packed bignum representation. A representation is *packed* when the radix β is chosen as $2^{\mathsf{bits\ t}}$, where $\mathsf{bits\ t}$ is the number of bits in a given type t for a_i. For example, $\beta = 2^{32}$ for 32-bit unsigned integers or $\beta = 2^{64}$ for 64-bit unsigned integers. We use this representation for our generic bignum library (Chapter 5) and field elements of Curve25519 (§3.4).

Unpacked bignum representation. A representation is *unpacked* when the radix β is chosen as a number strictly less than $2^{\mathsf{bits\ t}}$. In this case, the coefficients a_i are limited only to the machine word width, so we have extra space, or "nail" bits [2], to hold carries without propagating them to the next-higher coefficients when computing arithmetic operations. We use this representation for field elements of Poly1305 (§2.4 and §4.3.4) with $\beta = 2^{26}$ and Curve25519 (§3.4) with $\beta = 2^{51}$, where each a_i is stored in a 64-bit machine integer. Therefore, we can split the machine word into two parts, gray and blue, where the first corresponds to the extra space and the second stores $0 \leq a_i < \beta$.



To reflect the fact that we can store the extra carries in the gray space, we loosen the conditions on a_i. That is, when $\beta \leq a_i < 2^{\mathsf{bits\ t}}$, we represent a_i as $a_i = a_{hi} \cdot \beta + a_{lo}$, where $a_{hi} = \dfrac{a_i}{\beta}$ and $a_{lo} = a_i \ \%\ \beta$ are stored in the gray and blue parts, respectively. We use / to denote integer division (i.e., $\left\lfloor \dfrac{a_i}{\beta} \right\rfloor$) and % for the modulo operation.

Bignum evaluation. In order to express the semantics of bignum operations we introduce the *bn_v* function that reflects a given sequence a of positive length len to the mathematical integer as follows:

$$bn\_v\ a \ = \ \sum_{i=0}^{len-1} a_i \cdot \beta^i.$$

In the following, we will sometimes omit the function bn_v when it is obvious from the context. In other words, we implicitly identify a mathematical integer with its bignum representation.

[2] `https://www.manpagez.com/info/gmp/gmp-6.0.0a/gmp_63.php`

Functional correctness for bignum operations. For a bignum operation *bn_op*, we prove that the mathematical integer of its result is the same as if we had performed the mathematical operation *op* on the mathematical integers of its inputs. For example, if *bn_op* is a binary operation that takes as input two sequences a and b and returns another sequence c, we show that the following holds:

$$bn\_v\ (bn\_op\ a\ b) = op\ (bn\_v\ a)\ (bn\_v\ b).$$

2.4 Verifying Poly1305

2.4.1 Mathematical Description

The Poly1305 one-time MAC function [31] is standardized in IETF RFC7539 [94]. It takes a 32-byte key as input and splits it into two 128-bit integers s and r. It then splits the input message into 16-byte blocks, hence transforming it to a sequence of 128-bit integers $(m_1, m_2, \ldots, m_n)$; if the last block is partial, it is filled out with zeroes to obtain a full block.

The main computation in the Poly1305 MAC evaluates the following polynomial in the prime field $\mathbb{Z}_p$, where $p = 2^{130} - 5$:

$$a = (m_1 \cdot r^n + m_2 \cdot r^{n-1} + \ldots + m_n \cdot r)\ \%\ p.$$

In practice, this polynomial is evaluated block by block, by applying Horner's method to rearrange the polynomial as follows:

$$a = ((\ldots((0 + m_1) \cdot r + m_2) \cdot r + \ldots + m_n) \cdot r)\ \%\ p.$$

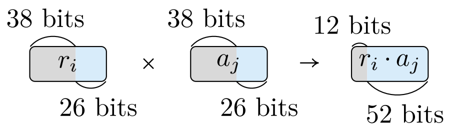
We maintain an accumulator a, initially set to 0, and to process each new block m_i, we first add it to the accumulator, and then multiply the result by r (all operations in $\mathbb{Z}_p$). Once the final block is processed, we compute $(s + a)\ \%\ 2^{128}$ to obtain the MAC.

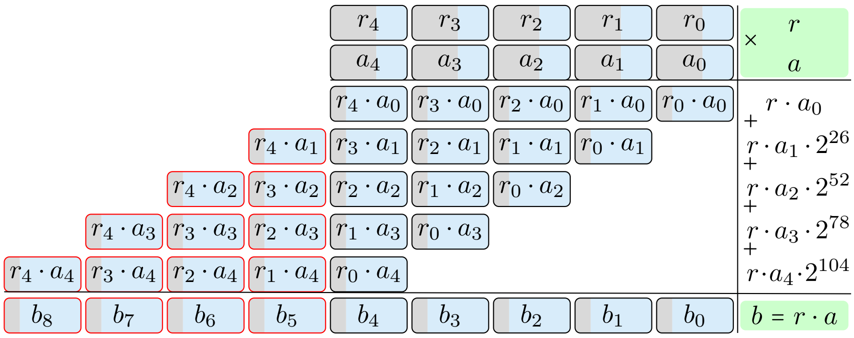
Field arithmetic with a radix-2^{26} representation. We encode an element of the Poly1305 field as a sequence of five 64-bit unsigned integers $[e_0; e_1; e_2; e_3; e_4]$, where each e_i is below 2^{26}, with the following evaluation function:

$$e_4 \cdot 2^{104} + e_3 \cdot 2^{78} + e_2 \cdot 2^{52} + e_1 \cdot 2^{26} + e_0.$$

In general, field multiplication consists of polynomial multiplication, modular reduction and carry propagation, where the three steps can be either sequential or interleaved. As field multiplication is the most performance-critical operation in the Poly1305 MAC, the radix is chosen to perform all three steps sequentially.

Let us first consider the polynomial multiplication of two field elements, r and a. It can be seen as schoolbook multiplication with no carry propagation, where we perform a double-wide multiplication of the form 26-bit × 26-bit → 52-bit:



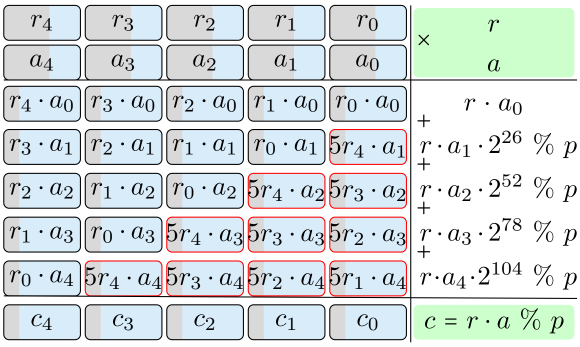

r_4 r_3 r_2 r_1 r_0
× r
a_4 a_3 a_2 a_1 a_0
a
$r_4 \cdot a_0$ $r_3 \cdot a_0$ $r_2 \cdot a_0$ $r_1 \cdot a_0$ $r_0 \cdot a_0$
$r \cdot a_0$
+
$r_4 \cdot a_1$ $r_3 \cdot a_1$ $r_2 \cdot a_1$ $r_1 \cdot a_1$ $r_0 \cdot a_1$
$r \cdot a_1 \cdot 2^{26}$
+
$r_4 \cdot a_2$ $r_3 \cdot a_2$ $r_2 \cdot a_2$ $r_1 \cdot a_2$ $r_0 \cdot a_2$
$r \cdot a_2 \cdot 2^{52}$
+
$r_4 \cdot a_3$ $r_3 \cdot a_3$ $r_2 \cdot a_3$ $r_1 \cdot a_3$ $r_0 \cdot a_3$
$r \cdot a_3 \cdot 2^{78}$
+
$r_4 \cdot a_4$ $r_3 \cdot a_4$ $r_2 \cdot a_4$ $r_1 \cdot a_4$ $r_0 \cdot a_4$
$r \cdot a_4 \cdot 2^{104}$
b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
$b = r \cdot a$

Then, we can perform reduction modulo $p = 2^{130} - 5$ of the product $b = r \cdot a$ by multiplying b_5 by 5 and adding the product to b_0, $b_6 \cdot 5$ to b_1 and so on:

$(2^{130} - 5)\ \%\ p = 0$, hence, $2^{130}\ \%\ p = 5$

$$(bn\_v\ b)\ \%\ p = ((b_8 \cdot 2^{78} + b_7 \cdot 2^{52} + b_6 \cdot 2^{26} + b_5) \cdot \underbrace{2^{130}}_{\%\ p\ =\ 5} + (b_4 \cdot 2^{104} + b_3 \cdot 2^{78} + b_2 \cdot 2^{52} + b_1 \cdot 2^{26} + b_0))\ \%\ p.$$

However, we can compute modular multiplication more efficiently by interleaving multiplication with reduction, relying on the distributivity of the modulo operator over addition:


r_4 r_3 r_2 r_1 r_0
× r
a_4 a_3 a_2 a_1 a_0
a
$r_4 \cdot a_0$ $r_3 \cdot a_0$ $r_2 \cdot a_0$ $r_1 \cdot a_0$ $r_0 \cdot a_0$
$r \cdot a_0$
+
$r_3 \cdot a_1$ $r_2 \cdot a_1$ $r_1 \cdot a_1$ $r_0 \cdot a_1$ $5r_4 \cdot a_1$
$r \cdot a_1 \cdot 2^{26}\ \%\ p$
+
$r_2 \cdot a_2$ $r_1 \cdot a_2$ $r_0 \cdot a_2$ $5r_4 \cdot a_2$ $5r_3 \cdot a_2$
$r \cdot a_2 \cdot 2^{52}\ \%\ p$
+
$r_1 \cdot a_3$ $r_0 \cdot a_3$ $5r_4 \cdot a_3$ $5r_3 \cdot a_3$ $5r_2 \cdot a_3$
$r \cdot a_3 \cdot 2^{78}\ \%\ p$
+
$r_0 \cdot a_4$ $5r_4 \cdot a_4$ $5r_3 \cdot a_4$ $5r_2 \cdot a_4$ $5r_1 \cdot a_4$
$r \cdot a_4 \cdot 2^{104}\ \%\ p$
c_4 c_3 c_2 c_1 c_0
$c = r \cdot a\ \%\ p$

Finally, the carry propagation is performed for the result of modular multiplication. It returns an integer $e = r \cdot a\ \%\ p = [e_0; e_1; e_2; e_3; e_4]$ less than the prime p, where each coefficient e_i is less than 2^{26}.

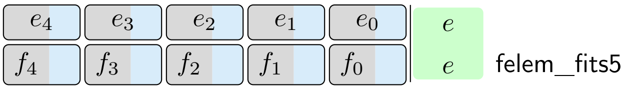
Functional correctness bugs. Even with these delayed carries, carry propagation continues to be a performance bottleneck for field multiplication, and high-performance implementations of Poly1305 implement many low-level optimizations, such as interleaving carry chains, and skipping some carry steps if the developer believes that a given coefficient is below a threshold. Such delicate optimizations have often led to functional correctness bugs [96, 97, 98]. These bugs are particularly hard to find by testing or auditing, since they only occur in low-probability corner cases deep inside arithmetic. Nevertheless, such bugs may be exploitable by a malicious adversary once they are discovered.

2.4.2 Verifying Field Arithmetic

This section considers the well-known lazy-reduction technique for optimizing carry propagation, which allows us to skip some carry steps until the final modular reduction, $(s + a)$ % 2^{128}. This means that the result of field multiplication is not necessarily less than the prime $p = 2^{130} - 5$, and its coefficients can be greater than or equal to 2^{26}. Other optimizations, such as interleaving carry chains, and performing multiplication of several field elements in parallel, are discussed in more detail in [33] and formally verified in §4.3.4.

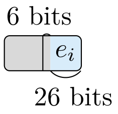
As we wish to optimize the carry propagation and still obtain the correct results, we introduce the felem_fits5 and felem_wide_fits5 predicates to ensure the absence of integer overflow during the computations. The predicates consist of upper bounds for each coefficient of a field element. The difference is that felem_fits5 assumes that we represent a field element as a sequence of five 32-bit integers, whereas felem_wide_fits5 is used for field multiplication, where we perform a double-wide multiplication of the form 26-bit × 26-bit → 52-bit. In other words, we compute the product of two 64-bit unsigned integers, a and b, as $(a \,\%\, 2^{32}) \cdot (b \,\%\, 2^{32})$, which fits into a 64-bit integer.

Following the notations presented in §2.3, let $M = 2^{26} - 1$, a maximum value that can be stored in the right blue part. The figure below depicts felem_fits5 for a field element e, with factors f_i such that $e_i \leq f_i \cdot M$ for all $i = 0 \ldots 4$:

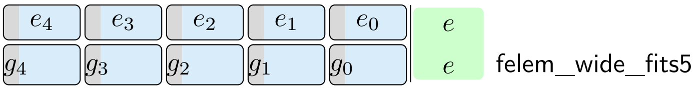


$$e_0 \leq f_0 \cdot M \ \wedge\ e_1 \leq f_1 \cdot M \ \wedge\ e_2 \leq f_2 \cdot M \ \wedge\ e_3 \leq f_3 \cdot M \ \wedge\ e_4 \leq f_4 \cdot M$$

We restrict f_i to be less than or equal to $2^{32-26} = 2^6$.



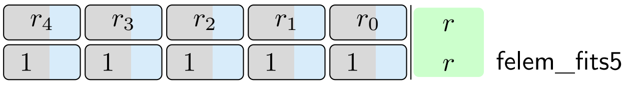
When performing a double-wide multiplication of a and b, such that $a \leq c \cdot M$ and $b \leq d \cdot M$, the upper bound for their product is $a \cdot b \leq c \cdot d \cdot M \cdot M$. If $c \leq 2^6$ and $d \leq 2^6$, then $c \cdot d \leq 2^{12}$. Therefore, felem_wide_fits5 for a field element e, depicted below, consists of the factors g_i such that $e_i \leq g_i \cdot M \cdot M$ with $g_i \leq 2^{12}$.



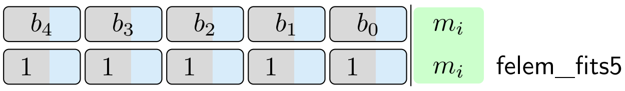
$$e_0 \leq g_0 \cdot M \cdot M \ \wedge\ e_1 \leq g_1 \cdot M \cdot M \ \wedge\ e_2 \leq g_2 \cdot M \cdot M \ \wedge\ e_3 \leq g_3 \cdot M \cdot M \ \wedge\ e_4 \leq g_4 \cdot M \cdot M$$

The invariant for polynomial evaluation. For simplicity, we prove functional correctness for field addition and multiplication with concrete upper bounds, as a key r, a message block m_i, and an accumulator a have the following invariant throughout the polynomial evaluation.

— Both r and m_i are less than 2^{130} and can be uniquely represented in radix-2^{26}, if do not use the extra space (the left gray part). Therefore, $f_i = 1$ for all $i = 0 \ldots 4$.

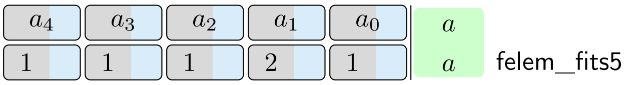


$r_0 \leq 1 \cdot M \;\wedge\; r_1 \leq 1 \cdot M \;\wedge\; r_2 \leq 1 \cdot M \;\wedge\; r_3 \leq 1 \cdot M \;\wedge\; r_4 \leq 1 \cdot M$



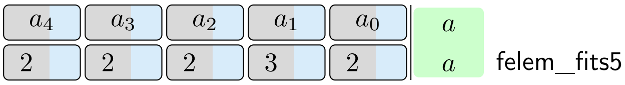
$b_0 \leq 1 \cdot M \;\wedge\; b_1 \leq 1 \cdot M \;\wedge\; b_2 \leq 1 \cdot M \;\wedge\; b_3 \leq 1 \cdot M \;\wedge\; b_4 \leq 1 \cdot M$

— When the accumulator a stores the result of field multiplication, it has all $f_i = 1$ except $f_1 = 2$, as we skip some carry steps (explained in the following).



$a_0 \leq 1 \cdot M \;\wedge\; a_1 \leq 2 \cdot M \;\wedge\; a_2 \leq 1 \cdot M \;\wedge\; a_3 \leq 1 \cdot M \;\wedge\; a_4 \leq 1 \cdot M$

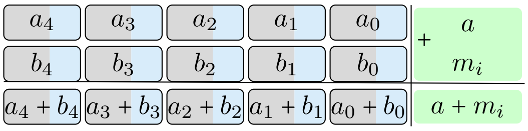
— When the accumulator a stores the result of field addition, it has all $f_i = 2$ except $f_1 = 3$, as we skip modular reduction and carry propagation steps (explained in the following).

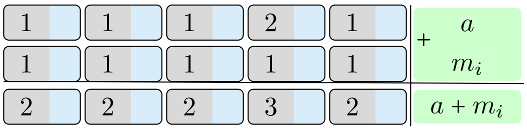


$a_0 \leq 2 \cdot M \;\wedge\; a_1 \leq 3 \cdot M \;\wedge\; a_2 \leq 2 \cdot M \;\wedge\; a_3 \leq 2 \cdot M \;\wedge\; a_4 \leq 2 \cdot M$

We now show that there is no integer overflow when computing the coefficients of intermediate and resulting polynomials with the above invariant, i.e., each f_i is under 2^6 and g_i is below 2^{12}.

In-place field addition. In our case, field addition is simply polynomial addition performed by adding corresponding coefficients.





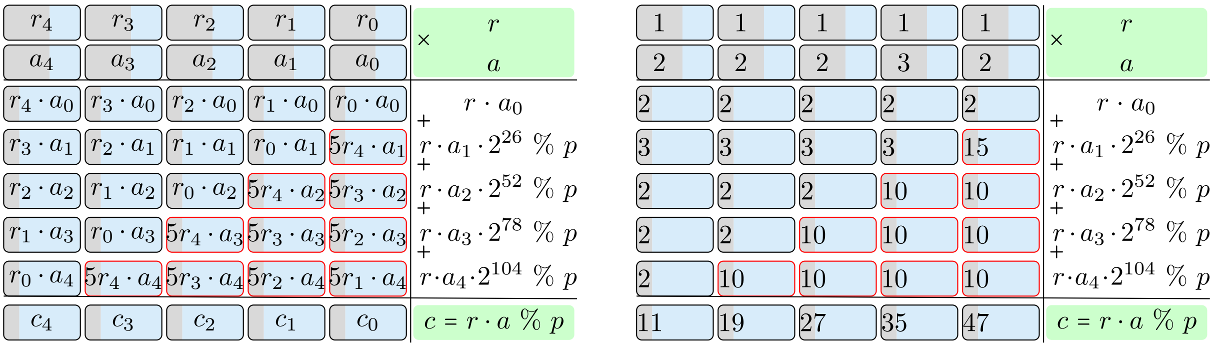
$a_0 + b_0 \leq 1 \cdot M + 1 \cdot M = 2 \cdot M$

$a_1 + b_1 \leq 2 \cdot M + 1 \cdot M = 3 \cdot M$

$\ldots$

On the right, we derive the upper bounds for the coefficients of the resulting polynomial, where a stores the result of field multiplication and m_i is a new message block. As any factor for the extra space is below 2^6, there is enough room to represent the result of field addition.

Field multiplication. Let us first infer the upper bounds for the coefficients of the intermediate and resulting polynomials when computing polynomial multiplication interleaved with modular reduction, where r is a key and a represents the result of field addition.


r_4 r_3 r_2 r_1 r_0
× r
a_4 a_3 a_2 a_1 a_0
a
$r_4 \cdot a_0$ $r_3 \cdot a_0$ $r_2 \cdot a_0$ $r_1 \cdot a_0$ $r_0 \cdot a_0$ $r \cdot a_0$
$r_3 \cdot a_1$ $r_2 \cdot a_1$ $r_1 \cdot a_1$ $r_0 \cdot a_1$ $5r_4 \cdot a_1$ + $r \cdot a_1 \cdot 2^{26}$ % p
$r_2 \cdot a_2$ $r_1 \cdot a_2$ $r_0 \cdot a_2$ $5r_4 \cdot a_2$ $5r_3 \cdot a_2$ + $r \cdot a_2 \cdot 2^{52}$ % p
$r_1 \cdot a_3$ $r_0 \cdot a_3$ $5r_4 \cdot a_3$ $5r_3 \cdot a_3$ $5r_2 \cdot a_3$ + $r \cdot a_3 \cdot 2^{78}$ % p
$r_0 \cdot a_4$ $5r_4 \cdot a_4$ $5r_3 \cdot a_4$ $5r_2 \cdot a_4$ $5r_1 \cdot a_4$ + $r \cdot a_4 \cdot 2^{104}$ % p
c_4 c_3 c_2 c_1 c_0 $c = r \cdot a$ % p
1 1 1 1 1 × r
2 2 2 3 2 a
2 2 2 2 2 $r \cdot a_0$
3 3 3 3 15 + $r \cdot a_1 \cdot 2^{26}$ % p
2 2 2 10 10 + $r \cdot a_2 \cdot 2^{52}$ % p
2 2 10 10 10 + $r \cdot a_3 \cdot 2^{78}$ % p
2 10 10 10 10 + $r \cdot a_4 \cdot 2^{104}$ % p
11 19 27 35 47 $c = r \cdot a$ % p

$$r_0 \cdot a_0 \leq (1 \cdot M) \cdot (2 \cdot M) = 2 \cdot M \cdot M$$

$$5 \cdot r_4 \cdot a_1 \leq 5 \cdot (1 \cdot M) \cdot (3 \cdot M) = 15 \cdot M \cdot M$$

$$5 \cdot r_3 \cdot a_2 \leq 5 \cdot (1 \cdot M) \cdot (2 \cdot M) = 10 \cdot M \cdot M$$

$$\ldots$$

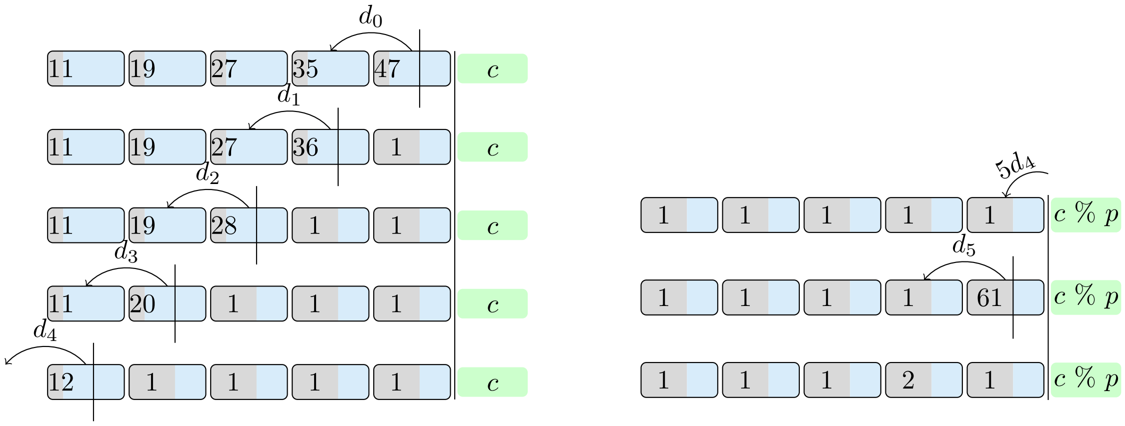
$$c_0 = r_0 \cdot a_0 + 5 \cdot r_4 \cdot a_1 + 5 \cdot r_3 \cdot a_2 + 5 \cdot r_2 \cdot a_3 + 5 \cdot r_1 \cdot a_4$$

$$c_0 \leq 2 \cdot M \cdot M + 15 \cdot M \cdot M + 10 \cdot M \cdot M + 10 \cdot M \cdot M + 10 \cdot M \cdot M = 47 \cdot M \cdot M$$

$$\ldots$$

As we can see, any factor for the extra space is less than 2^{12}, i.e., no integer overflow occurs during the computations.

Finally, in order to preserve the invariant for the accumulator a, we perform the carry propagation for the result of modular multiplication.



On the left side, we do not change the mathematical integer of the result, as we carry the extra bits from limb 0 to limb 1, then from limb 1 to limb 2, etc. For example, c_0 can be represented as $c_0 = \frac{c_0}{2^{26}} \cdot 2^{26} + c_0 \ \% \ 2^{26}$, where $d_0 = \frac{c_0}{2^{26}}$ is added to the next-higher coefficient and $c_0 \ \% \ 2^{26}$ becomes a new value for c_0. On the right, we do not change the mathematical integer of the result *modulo prime*. In order to have all $f_i = 1$, an additional round of the carry propagation is needed.

Specifying field arithmetic in F* (high-level specification). Our F* specification for the Poly1305 algorithm relies on arithmetic modulo the prime $2^{130} - 5$:

```
let prime = pow2 130 − 5
let felem = x:nat{x < prime}
let fadd (x y:felem) : felem = (x + y) % prime
let fmul (x y:felem) : felem = (x ∗ y) % prime
```

The prime modulus is a mathematical constant, and the type of field elements felem is defined using a *refinement type*, as the type of natural numbers (nat) less than the prime. Field operations are then defined using the mathematical operators for addition, multiplication and modulo. The full Poly1305 MAC is specified as the evaluation of a polynomial over this field (see §4.3.4, Figure 4.8). The specification is then compiled to OCaml and subjected to a substantial amount of testing, to detect specification bugs.

Bignum representation in F* (low-level specification). We introduce a low-level specification to prove only the functional correctness of the arithmetic functions for a chosen representation of a field element, which we encode as a tuple of five 64-bit *secret* integers.

```
let felem5 = (uint64 & uint64 & uint64 & uint64 & uint64)
```

Bignum evaluation in F* (from low-level to high-level specification). To reason about the semantics of field operations, we define the as_nat5 function that computes the mathematical integer of an encoded field element e. Whereas the feval5 function takes modulo prime of this value, i.e., it reflects our representation of a field element in the low-level specification (a tuple of five 64-bit integers) to the one in the high-level specification (a natural number less than the prime).

```
let as_nat5 (e:felem5) : nat = let (e0,e1,e2,e3,e4) = e in
  v e0 + v e1 ∗ pow2 26 + v e2 ∗ pow2 52 + v e3 ∗ pow2 78 + v e4 ∗ pow2 104
let feval5 (e:felem5) : felem = as_nat5 e % prime
```

Ensuring the absence of integer overflow in F*. As we need to control the upper bounds of e_i to ensure there is no integer overflow during the computations, we define the felem_fits5 and felem_wide_fits5 predicates shown in Figure 2.2. Both track the factors for the extra space. The difference is that felem_fits5 assumes that we represent a field element as a tuple of five 32-bit integers, whereas felem_wide_fits5 is used for field multiplication. The maximum value of the factors is $2^6 = 64$ and $2^{12} = 4096$, respectively.

Specifying and verifying field addition in F* (low-level specification). This section separates proofs from a specification making them more robust and modular (e.g., several facts about the specification can be split into small lemmas). Therefore, we can use modular addition +. instead of strict addition +! in the specification of field addition:

```
let add5 ((a0,a1,a2,a3,a4):felem5) ((b0,b1,b2,b3,b4):felem5) : felem5 =
  let a0 = a0 +. b0 in let a1 = a1 +. b1 in let a2 = a2 +. b2 in
  let a3 = a3 +. b3 in let a4 = a4 +. b4 in (a0,a1,a2,a3,a4)

val add5_lemma: a:felem5 → mi:felem5 → Lemma
  (requires felem_fits5 a (1,2,1,1,1) ∧ felem_fits5 mi (1,1,1,1,1))
  (ensures felem_fits5 (add5 a mi) (2,3,2,2,2) ∧ feval5 (add5 a mi) == fadd (feval5 a) (feval5 mi))
```

```
let nat5 = (nat & nat & nat & nat & nat)
let max26 = pow2 26 − 1
```

```
let scale32 = f_i:nat{f_i ≤ 64}
let scale32_5 = f:nat5{let (f0,f1,f2,f3,f4) = f in
  f0 ≤ 64 ∧ f1 ≤ 64 ∧ f2 ≤ 64 ∧
  f3 ≤ 64 ∧ f4 ≤ 64}

let felem_fits1 (e_i:uint64) (f_i:scale32) =
  v e_i ≤ f_i ∗ max26
let felem_fits5 (e:felem5) (f:scale32_5) =
  let (e0,e1,e2,e3,e4) = e in
  let (f0,f1,f2,f3,f4) = f in
  felem_fits1 e0 f0 ∧
  felem_fits1 e1 f1 ∧
  felem_fits1 e2 f2 ∧
  felem_fits1 e3 f3 ∧
  felem_fits1 e4 f4
```

```
let scale64 = g_i:nat{g_i ≤ 4096}
let scale64_5 = g:nat5{let (g0,g1,g2,g3,g4) = g in
  g0 ≤ 4096 ∧ g1 ≤ 4096 ∧ g2 ≤ 4096 ∧
  g3 ≤ 4096 ∧ g4 ≤ 4096}

let felem_wide_fits1 (e_i:uint64) (g_i:scale64) =
  v e_i ≤ g_i ∗ max26 ∗ max26
let felem_wide_fits5 (e:felem_wide5) (g:scale64_5) =
  let (e0,e1,e2,e3,e4) = e in
  let (g0,g1,g2,g3,g4) = g in
  felem_wide_fits1 e0 g0 ∧
  felem_wide_fits1 e1 g1 ∧
  felem_wide_fits1 e2 g2 ∧
  felem_wide_fits1 e3 g3 ∧
  felem_wide_fits1 e4 g4
```

Figure 2.2 – F* specification for the felem_fits5 and felem_wide_fits5 predicates used to ensure the absence of integer overflow when computing field addition and multiplication with an unpacked bignum representation.

The fadd function from the high-level specification takes as input two natural numbers less than the prime and returns a result of type felem. Hence, we invoke the feval5 function to state the functional correctness of add5 in the lemma add5_lemma, which F* can prove automatically up to the following equality.

```
as_nat5 (add5 a mi) == as_nat5 a + as_nat5 mi
```

To complete our proof, we apply the modulo operator to both sides of the above equality and invoke the distributivity lemma for the modulo operator over addition.

Specifying and verifying field multiplication in F* (low-level specification). Similar to the above, we specify field multiplication and prove its functional correctness. In practice, the intermediate products $r \cdot a_i \cdot 2^{26 \cdot i} \ \% \ p$ are accumulated rather than stored aside to obtain the final sum. However, for simplicity, we consider here only the multiplication of a bignum r by a limb a_i.

```
let smul5 ((r0,r1,r2,r3,r4):felem5) (a_i:uint64) : felem5 =
  let c0 = r0 ∗. a_i in let c1 = r1 ∗. a_i in let c2 = r2 ∗. a_i in
  let c3 = r3 ∗. a_i in let c4 = r4 ∗. a_i in (c0,c1,c2,c3,c4)

let nx5 (x:nat) : nat5 = (x,x,x,x,x)
val smul5_lemma: #m1:scale32_5 → #m2:scale32 → r:felem5 → a_i:uint64 → Lemma
  (requires felem_fits5 r m1 ∧ felem_fits1 a_i m2 ∧ m1 ∗^nx5 m2 ≤^nx5 4096)
  (ensures felem_wide_fits5 (smul5 a_i r) (m1 ∗^nx5 m2) ∧ as_nat5 (smul5 a_i r) == as_nat5 r ∗ v a_i)
```

The lemma smul5_lemma is defined in a generic way as it is invoked several times with different values for m1 and m2. The ∗^ and ≤^ functions are component-wise multiplication and comparison ("less than or equal to") operations, respectively. For example, the proof for the first product $r \cdot a_0$ in our field multiplication is as follows.

```
let (c0,c1,c2,c3,c4) = smul5 (r0,r1,r2,r3,r4) a0 in
smul5_lemma #(1,1,1,1,1) #2 (r0,r1,r2,r3,r4) a0;
assert (felem_wide_fits5 (c0,c1,c2,c3,c4) (2,2,2,2,2));
```

2.4.3 Implementing Poly1305

The Poly1305 MAC function can be computed using a streaming/incremental API or a one-shot API. The difference is that the latter processes the entire message, while the former can handle it in chunks. Nevertheless, we can identify a set of functions shared between the two implementations: allocation of an internal state ctx, its initialization (poly1305_init), processing a 16-byte block and the last (partial) block, and obtaining the tag (poly1305_finish).

We now build a verified implementation for the one-shot API specified in the IETF RFC7539 [94], whereas the incremental API is discussed in more detail in [106].

```
val poly1305_mac:
    tag:lbuffer uint8 16ul
  → len:size_t → msg:lbuffer uint8 len
  → key:lbuffer uint8 32ul → Stack unit
  (requires λ h →
    live h msg ∧ live h tag ∧ live h key ∧
    disjoint tag msg ∧ disjoint tag key)
  (ensures λ h0 _ h1 → modifies (loc tag) h0 h1 ∧
    as_seq h1 tag == Spec.Poly1305.poly1305_mac (as_seq h0 msg) (as_seq h0 key))

let poly1305_mac tag len msg key =
  push_frame ();
  let ctx = create 15ul (u64 0) in
  poly1305_init ctx key;
  poly1305_eval ctx len msg;
  poly1305_finish tag key ctx;
  pop_frame ()
```

The poly1305_mac function takes a message msg of length len and a 32-byte key key as input and writes the result in a 16-byte array tag. It stack-allocates the internal state ctx which is a mutable array containing three field elements: an accumulator a, a key r and a precomputed value $5 \cdot r$ for a field multiplication. The poly1305_init function encodes the first half of the given key into r, precomputes $5 \cdot r$ and sets a to zero. The poly1305_eval function evaluates the polynomial in the Poly1305 field. The poly1305_finish function first reduces the accumulator a modulo $2^{130} - 5$ and adds it to the second half of key. The final addition is performed modulo 2^{128}, and the result is stored in a tag array. For each function, we provide a type that captures memory safety, functional correctness and secret independence. For functional correctness, we prove that (1) each function conforms to its specification and (2) poly1305_init establishes the invariant from §2.4.2 for our internal state ctx, poly1305_eval preserves it, and poly1305_finish only expects it but does not necessary keep it. The invariant states the concrete upper bounds for the key r, precomputed value $5 \cdot r$ and accumulator a that stores the result of field multiplication.

Project	Programming Workflow	Examples

HACL* [127]	Spec ← Reference Impl
	Spec ← Reference Impl + Bignum (specialized)
		ChaCha20, SHA-256, SHA-512
		Ed25519, Poly1305, Curve25519

EverCrypt (Chapter 3)	*multiplexing* Spec ← Impl — Impl$_1$ Impl$_2$ . . . *agility* Spec.Agile — Spec$_1$ ← Impl.Agile — Impl$_1$ Spec$_2$ Impl$_2$	Curve25519 Hash, AEAD
HACL×N (Chapter 4)	Spec.Scalar ← Spec.Vec ← Impl.Vec — M32 M128 M256 M512 + Bignum.Vec (specialized)	ChaCha20, SHA2-mb, Blake2 Poly1305
Bignum Library (Chapter 5)	Spec ← Reference Impl + Bignum (generic)	FFDHE, RSA-PSS Ed25519 (scalar mul)

Table 2.1 – Programming workflow for HACL*, EverCrypt, HACL×N and our bignum library.

Extracting stateful code to C. The resulting C code for the poly1305_mac function is depicted below. As expected, the Poly1305 internal state ctx is an array allocated on the stack.

```
void poly1305_mac(uint8_t *tag, uint32_t len, uint8_t *msg, uint8_t *key) {
  uint64_t ctx[15U] = { 0U };
  poly1305_init(ctx, key);
  poly1305_update(ctx, len, msg);
  poly1305_finish(tag, key, ctx); }
```

When the length of the message len is greater than 4 GB, one could use a streaming/incremental API which is also available in HACL* [106].

2.5 Problem Statement

HACL*-v1 (2017). HACL* [127] is an open-source verified cryptographic library that implements the ChaCha20-Poly1305 AEAD, Curve25519 elliptic curve, Ed25519 signature scheme and SHA-2 family of hash functions. The library is written in Low* and compiles to portable C code for 32-bit and 64-bit platforms. Table 2 from [127] measures the performance of HACL* against OpenSSL (with assembly enabled and disabled), LibSodium (with assembly disabled) and TweetNaCl (portable C implementations). While HACL*'s code is as fast as reference C implementations in OpenSSL and LibSodium, it is 1.5-5.14× slower than the hand-optimized assembly. This work also verified a 128-bit vectorized implementation of the ChaCha20 stream cipher that exploits the internal parallelism of the algorithm and relies on compiler intrinsics for SIMD vector instructions. It results in a 2.2× speed-up and reduces the performance gap in comparison with OpenSSL's 256-bit vectorized assembly to 2.3×.

Motivation for HACL*-v2. In 2017, HACL* demonstrated that writing a verified cryptographic library is practical. The HACL* code was deployed in Mozilla's NSS cryptographic library. However, it required a significant manual effort to prove that the code is memory safe, secret independent, and functionally correct with respect to the high-level specification. The proof-to-code ratio for the customized bignum library of Curve25519, Ed25519 and Poly1305 was up to 6, while the factor for symmetric primitives was around 2. In addition, the library did not support other important standardized cryptographic primitives like the RSA-PSS signature scheme or P-256 elliptic curve.

HACL*-v2. In this thesis, we present a new approach towards building a formally verified high-performance multi-platform cryptographic library, where the motto is to *verify once* but *compile* and *specialize* many times to maximize programmer productivity. Thanks to inlining and partial evaluation in F*, we aggressively share the code between **(1)** assembly and C implementations (Chapter 3), **(2)** scalar and vectorized implementations (Chapter 4), and **(3)** 32-bit and 64-bit multi-precision arithmetic libraries (Chapter 5). For example, as reported in §4.5, the proof-to-code ratio for Poly1305 is 0.9, as portable and three vectorized implementations are compiled from one generic implementation written in Low*.

HACL*-v2 enhances the HACL*-v1 library with new formally-verified fast implementations such as the Blake2 family of hash functions, RSA-PSS and FFDH. It replaces the prior C implementations with faster implementations for Curve25519, Ed25519, SHA-2 and ChaCha20-Poly1305. For example, our verified vectorized implementations reduce the performance gap in comparison with the fastest hand-optimized assembly to 3-5% (§4.5) for ChaCha20-Poly1305.

Table 2.1 shows the difference between our three new projects and HACL*-v1 for the programming workflow. The EverCrypt cryptographic provider introduces support for agility (choosing between multiple algorithms for the same functionality) and multiplexing (choosing between multiple implementations of the same algorithm). Then, HACL×N presents a hybrid approach towards building a multi-platform library of vectorized cryptographic algorithms. As vectorization significantly changes the code structure, it was a challenge to prove that vectorized specification is functionally correct to the high-level (scalar) specification. Finally, our newly developed generic bignum library can be used in many situations, for example, when a programmer does not have the resources to implement a customized bignum library from scratch.

Chapter 3

EverCrypt: a Verified Cryptographic Provider

This chapter describes the research that was previously published in the following paper:

[107] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy*, pages 983–1002. IEEE Computer Society Press, May 2020

My contribution to this paper is the verified implementation of Curve25519 in Low described in Section 3.4.*

Application developers seldom write their own cryptographic code; instead they rely on a *cryptographic provider*, such as OpenSSL's `libcrypto` [100], Windows Cryptography API [87], or `libsodium` [6]. Developers expect their provider to be *comprehensive* in supplying all of the functionalities they need (asymmetric and symmetric encryption, signing, hashing, key derivation, . . .) for all the platforms they support.

Further, a modern cryptographic provider should be *agile*; that is, it should provide multiple algorithms (e.g., ChaCha20-Poly1305 and AES-GCM) for the same functionality (e.g., authenticated encryption) and all algorithms should employ a single unified API, to make it simple to change algorithms if one is broken.

A modern cryptographic provider should also support *multiplexing*, that is, the ability to choose between multiple implementations of the same algorithm. This allows the provider to employ high-performance implementations on popular hardware platforms (OpenSSL, for example, supports dozens), while still providing a fallback implementation that will work on any platform. Ideally, these disparate implementations should be exposed to the developer via a single unified API, so that she need not change her code when a new optimized version is deployed, and so that the provider can automatically choose the optimal implementation, rather than force the developer to do so.

Since a cryptographic provider is the linchpin of most security-sensitive applications, its *correctness* and *security* are crucial. However, for most applications (e.g., TLS, cryptocurrencies, or disk encryption), the provider is also on the critical path of the application's *performance*. Historically, it has been notoriously difficult to produce cryptographic code that is fast, correct, and secure (e.g., free of leaks via side channels). For instance, OpenSSL's `libcrypto` reported 24 vulnerabilities between January 1, 2016 and May 1, 2019 (Table C.1).

Cryptographic Software Verification. Such critical, complex code is a natural candidate for formal verification, which can mathematically guarantee correctness and security even for complex low-level implementations. Indeed, in recent years, multiple research groups have produced exciting examples of verified cryptographic code. However, previous work has not produced a verified cryptographic *provider* comparable to the unverified providers in use today.

Instead, some work verifies parts of individual algorithms, particularly portions of Curve25519. For example, Chen et al. [42] verify the correctness of a Montgomery ladder using Coq [119] and an SMT solver, while Tsai et al. [122] and Polyakov et al. [104] do so using an SMT solver and a computer algebra system. All three focus on implementations written in assembly-like languages. Almeida et al. [14], in contrast, use a domain-specific language (DSL) and verified compilation (via Coq, Dafny [78], and custom scripting) to verify memory safety and side-channel resistance, and, in more recent work [15], correctness for ChaCha20 and Poly1305 using EasyCrypt [28].

Other papers verify a single algorithm, e.g., SHA-256 [19], HMAC instantiated with SHA-256 [30], or HMAC-DRBG [125]. Each is written in Coq using the Verified Software Toolchain [18] and yields C code, which is compiled by CompCert [79]. Using the Foundational Cryptography Framework [102], they prove the cryptographic constructions secure.

The SAW tool [50] proves the correctness of C code (via LLVM) and Java code with respect to mathematical specifications written in Cryptol. Dawkins et al. use it to verify reference implementations and simple in-house implementations of AES, ZUC, FFS, ECDSA, and the SHA-384 inner loop [120], but they do not report performance, nor verify assembly.

Other works verify multiple algorithms for the same functionality, typically for elliptic curves. For example, Zinzindohoué et al. develop a generic library based on templates, and instantiate it for three popular curves [126], including Curve25519, but they report 100× performance overheads compared with C code. FiatCrypto [56] employs a verified, generic compilation scheme from a high-level Coq specification to C code for bignum arithmetic, which enables them to generate verified field arithmetic code for several curves. Even with aggressive optimization by an unverified C compiler, they still lag behind hand-tuned assembly implementations by 25% to 200%.

Finally, Hawblitzel et al. [67] verify a variety of algorithms (RSA signing and encryption, SHA-1, SHA-256, and HMAC) in Dafny, but they are compiled and verified as a part of a larger application, and they report performance overheads ranging from 30% to 100× compared with unverified implementations.

EverCrypt. Building on previous algorithm verification efforts [127, 36, 59], we present EverCrypt, a cryptographic provider that supports agility, multiplexing, and auto-configuration.

EverCrypt's agility means that multiple algorithms provably provide the same API to clients, and its multiplexing demonstrates multiple disparate implementations verifying against the same cryptographic specification. The API is carefully designed to be usable by both verified and unverified (e.g., C) clients. To support the former, we show that multiple layers of abstraction, both in the implementation and in the specification of cryptographic algorithms, naturally lead to agile code while also improving verification efficiency. We also demonstrate how the judicious use of generic programming can enhance developer productivity and simplify specifications by cleanly extracting commonality from related algorithms.

To illustrate the benefits of multiplexing, and as part of our efforts to ensure state-of-the-art performance for EverCrypt, we present new verified implementations of a large variety of algorithms. These include, e.g., Curve25519 [32] and hash functions, that transparently multiplex between a generic C implementation and a hand-tuned x64 assembly implementation verified using the Vale tool [36, 59]. When run on x64, they match or exceed the performance of the best unverified code, while the C implementations provide support across all other platforms (and offer performance competitive with unverified C code as well).

All EverCrypt code is proven safe and functionally correct. Safety means that the code respects memory abstractions (e.g., never reading outside the bounds of an array) and does not perform illegal operations (e.g., divide by zero). Functional correctness means the code's computational behaviour is precisely described by a pure mathematical function. Further, all EverCrypt code is proven timing-attack resistant; specifically, the sequence of instructions executed and the memory addresses accessed do not depend on secret inputs; more informally, EverCrypt provably implements "constant-time cryptography" [25]. EverCrypt does not (yet) include cryptographic proofs of security (e.g., we do not prove that counter-mode encryption is secure if the underlying cipher is secure). As future work, we plan to apply techniques similar to those of Bhargavan et al. [47] to provide such proofs for common security assumptions atop EverCrypt's functionalities.

3.1 Background: Verifying Cryptographic Assembly Code with Vale

EverCrypt unifies (via multiplexing and an agile API) two open-source projects, HACL* [127] and ValeCrypt [36, 59]. We choose these projects as starting points, since HACL* produces high-performance C code for cross-platform support, while ValeCrypt produces assembly code for maximum performance on specific hardware platforms. Both employ F* [118] for verification, which allows EverCrypt to reason about both in a single unified framework.

3.1.1 ValeCrypt

ValeCrypt is a collection of verified assembly programs written in Vale [36, 59], which allows developers to annotate assembly code with pre- and postconditions, loop invariants, lemmas, etc. to assist with verification (Fig. 3.1).

Vale models a well-structured subset of assembly language, with control flow restricted to blocks of instructions, if/then/else branching, and while loops. Although limited, this subset is well-suited to implementations of cryptographic primitives similar to those found in OpenSSL.

```
procedure mul1(ghost dst:arr64, ghost src:arr64)
  lets dst_ptr @= rdi; src_ptr @= rsi; b @= rdx;
       a := pow2_four(src[0], src[1], src[2], src[3]);
  reads dst_ptr; src_ptr; b;
  modifies rax; r8; r9; r10; r11; r12; r13; mem; efl;
  requires adx_enabled && bmi2_enabled;
    arrays_disjoint(dst, src) || dst == src;
    validDstAddrs64(mem, dst_ptr, dst, 4);
    validSrcAddrs64(mem, src_ptr, src, 4);
  ensures
    let d := pow2_five(dst[0], dst[1], dst[2], dst[3], rax);
    d == old(a * b);
```

Figure 3.1 – Type signature for ValeCrypt's implementation of multiplying a 256-bit number (in the `src` array) by a 64-bit number in `b`. The arrays themselves are "ghost" variables, i.e., used only for proof purposes. The signature first declares some local aliases using `lets` (e.g., the pointer to the `dst` array must be in the `rdi` register). The procedure then specifies its framing (the portions of state it reads/modifies). The preconditions show that it expects the CPU to support the ADX and BMI2 extensions; the input and output arrays cannot partially overlap; and the pointers provided are valid. It returns the result of the multiplication in a combination of the destination array and `rax`.

The latest version of Vale [59] can be viewed as a DSL that relies on deeply embedded hardware semantics formalized within F*, which also discharges proof obligations demonstrating the correctness and security of Vale assembly programs.

Vale implementations are verified for safety (programs do not crash), functional correctness (the implementations match their specification on all inputs), and robustness to cache-based and timing-based side-channel attacks.

ValeCrypt includes several implementations of cryptographic primitives. Some of these achieve good performance, such as 750 MB/s for AES-CBC [36] and 990 MB/s for AES-GCM [59], but this still falls short of the fastest OpenSSL assembly language code, which reaches up to 6400 MB/s for AES-GCM [59]. ValeCrypt's implementations are, by design, platform specific and do not offer fallbacks.

For full details on Vale see [36, 59], we use verified Vale code for field arithmetic in §3.4.2.

3.1.2 Relating Low* and Vale

Implementing cryptography in assembly enables the use of hardware-specific features, such as the SHA-EXT [65] and AES-NI [62] instruction sets. It also enables manual optimizations that general-purpose compilers might not detect. To achieve high-performance, EverCrypt therefore needs the ability to verifiably call Vale assembly routines from Low*. To support fine-grained interactions between C and assembly, we provide a verified interoperation framework to reconcile differences in their execution models (e.g., different memory models) while ensuring that specifications of verified Low* and Vale code match precisely. Our work is based closely on prior work relating Low* and Vale by Fromherz et al. [59].

In contrast to prior work that considers C programs interoperating with potentially malicious assembly language contexts [13], the setting in EverCrypt is simpler—C code verified in

Low* interacts with assembly code verified in Vale. We summarize the main features of our interoperation model below.

- Verified Low* programs call into verified Vale procedures or inline assembly fragments.
- Control transfers from Vale back to Low* only via returns; there are no callbacks from Vale to Low*.
- The only observable effects of Vale in Low* are:
  - updates to memory, observed atomically as the Vale program returns (since there are no callbacks, intermediate memory states are not observable);
  - the value in the `rax` register in the final machine state of the Vale program as it returns to Low*; and,
  - digital side-channels due to the trace of instructions or memory addresses accessed by the Vale program.

As such, Vale procedures extend the semantics of Low* with an atomic computational step with effects on memory and a single word-sized result (but with variable execution time as a potential side channel). The goal of our interoperation framework is to safely lift the fine-grained Vale semantics to a Low* specification, so that Low* programs containing atomic Vale steps can be verified within Low*'s program logic.

Modeling Interoperation. Abstractly, a verified Vale procedure satisfies a Hoare triple $\{P\}\ c\ \{Q\}$, meaning that for all Vale machine states s_0 that satisfy P, it is safe to evaluate the Vale instructions c, producing a final state that satisfies Q; i.e., the following property is provable in F*: $\forall s_0.\ P\ s_0 \implies Q\ (\mathsf{eval}\ c\ s_0)$, where eval is a definitional interpreter for the semantics of Vale in F*. We make use of this definitional interpreter to lift a Vale Hoare triple to Low*, as shown in the sketch below:

```
1 let call_assembly c arg1 ... argn
2   : Stack uint64 (requires lift_pre P) (ensures lift_post Q) =
3   let h0 = get () in
4   let s0 = initial_vale_state h0 arg1 ...argn in
5   let s1 = eval c s0 in
6   let rax, h1 = final_lowstar_state h0 s1 in
7   put h1; rax
```

The Low* function call_assembly models calling the Vale code c with arguments $\mathsf{arg}_1 \ldots \mathsf{arg}_n$. Operationally, the call is modeled as follows: at line 3, we retrieve the initial Low* heap h_0; at line 4, we construct the initial Vale state s_0 from h_0 and all the arguments; at line 5, we run the Vale definitional interpreter to obtain the final Vale state s_1; at line 6, we translate this s_1 back to a Low* heap h_1 and return value rax; finally, we update the Low* state atomically with h_1 and return rax. Most importantly, at line 2, we prove that whenever the Vale code satisfies $\{P\}\ c\ \{Q\}$, our operational model of Low*/Vale interoperation is soundly specified by the Low* computation type: Stack uint64 (requires lift_pre P) (ensures lift_post Q). Concretely, each call to assembly from Low* is extracted by our compiler as a C extern signature, whereas the assembly code itself is printed by Vale's simple assembly printer.

Trust assumptions. The definition of the call_assembly wrapper is at the core of our trusted assumptions: it defines the semantics of a call from Low* into Vale. Some salient parts of this trusted definition include the translation of the Low* memory into a Vale memory and back, and the parameter-passing convention. Of course, the Vale evaluator is part of the trusted assumptions about Vale itself, regardless of interoperation with Low*. Notably, the specification of call_assembly (line 2) is not trusted itself, since it is merely an abstraction that is proven to be satisfied by the operational definition. Additionally, we rely on the C compiler, assembler and linker to concretely implement the call as modeled by our formal development.

Relating memory models. The memory models used by Low* and Vale differ significantly. The Low* memory model stores values of structured types: the types include machine integers of various widths (8–128 bits) and signedness; and arrays of structured values (as in C, pointers are just singleton arrays). In contrast, Vale treats memory as just a flat array of bytes. At each call, we may pass several pointers from Low* to Vale; to do so, we assume the existence of a physical address map that assigns a Vale address for each shared pointer. Given this address map, we can build an explicit correspondence from the fragment of Low* memory containing the shared pointers to Vale's flat memory—this involves making explicit the layout of each structured type (e.g., endianness of machine integers), and the layout in contiguous memory of the elements of an array.

Modeling the calling convention. From Vale's perspective, arguments are received in specific registers and spilled on the stack if needed; in contrast, in Low* as in C, arguments are just named. As we construct the initial Vale state, we translate between these (platform-specific) conventions, e.g., on an x64 machine running Linux, the first argument of a function must be passed in the rdi register, and the second in rsi. Further, the wrapper requires that *callee-saved registers* (e.g., r15 for Windows on x64) have the same value when entering and exiting the Vale procedure. Aside from x64 standard calls on Windows and Linux, we also support custom calling conventions in support of inline assembly (subject to restrictions, e.g., the stack register rsp cannot be used to pass arguments, and distinct arguments must be passed in distinct registers). One of the subtleties of modeling the calling convention is to define it once, while accounting for all arities (notice the $\mathsf{arg}_1 \ldots \mathsf{arg}_n$, in the sketch of call_assembly of the previous section)—F*'s support for dependently typed arity-generic programming [17] makes this possible.

Lifting specifications generically from Vale to Low*. To preserve the verification guarantees of a Vale program when called from Low*, the Vale preconditions must be provable in the initial Low* state and arguments in scope. Dually, the Vale postconditions must suffice to continue the proof on the Low* side. A key feature of our interoperation layer is to lift Vale specifications along the mapping between Vale and Low* states, e.g., lift_pre and lift_post reinterpret soundly and generically (i.e., once and for all) pre- and postconditions on Vale's flat memory model and register contents in terms of Low*'s structured memory and named arguments in scope. Relating specifications between Vale and Low* is untrusted—thankfully so, since this is also perhaps the most complex part of our interoperation framework, for two reasons. First, Vale and Low* use subtly different core concepts (each optimized for their particular setting) including different types for integers, and different predicates for memory footprints, disjointness, and liveness of memory locations. Hence, relating their specifications involves working deep within the core of the semantic models of the two languages and proving compatibility properties among these different notions. This is only possible because both languages are embedded within the same underlying logic, i.e., F*. Second, because we model the calling convention generically for all arities, the relations between Vale and Low* core concepts must also be generic, since these state properties of the variable number and types of arguments in scope. However, the payoff for these technical proofs is that they are done once and for all, and their development cost is easily amortized by the relative convenience of instantiating the framework at a specific arity for each call from Low* to Vale.

Side-channel analysis. Whereas the relation between Hoare triples between Low* and Vale is fully mechanized within F*'s logic (as illustrated by the call_assembly sketch), we rely on a meta-theorem to connect the side-channel guarantees provided by Low* and Vale. Specifically, whereas Low*'s guarantees rely on enforcing constant-time properties using a syntactic analysis based on abstract types, Vale's constant-time guarantees arise from a certified taint analysis that is executed on the Vale instructions [36, 59]. This metatheorem (restated in Appendix A) is based on a notion of combined execution traces that include an abstraction of both Low* and Vale instruction sequences, and it proves that when a well-typed Low* program interoperates with a Vale program proven constant-time by the Vale taint analysis, the combined traces produced by these programs are independent of their secret inputs.

3.2 Crafting an Agile, Abstract API for EverCrypt

A key contribution of EverCrypt is the design of its API, which provides abstract specifications suitable for use both in verified cryptographic applications and in unverified code. While agility matters for security and functionality, we also find that it is an important principle to apply throughout our code: beneath the EverCrypt API, we use agility extensively to build generic implementations with a high degree of code and proof sharing between variants of related algorithms.

Figure 3.2 outlines the overall structure of our API and implementations for hashing algorithms—similar structures are used for other classes of algorithms. At the top left (in red), we have trusted, pure specifications of hashing algorithms. Our specifications are optimized for clarity, not efficiency. Nevertheless, we compile them to OCaml and test them using standard test vectors. We discuss specifications further in §3.2.1. To the right of the figure, we have verified optimized implementations. The top-level interface is EverCrypt.Hash, which multiplexes new, efficient imperative implementations written in Low* and Vale. Each of these implementations is typically proven correct against a low-level specification (e.g., Derived.SHA2_256) better suited to proofs of implementation correctness than the top-level Spec.Hash—these derived specifications are then separately proven to refine the top-level specifications. We discuss our top-level API in §3.2.2.

For reuse within our verified code, we identify several generic idioms. For instance, we share a generic Merkle-Damgård construction [86, 46] between all supported hash algorithms. Similarly, we obtain all the SHA2 variants from a generic template. The genericity saves verification effort at zero runtime cost—using F*'s support for partial evaluation, we extract fully specialized C and assembly implementations of our code (§3.3).

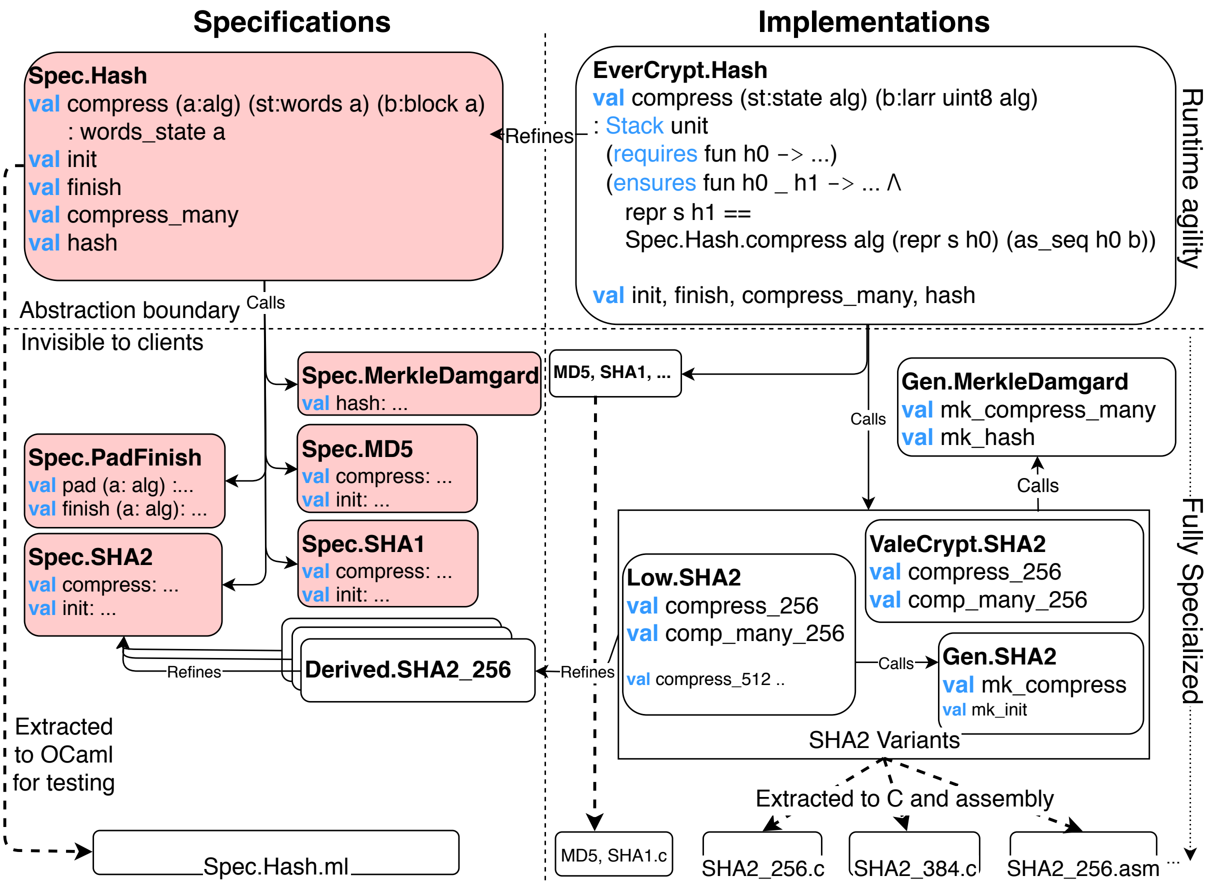


Figure 3.2 – The modular structure of EverCrypt (illustrated on hashing algorithms)

3.2.1 Writing Agile Specifications

We factor common structure shared by multiple specifications into "generic" functions parameterized by an algorithm parameter, and helper functions that branch on it to provide algorithm-specific details. This reduces the potential for errors, makes the underlying cryptographic constructions more evident, and provides a blueprint for efficient generic implementations (§3.3)

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

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

Although MD5 and SHA1 are known to be insecure [34, 123, 80], a practical provider must supply them for compatibility reasons. At the application level, cryptographic security theorems can be conditioned on the security of the algorithms used and, as such, would exclude MD5 or SHA1. Pragmatically, EverCrypt can also be configured to disable them, or even exclude their code at compile time.

All these algorithms use the Merkle-Damgård construction [86, 46] for hashing a bytestring by (1) slicing it into input blocks, with an encoding of its length in the final block; (2) calling a core, stateful compression function on each block; (3) extracting the hash from the final state. Further, the four members of the SHA2 family differ only on the lengths of their input blocks and resulting tags, and on the type and number of words in their intermediate state. Rather than writing different specifications, we define a generic state type parameterized by the algorithm:

```
let word alg = match alg with
  | MD5 | SHA1 | SHA2_224 | SHA2_256 → UInt32.t
  | SHA2_384 | SHA2_512 → UInt64.t
let words alg = m:seq (word alg){length m = words_length alg}
let block alg = b:bytes {length b = block_length alg}
```

Depending on the algorithm, the type word alg selects 32-bit or 64-bit unsigned integer words; words alg defines sequences of such words of the appropriate length; and block alg defines sequences of bytes of the appropriate block length. With these types, we write a generic SHA2 compression function that updates a hash state (st) by hashing an input block (b). Note, this definition illustrates the benefits of programming within a dependently typed framework—we define a single function that operates on either 32-bit or 64-bit words, promoting code and proof reuse and reducing the volume of trusted specifications.

```
module Spec.SHA2
let compress (alg:sha2_alg) (st:words alg) (b:block alg) : words_state alg =
  let block_words = words_of_bytes alg (block_length alg) b in
  let st’ = shuffle alg st block_words in
  seq_map2 (word_add_mod alg) st st’
```

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

3.2.2 EverCrypt’s Top-level API

Verified programming is a balancing act: programs must be specified precisely, but revealing too many details of an implementation breaks modularity and makes it difficult to revise or extend the code without also breaking clients. A guiding principle for EverCrypt’s API is to hide, through abstraction, as many specifics of the implementation as possible. We use abstraction in two flavors. *Specification* abstraction hides details of an algorithm’s specification from its client, e.g., although EverCrypt.Hash.compress is proven to refine Spec.Hash.compress, only the type signature of the latter, not its definition, is revealed to clients. In addition, *representation* abstraction hides details of an implementation’s data structures, e.g., the type used to store the hash’s internal state is hidden.

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

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

The interface of our low-level compress function is shown below, with some details elided.

```
module EverCrypt.Hash
val compress (s:state alg) (b:larr uint8 alg) : Stack unit
 (requires λ h0 → inv s h0 ∧ b ∈ h0 ∧ fp s h0 ‘disjoint‘ loc b)
 (ensures λ h0 _ h1 → inv s h1 ∧ modifies (fp s h0) h0 h1 ∧
     repr s h1 == Spec.Hash.compress alg (repr s h0) (as_seq h0 b))
```

Clients of compress must pass in an abstract state handle s (indexed by an implicit algorithm descriptor alg), and a mutable array b:larr uint8 alg holding a block of bytes to be added to the hash state. As a precondition, they must prove inv s h0, the abstract invariant of the state. This invariant is established initially by the state allocation routine, and standard framing lemmas ensure that the invariant still holds for subsequent API calls as long as any intervening heap updates are to disjoint regions of the heap. Separating allocation from initialization is a common low-level design decision, as it allows clients to reuse memory, an important optimization. In addition to the invariant, clients must prove that the block b is live; and that b does not overlap with the abstract footprint of s (the memory locations of the underlying hash state). The Stack unit annotation states that compress is free of memory leaks and returns the uninformative unit value (). As a postcondition, compress preserves the abstract invariant; it guarantees that only the internal hash state is modified; and, most importantly, that the final value held in the hash state corresponds *exactly* to the words computed by the pure specification Spec.Hash.compress. It is this last part of the specification that captures functional correctness, and justifies the safety of multiplexing several implementations of the same algorithm behind the API, inasmuch as they are verified to return the same results, byte for byte.

State abstraction is reflected to C clients as well, by compiling the state type as a pointer to an incomplete struct [114]. Hence, after erasing all pre- and post-conditions, our sample low-level interface yields an idiomatic C function declaration in the extracted evercrypt_hash.h listed below.

```
struct state_s;
typedef struct state_s state;
void compress (state *s, uint8 *b);
```

Given an abstract, agile, functionally correct implementation of our 6 hash algorithms, we develop and verify the rest of our API for hashes in a generic manner. We first build support for incremental hashing (similar to compress, but taking variable-sized bytestrings as inputs), then an agile standard-based implementation of keyed-hash message authentication codes (HMAC) and finally, on top of HMAC, a verified implementation of key derivation functions (HKDF) [75].

Appendix B lists the resulting C API and sample extracted code. We expect this API to be stable throughout future versions of EverCrypt. Thanks to agility, adding a new algorithm (e.g., a hash) boils down to extending an enumeration (e.g., the hash algorithm type) with a new case. This is a backward-compatible change that leaves function prototypes identical. Thanks to multiplexing, adding a new optimized implementation is purely an implementation matter that is dealt with automatically within the library, meaning once again that such a change is invisible to the client. Finally, thanks to abstract state and framing lemmas, EverCrypt can freely optimize its representation of state, leaving verified and unverified clients equally unscathed.

3.3 Agile Implementations of the API with Zero-cost Generic Programming

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

3.3.1 Implementing EverCrypt.Hash

The abstract type state alg is defined in F* as a pointer to a datatype holding algorithm-specific state representations, as shown below:

```
type state_s (a: alg) =
  match a with
  | SHA2_256_s: p:hash_state SHA2_256 → state_s SHA2_256
  | SHA2_384_s: p:hash_state SHA2_384 → state_s SHA2_384
  | ...
let state alg = pointer (state_s alg)
```

The state_s type is extracted to C as a tagged union, whose tag indicates the algorithm alg and whose value contains a pointer to the internal state of the corresponding algorithm. The union incurs no space penalty compared to, say, a single void∗, and avoids the need for dangerous casts from void∗ to one of uint32_t∗ or uint64_t∗. The tag allows an agile hash implementation to dynamically dispatch based on the algorithm, as shown below for compress:

```
let compress s blocks =
  match !s with
  | SHA2_256_s p → compress_multiplex_sha2_256 p blocks
  | SHA2_384_s p → compress_sha2_384 p blocks
  | ...
```

In this code, since we only have one implementation of SHA2-384, we directly call into Low*. For SHA2-256, however, since we have multiple implementations in Low* and Vale, we dispatch to a local multiplexer that selects the right one.

3.3.2 Partial Evaluation for Zero-cost Genericity

Abstract specifications and implementations, while good for encapsulation, modularity, and code reuse, can compromise the efficiency of executable code. We want to ensure that past the agile EverCrypt.Hash nothing impedes the run-time performance of our code. To that end, we now show how to efficiently derive specialized Low* code, suitable for calling by EverCrypt.Hash, by partially evaluating our verified source *code*, reducing away several layers of abstraction before further compilation. The C code thus emitted is fully specialized and abstraction-free, and branching on algorithm descriptors only happens above the specialized code, where the API demands support for runtime configurability (e.g., only at the top-level of EverCrypt.Hash). As such, we retain the full generality of the agile, multiplexed API, while switching efficiently and only at a coarse granularity between fast, abstraction-free implementations (Figure 3.2).

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

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

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

However, a bit of inlining and partial evaluation goes a long way. We program most of our *stateful* code in a case-generic manner. Just like in specifications, the stateful compression function is written once in a generic manner (in Gen.SHA2); we trigger code specialization at the top-level by defining all the concrete instances of our agile implementation, as follows:

```
module Low.SHA2
let compress_224 = Gen.SHA2.compress SHA2_224
let compress_256 = Gen.SHA2.compress SHA2_256
let compress_384 = Gen.SHA2.compress SHA2_384
let compress_512 = Gen.SHA2.compress SHA2_512
```

When extracting, say compress_256, F* will partially evaluate Gen.SHA2.compress on SHA2_256, eventually encountering word_logand SHA2_256 and reducing it to UInt32.logand. By the time all reduction steps have been performed, no agile code remains, and all functions and types that were parameterized over alg have disappeared, leaving specialized implementations for the types, operators and constants that are specific to SHA2-256 and can be compiled to efficient, idiomatic C code.

We take this style of partial evaluation one step further, and parameterize stateful code over algorithms *and* stateful functions. For instance, we program a generic, *higher-order* Merkle-Damgård hash construction [86, 46] instantiating it with specific compression functions, including multiple implementations of the same algorithm, e.g., implementations in Low* or Vale. Specifically, the mk_compress_many function is parameterized by a compression function f, which it applies repeatedly to the input.

```
val mk_compress_many (a:hash_alg) (f:compress_st a) : compress_many_st a
```

We obtain several instances of it, for the same algorithm, by applying it to different implementations of the same compression function, letting F* specialize it as needed.

```
let compress_many_256_vale: compress_many_st SHA2_256 =
  mk_compress_many SHA2_256 compress_sha2_256_vale
let compress_many_256: compress_many_st SHA2_256 =
  mk_compress_many SHA2_256 compress_sha2_256
```

This higher-order pattern allows for a separation of concerns: the many-block compression function need not be aware of *how* to multiplex between Low* and Vale, or even of *how many* choices there might be. We extend this higher-order style to our entire hash API: for instance, mk_hash generates a one-shot hash function. We then instantiate the entire set of functions, yielding two specialized APIs with no branching: one for Low* code, one for Vale code.

This technique is akin to C++ templates, in that both achieve zero-cost abstractions. F*, however, allows us to verify by construction that any specialization satisfies an instantiation of the generic specification, unlike C++ templates which perform textual expansion and repeated checks at each instantiation for typability in C++'s comparatively weak type system.

3.4 Verifying Curve25519

The Curve25519 elliptic curve [32], standardized as IETF RFC7748 [77], is quickly emerging as the default curve for modern cryptographic applications. It is the only elliptic curve supported by protocols like Signal [5] and Wireguard [51], and is one of two curves commonly used with Transport Layer Security (TLS) and Secure Shell (SSH). Curve25519 was designed to be fast, and many high-performance implementations optimized for different platforms have been published [32, 44, 54, 95] and added to mainstream cryptographic libraries.

3.4.1 Mathematical Description

Curve25519 can be implemented in about 500 lines of C. About half of this code consists of a customized bignum library that implements modular arithmetic over the field of integers modulo the prime $p = 2^{255} - 19$. The most performance-critical functions implement multiplication and squaring over this field.

Field arithmetic with a radix-2^{64} representation. Since each field element has up to 255 bits, it can be stored in 4 64-bit machine words, encoding a polynomial of the form:

$$e_3 \cdot 2^{192} + e_2 \cdot 2^{128} + e_1 \cdot 2^{64} + e_0$$

where each coefficient is less than 2^{64}. Multiplying (or squaring) field elements amounts to schoolbook multiplication with a 64-bit radix: whenever a coefficient in an intermediate polynomial goes beyond 64-bits, we need to carry over the extra bits to the next-higher coefficient. To

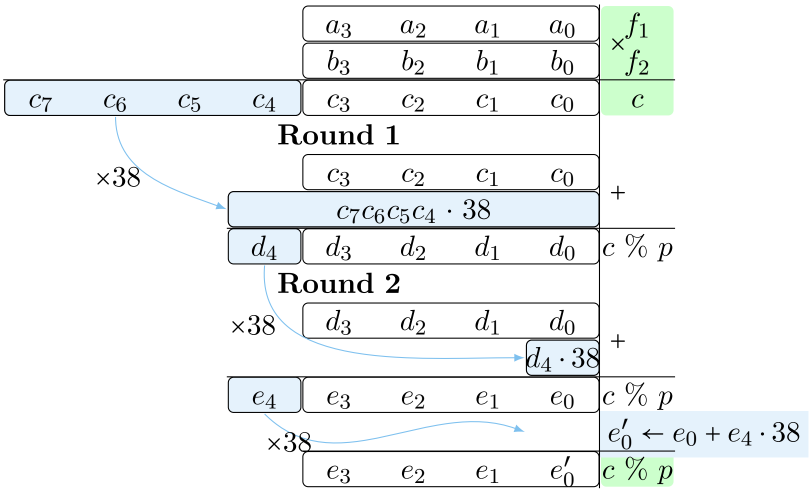


Figure 3.3 – Multiplication of two 256-bit numbers, f_1 and f_2, followed by lazy reduction modulo $p = 2^{255} - 19$ with a radix-2^{64} representation. The result is a 256-bit number.

avoid a timing side-channel, we must assume that every 64-bit addition may lead to a carry and propagate the (potentially zero) carry bit regardless.

To carry out lazy reduction modulo the prime (i.e., the result is not necessarily less than the prime) for the product c of two field elements, f_1 and f_2, we need to perform the following two rounds depicted in Figure 3.3. For the first round, we split the 512-bit number c into two 256-bit numbers, c_{high} and c_{low}, such that $c = c_{high} \cdot 2^{256} + c_{low}$. We then add its high part c_{high} multiplied by 38 to its low part c_{low}, as it can be derived from the following:

$$2 \cdot (2^{255} - 19) \,\%\, p = 0, \text{ hence, } 2^{256} \,\%\, p = 38$$

$$c \,\%\, p = (c_{high} \cdot \underbrace{2^{256}}_{\%\, p \,=\, 38} + c_{low}) \,\%\, p = (c_{high} \cdot 38 + c_{low}) \,\%\, p.$$

The result of the first round is stored in five 64-bit machine words $(d_4d_3d_2d_1d_0)_{2^{64}}$, where d_4 has at most 6 bits. Hence, we perform the second round, where $d_4 \cdot 38 < 2^{64}$ is added to $(d_3d_2d_1d_0)_{2^{64}}$. The addition results in a 257-bit number $(e_4e_3e_2e_1e_0)_{2^{64}}$ with $e_4 \leq 1$. To obtain a final 256-bit result, we need to update only e_0 with the value $e_0 + e_4 \cdot 38$, which fits into a 64-bit machine word. The latter can be justified by considering the two cases when adding $d_4 \cdot 38$ to d_0 gives a carry or no carry, as e_0 stores the result of $(d_0 + d_4 \cdot 38) \,\%\, 2^{64}$:

Case 1. if $d_0 + d_4 \cdot 38 \geq 2^{64}$ then $e_0 + e_4 \cdot 38 \leq e_0 + 38 = (d_0 + d_4 \cdot 38) \,\%\, 2^{64} + 38$

$= d_0 + d_4 \cdot 38 - 2^{64} + 38 \leq (2^{64} - 1) + (2^6 - 1) \cdot 38 - 2^{64} + 38 = 2^6 \cdot 38 - 1 < 2^{64}$

Case 2. if $d_0 + d_4 \cdot 38 < 2^{64}$ then $e_4 = 0$

For the first case, we have enough room to store the result of $(e_0 + e_4 \cdot 38)$, regardless of the value $e_4 \leq 1$. For the second case, $e_4 = 0$.

Field arithmetic with a radix-2^{51} representation. Propagating carries can be quite expensive, so a standard optimization is to delay carries by using an *unpacked* representation, with a field element stored in 5 64-bit machine words, each holding 51 bits, yielding a radix-2^{51} polynomial:

$$e_4 \cdot 2^{204} + e_3 \cdot 2^{153} + e_2 \cdot 2^{102} + e_1 \cdot 2^{51} + e_0$$

While this representation leads to 9 more 64x64 multiplications, since each product now has only 102 bits, it has lots of room to hold extra carry bits without propagating them until the final modular reduction. Modular arithmetic using this representation is implemented similar to the Poly1305 field arithmetic (§2.4).

Faster Curve25519 with Intel ADX. In 2017, Oliveira et al. [95] demonstrated a significantly faster implementation of Curve25519 on Intel platforms that support Multi-Precision Add-Carry Instruction Extensions, also called Intel ADX [101]. Unlike other fast Curve25519 implementations, Oliveira et al. use a radix-2^{64} representation and instead optimize the carry propagation code by carefully managing Intel ADX's second carry flag (Figure 3.4). The resulting performance improvement is substantial—at least 20% faster than prior implementations.

Oliveira et al. wrote their implementation mostly in assembly. A year after its publication, when testing and comparing this code against formally verified implementations taken from HACL* [127] and Fiat-Crypto [56], Donenfeld and others found several critical correctness bugs [53]. These bugs were fixed [49], with a minor loss of performance, but they raised concerns as to whether this Curve25519 implementation, with the best published performance, is trustworthy enough for deployment in mainstream applications.

3.4.2 Verifying Field Arithmetic

In EverCrypt, we include two new versions of Curve25519. The first is written in Low* and generates portable C code that uses a radix-2^{51} representation. The second relies on verified Vale assembly for low-level field arithmetic that uses a radix-2^{64} representation, inspired by Oliveira et al.'s work.

Notably, we carefully factor out the *generic* platform-independent Curve25519 code, including field inversion, curve operations, key encodings, etc., so that this code can be shared between our two implementations (Figure 3.5). In other words, we split our Curve25519 implementation into two logical parts: **(1)** the low-level field arithmetic, implemented both in Vale and in Low*, but verified against the same mathematical specification in F*, and **(2)** a high-level curve implementation in Low* that can use either of the two low-level field-arithmetic implementations.

This section covers the first part, whereas the second is discussed in more detail in §3.4.3.

Specifying field arithmetic in F*. The mathematical specification of the underlying finite-field arithmetic $\mathbb{Z}_p$ is simple and can be written as follows.

```
let prime : pos = pow2 255 − 19
let elem = x:nat{x < prime}
let ( +% ) (x y:elem) = (x + y) % prime
let ( −% ) (x y:elem) = (x − y) % prime
let ( ∗% ) (x y:elem) = (x ∗ y) % prime
```

It defines a constant prime as $2^{255} - 19$ and a type elem for a field element as a non-negative mathematical integer less than prime. It also defines the following field operations: addition (+%), subtraction (−%) and multiplication (∗%).

Specifying and verifying field arithmetic in Low*. In order to write a generic implementation for Curve25519, we introduce a Curve25519.Fields module that defines a shared in-

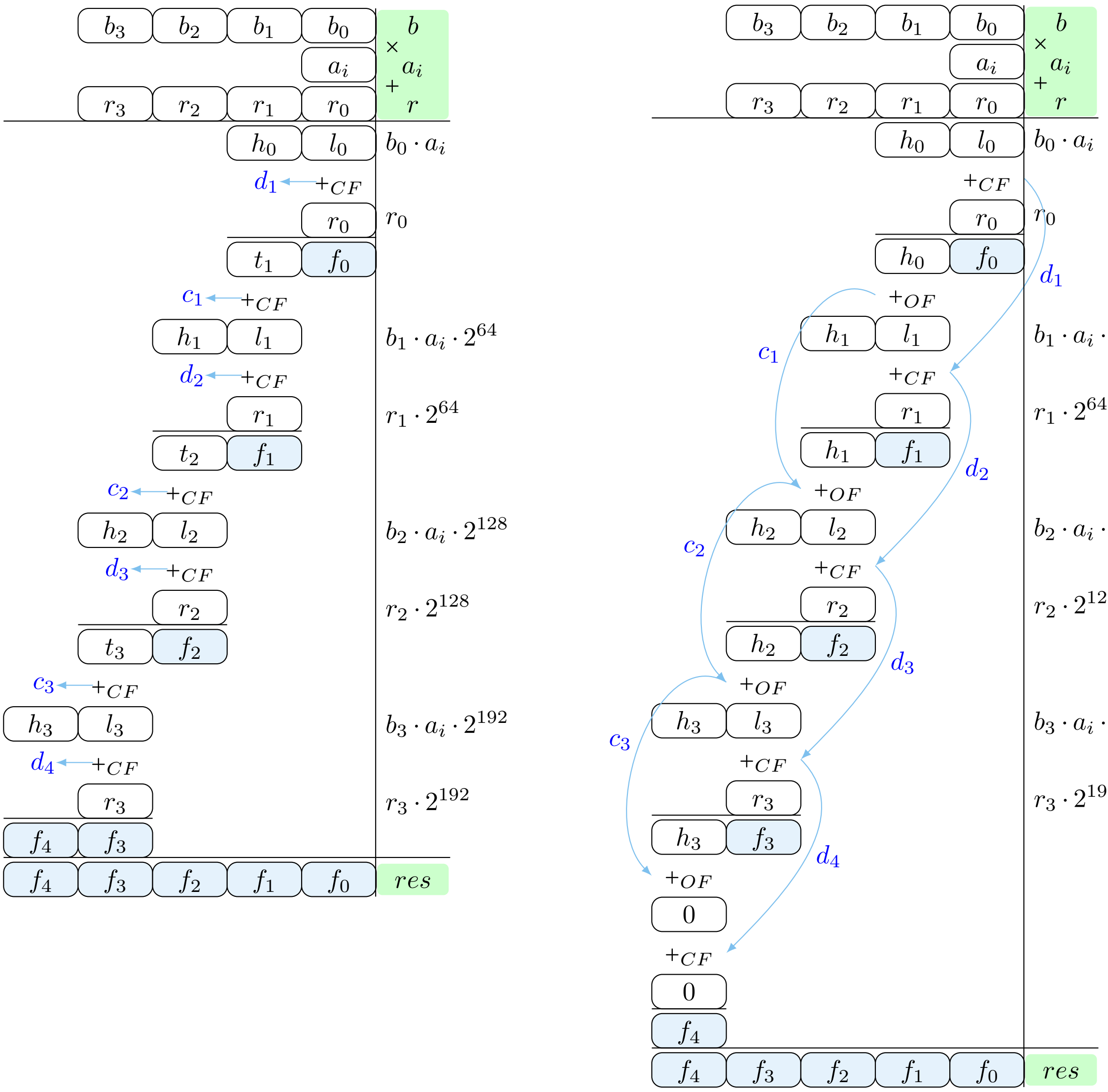


Figure 3.4 – 256 × 64-bit multiplication with a 256-bit accumulator.
On the left, using the ADC and ADD instructions, and on the right, using the ADX and MULX instructions. The ADC and ADD instructions compute addition with and without a carry, respectively, and share the same carry flag. The ADX instruction set supports two additions, ADOX and ADCX, with independent carry flags, which allows computing multiplication with two parallel carry chains.

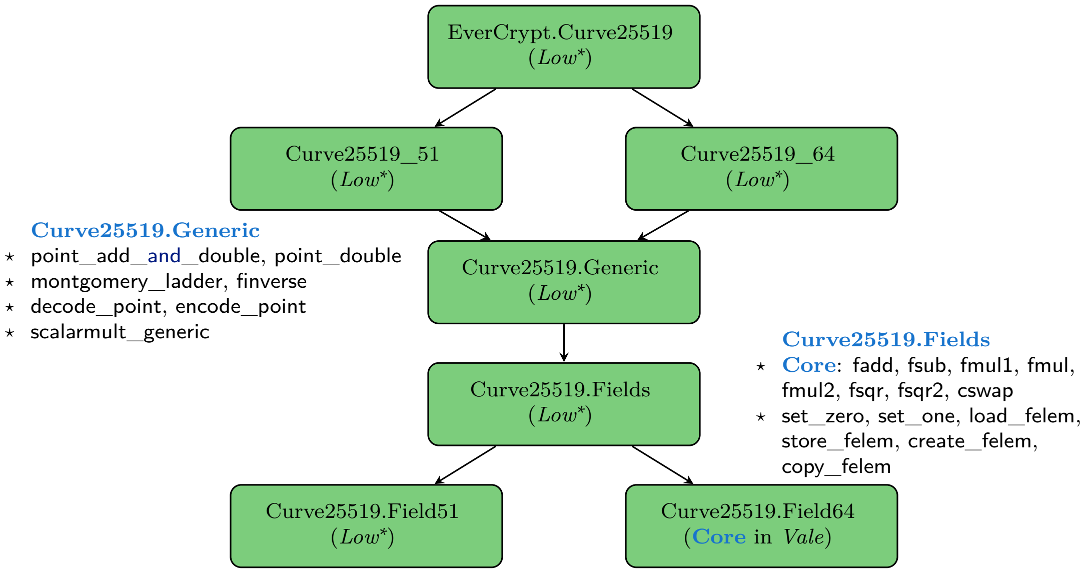


Figure 3.5 – Structure of Curve25519 in EverCrypt.

terface between our two implementations for the low-level arithmetic, Curve25519.Field51 and Curve25519.Field64. The first is written in Low* and uses a radix-2^{51} representation (M51), whereas the second is implemented in Vale and uses a radix-2^{64} representation (M64). The field_spec type enumerates both versions.

```
(∗ Curve25519.Fields ∗)
type field_spec = | M51 | M64
let nlimb (s:field_spec) = match s with | M51 → 5ul | M64 → 4ul
let felem (s:field_spec) = lbuffer uint64 (nlimb s)
```

Depending on a type field_spec value s, a field element is stored in a mutable array of either five 64-bit machine words (s=M51) or four 64-bit machine words (s=M64). The function as_nat computes the mathematical integer of a given array e in a given memory h, whereas feval reflects it into a field element, a non-negative mathematical integer less than prime.

```
(∗ Curve25519.Fields ∗)
let as_nat (#s:field_spec) (h:mem) (e:felem s): GTot nat = let e = as_seq h e in match s with
  | M51 → v e.[0] + v e.[1] ∗ pow2 51 + v e.[2] ∗ pow2 102 + v e.[3] ∗ pow2 153 + v e.[4] ∗ pow2 204
  | M64 → v e.[0] + v e.[1] ∗ pow2 64 + v e.[2] ∗ pow2 128 + v e.[3] ∗ pow2 192

let feval (#s:field_spec) (h:mem) (e:felem s) : GTot elem = as_nat h e % prime
```

Next, we define a shared signature for low-level arithmetic such as modular addition, subtraction, multiplication, and squaring. We also implement functions that can multiply and square two elements at the same time and can be used for curve operations shown in Figure 3.6. In Vale, we implement these functions using the double carry flags of Intel ADX.

For example, we specify field multiplication as a function fmul that takes two field elements, f1 and f2, computes their product modulo prime and writes the result in out.

For memory safety, the precondition of fmul requires the liveness and either disjointness or equality of the input and output arrays. The postcondition of fmul ensures that only out

$$P = (x_2/z_2, \_),\ Q = (x_3/z_3, \_),\ R = Q - P = (x_1, \_)$$

$$\begin{array}{rllll}
 & & a = x_2 + z_2 \quad c = x_3 + z_3 & & \\
 & & b = x_2 - z_2 \quad d = x_3 - z_3 & & \\
\mathsf{fsqr2} & \{ \ aa = a \cdot a & da = d \cdot a & \} & \mathsf{fmul2} \\
 & \{ \ bb = b \cdot b & cb = c \cdot b & \} & \\
 & e = aa - bb & x_3 = da + cb & & \\
\mathsf{fmul1} & [\ e_{121665} = e \cdot 121665 & z_3 = da - cb & & \\
 & aa\_e_{121665} = e_{121665} + aa & x_3 = x_3 \cdot x_3 & \} & \mathsf{fsqr2} \\
\mathsf{fmul2} & \{ \ x_2 = aa \cdot bb & z_3 = z_3 \cdot z_3 & \} & \\
 & \{ \ z_2 = e \cdot aa\_e_{121665} & z_3 = z_3 \cdot x_1 &] & \mathsf{fmul} \\
 & (x_2/z_2, \_) = P + P & (x_3/z_3, \_) = P + Q & &
\end{array}$$

Figure 3.6 – x-coordinate point doubling and addition for Curve25519 from RFC7748.

```
(* Curve25519.Fields *)
val fmul (#s:field_spec) (out f1 f2:felem s) : Stack unit
  (requires λ h → live h out ∧ live h f1 ∧ live h f2 ∧ fmul_pre h f1 f2 ∧
    eq_or_disjoint f1 f2 ∧ eq_or_disjoint f1 out ∧ eq_or_disjoint f2 out)
  (ensures λ h0 _ h1 → modifies (loc out) h0 h1 ∧
    fmul_post h1 out ∧ feval h1 out == feval h0 f1 *% feval h0 f2)

let fmul #s out f1 f2 = match s with
  | M51 → fmul_51 out f1 f2 (* from Curve25519.Field51 *)
  | M64 → fmul_64 out f1 f2 (* from Curve25519.Field64 *)
```

is modified, leaving other disjoint objects in memory unchanged. The functional correctness guarantees that the result out is the product of f1 and f2 followed by reduction modulo prime if, in addition, the fmul_pre predicate holds. The predicate contains the requirements for each specialized version of fmul that must be fulfilled before their call.

```
(* Curve25519.Fields *)
let fmul_pre (#s:field_spec) (h:mem) (f1 f2:felem s) : Type0 = match s with
  | M51 → f51_felem_fits h f1 (...) ∧ f51_felem_fits h f2 (...)
  | M64 → Vale.X64.CPU_Features_s.(adx_enabled ∧ bmi2_enabled)

let fmul_post (#s:field_spec) (h:mem) (f:felem s) : Type0 = match s with
  | M51 → f51_felem_fits h f (...)
  | M64 → ⊤
```

For s=M51, similar to the Poly1305 field arithmetic (§2.4), we introduce the f51_felem_fits predicate to ensure the absence of integer overflow during the computations. We also optimize the computations by skipping the modular reduction steps after addition and subtraction, as a sequence of arithmetic operations for point doubling and addition depicted in Figure 3.6 is fixed. Thus, we do not preserve a single invariant for a field element but derive a separate one for each function.

For s=M64, the fmul function expects the CPU to support the ADX and BMI2 instructions. We also implicitly specify the invariant for field elements with a radix-2^{64} representation, that is, fmul takes as input two fields elements, f1 and f2, under 2^{256} and outputs a field element out less than 2^{256}. This invariant always holds for a field element stored in four 64-bit machine words.

Specifying and verifying arithmetic in Vale. Let us consider a textbook multiplication of two 256-bit integers using a radix-2^{64} representation in more detail. This arithmetic operation amounts to a 256 × 64-bit multiplication with a 256-bit accumulator, depicted on the left side of Figure 3.4. The main computation has the form of $a_i \cdot b_i + t_i + r_i$, where a_i, b_i, t_i and r_i are 64-bit integers. The result fits into a 128-bit integer, as

$$a_i \cdot b_i + t_i + r_i \leq (2^{64} - 1) \cdot (2^{64} - 1) + (2^{64} - 1) + (2^{64} - 1) = 2^{128} - 1.$$

In assembly, we would first perform a double-wide multiplication of a_i and b_i and store the product in two 64-bit integers h_i and l_i, so that $h_i \cdot 2^{64} + l_i = a_i \cdot b_i$. Then, we would perform the addition of l_i and t_i that can yield a carry c_i, which we would have to immediately add to h_i before carrying out the next addition with r_i, as the ADD and ADC instructions share the same carry flag:

$$\begin{array}{ll}
(h_i, l_i) \leftarrow a_i \cdot b_i & \text{MUL, modifies } CF \\
(c_i, f_i) \leftarrow l_i + t_i & \text{ADD, } c_i \rightarrow CF \\
(\_, t_{i+1}) \leftarrow h_i + 0 + c_i & \text{ADC, } c_i \leftarrow CF \\
(d_{i+1}, f_i) \leftarrow f_i + r_i & \text{ADD, } d_{i+1} \rightarrow CF \\
(\_, t_{i+1}) \leftarrow t_{i+1} + 0 + d_{i+1} & \text{ADC, } d_{i+1} \leftarrow CF
\end{array}$$

Therefore, to compute $t_{i+1} \cdot 2^{64} + f_i = a_i \cdot b_i + t_i + r_i$, we would need to invoke the instructions sequentially in the following order: MUL, ADD, ADC, ADD, ADC.

However, this computation can be performed more efficiently using the double carry flags of Intel ADX [101], as depicted on the right side of Figure 3.4.

$$\begin{array}{ll}
(h_i, l_i) \leftarrow a_i \cdot b_i & \text{MULX, no flags are modified} \\
(c_i, f_i) \leftarrow l_i + h_{i-1} + c_{i-1} & \text{ADOX, } c_{i-1} \leftarrow OF,\ c_i \rightarrow OF \\
(d_{i+1}, f_i) \leftarrow f_i + r_i + d_i & \text{ADCX, } d_i \leftarrow CF,\ d_{i+1} \rightarrow CF
\end{array}$$

The ADOX and ADCX instructions compute addition with a carry. The difference is that ADOX uses the OF flag for the carry-in and carry-out, whereas ADCX uses the CF flag for the carry-in and carry-out. Hence, we can perform the multiplication with two independent carry chains.

As the ADX instruction set is not available from C, we use Vale to formally verify that this assembly code matches our F* specification for field arithmetic. *We do not describe the proof in detail because it was not part of this thesis work.*

3.4.3 Implementing Curve25519

In §3.4.2, we define a shared interface for field arithmetic, where all functions are parametrized by a type field_spec value s. As with EverCrypt.Hash implementation (§3.3.2), we use verified zero-cost generic programming techniques to write the platform-independent Curve25519 functions once in a generic manner. Then, we define the concrete instances of the top-level functions to generate specialized versions of Curve25519.

For example, a scalar multiplication function scalarmult_generic computes a shared secret shared, given a user's secret key my_priv and another user's public key their_pub. The function first decodes the secret and public keys and then encodes the result of invoking the Montgomery ladder function.

```
(* Curve25519.Generic *)
let scalarmult_generic (#s:field_spec) shared my_priv their_pub =
 ...; decode_point #s init their_pub;
 montgomery_ladder #s init my_priv init;
 encode_point #s shared init; ...

(* Curve25519_51 *)
let scalarmult_51 = scalarmult_generic #M51
(* Curve25519_64 *)
let scalarmult_64 = scalarmult_generic #M64
```

When extracting the concrete instance, F* partially evaluates scalarmult_generic on a given s, leaving no mention of it by inlining specialized implementations for the constants, types and operators that can be further compiled to idiomatic C code. In §4.2.3, we discuss in more detail how one can annotate functions with [@@Specialize] and [@@Inline] to control the call-graph of the extracted C code. The former means that a function should remain in the call-graph, whereas the latter has to be inlined in the caller's body.

Finally, based on the runtime configuration, EverCrypt selects the most suitable implementation of Curve25519:

```
(* EverCrypt.Curve25519 *)
let scalarmult shared my_priv their_pub =
  let has_bmi2 = EverCrypt.AutoConfig2.has_bmi2 () in
  let has_adx = EverCrypt.AutoConfig2.has_adx () in
  if EverCrypt.TargetConfig.hacl_can_compile_vale && has_bmi2 && has_adx then
    scalarmult_64 shared my_priv their_pub
  else
    scalarmult_51 shared my_priv their_pub
```

Fast Verified Curve25519 in EverCrypt. In Low*, we also optimize our ladder implementation to reduce the number of conditional swaps, and to skip point addition in cases where the bits of the secret key are fixed. With the aid of these optimizations, we are able to produce a mixed assembly-C implementation of Curve25519 that slightly outperforms that of Oliveira et al. but with formal proofs of memory safety, functional correctness, and secret independence for the assembly code, C code, and the glue code in between (§3.4.4).

3.4.4 Run-Time Performance of Curve25519

Table 3.1 measures the performance of our implementations of Curve25519 against that of other implementations, including OpenSSL, Erbsen et al. [56] (one of the fastest verified implementations), and Oliveira et al. [95] (one of the fastest unverified implementations). All measurements are collected on an Intel Kaby Lake i7-7560 using a Linux kernel crypto benchmarking suite [52]. OpenSSL cannot be called from the kernel and was benchmarked using the same script, but in user space. All code was compiled with GCC 7.3 with flags `-O3 -march=native -mtune=native`.

Implementation	Radix	Language	CPU cycles
donna64 [1]	2^{51}	64-bit C	159634
fiat-crypto [56]	2^{51}	64-bit C	145248
amd64-64 [42]	2^{51}	Intel x86_64 asm	143302
sandy2x [44]	$2^{25.5}$ (*)	Intel AVX asm	135660
EverCrypt portable (*this work*)	2^{51}	64-bit C	135636
openssl* [3]	2^{64}	Intel ADX asm	118604
Oliveira et al. [95]	2^{64}	Intel ADX asm	115122
EverCrypt targeted (*this work*)	2^{64}	64-bit C + Intel ADX asm	113614

Table 3.1 – Performance comparison between Curve25519 implementations, where fiat-crypto [56] and EverCrypt (*this work*) are formally verified.
(*) $2^{25.5}$ is a mixed-radix representation, where 26 bits are used in the first limb, 25 bits are used for the second limb, 26 bits are used for the third one and so on, with the following evaluation function: $e_0 + e_1 \cdot 2^{26} + e_2 \cdot 2^{51} + e_3 \cdot 2^{77} + \ldots$

Our results show that, with optimizations from Section 3.4, EverCrypt's combined Low*+Vale implementation narrowly exceeds that of Oliveira et al. by about 1%, which in turn exceeds that of OpenSSL by 3%. We also exceed the previous best verified implementation from Erbsen et al. by 28%. To the best of our knowledge, this makes EverCrypt's implementation of Curve25519 the fastest verified or unverified implementation on record. Notably, our optimizations also improve our portable C implementation, which is now the second fastest verified implementation, beating Erbsen et al.'s implementation by 7.1%.

3.5 Conclusions

The EverCrypt API is carefully designed to support algorithm agility and multiplexing, but seals many details behind an abstraction boundary to simplify verification for clients. We program its implementation generically to reduce specification bloat and increase code and proof reuse, without harming performance. Indeed, our targeted implementation of Curve25519 meets or exceeds the performance of state-of-the-art unverified implementations, and handily exceed previous verified implementations. Several algorithms from EverCrypt have been deployed in Mozilla's NSS cryptographic library, the Wireguard VPN, the Zinc crypto library for the Linux Kernel, and the Tezos blockchain.

In the following chapter, we discuss how to build a library of verified vectorized cryptographic implementations that produce more efficient code for hashes (Blake2 and SHA-2) and the ChaCha20-Poly1305 AEAD. All the code is integrated into the HACL* project and is available through the EverCrypt API.

Chapter 4

HACL×N: Verified Generic SIMD Crypto

This chapter describes the research that was previously published in the following paper:

[103] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. HACLxN: Verified generic SIMD crypto (for all your favourite platforms). In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 899–918. ACM Press, November 2020

I am the first author of this paper. I wrote a verified vectorized implementation of Poly1305 and proved that vectorized specifications for Poly1305, ChaCha20 and SHA2-mb are functionally equivalent to their original (scalar) specifications.

4.1 High-Performance SIMD Crypto

When a new cryptographic algorithm is standardized, the designers usually describe (and sometimes include as an appendix) a reference implementation that would work on any 32-bit computer. However, the algorithm and its parameters are often chosen carefully to admit platform-specific optimizations. For example, new authenticated encryption schemes like ChaCha20-Poly1305 and hash algorithms like Blake2 were deliberately designed to enable Single Instruction Multiple Data (SIMD) vectorization. Since most desktops and smartphones are now equipped with SIMD-enabled processors capable of computing over 4, 8, or 16 32-bit integers in parallel, the performance impact of vectorization can be dramatic.

SIMD Parallelization for Cryptographic Code. Consider ChaCha20, a counter-mode (CTR) encryption algorithm standardized in IETF RFC 7539. The OpenSSL library includes a reference implementation of ChaCha20 (written by Bernstein) in 122 lines of portable C code. When compiled with GCC, this code takes between 4-9 cycles to encrypt a byte on modern processors. For better performance, OpenSSL also includes a dozen other hand-written assembly implementations of ChaCha20 (totaling over 10K lines), for various generations of Intel and

ARM processors. Each implementation exploits platform-specific SIMD instructions to parallelize ChaCha20 for maximum speed. For example, encryption takes just 0.56 cycles per byte on a server with an AVX512-enabled Intel processor (see Table D.2).

The task of writing such optimized vectorized implementations can conceptually be divided into three stages. First, we apply high-level *algorithmic transformations* that rearrange the cryptographic computation in a way that allows multiple arithmetic operations to be performed in parallel. Some transformations are algorithm-specific while others are generic patterns that apply to multiple algorithms. In ChaCha20, for example, we can parallelize the inner block cipher (as intended by the designers), or we can transform the generic CTR loop to process multiple blocks at a time, or both. Once the algorithm has been parallelized, we implement it using low-level *platform-specific SIMD instructions*, relying on custom SIMD routines for commonly-used cryptographic operations like integer rotations and matrix transpositions. Finally, we can *hand-optimize the assembly code* for a target platform by reusing registers to avoid spills, rearranging instructions to exploit pipelining, etc.

Unfortunately, the optimized assembly code at the end of this process no longer reflects the high-level structure of the parallelized algorithm, making it unreadable and hard to audit. Furthermore, code reuse is minimal: we need to rewrite code for each platform from scratch, even if we are implementing the same algorithmic ideas, increasing the chances of unintended bugs. Hence, we end up with 10K lines of assembly code for ChaCha20 that only a few developers can audit and safely modify. Efficient implementations of algorithms like Poly1305 are even more complex and error-prone [29], since they have to interleave bignum arithmetic with SIMD vectorization, and hence have to account for the subtleties of both.

How can we be sure that all these platform-specific implementations are correct? Testing helps, but does not give complete coverage, and it requires significant resources to maintain test environments for multiple platforms. The challenge is to build high-assurance cryptographic libraries that are mechanically verified to be correct, memory safe, and secret independent ("constant-time").

Cryptographic Software Verification. Several recent works have explored different approaches towards building verified cryptographic software (see [21] for a more complete survey.)

Vale [36, 59], Jasmin [14, 15], and CryptoLine [60] can directly verify hand-optimized assembly implementations. Consequently, they do not need to sacrifice any performance or trust any compiler. Conversely, the assembly code they verify is neither portable nor reusable. One must rewrite and reverify new code for each platform, and the tool may not support all platforms (Vale and Jasmin do not currently support ARM or AVX512.) More generally, verifying low-level assembly involves quite a bit of work and becomes challenging for large cryptographic constructions and libraries. Consequently these tools are best suited to verify a few optimized implementations of important cryptographic primitives on chosen target platforms.

A different approach is to verify portable code written in a high-level language and rely on a compiler to convert it to assembly. The Verified Software Toolchain [19, 30] and the Cryptol/SAW framework [120] can prove the functional correctness of C (and Java) code against a high-level mathematical specification. HACL* [127] and Fiat-Crypto [56] verify cryptographic code written in verification-oriented high-level languages and compile it to portable C code. The advantage of this approach is that the target C code is readable, auditable, and portable. In addition, these frameworks can use high-level programming mechanisms to share code and proofs between different algorithms. The disadvantage is that the compiled code can be significantly slower than hand-written assembly [127]. Moreover, we either need to trust the C compiler or use a verified compiler like CompCert [79], which produces even slower code.

These approaches are not mutually exclusive. EverCrypt (Chapter 3) is a cryptographic provider that composes verified C code from HACL* with verified Intel assembly code from Vale to obtain best-in-class performance on Intel platforms for elliptic curves like Curve25519 and authenticated encryption schemes like AES-GCM.

Our Approach. In this chapter, we present a new hybrid approach towards building a multi-platform library of vectorized cryptographic algorithms, by following the high-level programming methodology of HACL* but compiling it to multiple platform-specific C implementations that rely on compiler intrinsics for SIMD vector instructions. Hence, we seek to preserve the portability, auditability, code and proof reuse enabled by high-level source code, while closing the performance gap with assembly implementations.

The main insight guiding our approach is that the high-level algorithmic transformations needed for SIMD vectorization can be implemented and verified *generically*, without relying on details of the underlying platform, and then automatically *specialized* for a given target platform. Our second observation is that modern C compilers are good enough (and constantly improving) at instruction scheduling and register allocation, so hand-optimizing assembly for each platform is often not necessary for performance.

HACL×N Workflow. Figure 4.1 depicts our high-level methodology as a sequence of programming, verification, and compilation tasks:

High-Level Specification We first write a succinct formal specification for each cryptographic algorithm in the F* language [118], by carefully transcribing the corresponding IETF or NIST standard. This specification is trusted but executable; it serves as a testable, readable, reference implementation that can be audited by cryptographers.

Generic Vector Library We extend HACL* with libraries for machine integers and vectors of integers, designed to enable generic SIMD programming. We implement the vector library as a trusted C header file that maps each vector operation to platform-specific vector instructions for ARM Neon, Intel AVX, AVX2, and AVX512.

SIMD Patterns for Crypto We identify, implement, and verify a series of reusable SIMD programming patterns commonly used in cryptographic algorithms, including generic constructions for multi-buffer parallelism, CTR encryption, and polynomial evaluation.

Verified Vectorized Implementations We build vectorized implementations of Blake2s, Blake2b, SHA-224, SHA-256, SHA-384, SHA-512, ChaCha20, and Poly1305, in Low* [108] (a subset of F*). These generic implementations are parameterized over a target vector size and can be instantiated with vectors of any size: $1, 2, 4, 8$, etc. (Vectors of size 1 correspond to scalar code.)

Target-Specific Compilation We exploit the meta-programming and compile-time specialization features of F* to translate our generic Low* implementations to custom C implementations for each target platform. The compiler generates both portable 32-bit C code and vectorized C

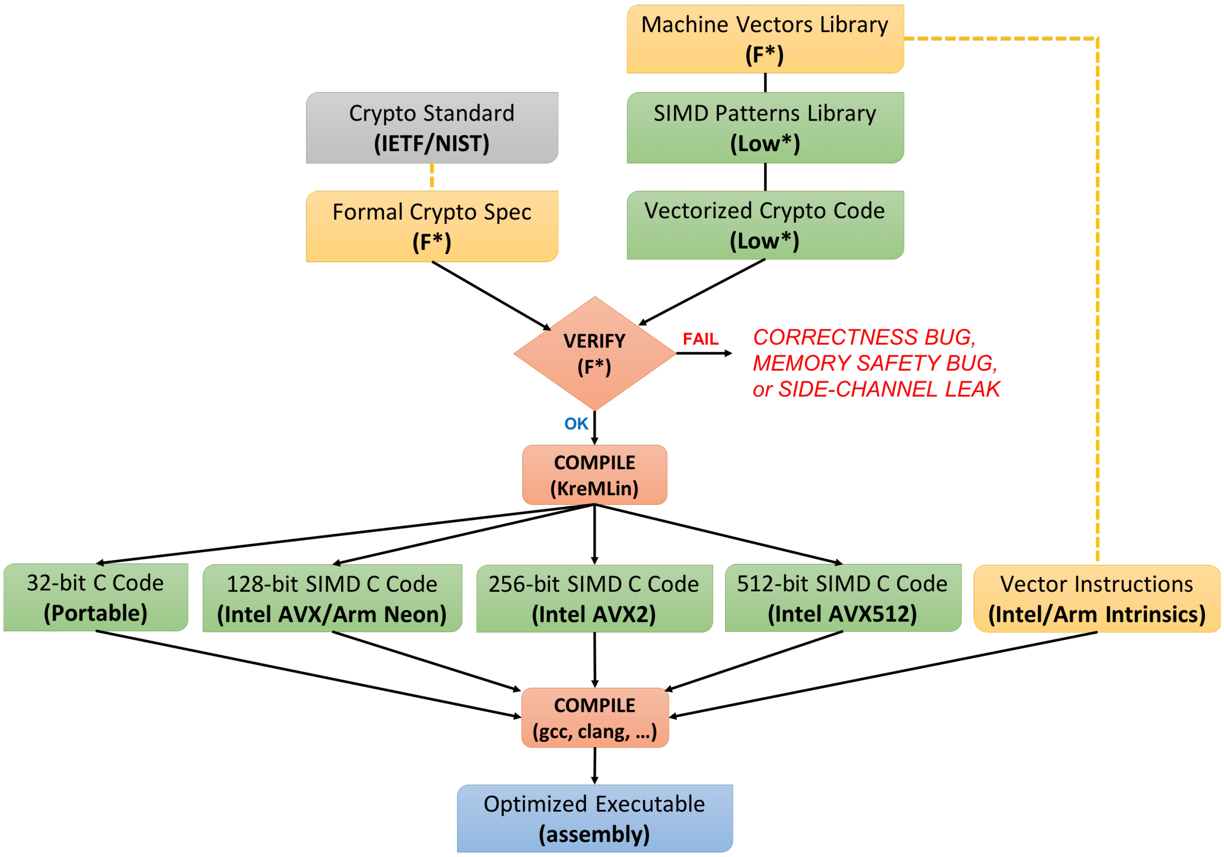


Figure 4.1 – HACL×N programming and verification workflow. We write SIMD cryptographic code in Low* [108] and prove it memory-safe, secret independent, and functionally correct with respect to a high-level formal specification in F* [118], before compiling it to target-specific C code linked with compiler intrinsics. (Code components in green are verified; those in yellow are trusted and carefully tested.)

code for ARM Neon and Intel AVX/AVX2/AVX512. Each C implementation can be compiled via GCC or CLANG to machine code.

Contributions. We apply this workflow on four families of cryptographic algorithms, but our methodology is more generally applicable to other algorithms, and even to non-cryptographic code. To the best of our knowledge, ours are the first verified vectorized cryptographic implementations on ARM Neon and AVX512, and the first verified implementations of vectorized Blake2 and SHA-2. The proof overhead of our method is significantly less than that of HACL*. Our vectorized ChaCha20-Poly1305 code is deployed in Mozilla Firefox, and our Blake2 code is used in Tezos.

Our goal is to build a usable *library* of verified cryptographic algorithms, not just verify a few isolated algorithms. We show how we can embed our vectorized algorithms into HACL* and safely compose them with previously verified C and assembly code [127, 36, 107]. The resulting library provides optimized verified code for many of the ciphersuites used in modern protocols like TLS 1.3, WireGuard, and Signal. We further show how to use our methodology to build optimized implementations of agile cryptographic applications, such as the upcoming HPKE standard [23].

Trusting the C Compiler. The chief limitation of our approach, compared to works that directly verify assembly, is the considerable trust we place in the C compiler. Mainstream C compilers frequently have bugs [124], and even if the code they generate is functionally correct, they may introduce side-channels that were not present in the original source [116], which can be dangerous for cryptographic code.

A natural alternative is to rely on a verified compiler like CompCert [79] which has been extended with side-channel preservation guarantees [26]. However, support for SIMD instructions in CompCert is still ongoing [16], and so we cannot use CompCert for this work. Another direction could be to develop a specialized compiler that can directly compile our cryptographic implementations to assembly code, say by building on a recent compiler from Low* to web assembly [105], and extending it with SIMD instructions [10].

In this work, we make the pragmatic design decision to trust mainstream C compilers like GCC and CLANG, while waiting for these future improvements in verified compilation. In exchange, we obtain portability, performance, and the ability to scale up verification to an entire library of vectorized algorithms. In practice, the libraries and applications that use our cryptographic code (e.g., Firefox, Tezos) already place a large amount of trust in the C compiler. From their viewpoint, compiler bugs are an inevitable cost of large-scale software development; they are much more concerned by security and functionality bugs in the source C code.

4.2 Write & Verify once; compile N times

HACL×N is a new SIMD-oriented extension of HACL*, where the motto is to *verify once*, but *compile and specialize* many times, in order to maximize programmer productivity. For example, when implementing SHA-2, we write a single *generic* implementation for all four variants (i.e., SHA-224, SHA-256, SHA-384, SHA-512) and specialize it to obtain both scalar and vectorized implementations for each variant on each platform, yielding sixteen verified C implementations in total. This aggressive approach towards code and proof reuse relies on an intentional, careful scaffolding of verified libraries and compilation techniques. This section describes how this methodology is implemented in HACL×N.

4.2.1 Abstract integer vectors for SIMD code

SIMD programming in C usually requires dealing with a patchwork of headers and compiler builtins, depending on the target instruction set (e.g., ARM Neon, Intel AVX2).

In order to establish clear interfaces and abstraction boundaries, we introduce a low-level machine vector library dubbed Lib.IntVector.Intrinsics that hides platform differences behind a shared interface. This module selects and axiomatizes vector operations that are general enough to be implemented using, say, both Neon and AVX. For example, it defines arithmetic and bitwise operations for 128-bit, 256-bit, and 512-bit vectors. We carefully audit its semantics, and perform rigorous testing to ensure that our specifications carefully capture the intrinsics' expected behaviour.

At compile-time, calls to this library are diverted to a hand-written C implementation that calls the corresponding SIMD intrinsics provided by the underlying platform. For example, the F* module defines vec128_xor; and its C implementation either calls _mm_xor_si128 (on Intel AVX) or veorq_u32 (on ARM Neon).

As the next step, in order to enable programmers to write generic vectorized code that works for any platform, we define a more abstract vector library of overloaded operators over all vector widths:

```
val vec_t: t:inttype → w:width → Type
val vec_v: #t:inttype → #w:width → vec t w → lseq (uint_t t SEC) w

val (+|): #t:inttype → #w:width → v1:vec_t t w → v2:vec_t t w → vec_t t w

val vec_add_mod_lemma: #t:inttype → #w:width → v1:vec_t t w → v2:vec_t t w →
  Lemma (vec_v (v1 +| v2) == map2 (+.) (vec_v v1) (vec_v v2))
```

The type vec_t is parametric over its width w, and the type t of its elements. The module only offers constructors for valid combinations of w and t, and defines abbreviations for commonly-used types like uint32x4 and uint32x8.

Similarly to the integer operator +., we define the overloaded operator +| which is the point-wise lifting of modular integer addition to vectors of any width. Just like with integers, both the type (vec_t) and the operations (+|) are reduced away by F* at compile-time, and replaced with calls to the low-level SIMD intrinsics. In order to reason about the semantics of vector operations, we use the vec_v function, which reflects a vector as a sequence of integers. For example, we specify vector addition +| in terms of point-wise addition (map2 (+.)) on the sequences returned by vec_v.

Towards Generic Code and Proofs. The design of our generic libraries is a key technical device that allows us to attain greater productivity when authoring verified code in HACL×N. By bringing together all variants of integers, arrays, and vectors into a few, well-documented, extensible libraries, we provide both newcomers and experts with intuitive yet powerful APIs that they can use to implement new cryptographic algorithms. Moreover, these APIs encourage programmers to write generic code that is succinct, readable, and easy to maintain.

For example, both Blake2 and SHA-2 have multiple variants, differing mainly in their internal integer representation (uint32 vs. uint64). Most cryptographic libraries contain independent code for each variant, with only the more popular variants (e.g., SHA-256) being optimized for SIMD platforms. Using our libraries, however, we write and verify generic implementations for Blake2 and SHA-2 which we instantiate to obtain optimized SIMD code for each variant.

We believe that our use of generic strong-typed integers and vectors is generally applicable as a software engineering pattern for cryptographic code, even in other languages like Rust and verification frameworks like Jasmin. Within Low*, we anticipate building many more such generic libraries in the future.

4.2.2 Representation-agnostic cryptographic code

We now illustrate how we can use our abstract integer and vector libraries to write generic implementations for algorithms, even when their internal data representations on different platforms are different. (We leave a detailed discussion of SIMD algorithmic techniques to §4.3 and discuss here only the compilation aspects.)

For example, our vectorized implementation of Poly1305 (§4.3.4) supports multiple vector architectures, identified by a type varch:

```
type varch = | M32 | M128 | M256 | M512

let poly1305_ctx (s: varch) = match s with
  | M32 → lbuffer MUT (vec_t U64 1) 25ul
  | M128 → lbuffer MUT (vec_t U64 2) 25ul | ...
```

All the types and code in the Poly1305 implementation are parameterized by a target varch value s. However, only a few of these definitions need to inspect s. The Poly1305 internal state representation (poly1305_ctx s) is defined as a mutable buffer holding five field elements where the width of these vectors depends on s. On scalar 32-bit machines (s=M32), each vector has one element (i.e., is a uint64), on ARM Neon and Intel AVX (s=M128), each vector has two elements, etc. Similarly, the number of blocks processed in parallel is different on each vector architecture:

```
inline_for_extraction
let blocklen (s:varch): int_t U32 PUB = match s with
  | M32 → 16ul
  | M128 → 32ul | ...
```

Scalar code processes 1 block (16 bytes) at a time, 128-bit vectorized code processes 2 blocks (32 bytes) at a time, etc.

Once we have set up these basic definitions, however, the vast bulk of the Poly1305 implementation is generic and works uniformly on all four architectures. It never needs to do a case analysis on s, except if we wanted to implement some platform-specific optimization. This code will not need to be updated even if we extend varch to support another vector size. For example, the poly1305_update function loops over the input data in blocklen-sized chunks, then processes each chunk to update the state:

```
inline_for_extraction
val poly1305_update: #s:varch →
  ctx:poly1305_ctx s → len:size_t → text:lbuffer uint8 len → Stack unit (..) (..)
```

Although the length of each chunk and the internal state representation both depend on s, the code and proof for poly1305_update is fully generic; it never relies on the actual value of s.

To compile poly1305_update to C, we must first instantiate its s parameter for a target architecture:

```
let update32 = poly1305_update #M32
```

At compile-time, F* processes the inline_for_extraction annotation on poly1305_update and replaces the call site with the function definition. It then repeatedly simplifies the code by partially applying functions whose s parameter is known, inlining types and functions, propagating constants, evaluating case analyses, and discarding unreachable branches, until all mentions of s have been eliminated. The resulting C code corresponds to a scalar implementation of Poly1305 that operates over arrays of uint64 values:

```
void Poly1305_32_update32(uint64_t *ctx, uint32_t len, uint8_t *text);
```

There is no verification cost associated to performing a partial application of poly1305_update to a concrete argument: the three other cases, for 128, 256 and 512-bit specialized variants of Poly1305, also come for free, needing no new code or proofs.

4.2.3 Large-scale program specialization

The technique described above suffers from one caveat: the entire algorithm must be inlined into the top-level function in order to get fully applied matches to appear and be reduced away by F*. While this is fine for a mid-size algorithm such as Poly1305, for a larger piece of code such as HPKE (§4.4.2), this would generate prohibitively large and unreadable C code.

We now address very-large scale genericity, and use the full power of Meta-F* to solve this problem. We have written a tactic (i.e., a meta-program) that takes an algorithm written in a normal style and transforms its entire call-graph, rewriting the algorithm in a form similar to C++ templatized code. After rewriting, the programmer can generate specialized versions of their code like we did above for Poly1305, with the added benefit that the structure of the call-graph is preserved, rather than inlined away.

The tactic takes upon the burden of rewriting the code in a slightly more convoluted form (described below), meaning there is no extra cost for its users. It is flexible, and allows the programmer to annotate their code to specify which functions should be inlined away and which ones should remain at extraction-time. At the time of writing, our tactic is the second largest Meta-F* program written (> 600 LoC), and is used in almost every algorithm in HACL×N.

Overview. We now illustrate the inner workings of our tactic on HPKE, a composite cryptographic construction described in §4.4.2. HPKE calls into a Diffie-Hellman (DH), an AEAD and a hash algorithm. HPKE is *agile* over the choice of these algorithms, and so our generic HPKE code is parameterized by a triplet of algorithms:

```
type hpke_index = dh_alg & aead_alg & hash_alg
```

To instantiate HPKE, the programmer first chooses a triplet of algorithms, and then chooses an implementation of each algorithm. For example, fixing aead_alg to be Chacha20Poly1305 still allows four possible ChaCha20-Poly1305 implementations, one for each degree of vectorization. By mixing and matching algorithms and implementations, we can build 54 combinations of HPKE. Our tactic allows us to verify HPKE once, for each possible triplet of *algorithms*, and enjoy specialization for free for any combination of implementations.

$$\mathsf{hpke} \overset{\mathsf{calls}}{\to} \mathsf{hpke}_{\mathsf{helper}} \overset{\mathsf{calls}}{\to} \mathsf{AEAD.encrypt}$$

The (simplified) call-graph of HPKE is described above, where hpke_helper was split out for proof modularity, but should not appear in the generated C code, as it would be too verbose.

Using F*'s custom annotations, the programmer decorates both hpke and AEAD.encrypt with [@@Specialize], and hpke_helper with [@@Inline]. Doing so, they indicate to the tactic that hpke and AEAD.encrypt should both remain in the call-graph, while hpke_helper is to be inlined in its callers' bodies.

Upon executing, the tactic traverses the call-graph, starting from hpke, and proceeds as follows. First, inlined functions are eliminated, leaving a call-graph only made up of specialized nodes. Then, hpke is rewritten to take as extra parameters function pointers for every specialized function that it calls into. We call this the convoluted form: the user could have written it manually, but the syntactic overhead would have been substantial. After rewriting, the signature of hpke becomes as follows:

```
let snd3 (_, x, _) = x
val hpke #i:hpke_index → encrypt:AEAD.encrypt_t (snd3 i) → . . .
```

Here, AEAD.encrypt_t a stands for the type of an AEAD encryption function for algorithm a, and . . . stands for the original arguments of the hpke function before the rewriting.

In addition to being applied to an index specifying the algorithms it depends on, hpke needs to also be applied to specialized *implementations* of these algorithms:

```
inline_for_extraction let aead_alg = Chacha20Poly1305
let encrypt_cp32: encrypt_t aead_alg = Chacha20Poly1305.encrypt #M32
let hpke_cp32 = HPKE.hpke (. . ., aead_alg, . . .) encrypt_cp32
```

The encrypt_cp32 function above is a specialized implementation of ChaCha20-Poly1305 admissible for any index (. . ., Chacha20Poly1305, . . .). The application of the tactic-rewritten hpke to a concrete index and encrypt_cp32 generates an implementation that calls the ChaCha20-Poly1305 algorithm, specifically its scalar implementation.

The shape of the call-graph is preserved, as hpke_cp32 calls into encrypt_cp32; furthermore, this technique allows swapping out encrypt_cp32 for any other variant, giving, e.g., hpke_cp256.

Using our tactic, the proof of HPKE is exclusively concerned with *algorithmic agility*, leaving implementation choices entirely up to the module that performs concrete instantiations. This enforces strong modularity, as HPKE need not be aware of the current or future implementations for a given algorithm.

The methodology applies, naturally, to all possible combinations of choices for DH, AEAD and hash, meaning we can obtain up to 54 specialized implementations of HPKE for free; 15 of these implementations are currently packaged within HACL×N.

Verification and Debugging. Note that the tactic is not part of the TCB, since whatever code the tactic generates is type-checked again by F*. This is by design: unlike, say, MTac2 [71], Meta-F* [83] does not allow the user to prove properties about tactics. This is a pragmatic design choice, trading provable correctness for ease-of-use and programmer productivity.

Tactics are just one tool in the utility belt of the F* programmer. We experimented with other strategies, such as type classes, but found that they imposed a lot of overhead on the user: if the call-graph has depth n, then the user needs to materialize $n-1$ type classes for each level of specialized functions. We thus found our "templatization" tactic to be simpler to use, and therefore better for proof productivity.

Debugging tactics and tactic-generate code is straightforward. Either the tactic itself fails, and F* points to the faulty line in the meta-program; or the generated code is ill-typed, in which

```
let g (alg:blake2_alg) (st:state alg) (a b c d:idx) (x y:word alg) : state alg =
  let st = st.[a] ← (st.[a] +. st.[b] +. x) in
  let st = st.[d] ← (st.[d] ^. st.[a]) >>>. (rotc alg 0) in
  let st = st.[c] ← (st.[c] +. st.[d]) in
  let st = st.[b] ← (st.[b] ^. st.[c]) >>>. (rotc alg 1) in
  let st = st.[a] ← (st.[a] +. st.[b] +. y) in
  let st = st.[d] ← (st.[d] ^. st.[a]) >>>. (rotc alg 2) in
  let st = st.[c] ← (st.[c] +. st.[d]) in
  let st = st.[b] ← (st.[b] ^. st.[c]) >>>. (rotc alg 3) in
  st
let mixing_core (alg:blake2_alg) (st:state alg) (m:state alg) : state alg =
  let st = g alg st 0 4 8 12 m.[0] m.[1] in
  let st = g alg st 1 5 9 13 m.[2] m.[2] in
  let st = g alg st 2 6 10 14 m.[4] m.[5] in
  let st = g alg st 3 7 11 15 m.[6] m.[7] in
  let st = g alg st 0 5 10 15 m.[8] m.[9] in
  let st = g alg st 1 6 11 12 m.[10] m.[11] in
  let st = g alg st 2 7 8 13 m.[12] m.[13] in
  let st = g alg st 3 4 9 14 m.[14] m.[15] in
  st
```

Figure 4.2 – F* specification for the core Blake2 computation.

case we can examine it like any other F* program. In practice, after some initial debugging, our tactic never generated ill-typed code (in over 20 use cases) and was used successfully by other collaborators.

4.3 SIMD Cryptographic Programming Patterns

We now identify a series of SIMD parallelization strategies and apply them to build and verify generic vectorized implementations for four families of cryptographic algorithms. The patterns we detail here are not meant to be exhaustive; for example, we do not cover bit- and byte-slicing. However, these patterns cover most of the standard vectorization strategies that we have seen used in popular cryptographic libraries. Although we apply each pattern only to a single algorithm family, we provide verified generic libraries that can be used to apply these patterns to other similar algorithms.

4.3.1 Exploiting Internal Parallelism (Blake2)

We first consider algorithms that are explicitly designed to allow their core operations to be parallelized. We illustrate this pattern for the Blake2 hash algorithm, but similar strategies apply to other cryptographic algorithms like ChaCha20 and Salsa20.

Formally Specifying Blake2. The Blake2 cryptographic hash algorithm [20] is standardized in IETF RFC 7693 [111]. We formalized this RFC in F* and the main types in the resulting specification are:

Blake2 has two variants, Blake2s and Blake2b; the first uses 32-bit words, whereas the latter uses 64-bit words. We specify the word type as an algorithm-dependent machine integer that

```
type blake2_alg = | Blake2s | Blake2b

let word_t (alg:blake2_alg) = match alg with
  | Blake2s → U32
  | Blake2b → U64

let word (alg:blake2_alg) = int_t (word_t alg) SEC
type state (alg:blake2_alg) = lseq (word alg) 16
```

is labeled as secret (SEC), which enforces that all operations on these words must be secret independent ("constant-time"). The Blake2 state, also called a working vector, is a 4 × 4 matrix of words, represented in the RFC as a sequence (lseq) of 16 words, laid out row-by-row.

To hash an input message, Blake2 first splits it into state-sized blocks (64 bytes for Blake2s, 128 bytes for Blake2b), and processes each block in sequence by calling a compression function. The core computation of the compression function is a loop that repeatedly loads a message block, permutes it according to a table, and then calls the mixing_core function depicted in Figure 4.2.

The mixing_core function in turn calls a shuffling function g 8 times. Each call takes 2 words from the message (x,y), and reads, shuffles, and writes four words in the Blake2 state (at indexes a,b,c,d), using a combination of modular addition (+.), xor (^.), and right-rotate (>>>.). We use overloaded operators that work for both uint32 and uint64, and this allows us to write a single generic, yet strongly-typed, formal specification for both Blake2s and Blake2b.

The resulting F* specification is executable and can be seen as a reference implementation of the RFC. We tested it against test vectors from the RFC and more comprehensive tests we added ourselves. Interestingly, our tests revealed a bug in a corner case of our specification when processing the last block. This bug was not exercised by the RFC test vectors, and this serves to reemphasize the need for mechanized specifications and formal verification.

Rearranging Code for 4-way Vectorization. Each of the first four calls to g in the mixing_core function read and modify a different column of the state matrix $((0,4,8,12),(1,5,9,13),\ldots)$. Hence, these calls can be executed in parallel [20]. The next four calls process different diagonals of the state and can also be executed in parallel. To exploit this 4-way parallelism inherent in Blake2, we rearrange the state to use vectors:

```
type vec_row (alg:blake2_alg) = vec_t (word_t alg) 4
type vec_state (alg:blake2_alg) = lseq (vec_row alg) 4

val to_vec (alg:blake2_alg) (st:state alg) : vec_state alg
val from_vec (alg:blake2_alg) (st:vec_state alg) : state alg
```

The state is now explicitly a matrix with four rows, and each row is a vector with four words. Based on this vectorized state, we can define a *vectorized specification* for Blake2 in F*. We will relate the two specifications using functions (to_vec, from_vec) that inter-convert between the original scalar state and its vectorized form.

The core Blake2 computations are rewritten as shown in Figure 4.3. The function g_vec applies the function g to each column in parallel. Using our vector library, the code for this function is remarkably similar to that of g; we simply replace the integer operations (+.,^.,>>>.)

```
let g_vec (alg:blake2_alg) (st:vec_state alg) (x y:vec_row alg) =
  let (a,b,c,d) = (0,1,2,3) in
  let st = st.[a] ← (st.[a] +| st.[b] +| x) in
  let st = st.[d] ← (st.[d] ^| st.[a]) >>>| (rotc alg 0) in
  let st = st.[c] ← (st.[c] +| st.[d]) in
  let st = st.[b] ← (st.[b] ^| st.[c]) >>>| (rotc alg 1) in
  let st = st.[a] ← (st.[a] +| st.[b] +| y) in
  let st = st.[d] ← (st.[d] ^| st.[a]) >>>| (rotc alg 2) in
  let st = st.[c] ← (st.[c] +| st.[d]) in
  let st = st.[b] ← (st.[b] ^| st.[c]) >>>| (rotc alg 3) in
  st
let diagonalize (alg:blake2_alg) (st:vec_state alg) : vec_state alg =
  let st = st.[1] ← vec_rotate_right_lanes st.[1] 1ul in
  let st = st.[2] ← vec_rotate_right_lanes st.[2] 2ul in
  let st = st.[3] ← vec_rotate_right_lanes st.[3] 3ul in
  st
let mixing_core_vec (alg:blake2_alg) (st:vec_state alg) (m:vec_state alg) : vec_state alg =
  let st = g_vec alg st m.[0] m.[1] in
  let st = diagonalize alg st in
  let st = g_vec alg st m.[2] m.[3] in
  let st = undiagonalize alg st in
  st
```

Figure 4.3 – 4-way vectorized specification for Blake2.

with their vector counterparts (+|,^|,>>>|) and we set the indexes a, b, c, d to column numbers $0, 1, 2, 3$.

The benefit of vectorization becomes clear in the mixing_core_vec function; it now calls g_vec only twice, since each call processes four columns at a time. If each vector operation has the same cost as a scalar operation, this transformation can provide up to a 4x performance improvement. However, one must account for the cost of loading, storing, and transforming vectors. For example, before the second call to g_vec we need to *diagonalize* the state, by rotating three of the row vectors, and undiagonalize it after.

The vectorized F* specification acts as an intermediate step between the original F* specification and the vectorized Low* implementation. We prove that the two specifications are equivalent via a series of lemmas. For example, we prove that mixing_core_vec computes the same function as mixing_core, but on the vectorized state:

```
∀(alg:blake2_alg) (st:state alg) (m:state alg).
  mixing_core alg st m == from_vec (mixing_core_vec alg (to_vec alg st) (to_vec alg m))
```

Implementing and Verifying Vectorized Blake2. Our Low* implementation of Blake2 closely follows the vectorized specification, but generalizes it further. On machines that support sufficiently wide vector instructions (128-bit for Blake2s, 256-bit for Blake2b), the implementation uses 4-way vectorization. On all other platforms, it defaults to scalar 32-bit code. By carefully structuring our code, we are able to define a single generic implementation for all four variants: scalar and vector, Blake2b and Blake2s.

The only other difference between the vectorized specification and our Low* code is that the code modifies the state in-place, instead of copying the state at each modification. We prove that this code is memory-safe (it does not read or write arrays outside their bounds) and we prove that it is functionally correct with respect to the vectorized F* specification, and by composing this with specification equivalence, we prove that it conforms to the original Blake2 specification.

Compiling to C with Vector Intrinsics. We compile the Low* code using KreMLin to obtain 4 C files: Blake2s_32.c, Blake2b_32.c, Blake2s_128.c, and Blake2b_256.c, each offering the same interface. The first two contain portable code that runs on any platform.

The C code in Blake2s_128.c is essentially a sequence of calls to 128-bit vector operations. This code can be linked with our library of vector intrinsics, and executed on any machine that supports Arm Neon or Intel AVX/AVX2/AVX512. For example, on Intel AVX, the C code for the first two shuffling operations of g_vec looks like:

```
st[0U] = _mm_add_epi32(st[0U], st[1U]);
st[0U] = _mm_add_epi32(st[0U], x);
st[3U] = _mm_xor_si128(st[3U], st[0U]);
st[3U] = _mm_xor_si128(_mm_slli_epi32(st[3U],32U−rotc0), _mm_srli_epi32(st[3U],rotc0))
```

Similarly, Blake2b_256 relies on AVX2 intrinsics and can be executed on any Intel AVX2/AVX512 machine. Performance numbers for all these implementations are given in §4.5. On Intel AVX512 processors, vectorization speeds up Blake2 by about 30%.

Further Platform-Specific Optimizations. We have focused on writing *generic* code to avoid duplication of coding and verification effort. However, one can sometimes get even better performance by writing platform-specific code for some operations.

Blake2 requires each input message block to be permuted multiple times according to a known permutation schedule. In our generic code, we implement these permutations naively, using vector loads. However, AVX512 offers more powerful *gather* instructions, so we implemented a special case of the message loading functions for AVX512. On our test machines, these instructions did not provide any performance benefit, but it is expected that these instructions will get faster in future processors. Another optimization, used by other Blake2 implementations, is to write custom AVX2 permutation code. This code is tedious and non-generic (about 300 lines of C), but can result in a significant speed-up. We did not implement this optimization in our code, leaving it for future work.

4.3.2 Multiple Input Parallelism (SHA-2)

The next pattern generally applies to any cryptographic algorithm when it is applied to a number of independent inputs (of the same size). We extensively use this pattern throughout our library. Here, we illustrate its use in our implementation of multi-buffer SHA-2.

Specifying the SHA-2 Family. The SHA-2 family of hash functions [7] is perhaps the most widely used cryptographic construction today. It is used as a core component within method authentication codes (HMAC), key derivation (PBKDF2, HKDF), signature schemes (Ed25519, ECDSA), and Merkle trees.

SHA-2 has four variants: SHA−224, SHA−256, SHA−384, and SHA−512. The first two use 32-bit words, whereas the last two use 64-bit words. Like in Blake2, we define a generic F*

```
let sha2 (a:sha2_alg) (in_len:size_nat) (input:lseq uint8 in_len) : lbytes (hash_len a) =
  let st0 = init a in
  let blocks = in_len / blocksize a in
  let st =
    repeati blocks (λ i st →
      let b = sub input (i ∗ blocksize a) (blocksize a) in
      compress_block a b st) st0 in
  let last_len = in_len % blocksize a in
  let last = sub input (in_len − last_len) last_len in
  let st = pad_compress_last a in_len last_len last st in
  emit a st
```

Figure 4.4 – F* specification for generic SHA-2 hash function.

specification for all four variants using our integer library. The SHA-2 state consists of 8 words and each block consists of 16 words (i.e., 64 or 128 bytes).

Our F* specification for the main sha2 hash function is depicted in Figure 4.4. It calls init to initialize the state (with some known constants); it then goes into a loop (repeati) that calls compress to mix each block of the input into the state; finally, it processes the last (partial) block by calling pad_compress_last and emits the output hash.

Multi-Buffer SHA-2. The sha2 function is not obviously parallelizable, since the output of each block is fed into the input of the next. But if we were willing to hash 4, 8, or 16 independent equal-sized inputs in parallel, performance could improve significantly. This strategy is called multi-buffer SHA-2 [64] and has been applied to other serial primitives like AES-CBC.

We write a generic vectorized specification for multi-buffer SHA-2, defining the vectorized state as an array of w-word vectors:

```
type vec_state (w:width) (alg:sha2_alg) = lseq (vec_t (word_t alg) w) 8
type multi_block (w:width) (alg:sha2_alg) = lseq (vec_t (word_t alg) w) 16
```

Seen as a $w \times 8$ matrix, each column of this state corresponds to one input message, and hence the state represents the intermediate SHA-2 state for w inputs. We process all w inputs block-by-block by calling a vectorized version of the compress function, which takes the vectorized state and a multi_block as input. Each multi_block corresponds to the ith blocks of each of the w inputs; hence it is an array of 16 vectors and can be seen as a $w \times 16$ matrix.

Writing and verifying the vectorized compress function follows a standard pattern. Like in the Blake2 g_vec function, we replace each integer operation with the corresponding vector operation. We then prove that this transformation results in a *mapped* version of compress: it independently compresses each column in parallel.

The main remaining task for multi-buffer SHA-2 is functions for loading the message blocks and then emitting the result. Both of these operations require matrix transpositions.

A Library for Transposing Vectors. When we load an input block using vector instructions, we naturally get these blocks loaded in row-wise form. When implementing multi-buffer SHA-256 with 128-bit vectors, for example, we process 4 inputs in parallel. We can efficiently load the 64-byte block from each input into 4 128-bit vectors, hence obtaining 16 vectors where vectors 0..3 contain data from input 0, 4..7 contain data from input 1 etc. To put this into the column-wise

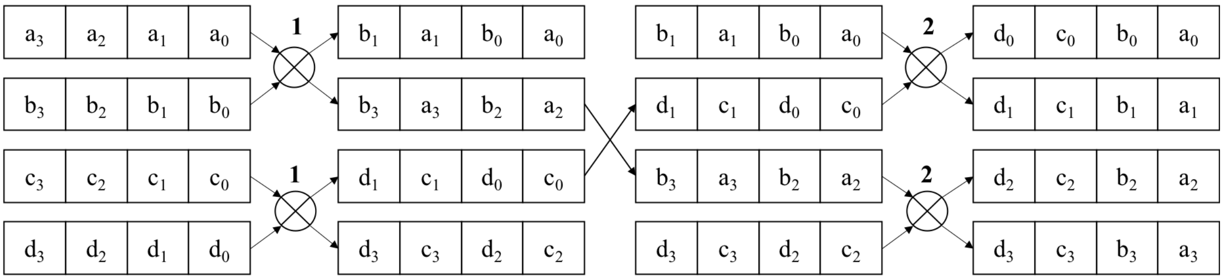


Figure 4.5 – Transposing a 4 × 4 vectorized state. Each pair of vectors is interleaved element by element, then each alternate pair is interleaved 2 at a time. Transposing a $n \times n$ vectorized matrix needs $n \log_2(n)$ interleavings.

multi_block format needed by vectorized compress, we need to transpose vectors $(0, 4, 8, 12)$ to obtain the first 4 vectors, then transpose $(1, 5, 9, 13)$ to obtain the next 4 vectors, and so on.

These kinds of transpositions are routinely needed in vectorized cryptographic code (see ChaCha20 below) and so we implemented and verified a generic library of vectorized transpositions called Lib.IntVector.Transpose. For each transposition, we prove that the result, seen as a matrix, is the transposition of the input.

A typical function provided by this library implements the 4 × 4 transposition depicted in Figure 4.5. It takes an array of 4 vectors each with 4 words as input. It uses a vector interleaving operation to interleave each pair of vectors element-by-element, leaving the low-half of the interleaved result in the first vector, putting the high-half in the second vector. (Both Arm and Intel platforms offer these kinds of interleaving instructions.) We then interleave each pair of alternate vectors 2-by-2 to obtain the final result.

Other functions in this library extend this pattern to 8 × 8 and 16 × 16 transpositions, and also for non-square matrices. The main complexity in writing and verifying these functions is in choosing the right sequence of vector operations (some interleaving instructions can be much more expensive than others).

Implementing and Compiling Multi-Buffer SHA-2. We build a generic implementation of SHA-2 in Low* that can be instantiated for all 4 SHA-2 algorithms and can be used with 4 or 8 inputs at a time. Hence, SHA-256 can be run on 4 inputs at a time on ARM Neon and 8 inputs at a time on Intel AVX2, while SHA-512 can be run on 4 inputs at a time on AVX2, and 8 at a time on AVX512.

The main complexity in writing and verifying this Low* code is that each function needs to input and manipulate a large number of buffers. For example, the Low* type for 4-buffer SHA-256 is depicted in Figure 4.6. It takes four equal-length buffers $(b0, b1, b2, b3)$ as input and four hash-length buffers $(r0, r1, r2, r3)$ as output. We require all 8 buffers to be live in the input heap, and we require the four output buffers to be disjoint. F* can then prove that the code for this function is memory safe, that it only modifies the four output buffers, and that the final value in each output buffer is the expected hash value of the corresponding input. To make these types easier to write and verify, we use a library of multi-buffer predicates like all_live, loc_pairwise_disjoint that are meta-evaluated into conjunctions of base predicates.

The performance results for all variants of multi-buffer SHA-2 are given in §4.5. On Intel

```
val sha256_4
  (r0 r1 r2 r3: lbuffer uint8 32ul)
  (len:size_t) (b0 b1 b2 b3: lbuffer uint8 len)
 : Stack unit
   (requires λ h0 →
     live h0 r0 ∧ live h0 r1 ∧ live h0 r2 ∧ live h0 r3 ∧
     live h0 b0 ∧ live h0 b1 ∧ live h0 b2 ∧ live h0 b3 ∧
     loc_pairwise_disjoint [ loc r0; loc r1; loc r2; loc r3 ])
   (ensures λ h0 _ h1 →
     modifies (loc r0 |+| loc r1 |+| loc r2 |+| loc r3) h0 h1 ∧
     as_seq h1 r0 == sha2 SHA2_256 (v len) (as_seq h0 b0) ∧
     as_seq h1 r1 == sha2 SHA2_256 (v len) (as_seq h0 b1) ∧
     as_seq h1 r2 == sha2 SHA2_256 (v len) (as_seq h0 b2) ∧
     as_seq h1 r3 == sha2 SHA2_256 (v len) (as_seq h0 b3))
```

Figure 4.6 – Low* API for 4-way vectorized multi-buffer SHA-256.

platforms, 4-buffer SHA-2 is about 2.6x faster than scalar SHA-2, and 8-buffer SHA-2 is up to 7x faster.

4.3.3 Counter Mode Encryption (ChaCha20)

We next consider a SIMD pattern that applies to all counter-mode encryption (CTR) algorithms, such as ChaCha20, AES, Salsa20, etc. More generally, we present a loop combinator called map_blocks, which maps a block-to-block function on some input data, and show how to parallelize any program that uses this combinator.

Specifying Generic CTR in F*. CTR is one of several block cipher modes of operation standardized by [55]. It is notably used in the two most popular authenticated encryption schemes: AES-GCM and ChaCha20-Poly1305. We specify CTR as a generic construction over any block cipher that meets the following interface:

```
type block = lbytes blocksize

val init: k:key → n:nonce → ctr0:nat → state
val key_block: st:state → i:nat → block
```

The block cipher must define a constant blocksize, and types for the key, nonce, and cipher state. It must define a function init to initialize the state, given a key, nonce, and initial counter ctr0. Finally, it must provide a function key_block that generates a block of key bytes given a block number i.

Given such a block cipher, we specify the CTR encryption algorithm as depicted in Figure 4.7. The function encrypt_block encrypts the ith message block by XORing it with the ith key block. The function encrypt_last pads the last (partial) block with zeroes and then encrypts it using encrypt_block.

Finally, the main encryption function ctr_encrypt (which is the same as the decryption function) initializes the state and calls the loop combinator map_blocks, which breaks the input msg into blocks, sequentially calls encrypt_block for each block, and calls encrypt_last for the last (partial) block.

```
let encrypt_block (st0:state) (i:nat) (b:block) : block =
  map2 (^.) (key_block st0 i) b

let encrypt_last (st0:state) (i:nat) (len:nat{len < blocksize}) (r:lbytes len) : lbytes len =
  let b = create blocksize (u8 0) in
  let b = update_sub b 0 len r in
  sub (encrypt_block st0 i b) 0 len

let ctr_encrypt (k:key) (n:nonce) (ctr0:nat) (msg:bytes) : cipher:bytes{length cipher == length msg} =
  let st0 = init k n ctr0 in
  map_blocks blocksize msg
    (encrypt_block st0)
    (encrypt_last st0)
```

Figure 4.7 – Generic F* Specification for CTR.

Multi-Input Parallelism for the Block Cipher. The map_blocks combinator exposes the inherent parallelism in CTR: it processes each block independently, and so can process any number of blocks in parallel. To exploit this parallelism, we first have to write a vectorized version of the block cipher:

```
type blocksize_v (w:width) = w * blocksize
type multi_block (w:width) = lbytes (blocksize_v w)

val init_v: w:width → k:key → n:nonce → ctr0:nat → vec_state w
val key_block_v: w:width → vec_state w → i:nat → multi_block w
```

Following the multi-input SIMD pattern, the vectorized block cipher processes w blocks at a time. It has an internal vectorized state vec_state that is initialized by the function init_v. The function key_block_v generates w consecutive key blocks. The main proof obligation is to show that these blocks correspond to the key blocks numbered i*w,i*w+1,...,(i+1)*w−1 in the original specification.

For ChaCha20, writing and verifying the vectorized block cipher code follows the same pattern as SHA-2. The ChaCha20 state is an array of 16 32-bit words, and so the vectorized state is an array of 16 vectors with w words each (each column corresponds to one input block). We replace each integer operation in the block cipher code with its vector equivalent, and we need to transpose the state before generating the output key blocks. By reusing library lemmas about vector operations and transpositions, we prove the correctness of the key_block_v function for ChaCha20 with relatively little effort.

Parallelizing CTR. Vectorizing the block cipher effectively yields a new block cipher with a larger blocksize. Hence, we can run the standard CTR algorithm over this vectorized block cipher, by processing w sequential blocks at a time. This results in a vectorized specification for ChaCha20. Our loop combinator library includes general lemmas about map_blocks, which allow us to prove the generic correctness of vectorized CTR (relying on a correctness lemma for the vectorized block cipher.) We instantiate this generic proof for vectorized ChaCha20, but the pattern can also be easily applied to other counter-mode encryption algorithms like AES-CTR.

Implementing and Compiling Vectorized ChaCha20. We implement Vectorized ChaCha20 in Low* in two steps. We first write a module for multi-block ChaCha20 that can process w blocks at the same time, for w=1,4,8,16. We then write a generic CTR module that uses the map_blocks to process w blocks at the same time.

The implementation introduces a new optimization in the vectorized code for encrypt_block, which loads w blocks of data from an input message, XORs it with w key blocks, and stores these blocks into the output ciphertext. Using vector instructions, we can implement this load-XOR-store loop generically and more efficiently than the byte-by-byte XOR in encrypt_block. In some cases, it is also beneficial to unroll this loop a few (say 4) times to take maximum advantage of instruction pipelining.

Because of our generic code structure, adding new platforms requires modest effort. For example, to add AVX512, the main additional effort was to add the relevant vector intrinsics and to define and verify a 16×16 transpose function, which is now in the library and can be used in other algorithms.

We note that this vectorization pattern is not the only one that applies to ChaCha20. The inner block cipher in ChaCha20 is inherently parallelizable (similarly to Blake2) and this parallelization has been exploited in prior work [33] and even verified [127]. However, in our experiments, we found that vectorizing CTR was generally more effective on our target platforms.

4.3.4 Polynomial Evaluation (Poly1305)

We now describe a SIMD pattern used in cryptographic algorithms like Poly1305 and GCM, which are written in terms of polynomial evaluation over a (large) field arithmetic. We show that these algorithms can be written using a loop combinator called repeat_blocks and detail how this combinator can be parallelized if the body of the loop satisfies some algebraic conditions.

Specifying Poly1305. The Poly1305 one-time MAC function [31] is standardized in IETF RFC7539 [94]. It takes a 32-byte key as input and splits into two 128-bit integers s and r. It then splits the input message into 16-byte blocks, hence transforming it to a sequence of 128-bit integers $(m_1, m_2 \dots m_n)$; if the last block is partial, it is filled out with zeroes to obtain a full block.

The main computation in the Poly1305 MAC evaluates the following polynomial in the prime field $\mathbb{Z}_p$, where $p = 2^{130} - 5$:

$$a = (m_1 \times r^n + m_2 \times r^{n-1} + \dots + m_n \times r) \mod p$$

In practice, this polynomial is evaluated block by block, by applying Horner's method to rearrange the polynomial as follows:

$$a = ((\dots((0 + m_1) \times r + m_2) \times r + \dots + m_n) \times r) \mod p$$

We maintain an accumulator a, initially set to 0, and to process each new block m_i, we first add it to the accumulator, and then multiply the result by r (all operations in $\mathbb{Z}_p$). Once the final block is processed, we compute $(s + a) \mod 2^{128}$ to obtain the MAC.

Figure 4.8 depicts our F* specification for the polynomial evaluation described above. It uses a loop combinator called repeat_blocks that splits the input into block-sized chunks. For each

```
let process_block (r:felem) (b:block) (acc:felem) : felem =
  fmul (fadd acc (encode b)) r

let process_last (r:felem) (len:nat{len < blocksize}) (b:lbytes len) (acc:felem) : felem =
  if len = 0 then acc else process_block r (pad_last len b) acc

let poly_eval (msg:bytes) (acc0 r:felem) : felem =
  repeat_blocks blocksize msg
    (process_block r)
    (process_last r)
  acc0
```

Figure 4.8 – F* specification for Poly1305 polynomial evaluation.

block, it calls process_block, which in turn calls the two field arithmetic operations in $\mathbb{Z}_p$: fadd to add an encoded block to the accumulator acc, and fmul to multiply the result with r. The function process_last pads and processes the last block. Our full F* specification for Poly1305 is not much larger than this; it only adds some concrete details from the RFC about encoding blocks and keys.

Parallelizing Polynomial Evaluation. Several prior works have observed (e.g., [33]) that the algebraic shape of polynomial evaluation lends itself to SIMD vectorization. For example, we can process blocks two-by-two by rewriting the polynomial as follows:

$$\begin{aligned}
a_1 &= (\ldots((m_1 \times r^2 + m_3) \times r^2 + m_5) \times r^2 + \ldots + m_{n-1}) \mod p\\
a_2 &= (\ldots((m_2 \times r^2 + m_4) \times r^2 + m_6) \times r^2 + \ldots + m_n) \mod p\\
a &= (a_1 \times r^2 + a_2 \times r) \mod p
\end{aligned}$$

Let's assume that n is even. We split the polynomial evaluation into two computations, one processes odd-numbered blocks, and the other processes even numbered blocks, but both computations are otherwise identical. We now have two accumulators (a_1, a_2) initialized to (m_1, m_2). We process two blocks (m_{2i-1}, m_{2i}) at a time by multiplying both (a_1, a_2) by r^2 and adding the result point-wise to (m_{2i-1}, m_{2i}). After processing n blocks, a final *normalization step* multiplies a_1 by r^2 and a_2 by r and adds them.

This refactored computation effectively computes two polynomials in parallel and it is easy to informally see why it is correct. We formalize and generalize this pattern as a vectorized specification of Poly1305 in F* that can process any number (e.g., 1/2/4/8) of blocks in parallel. Figure 4.9 depicts the vectorized specification.

The accumulator now has the type felem_v w, which represents a vector of w field elements. The function process_blocks_v evaluates w blocks in parallel by calling vectorized versions (fmul_v, fadd_v) of the field arithmetic functions. If less than w blocks of input are left, we call the process_last_v function that *normalizes* the vectorized accumulator to get a regular field element (felem), then calls the original (scalar) poly_eval function on the remaining input.

To set up the vectorized polynomial evaluation, poly_eval_v first precomputes r^w and stores it in a vector r_w whose elements all hold r^w. It then loads the initial accumulator acc0 into the 0th element of the vectorized accumulator acc0_v (all other elements are set to zero) and calls

```
let process_blocks_v (w:width) (r_w:felem_v w) (b:multi_block w) (acc:felem_v w) : felem_v w =
  fadd_v w (fmul_v w acc r_w) (encode_v w b)

let process_last_v (w:width) (r:felem)
  (len:nat{len < blocksize_v w}) (b:lbytes len) (acc:felem_v w) : felem =
  poly_eval b (normalize_v w r acc) r

let poly_eval_v (w:width) (msg:bytes) (acc0 r:felem) : felem =
  let r_w = pow_v w r in
  let acc0_v = to_acc_v w acc0 in
  repeat_blocks (blocksize_v w) msg
    (process_blocks_v w r_w)
    (process_last_v w r)
  acc0_v
```

Figure 4.9 – Generic vectorized specification for Poly1305.

repeat_blocks to process the input.

We generically prove, for all choices of w, that this vectorized specification is functionally equivalent to the original Poly1305 specification:

```
∀w msg acc0 r. poly_eval_v w msg acc0 r == poly_eval msg acc0 r
```

The proof relies on general lemmas about field arithmetic and the repeat_blocks combinator. We apply this lemma here to Poly1305 but it also applies to other polynomial MACs like GCM.

Implementing Multi-Input Field Arithmetic. The main effort of implementing and verifying (scalar or vectorized) Poly1305 is in the field arithmetic modulo p. Since Poly1305 uses a 130-bit field, a typical way of implementing a field element in Low* is as an array of 5 26-bit *limbs*, where each limb can grow to at most 64-bits. We then need to implement custom modular Bignum arithmetic (fadd, fmul) for this representation and prove it correct. In the original HACL* release, Poly1305 was one of the largest developments with 4716 lines, most of it dedicated to field arithmetic [127].

To implement vectorized Poly1305, we need to implement and verify a multi-input field arithmetic library that can add and multiply (fadd_v, fmul_v) multiple field elements in parallel. We take the original scalar Poly1305 code of HACL* and generalize it using the standard multi-input pattern. Each limb is represented by a 64-bit word, and a vectorized field element is an array of vectors, each of which has w words. All functions are parameterized by the vector width w and integer operations are replaced with vector ones. The correctness proofs are adapted for vectorized inputs and outputs.

While the multi-input algorithmic transformation is itself straightforward, applying it to thousands of lines of scalar Poly1305 was a challenge and constitutes our largest case study for the SIMD coding and verification patterns in this chapter. This is, however, a one-time cost. Once we vectorized all the field arithmetic in Poly1305, adding a new platform (such as AVX512) required only a modest amount of work. Furthermore, the verified vectorized bignum library we built for Poly1305 has many reusable components that can be used in other primitives like Curve25519 in future work.

	Portable	**Arm A64**	**Intel x64**			
Algorithm	**C code**	**Neon**	**AVX**	**AVX2**	**AVX512**	**Vale**
AEAD						
ChaCha20-Poly1305	✓ [127] (+)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
AES-GCM						✓ [36]
Hashes						
SHA-224,256	✓ [127] (+)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ [36]
SHA-384,512	✓ [127] (+)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
Blake2s, Blake2b	✓ [107] (+)	✓ (*)	✓ (*)	✓ (*)		
SHA3-224,256,384,512	✓ [107]					
HMAC and HKDF						
HMAC (SHA-2,Blake2)	✓ [127]	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
HKDF (SHA-2,Blake2)	✓ [127]	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
ECC						
Curve25519	✓ [127]					✓ [107]
Ed25519	✓ [127]					
P-256	✓ [107]					
High-level APIs						
Box	✓ [127]					
HPKE	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ (*)

Table 4.1 – Extending HACL* with vectorized crypto.
Implementations marked with (*) were newly developed for this chapter; those marked with a (+) replaced prior C implementations from [127]. These C implementations are composed with platform-specific Intel assembly code from Vale [36] (verified against the same specifications) to build the EverCrypt provider (Chapter 3). Vale assembly relies on AES-NI for AES-GCM, SHAEXT for SHA-2, and ADX+BMI2 instructions for Curve25519.

4.4 Cryptography for all your needs

While cryptographic algorithms are often designed, standardized, and implemented as independent components, they are typically deployed and used as part of composite constructions. For example, the ChaCha20 cipher is only safe to use in conjunction with a one-time MAC like Poly1305. The SHA-256 hash algorithm is used within HMAC, HKDF, and a number of signature schemes. So, even if the code for an individual algorithm is verified for memory safety, correctness, or side-channel resistance, these guarantees become quickly meaningless if the code is composed with a buggy algorithm, or if the API provided by the algorithm is easy for an application to misuse. Consequently, it is important for verified cryptographic code to be deployed as part of a comprehensive verified *library* of cryptographic constructions with safe usable APIs.

4.4.1 Integration and Deployment with HACL*

We contributed all the verified code developed in this chapter to the HACL* project, and helped to integrate it with the existing constructions and APIs in the HACL* library. Table 4.1 summarizes our contributions. For ChaCha20, Poly1305, Blake2, and SHA-2, our scalar code replaces the previous portable C code [127] and our vectorized implementations are offered as platform-dependent alternatives. For each platform, we also build verified implementations of the ChaCha20-Poly1305 AEAD construction, and we integrated our hash implementations into HMAC, HKDF, Ed25519, and ECDSA.

Crucially, for each algorithm, we ensure that each of our implementations meets the same high-level specifications as the original HACL* code, and retains the same API. Hence, verified applications built on top of HACL*, such as EverQuic [48], do not need to be re-verified.

For existing clients of the HACL* C library, such as the Linux Kernel, Mozilla Firefox, or the Tezos Blockchain, this means that the newer C code is a drop-in replacement, with no new specification or API to be reviewed. Indeed, some of the new vectorized code from HACL×N has already been deployed in production: Firefox now uses our vectorized ChaCha20-Poly1305 code and Tezos uses our vectorized Blake2, yielding measurable performance benefits.

HACL* includes a verified provider called EverCrypt (Chapter 3) that offers an agile, multiplexing API on top of both HACL* and Vale code. It uses CPU autodetection to dynamically dispatch API calls to the most efficient implementation for the platform the code is running on. We worked with the HACL* developers to make our HACL×N code available through the agile EverCrypt API. We strongly encourage clients use this verified, future-proof API: as more efficient implementations get added to HACL*, users of EverCrypt automatically get upgraded to faster variants.

4.4.2 HPKE: a verified application of HACL×N

We now illustrate how HACL×N serves as a platform for authoring verified cryptographic constructions and applications. We focus on Hybrid Public Key Encryption (HPKE), a new cryptographic construction that is undergoing standardization at the IRTF [23], and is already being used in several upcoming protocols [110, 22].

HPKE is a public-key encryption scheme with optional sender authentication: any sender who knows the HPKE public key of a recipient can encrypt a sequence of messages under this key and send it over the network; the recipient can use the corresponding private key to decrypt the messages. Optionally, the sender may also use a pre-shared symmetric key or a private Diffie-Hellman key to authenticate the message, but we do not support this feature.

At its core, HPKE relies on three components:

1. A key encapsulation mechanism (KEM) that generates a fresh secret shared between the sender and recipient and encapsulates (encrypts) it for the recipient's public key;
2. A key derivation function (KDF) that derives an encryption context containing a key and a nonce from the shared secret;
3. An authenticated encryption algorithm (AEAD) that uses the encryption key to encrypt and decrypt a sequence of messages.

Hence, computationally expensive public-key cryptography is only needed to initialize the encryption context, which can then be used to efficiently encrypt any amount of data.

HPKE is an *agile* scheme that supports multiple ciphersuites. The RFC recommends four KEMs (P-256, P-521, Curve25519, Curve448), two KDFs (HKDF-SHA256, HKDF-SHA512) and three AEADs (AES-GCM-128, AES-GCM-256, ChaCha20-Poly1305). Any combination is valid: HPKE thus has 24 possible ciphersuites, and many more *implementation* combinations.

Individually verifying all these would be intractable. However, using the integrated HACL* library, we can build a *generic* implementation of HPKE in 800 lines of code, in a way that is abstract in the choice of its KEM, KDF and AEAD implementation. To instantiate this code for a specific ciphersuite on a particular platform, we only need to provide implementations for

Algorithm	Intel Kaby Lake Laptop			Intel Xeon Workstation			ARM Raspberry Pi 3B+			Coding and Verification Effort				
	Our Code		Other	Our Code		Other	Our Code		Other	Scalar	Vec	Equiv	Low*	Out.
	Scalar	AVX2	Fastest	Scalar	AVX512	Fastest	Scalar	Neon	Fastest	Spec	Spec	Proof	Impl.	C
ChaCha20	3.73	0.77	0.75 (j)	5.74	0.56	0.56 (d)	8.69	5.19	4.49 (o)	151	182	819	510	4083
Poly1305	1.59	0.37	0.35 (j)	2.31	0.39	0.51 (j)	4.20	3.11	1.50 (o)	56 (arith)	122	370 +3594	2361	7136
Blake2b	2.56	2.26	2.02 (b)	3.97	3.13	2.84 (b)	6.99	–	6.02 (b)	430	441	324	1077	2824
Blake2s	4.32	3.34	3.06 (b)	6.63	4.52	4.11 (b)	11.42	15.30	9.80 (b)					
SHA$_{224,256}$	7.41	1.62×8	1.49×8 (o)	11.36	1.69×8	2.29×8 (o)	15.70	12.92×4	15.09 (o)	213	420	662	1360	4647
SHA$_{384,512}$	5.06	1.95×4	3.25 (o)	7.38	1.44×8	4.99 (o)	11.27	–	9.77 (o)					
Total (lines of specs, proofs, and code):										850	12242			18690

Table 4.2 – Evaluating HACL×N Performance and Development Effort
Performance (left): For each algorithm, we measure CPU cycles per byte when processing 16384 bytes of data. We list these numbers for our portable (scalar) C code, for our best-performing vectorized implementation on the machine, and for the fastest alternative implementation we tested: (j) refers to verified assembly code from Jasmin [15]; (o) is OpenSSL, (b) is code from the Blake2 team [20], (d) is code submitted by Romain Dolbeau to SUPERCOP. For multi-buffer SHA-2, the total cycle count is divided by the number of inputs processed in parallel (indicated by $\times N$).
Development Effort (right): All our specifications and proofs are written in F*, our implementations are written in Low* and then compiled to C. We calculate the size of each file in the development using `cloc`, discarding comments. The Poly1305 implementation includes a large field arithmetic component, which is separately listed. We write a single implementation of Blake2 and SHA-2 for all variants of these algorithms.

KEM, KDF and AEAD on that platform to perform program specialization (§4.2.3), so long as these meet our agile specifications. We instantiate and compile our code to obtain 15 verified variants of HPKE that build upon our new implementations of ChaCha20-Poly1305 and SHA-2, as well as previously verified implementations of AES-GCM, Curve25519 and P-256 from Vale and HACL*. Each instantiation consists of about 10 lines of F* code, and compiles to about 380 lines of C code.

To use our HPKE implementation, applications have two options. They can rely on the full HACL* library and call HPKE through the agile EverCrypt API. This way, clients automatically obtain the fastest implementation available on each platform, at the expense of extra run-time checks and a large codebase. Alternatively, they may directly use one of the 15 specialized HPKE variants distributed with HACL×N, packaged with just the code that it needs. For example, HACL×N provides a makefile that a user can use to compile just the code needed for the HPKE ciphersuite consisting of Curve25519, SHA-256, and ChaCha20-Poly1305 for ARM, resulting in a self-contained vectorized HPKE implementation with 3000 lines of C (compared to 100K lines for the full library).

4.5 Evaluation and Discussion

Benchmarking Performance. Appendix D presents detailed performance measurements and analysis for all our code obtained using both the SUPERCOP framework [2] and a user-space version of KBENCH9000 [52]. We benchmarked each algorithm on a low-end ARM Cortex-A53 device (Raspberry Pi 3B+, supporting NEON), a mainstream Intel i7-7560U laptop (Dell XPS13, supporting AVX2), and a high-end Intel Xeon Gold 5122 workstation (Dell Precision, supporting AVX512). We also benchmarked our code on 4 Amazon EC2 instances; two with Intel Xeon

CPUs, two with ARMv8 CPUs. We compared the performance of our code to popular libraries like OpenSSL and LibSodium, to optimized implementations contributed to SUPERCOP, to verified assembly code from Jasmin, and to reference implementations for each algorithm.

Table 4.2 summarizes the results. Our goal is to answer two questions: (1) what is the performance benefit of using our vectorized HACL×N code over the portable C code previously used in HACL*; (2) how does our code compare to optimized implementations, both verified and unverified, both in C and in hand-written assembly.

On AVX2, our vectorized code for ChaCha20, Poly1305, and SHA-2 is 3-5× faster than scalar code. On AVX512, the speed-up for these algorithms is 5-10×. Blake2 offers smaller speed-ups: 1.13-1.29× on AVX2, and 1.27-1.47× on AVX512. The reason for this modest improvement is that the portable C code for Blake2 is already very fast, and is heavily optimized by modern C compilers. On ARM Neon, the performance gains are more modest. ChaCha20 is 1.7× faster and Poly1305 is 1.4× faster than scalar code. Perhaps more surprisingly, the vectorized code for Blake2 provides no gains on low-end ARM devices, and is sometimes *worse* than scalar code. This is a known issue on ARM CPUs where the latency of vector shift instructions (used extensively in hash functions like Blake2) is quite high [8]. On higher-end ARM devices, like the Apple A9, and on upcoming ARM servers, we expect that vectorized code will reap significant benefits.

It is instructive to compare the performance of our ChaCha20-Poly1305 code with Jasmin's AVX2 assembly code, the only other verified implementation. On the laptop, our AVX2 code is 3-6% slower than Jasmin, and this performance gap is due to AVX2-specific instruction interleaving optimizations in the Jasmin code. However, Jasmin does not have an AVX512 implementation, so on Xeon workstations, our AVX512 code is significantly faster than Jasmin's AVX2. Interestingly, on these machines, even our AVX2 code is faster than Jasmin (see Table D.2), which indicates that the advantages of careful instruction interleaving do not carry over to other platforms. These measurements illustrate the tradeoff between our generic programming methodology and platform-specific assembly. On a specific platform, a skilled assembly programmer can eke out 5-10% extra speed from cryptographic code. However, we obtain both AVX2 and AVX512 implementations from the same verified source code, whereas one would have to re-write (and re-verify) a new AVX512 implementation in Jasmin. On ARM, our ChaCha20-Poly1305 code is the only verified implementation (Jasmin does not support ARM) and is 1.16-2.1× slower than hand-optimized OpenSSL assembly.

Our generic Blake2 code is between 10-15% slower than other implementations that include platform-specific permutation code. Our multi-buffer SHA-256 code is 9% slower than OpenSSL assembly on AVX2 but 35% faster on AVX512. OpenSSL does not provide multi-buffer implementations for other variants of SHA-2.

In summary, with the exception of hash functions on ARM devices, vectorization provides a measurable speed-up for all algorithms on all platforms. Furthermore, the C code extracted from our verified vectorized implementations is close in performance to the fastest available hand-optimized assembly code on each platform.

Estimating Developer Skill and Effort. From a software engineering viewpoint, we would like to answer two further questions: (1) what is the coding and verification effort of HACL×N compared to HACL* [127]; (2) what is the developer skill and work required to extend our library with new algorithms and new architectures.

Recall the workflow for developing and extending HACL×N (Figure 4.1). Table 4.2 tries to quantify the effort for the main steps in our workflow for each algorithm, in terms of lines of code and proof. If we measure verification overhead in terms of lines of proof for each line of generated C, HACL×N code has an overhead under 0.9X, compared to the 3× overhead in HACL* [127].

In ChaCha20, for example, our code and proofs total to 1511 lines, this is more than the 691 lines for scalar ChaCha20 in HACL*, but we are able to compile our code to a scalar implementation and 3 vectorized implementations, totaling 4083 lines of C (with a proof overhead of 0.37). Our largest development is Poly1305, totaling 6447 lines, which can itself be broken down into the field arithmetic (3594 lines) and polynomial evaluation (2853 lines). This generic implementation is about twice as large as the original scalar code in [127], but compiles to 4 C implementations, totaling 7136 lines of C (overhead 0.9). Moreover, we expect large parts of our vectorized bignum code to be reusable in other algorithms.

The skills required to extend HACL×N depend on the task. Compiling the code to C, specializing an algorithm for a platform, and making small modifications requires standard programming skills. Writing a high-level specification requires knowledge of the cryptographic algorithm and basic knowledge of functional programming. Implementing an algorithm requires knowledge of the F* language and type system but we ease the coding effort by providing a few well-documented libraries for integer, array, and vector operations. Extending HACL×N with a new platform requires knowledge of the new instruction set, and familiarity with the vector libraries. For example, PhD students in our team can usually write a high-level specification in a day, and a generic vectorized implementation in a week. Adding AVX512 to HACL×N took about a week.

Verifying an implementation against a high-level specification requires considerable skill in formal verification and a good familiarity with F* and Low*. For algorithms like ChaCha20, Blake2, and SHA-2, most of the subsequent effort is in proving specification equivalence for vectorization, and memory safety for low-level code. Using various library lemmas, these proofs typically take one week for each algorithm. Verifying code that interleaves vectorization with complex math can take significantly longer; it took the author about a month to write, verify and optimize Poly1305.

Relying on the C Compiler. All our performance measurements above relied on mainstream compilers like GCC and CLANG running with all their optimizations (−O3). Adding large unverified compilers like GCC into the trusted computing base (TCB) for a cryptographic library is risky, since they may be buggy [124] and may introduce side channels that were not present in the C code [116].

To estimate the impact of compiler optimizations on HACL×N performance, we measured the performance of our code with different C compilers at various optimization levels. Appendix E presents the detailed results, but to summarize, most of the performance of our code depends only on well-understood compiler optimizations enabled in O1 and O2, such as constant propagation, inlining, loop unrolling, and dead store elimination. Hence, a verified compiler that implemented just these optimizations could get close to the performance of GCC while eliminating the compiler from our TCB.

We measured the performance of the CompCert verified compiler [79], which implements a few standard optimizations. On our scalar code (CompCert does not yet support SIMD), we found that CompCert is about 2× slower than GCC and 30% slower than CLANG at `O1`. However, CompCert is an active project and we hope to benefit from ongoing improvements for performance, side channel resistance [26], and SIMD support [16].

Our F*-based ecosystem allows a variety of approaches to be brought together to build high-assurance high-performance cryptographic applications. We can write verified cryptographic code in Low* and compile it via GCC or CompCert, depending on the TCB. Performance-critical functions can be written in Vale assembly, verified in F*, and embedded back in HACL* [107]. Finally, we can write and verify protocol code in F* and compose it with our verified crypto to obtain fully verified protocol implementations [105, 48].

4.6 Deployment and Future Work

HACL×N has been integrated into the HACL* cryptographic library and all our code is publicly available at:

`https://github.com/project-everest/hacl-star`

Our vectorized ChaCha20 and Poly1305 implementations have been deployed in the NSS cryptographic library used by Mozilla Firefox, and in the TLS stack used in Microsoft's msQuic implementation. Our vectorized Blake2 code is being deployed in the Tezos blockchain. Other deployments are ongoing.

Each deployment induces a new workflow that exercises different aspects of our verified codebase. For example, integrating our code into NSS requires specification and code review by the NSS developers. Consequently, a good amount of our engineering effort goes into generating readable C code from KreMLin, in a way that follows NSS coding guidelines. The code is then subjected to static analysis tools that check for unused variables, dead code and other issues that sometimes require fixes in the Low* source code. Finally, once our C code passes the audit, it is integrated into the NSS continuous integration (CI) infrastructure, where it is regularly tested on a large number of platforms, against both hand-written unit tests and test frameworks like Wycheproof [4]. The code is then pushed to the main NSS branch and included in Firefox (a few thousand users) to find early deployment problems. After 2-4 weeks, it is deployed to Firefox Beta (a few million users) where more platform compatibility issues may be found due to the increased coverage. A month later, if no issues are found, the code is released in the Firefox browser (about 250 million users.)

The above workflow requires close coordination between NSS and HACL* developers over an extended period of time. A similar level of engagement is needed for successful deployments in Tezos and msQuic. This additional time and effort should be seen as the cost of transferring verified code from a research project like ours to real-world software applications.

This chapter has focused on a few algorithms, but we are working on extending the library with many more vectorized implementations, following the same SIMD patterns we have discussed here. We also plan to optimize our code better for low-end ARM devices and investigate new vectorization strategies for such platforms.

Chapter 5

A Verified Bignum Library

Many cryptographic primitives rely on modular arithmetic, which is the basis for more complex mathematical operations such as modular exponentiation and elliptic curve scalar multiplication. As we have seen, we can build an efficient verified implementation of Curve25519 and Poly1305 with the help of the customized bignum library, enabling modular-reduction optimizations as *the modulus is known in advance.* However, in many cases, we still need access to a verified bignum library that is generic, portable, and relatively fast, i.e., does not depend on a modulus form and works for any platform without sacrificing performance. For example, in RSA encryption and signing, which are still commonly used in Transport Layer Security (TLS) and Internet Protocol Security (IPsec), *the modulus is not known in advance, it is not prime, and even its size is unknown.*

Formally Verified Bignum Libraries. As we have previously seen, many research projects have worked on formally verified implementations of the Poly1305 one-time MAC and elliptic curves, such as Curve25519 and NIST P-256, i.e., where the modulus is known in advance. To achieve that, various approaches have been developed to verify highly-optimized assembly [36, 59, 14, 15, 60, 112, 113], portable C implementations [56, 127, 82] or their combination [107]. Some work has been done on formal verification of a general purpose arbitrary-precision integer arithmetic library [92, 11, 84], which in most cases is not suitable for cryptography as it does not run in constant-time.

Our Approach. We follow the methodology described in Chapter 2 and depicted in Figure 2.1 to obtain verified portable implementations for modular-arithmetic-based cryptographic code. As it is the case for all cryptographic code in HACL*, we aim to accomplish the following verification and security goals for our bignum library: memory safety, functional correctness against a high-level F* specification, and secret independence. We separate between memory safety and functional correctness reasoning by introducing a low-level F* specification, where we prove the functional correctness of arithmetic algorithms for a chosen model of low-level data representation.

Our Contribution. We built and applied our generic bignum library in both the Ed25519 and RSA-PSS signature schemes and the Finite-Field Diffie-Hellman key exchange over standard groups. Our bignum library can also be applicable in other algorithms and situations where a modulus is not known in advance or a programmer does not have the resources to implement a

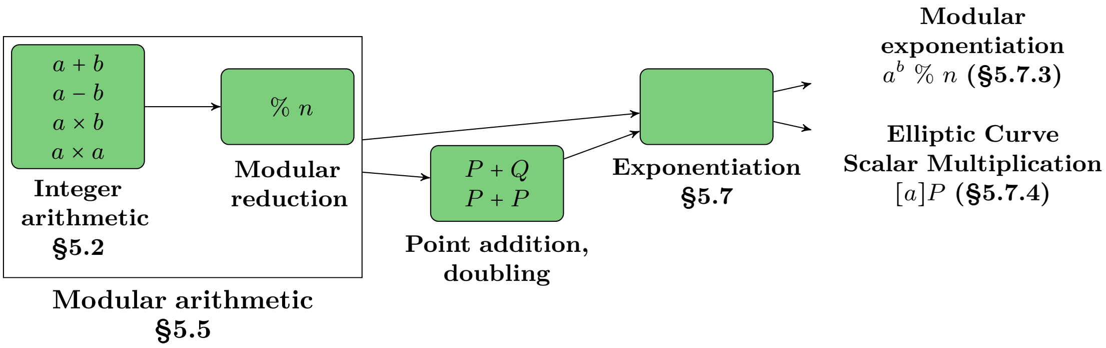


Figure 5.1 – High-level specification for Modular Exponentiation and Elliptic Curve Scalar Multiplication.

customized bignum library from scratch.

5.1 Bignum Library: an Overview

Our goal is to build a verified bignum library that covers all arithmetic operations needed for modular exponentiation and elliptic curve scalar multiplication. We identify three main layers that can be shared between the two operations. The first consists of the basic arithmetic operations: addition, subtraction, multiplication, and squaring. The second contains modular arithmetic, which can be implemented efficiently by interleaving integer arithmetic computations with modular reduction. The final layer is exponentiation, defined as a repeated application of a commutative monoid operation or, when an inverse operation is fast, an abelian group operation (§5.7). All the layers are shown in Figure 5.1.

Figure 5.1 can be seen as a high-level specification for modular exponentiation and scalar multiplication, as it is independent of the underlying bignum representation. For the former, the commutative monoid operation is a modular multiplication with the identity element 1. For the latter, the abelian group operation is a point addition with the identity element “point at infinity”. The point addition is defined using field arithmetic operations: addition, subtraction, multiplication and squaring.

We divide our bignum library into two parts, “Modular arithmetic” and “Exponentiation”. The latter is defined in a generic way and covers both modular exponentiation and scalar multiplication. We do not provide generic formulas for “Point addition and doubling” as there exist, for example, complete, uniform, and efficient formulas for Edwards Curves [69].

We start by considering the algorithms for integer and modular arithmetic that use a packed representation. The API for integer and modular arithmetic is presented in Sections 5.2 and 5.5, respectively. Section 5.3 covers the schoolbook algorithms for addition, subtraction and multiplication. Section 5.4 introduces more efficient algorithms for multiplication, such as squaring and Karatsuba multiplication. Then, in Sections 5.5 and 5.6, we discuss Montgomery reduction as a fast way of computing modular reduction.

Then, we introduce the generic methods for computing an exponentiation such as binary, fixed-window, and double fixed-window methods (§5.7), presented in the increasing order of their efficiency. In §5.7.3, we show how to compute modular exponentiation using Montgomery arithmetic and in §5.7.4, scalar multiplication for Ed25519 using the customized bignum library from Section 3.4.

Finally, we apply our bignum library in both the Ed25519 and RSA-PSS signature schemes and the Finite-Field Diffie-Hellman key exchange (§5.8) and report the performance numbers and verification effort in §5.9.

5.2 Low* API for Integer Arithmetic

For convenience, we recall the definitions given in §2.3.

Bignum representation. In this chapter, we encode a non-negative mathematical integer A as a fixed-length sequence of unsigned machine integers $a = [a_0; a_1; \ldots; a_{len-1}]$ such that

$$A = \sum_{i=0}^{len-1} a_i \cdot \beta^i, \text{ where}$$

— a radix $\beta = 2^{32}$ is used for 32-bit unsigned integers and

a radix $\beta = 2^{64}$ is used for 64-bit unsigned integers,

— a positive integer len is the length of the sequence and $A < \beta^{len}$,

— $\forall\ 0 \le i < len$, a limb a_i is the i-th element of the sequence and $0 \le a_i < \beta$.

Such a representation is *unique* and is called a *little-endian representation in radix-β*, i.e., the least significant limb is at index 0.

Bignum evaluation. In order to express the semantics of bignum operations, we define the bn_v function that reflects a given sequence a of positive length len to the mathematical integer as follows:

$$bn\_v\ a = \sum_{i=0}^{len-1} a_i \cdot \beta^i.$$

Functional correctness for bignum operations. For a bignum operation bn_op, we prove that the mathematical integer of its result is the same as if we had performed the mathematical operation op on the mathematical integers of its inputs. For example, if bn_op is a binary operation that takes as input two sequences a and b and returns another sequence c, we show that the following holds:

$$bn\_v\ (bn\_op\ a\ b) = op\ (bn\_v\ a)\ (bn\_v\ b).$$

In the following, we will sometimes omit the function bn_v when it is obvious from the context. In other words, we implicitly identify a mathematical integer with its bignum representation.

Bignum representation and operations in F* (high-level specification). The mathematical specification for the basic arithmetic and comparison operations is simple and can be written in F* as follows:

The type `int` represents unbounded mathematical integers in F*, and the type `nat` is an abbreviation for non-negative integers (`let nat = x:int{x ≥ 0}`). For our library, we define the

```
let add (a b:nat) : nat = a + b (∗ addition ∗)
let sub (a b:nat) : int = a − b (∗ subtraction ∗)
let mul (a b:nat) : nat = a ∗ b (∗ multiplication ∗)
let lt (a b:nat) : bool = a < b (∗ less than ∗)
let eq (a b:nat) : bool = a = b (∗ equal ∗)
```

operations over all pairs of natural numbers, i.e., we do not require a to be greater than or equal to b for the subtraction function.

Bignum representation in F* (low-level specification). We define a generic bignum type bignum t len as a sequence of *secret* integers, where the parameter t is either U32 or U64, and len is the length of the sequence. U32 and U64 correspond to 32-bit and 64-bit unsigned integers respectively.

```
let limb_t = t:inttype{t = U32 ∨ t = U64}
let limb (t:limb_t) = uint_t t SEC
let bignum (t:limb_t) (len:nat) = b:seq (limb t){length b = len}
```

Bignum evaluation in F* (from low-level to high-level specification). We define the bn_v function in F* via a total recursive function eval as follows.

```
let rec eval #t #len (b:bignum t len) (i:nat{i ≤ len}) : Tot nat =
  if i = 0 then 0 else eval b (i − 1) + v b.[i − 1] ∗ pow2 (bits t ∗ (i − 1))

let bn_v #t #len (b:bignum t len) = eval b len
```

The bn_v and eval functions mark arguments t and len as implicit with the # symbol, indicating that the F* type-checker can infer them automatically. The bits t function returns the number of bits in a given type t (32 for t = U32 and 64 for t = U64). The pow2 n function computes a power of two, i.e., 2^n. Finally, the v function reflects a machine integer to the mathematical integer.

Bignum representation in Low* (stateful code). We represent a bignum as an array of machine integers, where the length len is within the representable range of a size_t type, a public 32-bit unsigned integer.

```
let lbignum (t:limb_t) (len:size_t) = lbuffer (limb t) len
```

Bignum evaluation in Low* (from stateful code to high-level specification). The bignum evaluation function lbn_v maps the contents of a given array b in a given memory h to the mathematical integer.

```
let lbn_v #t #len (h:mem) (b:lbignum t len) : GTot nat = bn_v (as_seq h b)
```

Low* API for bignum operations. There are different algorithms to compute the same arithmetic operation and several ways of implementing the same algorithm. Nevertheless, each implementation must conform to its high-level specification. So, our goal is to show that our implementation type-checks against the interface that we specify for the following bignum operations:

— basic arithmetic operations: addition, subtraction, multiplication, and squaring;

— constant-time comparison operations;

— conversion functions between natural numbers, bytes, and our bignum representation.

Figure 5.2 depicts our Low* API for the basic arithmetic and constant-time comparison operations that we describe in more detail in the following. For simplicity, we assume that bignums a and b have the same length len of the following type.

```
let bn_len (t:limb_t) = len:size_t{0 < v len ∧ 2 ∗ bits t ∗ v len ≤ max_size_t}
```

The refinement on len ensures that there is no integer overflow in intermediate computations, e.g., when representing the integer (2 · bits t · v len) as required for Montgomery arithmetic (§5.5).

Addition and subtraction. The bn_add function performs a bignum addition of a and b. The function returns a carry c_out and writes the result in an array res of length len. The precondition requires the liveness and either disjointness or equality of the input and output arrays. The postcondition ensures that only the array res is modified, leaving other disjoint objects in memory unchanged. The functional correctness guarantees that multiplying the carry c_out by $2^{\text{bits t} * \text{v len}}$ and adding the product to res results in the bignum addition of a and b. This means that the function is *memory safe* and *functionally correct* for the following cases:

— bn_add len a b res, where a, b and res are pairwise disjoint

— bn_add len a a a

— bn_add len a a res, where a and res are disjoint

— bn_add len a b a, where a and b are disjoint

— bn_add len a b b, where a and b are disjoint.

The signature of the subtraction function bn_sub is similar to bn_add, except for functional correctness:

```
lbn_v h1 res − v c_out ∗ pow2 (bits t ∗ v len) == lbn_v h0 a − lbn_v h0 b
```

The array res stores the result of subtracting b from a, followed by reduction modulo $2^{\text{bits t} * \text{v len}}$, and c_out is a sign bit. That is, our bignum representation and operations are defined in the same way as unsigned integers and arithmetic operations in C.

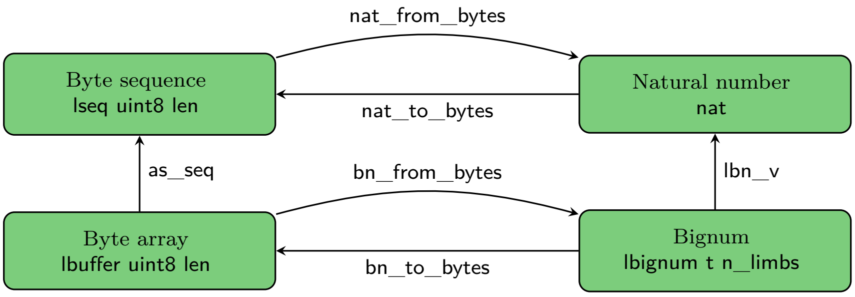
Multiplication and squaring. The bn_mul function performs a bignum multiplication of a and b and writes the result in an array res of twice the length len. Note that we require the input arrays a and b to be disjoint from the array res. The squaring function bn_sqr is a special case of bn_mul where both operands, a and b, are equal.

Constant-time comparison. The bn_lt_mask function returns $2^{\text{bits t}} - 1$ if a is less than b and zero otherwise. The bn_eq_mask function returns $2^{\text{bits t}} - 1$ if a is equal to b and zero otherwise. The modifies0 clause indicates that nothing is modified in memory.

Conversion functions between bytes and our bignum representation. Figure 5.3 depicts our Low* API for loading and serializing a bignum from either a big- or little-endian memory representation, i.e., from a sequence of bytes b of positive length len. The difference between the two representations is in the byte order. The former stores the most significant byte b_{len-1}

```
let carry (t:limb_t) = x:limb t{v x == 0 ∨ v x == 1}
let mask (t:limb_t) = x:limb t{v x == 0 ∨ v x == ones_v t}

val bn_add (#t:limb_t) (len:bn_len t) (a b res:lbignum t len) : Stack (carry t)
  (requires λ h → live h a ∧ live h b ∧ live h res ∧
    eq_or_disjoint a b ∧ eq_or_disjoint a res ∧ eq_or_disjoint b res)
  (ensures λ h0 c_out h1 → modifies (loc res) h0 h1 ∧
    (∗ Functional correctness of bignum addition ∗)
    v c_out ∗ pow2 (bits t ∗ v len) + lbn_v h1 res == lbn_v h0 a + lbn_v h0 b)

val bn_sub (#t:limb_t) (len:bn_len t) (a b res:lbignum t len) : Stack (carry t)
  (requires λ h → live h a ∧ live h b ∧ live h res ∧
    eq_or_disjoint a b ∧ eq_or_disjoint a res ∧ eq_or_disjoint b res)
  (ensures λ h0 c_out h1 → modifies (loc res) h0 h1 ∧
    (∗ Functional correctness of bignum subtraction ∗)
    lbn_v h1 res − v c_out ∗ pow2 (bits t ∗ v len) == lbn_v h0 a − lbn_v h0 b)

val bn_mul (#t:limb_t) (len:bn_len t) (a b:lbignum t len) (res:lbignum t (len +! len)) : Stack unit
  (requires λ h → live h a ∧ live h b ∧ live h res ∧
    disjoint res a ∧ disjoint res b ∧ eq_or_disjoint a b)
  (ensures λ h0 _ h1 → modifies (loc res) h0 h1 ∧
    (∗ Functional correctness of bignum multiplication ∗)
    lbn_v h1 res == lbn_v h0 a ∗ lbn_v h0 b)

val bn_sqr (#t:limb_t) (len:bn_len t) (a:lbignum t len) (res:lbignum t (len +! len)) : Stack unit
  (requires λ h → live h a ∧ live h res ∧ disjoint res a)
  (ensures λ h0 _ h1 → modifies (loc res) h0 h1 ∧
    (∗ Functional correctness of bignum squaring ∗)
    lbn_v h1 res == lbn_v h0 a ∗ lbn_v h0 a)

val bn_lt_mask (#t:limb_t) (len:bn_len t) (a b:lbignum t len) : Stack (mask t)
  (requires λ h → live h a ∧ live h b ∧ eq_or_disjoint a b)
  (ensures λ h0 r h1 → modifies0 h0 h1 ∧
    (∗ Functional correctness of constant−time bignum comparison ∗)
    (if lbn_v h0 a < lbn_v h0 b then v r == ones_v t else v r == 0))

val bn_eq_mask (#t:limb_t) (len:bn_len t) (a b:lbignum t len) : Stack (mask t)
  (requires λ h → live h a ∧ live h b ∧ eq_or_disjoint a b)
  (ensures λ h0 r h1 → modifies0 h0 h1 ∧
    (∗ Functional correctness of constant−time bignum equality ∗)
    (if lbn_v h0 a = lbn_v h0 b then v r == ones_v t else v r == 0))
```

Figure 5.2 – Low* API for the basic arithmetic and constant-time comparison operations.



```
let blocks (x:size_t{0 < v x}) (m:size_t{0 < v m}) = (x −. 1ul) /. m +. 1ul
let bytes_len (t:limb_t) = len:size_t{0 < v len ∧
  numbytes t ∗ v (blocks len (size (numbytes t))) ≤ max_size_t}
let n_limbs (t:limb_t) (len:bytes_len t) = blocks len (size (numbytes t))

(∗ two versions: for big−endian and little−endian numbers ∗)
val bn_from_bytes (#t:limb_t) (len:bytes_len t) (b:lbuffer uint8 len)
  (c:lbignum t (n_limbs t len)) : Stack unit
  (requires λ h → live h b ∧ live h c ∧ disjoint c b)
  (ensures λ h0 _ h1 → modifies (loc c) h0 h1 ∧
    (∗ Functional correctness of loading a natural number from byte array
    into the bignum representation ∗)
    lbn_v h1 c == nat_from_bytes (as_seq h0 b))

(∗ two versions: for big−endian and little−endian byte arrays ∗)
val bn_to_bytes (#t:limb_t) (len:bytes_len t) (c:lbignum t (n_limbs t len))
  (b:lbuffer uint8 len) : Stack unit
  (requires λ h → live h c ∧ live h b ∧ disjoint b c ∧
    lbn_v h c < pow2 (8 ∗ v len))
  (ensures λ h0 _ h1 → modifies (loc b) h0 h1 ∧
    (∗ Functional correctness of serializing a natural number into byte array
    from the bignum representation ∗)
    as_seq h1 b == nat_to_bytes (v len) (lbn_v h0 c))
```

Figure 5.3 – Low* API for conversion functions between natural numbers, bytes, and our bignum representation.

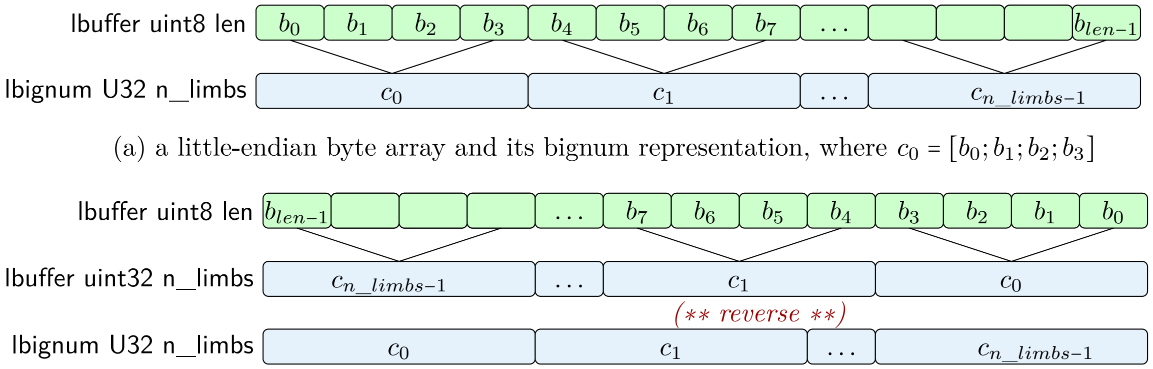
at the smallest memory address, while the latter stores the least significant byte b_0 there. Both representations are depicted in Figure 5.4.

The function nat_from_bytes returns the mathematical integer of a byte sequence b, while the function nat_to_bytes converts it back:

$$\text{nat\_from\_bytes}\ b = \sum_{i=0}^{\text{len}-1} b_i \cdot (2^8)^i$$

$$\text{nat\_to\_bytes len (nat\_from\_bytes b)} == \text{b}.$$

The functions bn_from_bytes and bn_to_bytes implement nat_from_bytes and nat_to_bytes, respectively, for the chosen bignum representation in Low*. For example, in order to convert the input bytes b to our bignum representation, bn_from_bytes_le splits them into $\frac{\text{bits t}}{8}$-byte blocks. If the last block is partial, it is filled with zeros to obtain a full block. Each block is then stored


lbuffer uint8 len
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7 . . . b_{len-1}
lbignum U32 n_limbs
c_0 c_1 . . . $c_{n\_limbs-1}$
lbuffer uint8 len
b_{len-1} . . . b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
lbuffer uint32 n_limbs
$c_{n\_limbs-1}$ . . . c_1 c_0
(** reverse **)
lbignum U32 n_limbs
c_0 c_1 . . . $c_{n\_limbs-1}$

(a) a little-endian byte array and its bignum representation, where $c_0 = [b_0; b_1; b_2; b_3]$

(b) a big-endian byte array and its bignum representation, where $c_0 = [b_3; b_2; b_1; b_0]$

Figure 5.4 – Conversion between bytes and our bignum representation.

as a bits t-bit integer c_i. We relate the input b and output c in the following way:

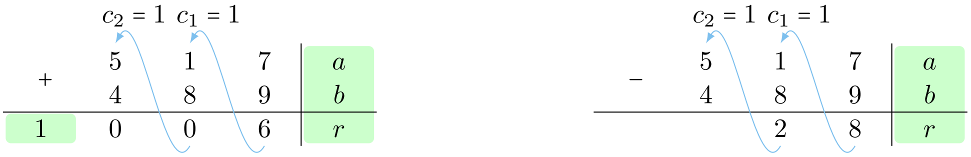
$$\text{bn\_v}\ c = \sum_{i=0}^{\text{n\_limbs}-1} c_i \cdot \beta^i, \text{ where } c_i = \sum_{j=0}^{nb-1} b_{nb\cdot i+j} \cdot (2^8)^j,$$

$$nb = \frac{\text{bits t}}{8} \text{ and } \text{n\_limbs} = \frac{\text{len} - 1}{nb} + 1 = \left\lceil \frac{len}{nb} \right\rceil.$$

5.3 Verifying Schoolbook Algorithms

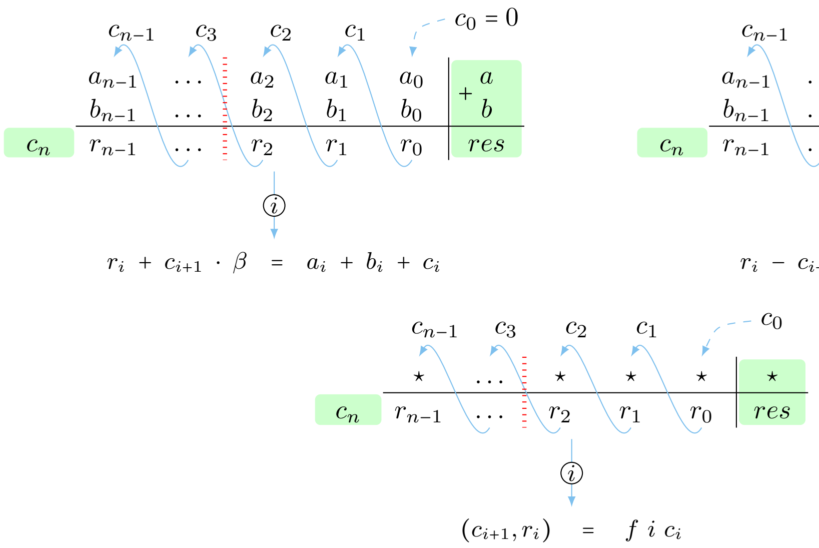
5.3.1 Addition and Subtraction

Let us recall what it means to carry out addition and subtraction using the schoolbook methods by taking as an example two multi-digit numbers, 517 and 489.



Both operations perform single-digit computations either of the form $a_i + b_i + c_i$ or of the form $a_i - b_i - c_i$ by composing and decomposing numbers in the radix-10 notation, starting from the least significant digit.

$$\begin{aligned}
&517 + 489 = \\
&(7 \cdot 10^0 + 1 \cdot 10^1 + 5 \cdot 10^2) + \\
&(9 \cdot 10^0 + 8 \cdot 10^1 + 4 \cdot 10^2) = \\
&\quad [10^i : a_i + b_i + c_i = r_i + c_{i+1} \cdot 10;\ \text{keep } r_i \text{, carry } c_{i+1} \text{ to the next-higher coeffient}] \\
&\quad [10^0 : 7 + 9 = 16 = 6 + 10;\ \text{keep 6, carry 1 to the next-higher coeffient}] \\
&\quad [10^1 : 1 + 8 + 1 = 10 = 0 + 10;\ \text{keep 0, carry 1 to the next-higher coeffient}] \\
&\quad [10^2 : 5 + 4 + 1 = 10 = 0 + 10;\ \text{keep 0, carry 1 to the next-higher coeffient}] \\
&\quad [10^3 : 1;\ \text{keep 1}] \\
&6 \cdot 10^0 + 0 \cdot 10^1 + 0 \cdot 10^2 + 1 \cdot 10^3 = 1006
\end{aligned}$$



c_{n-1} c_3 c_2 c_1 $c_0 = 0$
a_{n-1} ... a_2 a_1 a_0 + a
b_{n-1} ... b_2 b_1 b_0 b
c_n r_{n-1} ... r_2 r_1 r_0 res
i
$r_i + c_{i+1} \cdot \beta = a_i + b_i + c_i$
c_{n-1} c_3 c_2 c_1 $c_0 = 0$
a_{n-1} ... a_2 a_1 a_0 − a
b_{n-1} ... b_2 b_1 b_0 b
c_n r_{n-1} ... r_2 r_1 r_0 res
i
$r_i - c_{i+1} \cdot \beta = a_i - b_i - c_i$
c_{n-1} c_3 c_2 c_1 c_0
⋆ ... ⋆ ⋆ ⋆ ⋆
c_n r_{n-1} ... r_2 r_1 r_0 res
i
$(c_{i+1}, r_i) = f\ i\ c_i$

Figure 5.5 – Schoolbook addition and subtraction share a common pattern that we define as a generate_elems combinator.

$517 - 489 =$

$(7 \cdot 10^0 + 1 \cdot 10^1 + 5 \cdot 10^2) -$

$(9 \cdot 10^0 + 8 \cdot 10^1 + 4 \cdot 10^2) =$

$[10^i : a_i - b_i - c_i = r_i - c_{i+1} \cdot 10;$ keep r_i, borrow c_{i+1} from the next-higher coeffient$]$

$[10^0 : 7 - 9 = -2 = 8 - 10;$ keep 8, borrow 1 from the next-higher coeffient$]$

$[10^1 : 1 - 8 - 1 = -8 = 2 - 10;$ keep 2, borrow 1 from the next-higher coeffient$]$

$[10^2 : 5 - 4 - 1 = 0;$ keep 0$]$

$8 \cdot 10^0 + 2 \cdot 10^1 + 0 \cdot 10^2 = 28$

We can identify a common pattern shared between the two operations, addition and subtraction of n-digit numbers a and b, as a loop combinator generate_elems that generates n elements by iteratively applying a function f with an accumulator c_i. For addition, such a function f can be defined as a single-digit addition with a carry that adds the i-th digits of two multi-digit numbers a and b with a carry c_i and yields a result r_i and a carry-out c_{i+1}, such that

$$r_i + c_{i+1} \cdot 10 = a_i + b_i + c_i.$$

For subtraction, it can be defined as a single-digit subtraction with a borrow that subtracts the i-th digit of a multi-digit number b and a borrow c_i from the i-th digit of a multi-digit number a and returns a result r_i and a carry-out c_{i+1}, such that

$$r_i - c_{i+1} \cdot 10 = a_i - b_i - c_i.$$

While this was presented on the familiar radix 10, we can generalize it to any bignum representation in radix-β, as shown in Figure 5.5.

Specifying and implementing a loop combinator generate_elems in F*. We specify the loop combinator generate_elems as a total recursive function that takes a function f and an initial value c0 and yields a final value c and a sequence containing n elements.

```
let rec generate_elems (#t:Type0) (#a:Type0) (max:nat) (n:nat{n ≤ max})
  (f:(i:nat{i < max} → a → a & t)) (c0:a) : a & s:seq t{length s = n} =
  if n = 0 then (c0, Seq.empty)
  else begin
    let (c1, res1) = generate_elems max (n − 1) f c0 in
    let (c, r) = f (n − 1) c1 in
    (c, Seq.snoc res1 r) end
```

The base case (when n = 0) returns the accumulator's initial value c0 and an empty sequence. The recursive case first splits the result of the recursive call into an accumulator c1 and a sequence res1 containing n−1 elements. It then calls the given function f on the index n−1 and accumulator c1, obtaining the final value c and an element r, which is added to the end of the resulting sequence res1.

While our F* specification uses the recursive definition of the generate_elems combinator to prove the functional correctness of the arithmetic operations by induction on n later, our Low* implementation replaces the recursion with a *for* loop that we unroll four times to speed it up.

Specifying addition in F* (low-level specification). Our F* specification for a bignum addition simply calls the loop combinator generate_elems defined above.

```
val addcarry (#t:limb_t) (c_in:carry t) (a b:limb t) : Pure (carry t & limb t) (requires ⊤)
  (ensures λ(c_out, r) → v r + v c_out ∗ pow2 (bits t) = v a + v b + v c_in)

let bn_add #t #len (a b:bignum t len) : carry t & bignum t len =
  generate_elems len len (λ i c_i → addcarry c_i a.[i] b.[i]) (uint #t 0)
```

The combinator is invoked with an initial value set to zero and a lambda function (λ i c_i → ...) that in each iteration i calls a function addcarry on the accumulator c_i and i-th elements of the two input bignums, a and b, of length len.

Verifying addition in F* (from low-level to high-level specification). We prove that the function bn_add computes a bignum addition of a and b by induction on i defined as a natural number less than or equal to len. Our induction hypothesis is that the addition of $a_{0...i}$ and $b_{0...i}$ is computed after the i-th loop iteration. We use the notation $a_{j...(j+k)}$ to denote a sub-sequence of a starting from index j, with length k. Note that $a_{0...i}$ corresponds to $(bn\_v\ a)\ \%\ \beta^i$ or (eval a i) in F*, as we use a little-endian representation in radix-β for our bignum representation. Setting i to len concludes our proof for the lemma bn_add_lemma.

```
val bn_add_loop_lemma (#t:limb_t) (#len:nat) (a b:bignum t len) (i:nat{i ≤ len}) :
  Lemma (let (c, res) = generate_elems len i (λ i c_i → addcarry c_i a.[i] b.[i]) (uint #t 0) in
    v c ∗ pow2 (bits t ∗ i) + eval res i = eval a i + eval b i)

let bn_add_lemma #t #len (a b:bignum t len) :
  Lemma (let (c, res) = bn_add a b in
    v c ∗ pow2 (bits t ∗ len) + bn_v res = bn_v a + bn_v b) = bn_add_loop_lemma a b len
```

The proof of the base case (when $i = 0$) is straightforward. The inductive case first assumes that the induction hypothesis holds for $(i - 1)$ and then shows that it is also proper for i.

$$\begin{aligned}
&\mathsf{res}_{0\ldots i} \\
&= \mathsf{res}_{0\ldots(i-1)} + 2^{\mathsf{bits\ t}\cdot(i-1)} \cdot \mathsf{res}.[i-1] && \text{[by the def. of } \mathsf{eval}\text{]} \\
&= \mathsf{res1}_{0\ldots(i-1)} + 2^{\mathsf{bits\ t}\cdot(i-1)} \cdot \mathsf{r} && \text{[by the def. of } \mathsf{Seq.snoc}\text{]} \\
&= \mathsf{res1}_{0\ldots(i-1)} + 2^{\mathsf{bits\ t}\cdot(i-1)} \cdot (\mathsf{a}.[i-1] + \mathsf{b}.[i-1] + \mathsf{c1} - \mathsf{c} \cdot 2^{\mathsf{bits\ t}}) && \text{[by the def. of } \mathsf{addcarry}\text{]} \\
&= \mathsf{a}_{0\ldots(i-1)} + \mathsf{b}_{0\ldots(i-1)} - 2^{\mathsf{bits\ t}\cdot(i-1)} \cdot \mathsf{c1} && \text{[by the induction hypothesis]} \\
&\quad +2^{\mathsf{bits\ t}\cdot(i-1)} \cdot (\mathsf{a}.[i-1] + \mathsf{b}.[i-1] + \mathsf{c1} - \mathsf{c} \cdot 2^{\mathsf{bits\ t}}) \\
&= \mathsf{a}_{0\ldots i} + \mathsf{b}_{0\ldots i} - 2^{\mathsf{bits\ t}\cdot i} \cdot \mathsf{c} && \text{[by the def. of } \mathsf{eval}\text{]}
\end{aligned}$$

Specifying and verifying subtraction in F* (low-level specification). Our F* specification and proof of the functional correctness for a bignum subtraction are very similar to what we have described for the bignum addition above.

```
val subborrow (#t:limb_t) (c_in:carry t) (a b:limb t) : Pure (carry t & limb t) (requires ⊤)
  (ensures λ(c_out, r) →v r − v c_out ∗ pow2 (bits t) = v a − v b − v c_in)

let bn_sub #t #len (a b:bignum t len) : carry t & bignum t len =
  generate_elems len len (λ i c_i →subborrow c_i a.[i] b.[i]) (uint #t 0)

val bn_sub_lemma (#t:limb_t) (#len:nat) (a b:bignum t len) :
  Lemma (let (c, res) = bn_sub a b in
    bn_v res − v c ∗ pow2 (bits t ∗ len) = bn_v a − bn_v b)
```

The generate_elems combinator is called with an initial value set to zero and a lambda function (λi c_i →...) that in each iteration i calls a subborrow function on the accumulator c_i and i-th elements of the input bignums, a and b, of length len. Having this, we can prove that the bn_sub function computes a subtraction of a bignum b from a bignum a.

5.3.2 Multiplication

We now consider the schoolbook method for computing a multiplication of two multi-digit numbers, 517 and 489.

$$\begin{array}{cccccc|l}
 & & & 5 & 1 & 7 & a \\
 & & \times & 4 & 8 & 9 & b \\
\hline
 & & 4 & 6 & 5 & 3 & a \cdot b_0 \\
 & 4 & 1 & 3 & 6 & & + a \cdot b_1 \cdot 10 \\
2 & 0 & 6 & 8 & & & + a \cdot b_2 \cdot 10^2 \\
\hline
2 & 5 & 2 & 8 & 1 & 3 & r
\end{array}$$

It first calculates all the intermediate products $517 \cdot 9 \cdot 10^0$, $517 \cdot 8 \cdot 10^1$, $517 \cdot 4 \cdot 10^2$ and then adds them together:

$$517 \cdot 489 = 517 \cdot (9 \cdot 10^0 + 8 \cdot 10^1 + 4 \cdot 10^2) = 517 \cdot 9 \cdot 10^0 + 517 \cdot 8 \cdot 10^1 + 517 \cdot 4 \cdot 10^2.$$

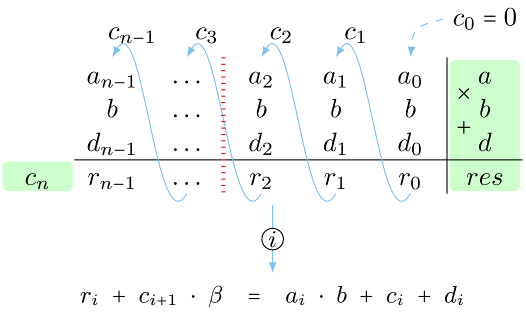

$c_0 = 0$
c_{n-1}
c_3
c_2
c_1
a_{n-1} ... a_2 a_1 a_0
b ... b b b
d_{n-1} ... d_2 d_1 d_0
c_n r_{n-1} ... r_2 r_1 r_0
× a
+ b
d
res
i
$r_i + c_{i+1} \cdot \beta = a_i \cdot b + c_i + d_i$

Figure 5.6 – The loop combinator generate_elems can be applied to a multiplication of a bignum by a limb with an in-place addition.

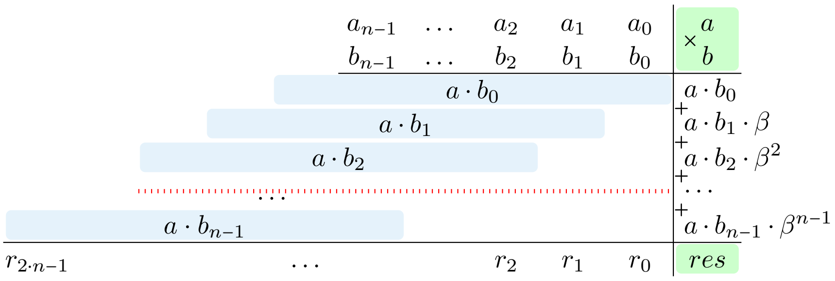
For a multiplication of a multi-digit number by a single digit b, we carry out single-digit computations of the form $a_i \cdot b + c_i$ by composing and decomposing numbers in the radix-10 notation, starting from the least significant digit.

$$
\begin{aligned}
&517 \cdot 9 \cdot 10^0 = \\
&(7 \cdot 10^0 + 1 \cdot 10^1 + 5 \cdot 10^2) \cdot 9 = \\
&\quad [10^i : a_i \cdot b + c_i = r_i + c_{i+1} \cdot 10;\ \text{keep } r_i\text{, carry } c_{i+1} \text{ to the next-higher coefficient}] \\
&\quad [10^0 : 7 \cdot 9 = 63 = 3 + 6 \cdot 10;\ \text{keep 3, carry 6 to the next-higher coefficient}] \\
&\quad [10^1 : 1 \cdot 9 + 6 = 15 = 5 + 1 \cdot 10;\ \text{keep 5, carry 1 to the next-higher coefficient}] \\
&\quad [10^2 : 5 \cdot 9 + 1 = 46 = 6 + 4 \cdot 10;\ \text{keep 6, carry 4 to the next-higher coefficient}] \\
&\quad [10^3 : 4;\ \text{keep 4}] \\
&3 \cdot 10^0 + 5 \cdot 10^1 + 6 \cdot 10^2 + 4 \cdot 10^3 = 4653
\end{aligned}
$$

A naive implementation of this method requires storing all the intermediate products before adding them, but we can accumulate them instead. Thus, we can immediately add the properly shifted single-digit products of $517 \cdot 8$ to the result of multiplying 517 by 9.

$$
\begin{aligned}
&517 \cdot 9 \cdot 10^0 + 517 \cdot 8 \cdot 10^1 = \\
&(3 \cdot 10^0 + 5 \cdot 10^1 + 6 \cdot 10^2 + 4 \cdot 10^3) + \\
&(0 \cdot 10^0 + 7 \cdot 10^1 + 1 \cdot 10^2 + 5 \cdot 10^3) \cdot 8 = \\
&\quad [10^i : a_i \cdot b + c_i + d_i = r_i + c_{i+1} \cdot 10;\ \text{keep } r_i\text{, carry } c_{i+1} \text{ to the next-higher coefficient}] \\
&\quad [10^0 : 3;\ \text{keep 3}] \\
&\quad [10^1 : 7 \cdot 8 + 5 = 61 = 1 + 6 \cdot 10;\ \text{keep 1, carry 6 to the next-higher coefficient}] \\
&\quad [10^2 : 1 \cdot 8 + 6 + 6 = 20 = 0 + 2 \cdot 10;\ \text{keep 0, carry 2 to the next-higher coefficient}] \\
&\quad [10^3 : 5 \cdot 8 + 2 + 4 = 46 = 6 + 4 \cdot 10;\ \text{keep 6, carry 4 to the next-higher coefficient}] \\
&\quad [10^4 : 4;\ \text{keep 4}] \\
&3 \cdot 10^0 + 1 \cdot 10^1 + 0 \cdot 10^2 + 6 \cdot 10^3 + 4 \cdot 10^4 = 46013
\end{aligned}
$$

Figure 5.6 shows how we can apply the loop combinator generate_elems defined in §5.3.1 to a



a_{n-1} ... a_2 a_1 a_0
b_{n-1} ... b_2 b_1 b_0
a × b
$a \cdot b_0$
$a \cdot b_1$
$a \cdot b_2$
...
$a \cdot b_{n-1}$
$a \cdot b_0$
+ $a \cdot b_1 \cdot \beta$
+ $a \cdot b_2 \cdot \beta^2$
+ ...
+ $a \cdot b_{n-1} \cdot \beta^{n-1}$
$r_{2 \cdot n-1}$ ... r_2 r_1 r_0
res

Figure 5.7 – Schoolbook multiplication.

multiplication of a multi-digit number a by a single digit b with or without performing an in-place addition with another multi-digit number d. For the latter, we can simply set d to zero and thus obtain the first case. For the former, the loop combinator takes a function that multiplies the i-th digit of a by b and then adds the result to the sum of a single digit c_i and the i-th digit of d. The function returns a tuple of single-digit numbers r_i and c_{i+1}, such that

$$r_i + c_{i+1} \cdot 10 = a_i \cdot b + c_i + d_i.$$

Note that the result of $a_i \cdot b + c_i + d_i$ fits into a two-digit number, since

$$a_i \leq 9 \wedge b \leq 9 \wedge c_i \leq 9 \wedge d_i \leq 9 \Rightarrow a_i \cdot b + c_i + d_i \leq 9 \cdot 9 + 9 + 9 = 99 = 9 + 9 \cdot 10.$$

Finally, Figure 5.7 depicts the schoolbook multiplication for any radix β, using the following formula:

$$a \cdot b = a \cdot (b_0 + b_1 \cdot \beta + b_2 \cdot \beta^2 + \ldots + b_{n-1} \cdot \beta^{n-1}) = a \cdot b_0 + a \cdot b_1 \cdot \beta + a \cdot b_2 \cdot \beta^2 + \ldots + a \cdot b_{n-1} \cdot \beta^{n-1}.$$

Complexity. The schoolbook multiplication requires n^2 double-wide multiplications (e.g., 64-bit × 64-bit → 128-bit). As we have seen in §3.4, the algorithm can be efficiently implemented using the MULX and ADX instructions (Figure 3.4).

Specifying and verifying multiplication by a limb with an in-place addition in F* (low-level specification). The function bn_mul1_add uses the generate_elems loop combinator to specify a multiplication of a bignum a of length len by a limb b with an in-place addition with another bignum d of the same length len. The result has at most (len + 1) limbs which we represent as a bignum res of length len with the most significant limb c aside.

```
val mul_wide_add2 (#t:limb_t) (a b c d:limb t) : Pure (limb t & limb tt) (requires ⊤)
  (ensures λ (hi, lo) → v lo + v hi ∗ pow2 (bits t) = v a ∗ v b + v c + v d)

let bn_mul1_add #t #len (b:limb t) (a d:bignum t len) : limb t & bignum t len =
  generate_elems len len (λ i c_i → mul_wide_add2 a.[i] b c_i d.[i]) (uint #t 0)

val bn_mul1_add_lemma (#t:limb_t) (#len:nat) (a d:bignum t len) (b:limb t) :
  Lemma (let (c, res) = bn_mul1_add b a d in
    v c ∗ pow2 (bits t ∗ len) + bn_v res = bn_v a ∗ v b + bn_v d)
```

The loop combinator is invoked with an initial value set to zero and a lambda function (λ i c_i → . . .) that in each iteration i calls a function mul_wide_add2 on the accumulator c_i, limb b, and i-th elements of the input bignums, a and d. The proof of the functional correctness is similar to the proof of the lemma bn_add_lemma from §5.3.1.

Specifying and verifying multiplication in F* (low-level specification). In the schoolbook multiplication, in each iteration i we first compute a multiplication of a bignum a by the i-th element of a bignum b. Then, we shift the product towards the most significant limb i times. Finally, we add the result to the accumulator value acc of length (len + len), i.e., the value $acc + a \cdot b_i \cdot \beta^i$ is calculated. In F*, we can specify this process for one iteration as a function bn_mul1_lshift_add that adds the product $a \cdot b_i$ to acc starting from index i as follows.

```
let bn_mul1_lshift_add #t #len #accLen (a:bignum t len) (b_i:limb t)
  (i:nat{i + len ≤ accLen}) (acc:bignum t accLen) : limb t & bignum t accLen =
  let (c, acc_i) = bn_mul1_add b_i a (slice acc i (i + len)) in
  (c, update_slice acc i (i + len) acc_i)
```

The function (slice acc i (i + len)) returns a sub-sequence of acc starting from index i, of length len. The function (update_slice acc i (i + len) acc_i) updates the input sequence acc with a sequence acc_i of length len starting from index i. Note that we do not add a limb c to the next-higher coefficient and do not modify other elements of a resulting sequence acc beyond the index (len + i), as we know that for the schoolbook multiplication, we can store the limb c directly at index (len + i). The semantics of the bn_mul1_lshift_add function can be expressed as follows.

```
val bn_mul1_lshift_add_lemma (#t:limb_t) (#len:nat) (#accLen:nat)
  (a:bignum t len) (b_i:limb t) (i:nat{i + len ≤ accLen}) (acc:bignum t accLen) :
  Lemma (let (c, res) = bn_mul1_lshift_add a b_i i acc in
   v c ∗ pow2 (bits t ∗ (len + i)) + eval res (len + i) =
       eval acc (len + i) + bn_v a ∗ v b_i ∗ pow2 (bits t ∗ i) ∧
   slice res (len + i) accLen == slice acc (len + i) accLen)
```

The proof of the lemma bn_mul1_lshift_add_lemma is presented below.

$$
\begin{aligned}
& \mathsf{res}_{0 \ldots (len+i)} \\
& = \mathsf{res}_{0 \ldots i} + 2^{\mathsf{bits\ t} \cdot i} \cdot \mathsf{res}_{i \ldots (i+len)} && \text{[by the def. of eval]} \\
& = \mathsf{acc}_{0 \ldots i} + 2^{\mathsf{bits\ t} \cdot i} \cdot \mathsf{acc\_i} && \text{[by the def. of update\_slice]} \\
& = \mathsf{acc}_{0 \ldots i} + 2^{\mathsf{bits\ t} \cdot i} \cdot (\mathsf{a} \cdot \mathsf{b\_i} + \mathsf{acc}_{i \ldots (i+len)} - \mathsf{c} \cdot 2^{\mathsf{bits\ t} \cdot len}) && \text{[by bn\_mul1\_add\_lemma]} \\
& = \mathsf{acc}_{0 \ldots (i+len)} + 2^{\mathsf{bits\ t} \cdot i} \cdot \mathsf{a} \cdot \mathsf{b\_i} - \mathsf{c} \cdot 2^{\mathsf{bits\ t} \cdot (len+i)} && \text{[by the def. of eval]}
\end{aligned}
$$

The complete F* specification of the schoolbook multiplication is as follows.

```
let rec bn_mul_loop #t #len (a b:bignum t len) (i:nat{i ≤ len}) : bignum t (len + len) =
  if i = 0 then create (len + len) (uint #t 0)
  else begin
    let acc1 = bn_mul_loop a b (i − 1) in
    let (c, acc) = bn_mul1_lshift_add a b.[i − 1] (i − 1) acc1 in
    acc.[len + i − 1] ← c end

let bn_mul #t #len (a b:bignum t len) = bn_mul_loop a b len
```

The bn_mul function calls a total recursive function bn_mul_loop on the input bignums, a and b, and index len. In the base case, the bn_mul_loop function returns an initial value of the accumulator, a sequence of length (len + len) filled with zeros. The recursive case calls the bn_mul1_lshift_add function on the input bignum a, (i − 1)-th element of a bignum b, index (i − 1), and result of recursively calling bn_mul_loop on the input bignums, a and b, and index (i − 1). Finally, we split the result into a bignum acc and a limb c, which is then stored at index (len + i − 1) of the resulting bignum acc.

We prove the functional correctness of the bn_mul_loop function by induction on i. Our induction hypothesis is that a multiplication of a by $b_{0 \ldots i}$ is computed after the i-th loop iteration. Setting i to len concludes our proof for the lemma bn_mul_lemma.

```
val bn_mul_loop_lemma (#t:limb_t) (#len:nat) (a b:bignum t len) (i:nat{i ≤ len}) :
  Lemma (eval (bn_mul_loop a b i) (len + i) = bn_v a ∗ eval b i)

let bn_mul_lemma #t #len (a b:bignum t len) :
  Lemma (bn_v (bn_mul a b) = bn_v a ∗ bn_v b) = bn_mul_loop_lemma a b len
```

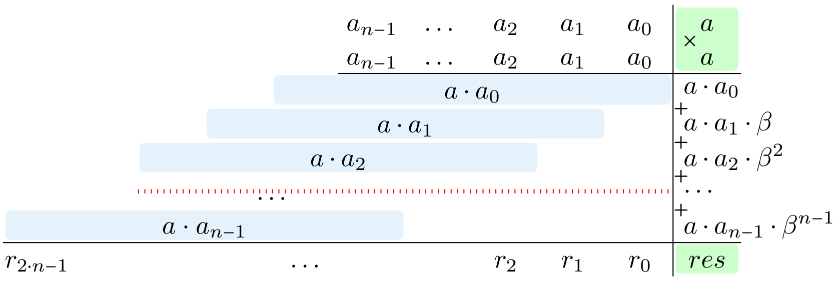
The proof of the base case is straightforward. The inductive case can be proved as follows, where acc = bn_mul_loop a b i:

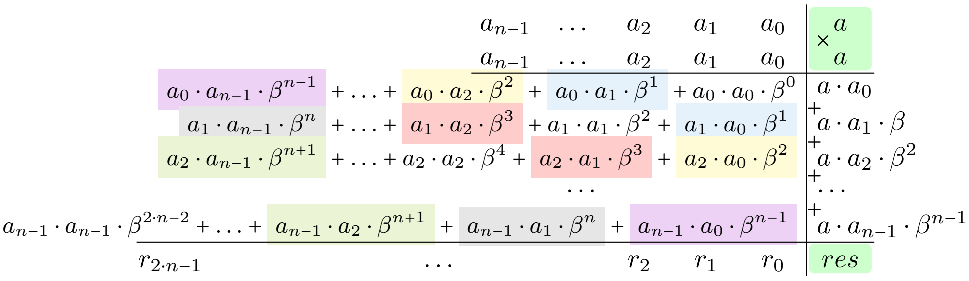
$$
\begin{aligned}
& \mathsf{acc}_{0 \ldots (len+i)} \\
&= \mathsf{acc}_{0 \ldots (len+i-1)} + 2^{\mathsf{bits\ t} \cdot (len+i-1)} \cdot \mathsf{acc}.[\mathsf{len{+}i{-}1}] && \text{[by the def. of eval]} \\
&= \mathsf{acc}_{0 \ldots (len+i-1)} + 2^{\mathsf{bits\ t} \cdot (len+i-1)} \cdot \mathsf{c} && \text{[by the def. of .[]]} \\
&= \mathsf{acc1}_{0 \ldots (len+i-1)} + 2^{\mathsf{bits\ t} \cdot (i-1)} \cdot \mathsf{a} \cdot \mathsf{b}.[i-1] && \text{[by bn\_mul1\_lshift\_add\_lemma]} \\
&= \mathsf{a} \cdot \mathsf{b}_{0 \ldots (i-1)} + 2^{\mathsf{bits\ t} \cdot (i-1)} \cdot \mathsf{a} \cdot \mathsf{b}.[i-1] && \text{[by the induction hypothesis]} \\
&= \mathsf{a} \cdot \mathsf{b}_{0 \ldots i} && \text{[by the def. of eval]}
\end{aligned}
$$

5.4 Verifying Optimized Algorithms for Multiplication

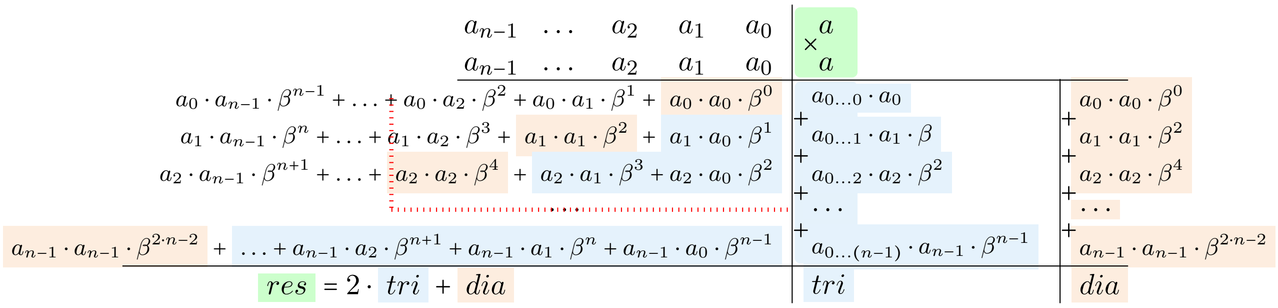
5.4.1 Squaring

Squaring is a special case of multiplication where both operands are equal. If we expand the definition bn_v for a bignum a in the schoolbook multiplication presented in Figure 5.7, we can observe that the terms $a_i \cdot a_j \cdot \beta^{i+j}$ and $a_j \cdot a_i \cdot \beta^{j+i}$ are identical.


a_{n-1} ... a_2 a_1 a_0
a_{n-1} ... a_2 a_1 a_0
a
×
a
$a \cdot a_0$
$a \cdot a_1$
$a \cdot a_2$
...
$a \cdot a_{n-1}$
$a \cdot a_0$
+ $a \cdot a_1 \cdot \beta$
+ $a \cdot a_2 \cdot \beta^2$
+ ...
+ $a \cdot a_{n-1} \cdot \beta^{n-1}$
$r_{2 \cdot n-1}$ ... r_2 r_1 r_0
res


a_{n-1} … a_2 a_1 a_0
$\times$ a
a_{n-1} … a_2 a_1 a_0
a
$a_0 \cdot a_{n-1} \cdot \beta^{n-1}$ + … + $a_0 \cdot a_2 \cdot \beta^2$ + $a_0 \cdot a_1 \cdot \beta^1$ + $a_0 \cdot a_0 \cdot \beta^0$
$a \cdot a_0$
$a_1 \cdot a_{n-1} \cdot \beta^n$ + … + $a_1 \cdot a_2 \cdot \beta^3$ + $a_1 \cdot a_1 \cdot \beta^2$ + $a_1 \cdot a_0 \cdot \beta^1$
+ $a \cdot a_1 \cdot \beta$
$a_2 \cdot a_{n-1} \cdot \beta^{n+1}$ + … + $a_2 \cdot a_2 \cdot \beta^4$ + $a_2 \cdot a_1 \cdot \beta^3$ + $a_2 \cdot a_0 \cdot \beta^2$
+ $a \cdot a_2 \cdot \beta^2$
…
+ …
$a_{n-1} \cdot a_{n-1} \cdot \beta^{2 \cdot n-2}$ + … + $a_{n-1} \cdot a_2 \cdot \beta^{n+1}$ + $a_{n-1} \cdot a_1 \cdot \beta^n$ + $a_{n-1} \cdot a_0 \cdot \beta^{n-1}$
+ $a \cdot a_{n-1} \cdot \beta^{n-1}$
$r_{2 \cdot n-1}$ … r_2 r_1 r_0
res

Instead of computing the products $a_i \cdot a_j \cdot \beta^{i+j}$ and $a_j \cdot a_i \cdot \beta^{j+i}$ twice, we can calculate such a term once and then double it. The figure below depicts how to compute a squaring for a bignum a. It first accumulates the products $a_{0 \ldots i} \cdot a_i \cdot \beta^i$ and then doubles the accumulator value tri, which is finally added to the sum of the products $a_i \cdot a_i \cdot \beta^{i+i}$ from the main diagonal.


a_{n-1} … a_2 a_1 a_0
$\times$ a
a_{n-1} … a_2 a_1 a_0
a
$a_0 \cdot a_{n-1} \cdot \beta^{n-1}$ + … + $a_0 \cdot a_2 \cdot \beta^2$ + $a_0 \cdot a_1 \cdot \beta^1$ + $a_0 \cdot a_0 \cdot \beta^0$
$a_{0 \ldots 0} \cdot a_0$
$a_0 \cdot a_0 \cdot \beta^0$
$a_1 \cdot a_{n-1} \cdot \beta^n$ + … + $a_1 \cdot a_2 \cdot \beta^3$ + $a_1 \cdot a_1 \cdot \beta^2$ + $a_1 \cdot a_0 \cdot \beta^1$
+ $a_{0 \ldots 1} \cdot a_1 \cdot \beta$
+ $a_1 \cdot a_1 \cdot \beta^2$
$a_2 \cdot a_{n-1} \cdot \beta^{n+1}$ + … + $a_2 \cdot a_2 \cdot \beta^4$ + $a_2 \cdot a_1 \cdot \beta^3$ + $a_2 \cdot a_0 \cdot \beta^2$
+ $a_{0 \ldots 2} \cdot a_2 \cdot \beta^2$
+ $a_2 \cdot a_2 \cdot \beta^4$
+ …
+ …
$a_{n-1} \cdot a_{n-1} \cdot \beta^{2 \cdot n-2}$ + … + $a_{n-1} \cdot a_2 \cdot \beta^{n+1}$ + $a_{n-1} \cdot a_1 \cdot \beta^n$ + $a_{n-1} \cdot a_0 \cdot \beta^{n-1}$
+ $a_{0 \ldots (n-1)} \cdot a_{n-1} \cdot \beta^{n-1}$
+ $a_{n-1} \cdot a_{n-1} \cdot \beta^{2 \cdot n-2}$
$res = 2 \cdot tri + dia$
tri
dia

Complexity. The squaring algorithm requires the following number of double-wide multiplications:

$$\frac{n \cdot (0 + n - 1)}{2} + n = \frac{n \cdot n - n + 2 \cdot n}{2} = \frac{n \cdot (n+1)}{2}.$$

Specifying squaring in F* (low-level specification). The F* specification for computing the first sum of the products $a_{0 \ldots i} \cdot a_i \cdot \beta^i$ is very similar to the F* specification for the bignum multiplication bn_mul_loop from §5.3.2. The difference is that the bn_mul1_lshift_add function is invoked with a sub-sequence of a starting from index 0, of length (i − 1), rather than the entire sequence a. The result is split into a bignum acc and a limb c, which is stored at index (i + i − 2) of the resulting bignum acc.

```
let rec bn_tri #t #len (a:bignum t len) (i:nat{i ≤ len}) : bignum t (len + len) =
  if i = 0 then create (len + len) (uint #t 0)
  else begin
    let acc1 = bn_tri a (i − 1) in
    let (c, acc) = bn_mul1_lshift_add (slice a 0 (i − 1)) a.[i − 1] (i − 1) acc1 in
    acc.[i + i − 2] ←c end
```

The F* specification for computing the second sum of the terms $a_i \cdot a_i \cdot \beta^{i+i}$ is represented by a total recursive function bn_dia that expects a bignum a of length len and index i and outputs a bignum of length (len + len).

The base case of the bn_dia function returns the initial value of the accumulator, a sequence of length (len + len) filled with zeros. The recursive case calls a double-wide multiplication

```
val mul_wide (#t:limb_t) (a b:limb t) : Pure (limb t & limb t) (requires ⊤)
  (ensures λ(hi, lo) → v lo + v hi ∗ pow2 (bits t) = v a ∗ v b)

let rec bn_dia #t #len (a:bignum t len) (i:nat{i ≤ len}) : bignum t (len + len) =
  if i = 0 then create (len + len) (uint #t 0)
  else begin
    let acc1 = bn_dia a (i − 1) in
    let (hi, lo) = mul_wide a.[i − 1] a.[i − 1] in
    let acc = acc1.[i + i − 2] ← lo in
    acc.[i + i − 1] ← hi end
```

function mul_wide on the (i—1)-th element of a. The result is split into two limbs, hi and lo, that are stored at indices (i + i − 2) and (i + i − 1) of a sequence acc1. The sequence acc1 results from recursively calling the function bn_dia on the input bignum a and index (i − 1).

Finally, we can specify the squaring function bn_sqr that doubles the result of the bn_tri function and adds it to the output of the bn_dia function.

```
let bn_sqr #t #len (a:bignum t len) : bignum t (len + len) =
  let tri = bn_tri a len in
  let dia = bn_dia a len in
  let (c0, res0) = bn_add tri tri in
  let (c1, res1) = bn_add res0 dia in
  res1
```

Note that both carry-out limbs, c0 and c1, are equal to zero, which we prove in the following subsection.

Verifying squaring in F* (from low-level to high-level specification). Now we prove that the bn_sqr function performs a squaring operation, i.e., the result's mathematical integer is equal to the input's mathematical integer squared.

```
val bn_sqr_lemma (#t:limb_t) (#len:nat) (a:bignum t len) :
  Lemma (bn_v (bn_sqr a) = bn_v a ∗ bn_v a)
```

The lemma bn_sqr_loop_lemma proves the above lemma by induction on i. We conclude our proof of the lemma bn_sqr_lemma by setting i to len and calling the lemma bn_add_lemma twice from §5.3.1.

```
val bn_sqr_loop_lemma (#t:limb_t) (#len:nat) (a:bignum t len) (i:nat{i ≤ len}) :
  Lemma (2 ∗ eval (bn_tri a i) (i + i) + eval (bn_dia a i) (i + i) = eval a i ∗ eval a i)
```

Our induction hypothesis is that the squaring of $a_{0...i}$ is computed after combining the results of calling the bn_tri and bn_dia functions on the input bignum a and index i in a certain way.

The proof of the base case is straightforward. The inductive case can be justified as follows.

$$
\begin{aligned}
&\mathsf{a}_{0\ldots i}\cdot\mathsf{a}_{0\ldots i}\\
&= \mathsf{a}_{0\ldots(i-1)}\cdot\mathsf{a}_{0\ldots(i-1)}\\
&\quad +2\cdot\mathsf{a}_{0\ldots(i-1)}\cdot\mathsf{a}.[i-1]\cdot 2^{\mathsf{bits\ t}\cdot(i-1)}\\
&\quad +\mathsf{a}.[i-1]\cdot\mathsf{a}.[i-1]\cdot 2^{\mathsf{bits\ t}\cdot(i+i-2)} && [\text{by the def. of } \mathsf{eval}]\\
&= 2\cdot(\mathsf{bn\_tri\ a\ (i-1)})_{0\ldots(i+i-2)}+(\mathsf{bn\_dia\ a\ (i-1)})_{0\ldots(i+i-2)}\\
&\quad +2\cdot\mathsf{a}_{0\ldots(i-1)}\cdot\mathsf{a}.[i-1]\cdot 2^{\mathsf{bits\ t}\cdot(i-1)}\\
&\quad +\mathsf{a}.[i-1]\cdot\mathsf{a}.[i-1]\cdot 2^{\mathsf{bits\ t}\cdot(i+i-2)} && [\text{by the induction hypothesis}]\\
&= 2\cdot(\mathsf{bn\_tri\ a\ (i-1)})_{0\ldots(i+i-2)}+(\mathsf{bn\_dia\ a\ i})_{0\ldots(i+i)}\\
&\quad +2\cdot\mathsf{a}_{0\ldots(i-1)}\cdot\mathsf{a}.[i-1]\cdot 2^{\mathsf{bits\ t}\cdot(i-1)} && [\text{by the def. of } \mathsf{bn\_dia}]\\
&= 2\cdot(\mathsf{bn\_tri\ a\ i})_{0\ldots(i+i)}+(\mathsf{bn\_dia\ a\ i})_{0\ldots(i+i)} && [\text{by the def. of } \mathsf{bn\_tri}]
\end{aligned}
$$

Knowing that $2\cdot\mathsf{tri}+\mathsf{dia}=\mathsf{a}\cdot\mathsf{a}$, we can show that both carry-out limbs, c0 and c1, from the bn_sqr function are equal to zero:

$$(\mathsf{c0}+\mathsf{c1})\cdot 2^{\mathsf{bits\ t}\cdot(len+len)}+\underbrace{\mathsf{res1}}_{<2^{\mathsf{bits\ t}\cdot(len+len)}}=\underbrace{2\cdot\mathsf{tri}+\mathsf{dia}}_{<2^{\mathsf{bits\ t}\cdot(len+len)}}\Rightarrow(\mathsf{c0}+\mathsf{c1})=0.$$

5.4.2 Karatsuba Multiplication

The Karatsuba multiplication [72, 73] is a divide-and-conquer algorithm that recursively breaks down the multiplication of two n-digit numbers, a and b, into three multiplications of two $\frac{n}{2}$-digit numbers. For simplicity, we assume that the number of digits n is *even*.

The algorithm uses the following representation for a and b:

$$a=a_1\cdot 10^p+a_0 \qquad\qquad b=b_1\cdot 10^p+b_0,$$

where $p=\frac{n}{2}$ and a_0, a_1, b_0 and b_1 are less than 10^p.

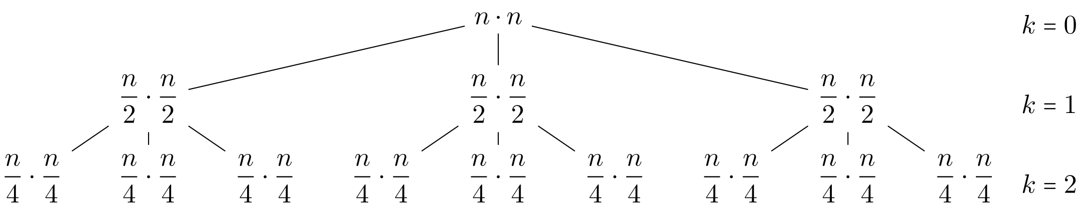
To compute the product of a and b with three multiplications, it expresses the coefficient $(a_1\cdot b_0+a_0\cdot b_1)$ in terms of three other products, including $a_1\cdot b_1$ and $a_0\cdot b_0$:

$$a\cdot b=(a_1\cdot 10^p+a_0)\cdot(b_1\cdot 10^p+b_0)=a_1\cdot b_1\cdot 10^{2\cdot p}+(a_1\cdot b_0+a_0\cdot b_1)\cdot 10^p+a_0\cdot b_0$$

$$
\begin{aligned}
a_1\cdot b_0+a_0\cdot b_1 &= a_1\cdot b_1+a_0\cdot b_0-(a_0-a_1)\cdot(b_0-b_1) && \textbf{(Subtractive variant)}\\
a_1\cdot b_0+a_0\cdot b_1 &= (a_0+a_1)\cdot(b_0+b_1)-a_1\cdot b_1-a_0\cdot b_0 && \textbf{(Additive variant)}
\end{aligned}
$$

These formulas are known as the subtractive and additive variants of the Karatsuba multiplication, respectively. The recursive case of the Karatsuba algorithm combines the results of the three recursive calls of the Karatsuba multiplication on two $\frac{n}{2}$-digit numbers, i.e., $a_0\cdot b_0$, $a_1\cdot b_1$, and $(a_0\pm a_1)\cdot(b_0\pm b_1)$, using one of the formulas above. The base case invokes, for example, a schoolbook multiplication.

Complexity. Without considering the cost of the addition operation, the number of double-wide multiplications required for the Karatsuba multiplication is shown in the following figure, where k corresponds to how many times we recursively apply the Karatsuba algorithm.


$n \cdot n$
$\frac{n}{2} \cdot \frac{n}{2}$
$\frac{n}{4} \cdot \frac{n}{4}$
$k = 0$
$k = 1$
$k = 2$

— $k = 0$: 1 schoolbook multiplication of two n-digit numbers

— $k = 1$: 3 schoolbook multiplications of two $\frac{n}{2}$-digit numbers

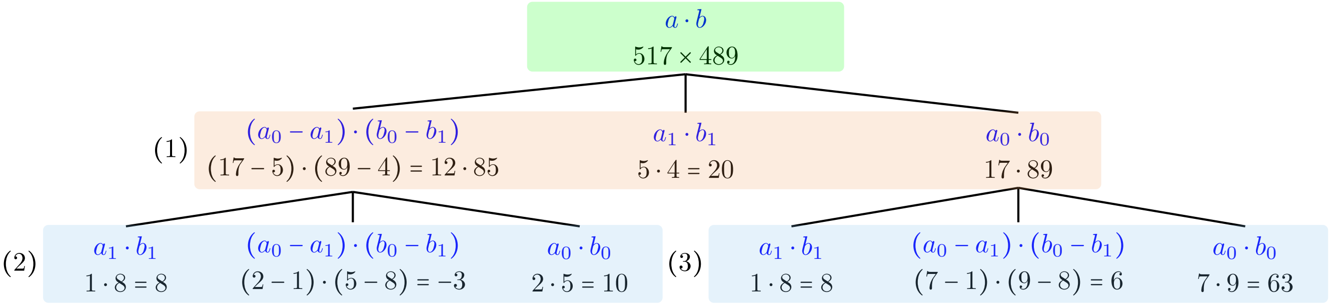
— $k = 2$: 9 schoolbook multiplications of two $\frac{n}{4}$-digit numbers

— k: 3^k schoolbook multiplications of two $\frac{n}{2^k}$-digit numbers

In total, the cost of the Karatsuba multiplication is $3^k \cdot \left(\frac{n}{2^k}\right)^2$ double-wide multiplications.

When the base case is a single-digit multiplication and $n = 2^k$, we can compute c such that $n^c = 3^k \cdot \left(\frac{n}{2^k}\right)^2$, i.e., $c = \log_2 3 \approx 1.58$.

Subtractive variant of the Karatsuba multiplication. Let us now multiply two multi-digit numbers, 517 and 489, using the subtractive variant of the Karatsuba multiplication. The base case is a single-digit multiplication.



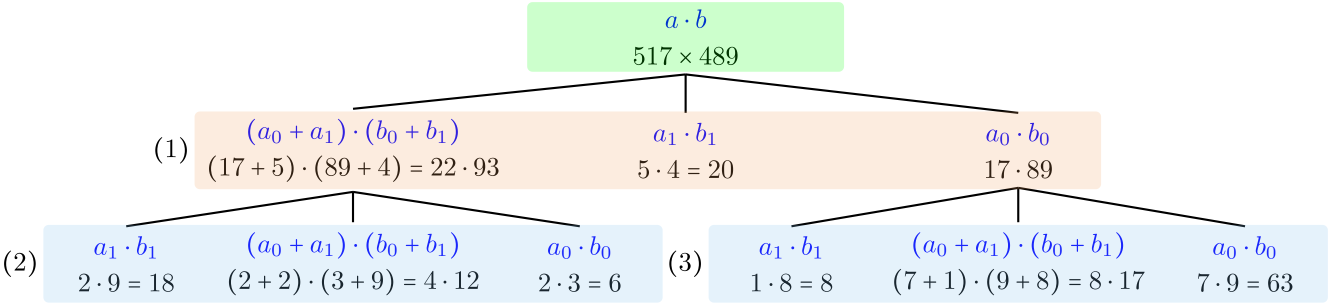
We apply the subtractive formula for $p = 2$ and $p = 1$ as follows:

$$\begin{array}{lllllllll}
 & a = a_1 \cdot 10^p + a_0 & b = b_1 \cdot 10^p + b_0 & & & & & \\
(1): & a = 5 \cdot 10^2 + 17 & b = 4 \cdot 10^2 + 89 & [p = 2, & a_1 = 5, & a_0 = 17, & b_1 = 4, & b_0 = 89] \\
(2): & a = 1 \cdot 10 + 2 & b = 8 \cdot 10 + 5 & [p = 1, & a_1 = 1, & a_0 = 2, & b_1 = 8, & b_0 = 5] \\
(3): & a = 1 \cdot 10 + 7 & b = 8 \cdot 10 + 9 & [p = 1, & a_1 = 1, & a_0 = 7, & b_1 = 8, & b_0 = 9]
\end{array}$$

$$\begin{aligned}
& a \cdot b = a_1 \cdot b_1 \cdot 10^{2 \cdot p} + (a_1 \cdot b_1 + a_0 \cdot b_0 - (a_0 - a_1) \cdot (b_0 - b_1)) \cdot 10^p + a_0 \cdot b_0 \\
& (3) : a \cdot b = 8 \cdot 10^2 + (8 + 63 - 6) \cdot 10 + 63 = 1513 \\
& (2) : a \cdot b = 8 \cdot 10^2 + (8 + 10 + 3) \cdot 10 + 10 = 1020 \\
& (1) : a \cdot b = 20 \cdot 10^4 + (20 + 1513 - 1020) \cdot 10^2 + 1513 = 252813
\end{aligned}$$

As we can see, the product of $(a_0 - a_1)$ and $(b_0 - b_1)$ may be a negative number. To avoid this situation, we can compute the absolute values for $|a_0 - a_1|$ and $|b_0 - b_1|$ and keep their sign aside. If both expressions have the same sign, we subtract $|a_0 - a_1| \cdot |b_0 - b_1|$ from the sum of $a_1 \cdot b_1$ and $a_0 \cdot b_0$ and add them together otherwise.

Additive variant of the Karatsuba multiplication. Let us carry out the multiplication for the same example but using the additive variant of the Karatsuba multiplication.



We apply the additive formula for $p = 2$ and $p = 1$ as follows:

$$\begin{array}{llllllll}
 & a = a_1 \cdot 10^p + a_0 & b = b_1 \cdot 10^p + b_0 & & & & & \\
(1): & a = 5 \cdot 10^2 + 17 & b = 4 \cdot 10^2 + 89 & [p = 2, & a_1 = 5, & a_0 = 17, & b_1 = 4, & b_0 = 89] \\
(2): & a = 2 \cdot 10 + 2 & b = 9 \cdot 10 + 3 & [p = 1, & a_1 = 2, & a_0 = 2, & b_1 = 9, & b_0 = 3] \\
(3): & a = 1 \cdot 10 + 7 & b = 8 \cdot 10 + 9 & [p = 1, & a_1 = 1, & a_0 = 7, & b_1 = 8, & b_0 = 9]
\end{array}$$

$$\begin{aligned}
& a \cdot b = a_1 \cdot b_1 \cdot 10^{2 \cdot p} + ((a_0 + a_1) \cdot (b_0 + b_1) - a_1 \cdot b_1 - a_0 \cdot b_0) \cdot 10^p + a_0 \cdot b_0 \\
& (3) : a \cdot b = 8 \cdot 10^2 + (8 \cdot 17 - 8 - 63) \cdot 10 + 63 = 1513 \\
& (2) : a \cdot b = 18 \cdot 10^2 + (4 \cdot 12 - 18 - 6) \cdot 10 + 6 = 2046 \\
& (1) : a \cdot b = 20 \cdot 10^4 + (2046 - 20 - 1513) \cdot 10^2 + 1513 = 252813
\end{aligned}$$

Here we can see that the result of $(a_0 + a_1)$ and $(b_0 + b_1)$ may overflow, i.e., we need to deal with carries when multiplying $(a_0 + a_1)$ by $(b_0 + b_1)$. Hence, the subtractive variant of the Karatsuba multiplication is more straightforward to implement in practice.

Karatsuba squaring. We can also compute the so-called Karatsuba squaring for an n-digit number a by combining the results of the three recursive calls of the Karatsuba squaring on an $\frac{n}{2}$-digit number, i.e., $a_0 \cdot a_0$, $a_1 \cdot a_1$, and $(a_0 \pm a_1) \cdot (a_0 \pm a_1)$. The base case invokes, for example, a squaring function from §5.3.1.

$$a \cdot a = (a_1 \cdot 10^p + a_0) \cdot (a_1 \cdot 10^p + a_0) = a_1 \cdot a_1 \cdot 10^{2 \cdot p} + (a_1 \cdot a_0 + a_0 \cdot a_1) \cdot 10^p + a_0 \cdot a_0$$

$$\begin{aligned}
a_1 \cdot a_0 + a_0 \cdot a_1 &= a_1 \cdot a_1 + a_0 \cdot a_0 - (a_0 - a_1) \cdot (a_0 - a_1) \quad && \textbf{(Subtractive variant)} \\
a_1 \cdot a_0 + a_0 \cdot a_1 &= (a_0 + a_1) \cdot (a_0 + a_1) - a_1 \cdot a_1 - a_0 \cdot a_0 \quad && \textbf{(Additive variant)}
\end{aligned}$$

Note that the above applies to any bignum representation in radix-β.

Specifying and verifying the Karatsuba multiplication in F* (low-level specification). We specify and verify the Karatsuba multiplication using a total recursive function bn_karatsuba_mul that takes two bignums, a and b, of length len and returns a bignum res of length (len + len), the result of their product. We explicitly write the decreases clause with a bignum length len as a parameter, that strictly decreases on each recursive call, to help the F* type-checker prove the termination of our recursive function.

```
let rec bn_karatsuba_mul #t #len (a b:bignum t len) :
  Tot (res:bignum t (len + len){bn_v res = bn_v a ∗ bn_v b}) (decreases len) =
  if len < bn_mul_threshold || len % 2 = 1 then bn_mul a b
  else begin
    let (a1, a0) = bn_halves a in let (s0, t0) = bn_sign_abs a0 a1 in
    let (b1, b0) = bn_halves b in let (s1, t1) = bn_sign_abs b0 b1 in

    let t23 = bn_karatsuba_mul t0 t1 in
    let r01 = bn_karatsuba_mul a0 b0 in
    let r23 = bn_karatsuba_mul a1 b1 in

    let (c5, t45) = bn_middle_karatsuba s0 s1 r01 r23 t23 in
    let (c, res) = bn_karatsuba_res r01 r23 t45 c5 in
    res end
```

The base case of the bn_karatsuba_mul function is when the bignum length len is either less than the threshold value bn_mul_threshold or odd. The parameter value bn_mul_threshold depends on the schoolbook multiplication and addition cost, making it difficult to choose a default value. By benchmarking the code with different values for bn_mul_threshold, we set this parameter to 32. By calling the lemma bn_mul_lemma from §5.3.2, we prove that the result is the product of the input bignums, a and b.

The recursive case first calls the bn_halves function on a bignum a, which returns a tuple of two bignums, a1 and a0, with length (len / 2), such that the following postcondition holds.

```
val bn_halves (#t:limb_t) (#len:nat) (a:bignum t len) :
  Pure (tuple2 (bignum t (len / 2)) (bignum t (len / 2))) (requires len % 2 = 0)
  (ensures λ (a1, a0) → let pp = pow2 (len / 2 ∗ bits t) in
    bn_v a0 = bn_v a % pp ∧ bn_v a1 = bn_v a / pp ∧
    bn_v a0 < pp ∧ bn_v a1 < pp ∧ bn_v a = bn_v a1 ∗ pp + bn_v a0)
```

The a0 and a1 bignums result from calling the (slice a i1 i2) function, which returns a subsequence of a starting from index i1, with length (i2 − i1). Since we use a little-endian representation for our bignums, a1 = slice a (len / 2) len and a0 = slice a 0 (len / 2).

Then, it calls the bn_sign_abs function on two bignums, a0 and a1, which returns a tuple containing a sign s0 and a bignum t0 of length (len / 2). The bignum t0 reflects the absolute value for (a0 – a1). The sign s0 is equal to one if the value is a negative number and zero otherwise. Note that an implementation of this function must compute both cases, i.e., (a0 – a1) and (a1 – a0), and then select the result in a constant-time way.

```
let abs (a:nat) (b:nat) : nat = if a < b then b − a else a − b

val bn_sign_abs (#t:limb_t) (#len:nat) (a b:bignum t len) :
  Pure (tuple2 (carry t) (bignum t len)) (requires ⊤)
  (ensures λ (sign, res) → bn_v res = abs (bn_v a) (bn_v b) ∧
    v sign = (if bn_v a < bn_v b then 1 else 0))
```

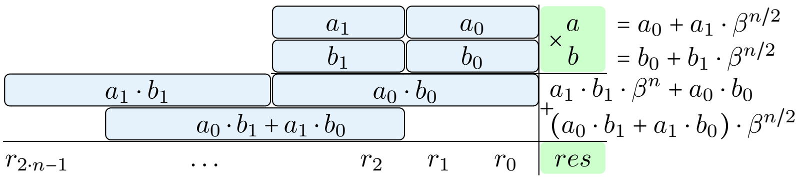

a_1
a_0
a
×
b
$= a_0 + a_1 \cdot \beta^{n/2}$
b_1
b_0
$= b_0 + b_1 \cdot \beta^{n/2}$
$a_1 \cdot b_1$
$a_0 \cdot b_0$
$a_1 \cdot b_1 \cdot \beta^n + a_0 \cdot b_0$
$a_0 \cdot b_1 + a_1 \cdot b_0$
+
$(a_0 \cdot b_1 + a_1 \cdot b_0) \cdot \beta^{n/2}$
$r_{2 \cdot n-1}$
$\dots$
r_2
r_1
r_0
res

Figure 5.8 – Karatsuba multiplication.

The same is done for a bignum b. Next, it recursively calls the bn_karatsuba_mul function three times to compute the following products: (t0 · t1), (a0 · b0), (a1 · b1). Finally, it computes the results according to the subtractive formula of the Karatsuba multiplication. The bn_middle_karatsuba function computes the middle coefficient of the formula, (a0 · b1 + a1 · b0).

```
val bn_middle_karatsuba (#t:limb_t) (#len:nat) (s0 s1:carry t) (r01 r23 t23:bignum t len) :
  Pure (tuple2 (limb t) (bignum t len)) (requires ⊤)
  (ensures λ(c, res) →
    let c0, k0 = bn_add r01 r23 in
    let c1, k1 = bn_sub k0 t23 in let c1’ = c0 −. c1 in
    let c2, k2 = bn_add k0 t23 in let c2’ = c0 +. c2 in

    if v s0 = v s1 then v c = v c1’ ∧ bn_v res = bn_v k1 else v c = v c2’ ∧ bn_v res = bn_v k2)
```

As the result of (a0 · b1 + a1 · b0) may overflow, the bn_middle_karatsuba function returns a carry and a bignum of length len. Note that an implementation of this function must also compute both cases, i.e., when the product of (a0 − a1) · (b0 − b1) is a negative number and when not, and then select the result in a constant-time way.

The bn_karatsuba_res function computes the final sum, as shown in Figure 5.8. It first stores a bignum r01 (= a0 · b0) followed by a bignum r23 (= a1 · b1) and then adds a bignum t45 (= (a0 · b1 + a1 · b0) % $2^{\text{bits t}\cdot\text{len}}$) to the result, starting from index (len / 2). We compute the addition for len elements only and then propagate the carry from this operation and the input carry c5 (= (a0 · b1 + a1 · b0) / $2^{\text{bits t}\cdot\text{len}}$) towards the most significant limb. Note that the final carry c is equal to zero as we compute the product of a and b, which is less than $2^{\text{bits t}\cdot(\text{len}+\text{len})}$.

```
val bn_karatsuba_res (#t:limb_t) (#len:nat) (r01 r23 t45:bignum t len) (c5:carry t) :
  Pure (tuple2 (carry t) (bignum t (len + len))) (requires ⊤)
  (ensures λ(c, res) → let pp = pow2 (len / 2 ∗ bits t) in
    bn_v res + v c ∗ pow2 (bits t ∗ (len + len)) =
    bn_v r23 ∗ (pp ∗ pp) + (v c5 ∗ pow2 (bits t ∗ len) + bn_v t45) ∗ pp + bn_v r01)
```

5.5 Low* API for Modular Arithmetic

In this section, we present an API for basic modular arithmetic functions: addition, subtraction, and multiplication. We assume that all computations are performed for a fixed modulus n, a positive number, with no special form. Each function takes as input two non-negative numbers, a and b, that are less than n and outputs a non-negative number c that is also less than n.

Modular reduction for addition and subtraction is simple, as we need to carry out either a conditional addition or a conditional subtraction.

```
let nat_mod n = a:nat{a < n}

let mod_add (n:pos) (a b:nat_mod n) : c:nat_mod n{c = (a + b) % n} =
  let res = a + b in if n ≤ res then res − n else res

let mod_sub (n:pos) (a b:nat_mod n) : c:nat_mod n{c = (a − b) % n} =
  let res = a − b in if res < 0 then res + n else res
```

Modular multiplication can be computed as a remainder of the Euclidean division of a non-negative number $a \cdot b$ by n:

$$a \cdot b \ \% \ n = a \cdot b - \frac{a \cdot b}{n} \cdot n \ \wedge \ 0 \leq a \cdot b \ \% \ n < n.$$

In order to avoid an operation as expensive as the division of $a \cdot b$ by n, we can use another way of representing a residue, the so-called Montgomery form [88, 37]. It encodes a and b as $a \cdot r \ \% \ n$ and $b \cdot r \ \% \ n$, respectively, where a constant $r > n$ is coprime to n, i.e., $gcd\ (r, n) = 1$, which guarantees that a modular multiplicative inverse of r, $r^{-1} \ \% \ n$, exists. The constant r is chosen in such a way that a division by r is fast. *In other words, we replace the expensive division by n with the fast division by r.* For example, for $\beta = 2^{32}$ or $\beta = 2^{64}$, we can compute a modular exponentiation $a^b \ \% \ n$ in Montgomery form for an odd modulus n by setting r to a power of 2 so that a division by r is a right-shift operation.

Montgomery arithmetic. Let us consider Montgomery arithmetic, where all computations are performed in Montgomery form to reduce the overhead of converting integers to and from Montgomery representation. Modular addition and subtraction when using the Montgomery form remain unchanged:

$$\begin{aligned}(a \cdot r \ \% \ n + (b \cdot r \ \% \ n)) \ \% \ n &= (a + b) \cdot r \ \% \ n \\ (a \cdot r \ \% \ n - (b \cdot r \ \% \ n)) \ \% \ n &= (a - b) \cdot r \ \% \ n.\end{aligned}$$

However, modular multiplication in Montgomery form requires removing the extra factor r:

$$(a \cdot r \ \% \ n \cdot (b \cdot r \ \% \ n)) \cdot r^{-1} \ \% \ n = (a \cdot b) \cdot r \ \% \ n.$$

Modular multiplication by r^{-1} is called Montgomery reduction, redc, which can also be used for converting an integer a to and from Montgomery form. For the former, a constant $(r^2 \ \% \ n)$ can be precomputed for a fixed modulus n.

$$\begin{aligned}&\text{to\_mont: } a \cdot (r^2 \ \% \ n) \cdot r^{-1} \ \% \ n = a \cdot r \ \% \ n \\ &\text{from\_mont: } (a \cdot r \ \% \ n) \cdot r^{-1} \ \% \ n = a \ \% \ n = a\end{aligned}$$

In the following, we use the notation $aM (= a \cdot r \ \% \ n)$ for the Montgomery form of an integer a and continue to employ a for the regular representation.

Functional correctness for Montgomery arithmetic. The figure below depicts *functional* *correctness* for Montgomery arithmetic. The result of invoking an operation op with two numbers, a and b, is the same as first converting them to their Montgomery form, aM and bM, and then transforming the result of the mont_op call on aM and bM back to the regular representation, c.

$$\begin{tikzcd}
aM, bM \arrow[r, "\text{mont\_op}"] & cM \arrow[d, "\text{from\_mont}"] \\
a, b \arrow[u, "\text{to\_mont}"] \arrow[r, "\text{op}"'] & c
\end{tikzcd}$$

op	mont_op
$(a+b)\ \%\ n$	$(aM+bM)\ \%\ n$
$(a-b)\ \%\ n$	$(aM-bM)\ \%\ n$
$a \cdot b\ \%\ n$	redc $(aM \cdot bM)$

Montgomery arithmetic in F* (high-level specification). We use a high-level specification for Montgomery arithmetic to prove some properties that are independent of implementation details. We start with a signature for Montgomery reduction that takes as input a modulus n, a constant r, a modular multiplicative inverse d of r, and a number $c < n \cdot n$. The function redc returns a number k which is less than n and equal to $c \cdot d\ \%\ n$.

```
val redc (n r:pos) (d:int{r ∗ d % n = 1}) (c:nat{c < n ∗ n}) : k:nat_mod n{k = c ∗ d % n}
```

The F* specification for converting a non-negative integer $a < n$ to and from Montgomery form is simple and is presented below.

```
let to_mont (n r:pos) (d:int{r ∗ d % n = 1}) (a:nat_mod n) : aM:nat_mod n{aM = a ∗ r % n} =
  redc n r d (a ∗ (r ∗ r % n))

let from_mont (n r:pos) (d:int{r ∗ d % n = 1}) (aM:nat_mod n) : a:nat_mod n{a = aM ∗ d % n} =
  redc n r d aM
```

It is easy to see that for an integer $a < n$, the result of converting its Montgomery form back to the regular representation is the same number a:

```
val lemma_mont_id (n r:pos) (d:int{r ∗ d % n = 1}) (a:nat_mod n) :
  Lemma (from_mont n r d (to_mont n r d a) = a)
```

The lemma above allows us to prove functional correctness for Montgomery arithmetic without invoking the function to_mont since we can uniquely identify numbers a and b by calling a function from_mont on inputs aM and bM, respectively.

```
val mont_add_lemma (n r:pos) (d:int{r ∗ d % n = 1}) (aM bM:nat_mod n) :
  Lemma (from_mont n r d ((aM + bM) % n) = (from_mont n r d aM + from_mont n r d bM) % n)

val mont_sub_lemma (n r:pos) (d:int{r ∗ d % n = 1}) (aM bM:nat_mod n) :
  Lemma (from_mont n r d ((aM − bM) % n) = (from_mont n r d aM − from_mont n r d bM) % n)

val mont_mul_lemma (n r:pos) (d:int{r ∗ d % n = 1}) (aM bM:nat_mod n) :
  Lemma (from_mont n r d (redc n r d (aM ∗ bM)) = (from_mont n r d aM ∗ from_mont n r d bM) % n)
```

Low* API for Montgomery arithmetic. As we have seen, we can save some calculations by storing the precomputed Montgomery constants for a fixed modulus n, e.g., $r^2\ \%\ n$. So we can define a structure that holds a bignum length len, a modulus n and a constant r2 as follows.

```
type bn_mont_ctx (t:limb_t) = {
  len: bn_len t;
  n: lbignum t len;
  r2: lbignum t len;
  ...(∗ some other precomputed constants ∗)}
```

We do not provide all the details about elements of type bn_mont_ctx, as we can treat them as opaque values containing at least a bignum length *len* and a modulus n:

```
let pbn_mont_ctx (t:limb_t) = pointer (bn_mont_ctx t)
let bn_mont_n (#t:limb_t) (h:mem) (k:pbn_mont_ctx t) = lbn_v h (deref h k).n
let bn_mont_len (#t:limb_t) (h:mem) (k:pbn_mont_ctx t) = (deref h k).len
```

The type pbn_mont_ctx defines a pointer to a bn_mont_ctx structure. The ghost total function (deref h k) is the dereference operator that returns the value of the variable pointed to this pointer k in a given memory h. A dot . is an access operator to a member of the variable structure.

Creating a Montgomery context. The function bn_mont_init stack-allocates and initializes a bn_mont_ctx structure. It takes as input a bignum n of length len and outputs a pointer to the structure.

```
let bn_mont_ctx_pre (#t:limb_t) (#len:bn_len t) (h:mem) (n:lbignum t len) =
  1 < lbn_v h n ∧ lbn_v h n % 2 = 1

let pbn_mont_ctx_inv (#t:limb_t) (h:mem) (k:pbn_mont_ctx t) =
  let k1 = deref h k in
  live h k ∧ live h k1.n ∧ live h k1.r2 ∧
  disjoint k k1.n ∧ disjoint k k1.r2 ∧ disjoint k1.n k1.r2 ∧
  bn_mont_ctx_pre h k1.n ∧
  lbn_v h k1.r2 = pow2 (2 ∗ bits t ∗ v k1.len) % lbn_v h k1.n ∧
  ...(∗ some other precomputed constants ∗)

inline_for_extraction noextract
val bn_mont_init (#t:limb_t) (len:bn_len t) (n:lbignum t len) : StackInline (pbn_mont_ctx t)
  (requires λ h → live h n ∧ bn_mont_ctx_pre h n)
  (ensures λ h0 k h1 → modifies0 h0 h1 ∧
    fresh_loc (loc k |+| (loc (deref h1 k).n |+| loc (deref h1 k).r2)) h0 h1 ∧
    pbn_mont_ctx_inv h1 k ∧ bn_mont_len h1 k = len ∧ bn_mont_n h1 k = lbn_v h0 n)
```

In addition to liveness, the precondition requires n to be an odd number greater than one. In this case, a modular multiplicative inverse of $r = 2^{\text{bits t}\cdot\text{v len}}$ exists.

The postcondition ensures that the following properties are fulfilled:

— none of the existing memory locations were modified (modifies0 h0 h1)

— locations for a pointer k and its fields, n and r2, are fresh

— liveness and disjointness of a pointer k and of the arrays in its fields n and r2

— the value of the field r2 of the dereferenced result is $r^2 \% n$

— the value of the field n of the dereferenced result equals the function parameter n

— the value of the field len of the dereferenced result equals the function parameter len.

The bn_mont_init function is annotated with the Low* StackInline effect meaning that the function can be inlined in a function body that has either the Low* Stack effect or Heap effect.

Converting a bignum to and from Montgomery form. Figure 5.9 depicts our Low* API for converting a bignum to and from Montgomery form. The functions bn_to_mont and bn_from_mont take as input a pointer k to a bn_mont_ctx structure and two bignums, a and aM. The precondition requires the liveness and disjointness of a, aM, k and its fields, n and r2. The length of a and aM must be equal to the length of a modulus n. The predicate pbn_mont_ctx_inv must be fulfilled for the pointer k.

The ghost total function bn_mont_v is a composition of the lbn_v and from_mont functions. It takes a bignum $\mathsf{aM} < \mathsf{n}$ in Montgomery form and returns the mathematical integer $\mathsf{a} < \mathsf{n}$ in the regular representation. We use this function to specify the semantics of Montgomery arithmetic in Low*.

The function bn_to_mont takes a bignum $\mathsf{a} < \mathsf{n}$ and converts it to its Montgomery form $\mathsf{aM} < \mathsf{n}$, while the function bn_from_mont transforms it back.

Montgomery addition, subtraction, multiplication, and squaring. Each function in our Low* API for Montgomery arithmetic has the same signature, except for functional correctness.

For example, the signature for Montgomery multiplication is as follows.

```
val bn_mont_mul (#t:limb_t) (k:pbn_mont_ctx t) (aM bM cM:buffer (limb t)) : Stack unit
  (requires λ h → pbn_mont_ctx_inv h k ∧
    bn_lt_n_footprint h k aM ∧ bn_lt_n_footprint h k bM ∧ bn_footprint h k cM ∧
    eq_or_disjoint aM bM ∧ eq_or_disjoint aM cM ∧ eq_or_disjoint bM cM)
  (ensures λ h0 _ h1 → modifies (loc cM) h0 h1 ∧ bn_lt_n_footprint h1 k cM ∧
   (* Functional correctness of Montgomery multiplication *)
    bn_mont_v h1 k cM = (bn_mont_v h0 k aM ∗ bn_mont_v h0 k bM) % bn_mont_n h0 k)
```

All the operations are performed in the Montgomery domain and the bn_mont_ctx structure remains unchanged throughout these computations.

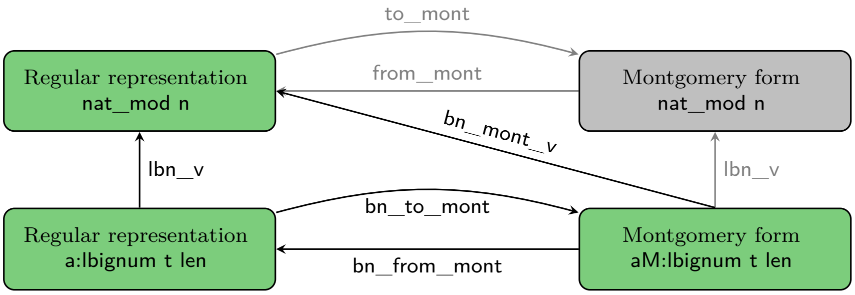
5.6 Verifying Modular Arithmetic

5.6.1 Modular Addition and Subtraction

As we discussed in §5.5, a high-level specification for modular reduction for addition and subtraction is straightforward, but it needs to perform either a conditional addition or a conditional subtraction. For the code, we need to compute both operations and select the result in a constant-time way.

Constant-time selection. A bn_mask_select function takes two bignums, a and b, of length len and a mask m. It returns a bignum b if $\mathsf{m} = 0$ and a bignum a if $\mathsf{m} = \mathsf{ones\_v\ t}$.

```
let bn_mask_select #t #len (m:limb t) (a b:bignum t len) : Pure (bignum t len) (requires ⊤)
  (ensures λ res → (v m = 0 ⟹ res == b) ∧ (v m = ones_v t ⟹ res == a)) =
  map2 len (mask_select m) a b
```



```
let bn_footprint (#t:limb_t) (h:mem) (k:pbn_mont_ctx t) (a:buffer (limb t)) =
  let k1 = deref h k in length a = v (bn_mont_len h k) ∧
  live h a ∧ disjoint a k ∧ disjoint a k1.n ∧ disjoint a k1.r2

let bn_lt_n_footprint (#t:limb_t) (h:mem) (k:pbn_mont_ctx t) (a:buffer (limb t)) =
  bn_footprint h k a ∧ lbn_v h a < bn_mont_n h k

val bn_mont_v (#t:limb_t) (h:mem) (k:pbn_mont_ctx t{pbn_mont_ctx_inv h k})
  (aM:buffer (limb t){bn_lt_n_footprint h k aM}) : GTot (a:nat{a < bn_mont_n h k})

val bn_to_mont (#t:limb_t) (k:pbn_mont_ctx t) (a aM:buffer (limb t)) : Stack unit
  (requires λ h → pbn_mont_ctx_inv h k ∧
    bn_lt_n_footprint h k a ∧ bn_footprint h k aM ∧ disjoint a aM)
  (ensures λ h0 _ h1 → modifies (loc aM) h0 h1 ∧ bn_lt_n_footprint h1 k aM ∧
    bn_mont_v h1 k aM = lbn_v h0 a)

val bn_from_mont (#t:limb_t) (k:pbn_mont_ctx t) (aM a:buffer (limb t)) : Stack unit
  (requires λ h → pbn_mont_ctx_inv h k ∧
    bn_lt_n_footprint h k aM ∧ bn_footprint h k a ∧ disjoint a aM)
  (ensures λ h0 _ h1 → modifies (loc a) h0 h1 ∧ bn_lt_n_footprint h1 k a ∧
    lbn_v h1 a = bn_mont_v h0 k aM)
```

Figure 5.9 – Low* API for conversion functions between Montgomery and regular representations.

The bn_mask_select function invokes a map2 function that constructs a resulting bignum by calling a mask_select function at each iteration i on the mask m and the i-th elements of a and b. The mask_select function can be implemented in various ways, for example:

```
val mask_select (#t:limb_t) (m a b:limb t) : Pure (limb t) (requires ⊤)
  (ensures λc → (v m = 0 ⟹ c == b) ∧ (v m = ones_v t ⟹ c == a))
let mask_select #t (m a b:limb t) = logxor b (logand m (logxor a b))
(* or *)
let mask_select #t (m a b:limb t) = logor (logand m a) (logand (lognot m) b)
```

Specifying and verifying a modular addition in F* (low-level specification). Let us recall a high-level specification for an addition modulo n that takes as input two numbers $a < n$ and $b < n$ and returns a number $c < n$ such that $c = (a + b) \% n$.

```
let mod_add (n:pos) (a b:nat_mod n) : c:nat_mod n{c = (a + b) % n} =
  let res = a + b in if n ≤ res then res − n else res
```

Looking at the specification, we define a bn_mod_add function that expects bignums n, a, b of length len and returns a bignum of the same length len. As expected, it computes both addition and subtraction and selects the result according to the mask m. The lemma bn_mod_add_lemma proves that the mask is chosen correctly.

```
let bn_mod_add #t #len (n a b:bignum t len) : bignum t len =
  let c0, r0 = bn_add a b in
  let c1, r1 = bn_sub r0 n in
  let m = c0 −. c1 in
  bn_mask_select m r0 r1

val bn_mod_add_lemma (#t:limb_t) (#len:nat) (n a b:bignum t len) : Lemma
  (requires bn_v a < bn_v n ∧ bn_v b < bn_v n)
  (ensures bn_v (bn_mod_add n a b) = (bn_v a + bn_v b) % bn_v n)
```

The proof of the lemma bn_mod_add_lemma is derived from the following two equalities:

$$\mathsf{r0} + \mathsf{c0} \cdot 2^{\mathsf{bits\ t \cdot len}} = \mathsf{a} + \mathsf{b} \quad \wedge \quad \mathsf{r1} + (\mathsf{c0} - \mathsf{c1}) \cdot 2^{\mathsf{bits\ t \cdot len}} = \mathsf{a} + \mathsf{b} - \mathsf{n}.$$

Case 1. If $\mathsf{a} + \mathsf{b} < \mathsf{n}$ then $\mathsf{c0} = 0 \ \wedge\ \mathsf{c1} = 1 \ \Rightarrow\ \mathsf{m} = \mathsf{ones\_v\ t} \ \wedge\ \mathsf{res} = \mathsf{r0}$.

Case 2. If $\mathsf{a} + \mathsf{b} \geq \mathsf{n}$ then

$$\underbrace{\mathsf{r1}}_{<2^{\mathsf{bits\ t \cdot len}}} + (\mathsf{c0} - \mathsf{c1}) \cdot 2^{\mathsf{bits\ t \cdot len}} = \underbrace{\mathsf{a} + \mathsf{b} - \mathsf{n}}_{<2^{\mathsf{bits\ t \cdot len}}} \ \Rightarrow\ \mathsf{c0} - \mathsf{c1} = 0 \ \Rightarrow\ \mathsf{m} = 0 \ \wedge\ \mathsf{res} = \mathsf{r1}.$$

Specifying and verifying a modular subtraction in F* (low-level specification). Similar to the modular addition, a bn_mod_sub function takes as input bignums n, a, b of length len and yields a bignum res of the same length len. If $a < n$ and $b < n$ then $res = (a - b) \% n$, as proven in the lemma bn_mod_sub_lemma.

```
let bn_mod_sub #t #len (n a b:bignum t len) : bignum t len =
  let c0, r0 = bn_sub a b in
  let c1, r1 = bn_add r0 n in
  let m = uint #t 0 −. c0 in
  bn_mask_select m r1 r0
```

```
val bn_mod_sub_lemma (#t:limb_t) (#len:nat) (n a b:bignum t len) : Lemma
  (requires bn_v a < bn_v n ∧ bn_v b < bn_v n)
  (ensures bn_v (bn_mod_sub n a b) = (bn_v a − bn_v b) % bn_v n)
```

The proof of the lemma bn_mod_sub_lemma relies on the following two equalities:

$$\mathsf{r0} - \mathsf{c0} \cdot 2^{\mathsf{bits\ t \cdot len}} = \mathsf{a} - \mathsf{b} \quad \wedge \quad \mathsf{r1} + (\mathsf{c1} - \mathsf{c0}) \cdot 2^{\mathsf{bits\ t \cdot len}} = \mathsf{a} - \mathsf{b} + \mathsf{n}.$$

Case 1. If $0 \le \mathsf{a} - \mathsf{b}$ then $\mathsf{c0} = 0 \;\Rightarrow\; \mathsf{m} = 0 \;\wedge\; \mathsf{res} = \mathsf{r0}$.

Case 2. If $\mathsf{a} - \mathsf{b} < 0$ then $\mathsf{c0} = 1 \;\Rightarrow\; \mathsf{m} = \mathsf{ones\_v\ t}$

$$\underbrace{\mathsf{r1}}_{<2^{\mathsf{bits\ t \cdot len}}} + (\mathsf{c1} - \mathsf{c0}) \cdot 2^{\mathsf{bits\ t \cdot len}} = \underbrace{\mathsf{a} - \mathsf{b} + \mathsf{n}}_{<2^{\mathsf{bits\ t \cdot len}}} \;\Rightarrow\; \mathsf{c1} - \mathsf{c0} = 0 \;\Rightarrow\; \mathsf{res} = \mathsf{r1}.$$

5.6.2 Montgomery Multiplication

Montgomery reduction replaces a costly division operation by n with a division by $r > n$, which is coprime to n. Usually, r is chosen as β^l, where l is the number of digits of n. The algorithm computes $a \cdot r^{-1} \ \% \ n$ for a $2l$-digit number $a < r \cdot n$. It first makes the number a divisible by r by adding a multiple of n. Then, it divides $a + n \cdot x$ by r, with the result being so close to n that only a final conditional subtraction is needed. The number x is chosen as $(a \ \% \ r) \cdot n' \ \% \ r$ where $(1 + n \cdot n') \ \% \ r = 0$ and the following is fulfilled.

$$(a + n \cdot x) \ \% \ r =$$
$$(a + n \cdot ((a \ \% \ r) \cdot n' \ \% \ r)) \ \% \ r = (a + n \cdot (a \cdot n')) \ \% \ r = (a \cdot (1 + n \cdot n')) \ \% \ r = 0$$

$$(a + n \cdot x) \ / \ r =$$
$$(a + n \cdot ((a \ \% \ r) \cdot n' \ \% \ r)) \ / \ r \le (r \cdot n - 1 + n \cdot (r - 1)) \ / \ r < (2 \cdot r \cdot n) \ / \ r = 2 \cdot n$$

$$\begin{array}{c|cccccccccc|l}
 & a_{2 \cdot l-1} & \dots & a_{l+2} & a_{l+1} & a_l & a_{l-1} & \dots & a_2 & a_1 & a_0 & a \\
 & \multicolumn{10}{r|}{n \cdot ((a \ \% \ \beta^l) \cdot n' \ \% \ \beta^l)} & + \; n \cdot ((a \ \% \ \beta^l) \cdot n' \ \% \ \beta^l) \\
\hline
c & b_{2 \cdot l-1} & \dots & b_{l+2} & b_{l+1} & b_l & 0 & \dots & 0 & 0 & 0 & b
\end{array}$$

Given a multiplicative inverse of r, d such that $r \cdot d \ \% \ n = 1$, we can check that the above conforms to our signature for Montgomery reduction redc from §5.5.

$$(a + n \cdot x) \ / \ r \ \% \ n = (a + n \cdot x) \ / \ r \cdot (r \cdot d) \ \% \ n = (a + n \cdot x) \cdot d \ \% \ n = a \cdot d \ \% \ n$$

Example. Let us now compute $517 \cdot 489$ modulo 531 using Montgomery reduction. It could be done by calling to_mont on the result of a redc invocation with $517 \cdot 489$. The final value is equal to $517 \cdot 489 \ \% \ 531$ as follows from the lemma lemma_mont_mod.

```
(∗ to_mont n r d (redc n r d a) = (a ∗ d % n) ∗ r % n = a % n ∗)
val lemma_mont_mod (n r:pos) (d:int{r ∗ d % n = 1}) (a:nat{a < n ∗ n}) :
  Lemma (to_mont n r d (redc n r d a) = a % n)
```

To compute 517·489, we can employ one of the multiplication algorithms we discussed before: either schoolbook or Karatsuba multiplication.

Parameters n' and d can be precomputed once for given n and r using the extended Euclidean algorithm.

$$
\begin{aligned}
& n \cdot (-n') + r \cdot d = 1 \\
& n = 531, r = 1000 \Rightarrow d = -197, n' = -371 \\
& (1 + n \cdot n') \ \% \ r = (1 + 531 \cdot 629) \ \% \ 10^3 = 0 \\
& r \cdot d \ \% \ n = 1000 \cdot 334 \ \% \ 531 = 1
\end{aligned}
$$

Then, we compute Montgomery reduction for $517 \cdot 489 = 252813$.

$$
\begin{array}{cccccc|l}
2 & 5 & 2 & 8 & 1 & 3 & a \\
2 & 0 & 0 & 1 & 8 & 7 & + 531 \cdot (813 \cdot 629 \ \% \ 10^3) \\
\hline
4 & 5 & 3 & 0 & 0 & 0 & b
\end{array}
$$

$$(a + n \cdot x) \ / \ r \ \% \ n = 453000 \ / \ 1000 \ \% \ 531 = 453 \ (= \mathsf{redc\ n\ r\ d}\ 252813)$$

Finally, with a precomputed Montgomery constant $r^2 \ \% \ n = 127$, we calculate Montgomery reduction for $127 \cdot 453 = 57531$.

$$
\begin{array}{cccccc|l}
 & 5 & 7 & 5 & 3 & 1 & a \\
5 & 3 & 0 & 4 & 6 & 9 & + 531 \cdot (531 \cdot 629 \ \% \ 10^3) \\
\hline
5 & 8 & 8 & 0 & 0 & 0 & b
\end{array}
$$

$$(a + n \cdot x) \ / \ r \ \% \ n = 588000 \ / \ 1000 \ \% \ 531 = 588 - 531 = 57 \ (= \mathsf{to\_mont\ n\ r\ d}\ 453)$$

The final value is 57 which coincides with the result of the regular modular reduction:

$$517 \cdot 489 \ \% \ 531 = 57.$$

Word-based Montgomery reduction. One way to avoid computing n' is to use word-based Montgomery reduction. It takes l iterations to make a divisible by $r = \beta^l$ and requires a precomputed Montgomery constant μ such that $(1 + n \cdot \mu) \ \% \ \beta = 0$.

$$
\begin{array}{c|cccccccccc|ll}
 & a^0_{2\cdot l-1} & \cdots & & a^0_{l+1} & a^0_l & a^0_{l-1} & \cdots & a^0_2 & a^0_1 & a^0_0 & a^0 & \\
 & & & & & \multicolumn{6}{c|}{n \cdot (a^0_0 \cdot \mu \ \% \ \beta)} & + n \cdot (a^0_0 \cdot \mu \ \% \ \beta) & \\
\hline
c_1 & a^1_{2\cdot l-1} & \cdots & a^1_{l+2} & a^1_{l+1} & a^1_l & a^1_{l-1} & \cdots & a^1_2 & a^1_1 & 0 & a^1 & + c_1 \cdot \beta^{2\cdot l} \\
 & & & & \multicolumn{6}{c}{n \cdot (a^1_1 \cdot \mu \ \% \ \beta)} & & + n \cdot (a^1_1 \cdot \mu \ \% \ \beta) \cdot \beta & \\
\hline
c_2 & a^2_{2\cdot l-1} & \cdots & a^2_{l+2} & a^2_{l+1} & a^2_l & a^2_{l-1} & \cdots & a^2_2 & 0 & 0 & a^2 & + c_2 \cdot \beta^{2\cdot l} \\
 & & & \multicolumn{6}{c}{n \cdot (a^2_2 \cdot \mu \ \% \ \beta)} & & & + n \cdot (a^2_2 \cdot \mu \ \% \ \beta) \cdot \beta^2 & \\
\hline
 & \multicolumn{10}{c|}{\cdots\cdots} & \cdots & \\
\hline
c_{l-1} & a^{l-1}_{2\cdot l-1} & \cdots & a^{l-1}_{l+2} & a^{l-1}_{l+1} & a^{l-1}_l & a^{l-1}_{l-1} & \cdots & 0 & 0 & 0 & a^{l-1} & + c_{l-1} \cdot \beta^{2\cdot l} \\
 & \multicolumn{6}{c}{n \cdot (a^{l-1}_{l-1} \cdot \mu \ \% \ \beta)} & & & & & + n \cdot (a^{l-1}_{l-1} \cdot \mu \ \% \ \beta) \cdot \beta^{l-1} & \\
\hline
c_l & a^l_{2\cdot l-1} & \cdots & a^l_{l+2} & a^l_{l+1} & a^l_l & 0 & \cdots & 0 & 0 & 0 & a^l & + c_l \cdot \beta^{2\cdot l}
\end{array}
$$

The algorithm looks very similar to the schoolbook multiplication. In each iteration i, it first computes a multiplication of n by the i-th element of the updated value of a^i multiplied by μ followed by a reduction modulo β. Then, it shifts the product toward the most significant limb i times. Finally, it adds the result to $c_i \cdot \beta^{2 \cdot l} + a^i$, i.e., the value $(c_i \cdot \beta^{2 \cdot l} + a^i + n \cdot (a^i_i \cdot \mu \ \% \ \beta) \cdot \beta^i)$ is computed. This makes a^{i+1}_i equal 0 (note that a^i_j means the j-th element of a at iteration i):

$$\begin{aligned} a^{i+1}_i &= (c_i \cdot \beta^{2 \cdot l} + a_i + n \cdot (a^i_i \cdot \mu \ \% \ \beta) \cdot \beta^i) \ / \ \beta^i \ \% \ \beta \\ &= (a^i_i + n \cdot (a^i_i \cdot \mu \ \% \ \beta)) \ \% \ \beta = (a^i_i \cdot (1 + n \cdot \mu)) \ \% \ \beta = 0. \end{aligned}$$

We can prove by induction on i that the following is fulfilled. Setting i to l gives us the same post-conditions for the result as for Montgomery reduction with n'.

$$\begin{aligned} &(c_i \cdot \beta^{2 \cdot l} + a^i) \ \% \ n = a^0 \ \% \ n \\ &(c_i \cdot \beta^{2 \cdot l} + a^i) \ \% \ \beta^i = 0 \\ &(c_i \cdot \beta^{2 \cdot l} + a^i) \le a^0 + n \cdot (\beta^i - 1) \land c_i \le 1 \end{aligned}$$

Example. Let us now consider the same example, computing $517 \cdot 489$ modulo 531, but using word-based Montgomery reduction. Parameter μ is easy to calculate:

$$(1 + 531 \cdot \mu) \ \% \ 10 = (1 + 1 \cdot \mu) \ \% \ 10 = 0 \Rightarrow \mu = 9.$$

Let us show only how to carry out Montgomery reduction for $517 \cdot 489 = 252813$.

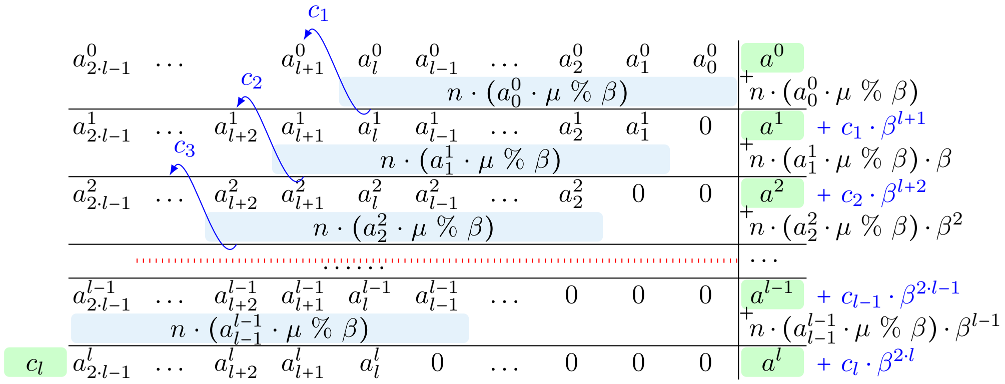
2	5	2	8	1	3	a^0
		3	7	1	7	$+ \ 531 \cdot (3 \cdot 9 \ \% \ 10)$
2	5	6	5	3	0	a^1
	3	7	1	7		$+ \ 531 \cdot (3 \cdot 9 \ \% \ 10) \cdot 10$
2	9	3	7	0	0	a^2
1	5	9	3			$+ \ 531 \cdot (7 \cdot 9 \ \% \ 10) \cdot 10^2$
4	5	3	0	0	0	a^3

The result is $453000 \ / \ 10^3 \ \% \ 531 = 453$ which matches with the value from the example above.

Complexity. The word-based Montgomery reduction algorithm requires the same number of double-wide multiplications as for the schoolbook multiplication, l^2, and, in addition, l modular multiplications (e.g., 64-bit × 64-bit → 64-bit).

Specifying and verifying word-based Montgomery reduction in F* (low-level specification). As we already pointed out, word-based Montgomery reduction is very similar to the schoolbook multiplication from §5.3.2. Let us recap our F* specification for the latter.

```
let rec bn_mul_loop #t #len (a b:bignum t len) (i:nat{i ≤ len}) : bignum t (len + len) =
  if i = 0 then create (len + len) (uint #t 0)
  else begin
    let acc1 = bn_mul_loop a b (i − 1) in
    let (c, acc) = bn_mul1_lshift_add a b.[i − 1] (i − 1) acc1 in
    acc.[len + i − 1] ← c end
```


c_1
c_2
c_3
$a^0_{2\cdot l-1}$ … a^0_{l+1} a^0_l a^0_{l-1} … a^0_2 a^0_1 a^0_0
a^0
$n\cdot(a^0_0\cdot\mu \,\%\, \beta)$
+ $n\cdot(a^0_0\cdot\mu \,\%\, \beta)$
$a^1_{2\cdot l-1}$ … a^1_{l+2} a^1_{l+1} a^1_l a^1_{l-1} … a^1_2 a^1_1 0
a^1 $+\, c_1\cdot\beta^{l+1}$
$n\cdot(a^1_1\cdot\mu \,\%\, \beta)$
+ $n\cdot(a^1_1\cdot\mu \,\%\, \beta)\cdot\beta$
$a^2_{2\cdot l-1}$ … a^2_{l+2} a^2_{l+1} a^2_l a^2_{l-1} … a^2_2 0 0
a^2 $+\, c_2\cdot\beta^{l+2}$
$n\cdot(a^2_2\cdot\mu \,\%\, \beta)$
+ $n\cdot(a^2_2\cdot\mu \,\%\, \beta)\cdot\beta^2$
……
…
$a^{l-1}_{2\cdot l-1}$ … a^{l-1}_{l+2} a^{l-1}_{l+1} a^{l-1}_l a^{l-1}_{l-1} … 0 0 0
a^{l-1} $+\, c_{l-1}\cdot\beta^{2\cdot l-1}$
$n\cdot(a^{l-1}_{l-1}\cdot\mu \,\%\, \beta)$
+ $n\cdot(a^{l-1}_{l-1}\cdot\mu \,\%\, \beta)\cdot\beta^{l-1}$
c_l $a^l_{2\cdot l-1}$ … a^l_{l+2} a^l_{l+1} a^l_l 0 … 0 0 0
a^l $+\, c_l\cdot\beta^{2\cdot l}$

$$\begin{aligned}&(1+n\cdot\mu)\ \%\ \beta = 0\ \wedge\ c_i \le 1\ \wedge\\ &(a^i + c_i\cdot\beta^{l+i})\ \%\ n = a^0\ \%\ n\ \wedge\\ &(a^i + c_i\cdot\beta^{l+i})\ \%\ \beta^i = 0\ \wedge\\ &(a^i + c_i\cdot\beta^{l+i}) \le a^0 + n\cdot(\beta^i - 1)\end{aligned}$$

Figure 5.10 – Montgomery reduction.

The main difference is that we start with an accumulator $acc = a^0$ that is not filled with zeroes. So we cannot directly store a limb c at index (len + i − 1). Instead, we perform an addition to the contents of the sequence at that index, which can yield a carry. We do not propagate the carry towards the most significant limb, so we have to pass it to the next loop iteration. We now need to deal not with a simple addition, but with an addition with the carry from the previous iteration. Figure 5.10 depicts the optimization for the algorithm. Our F* specification is as follows.

```
let carry_bn (t:limb_t) (len:nat) = tuple2 (carry t) (bignum t (len + len))
let rec bn_redc_loop #t #len (n:bignum t len) (mu:limb t)
  (i:nat{i ≤ len}) ((c, a): carry_bn t len) : carry_bn t len =
  if i = 0 then (c, a)
  else begin
    let (c1, a1) : carry_bn t len = bn_redc_loop n mu (i − 1) (c, a) in
    let (t1, a2) = bn_mul1_lshift_add n (mu ∗. a1.[i − 1]) (i − 1) a1 in
    let (c2, r) = addcarry c1 t1 a2.[len + i − 1] in
    let res = a2.[len + i − 1] ← r in
    (c2, res) end
```

The functional correctness of the bn_redc_loop function can be proven by induction on i with the following induction hypothesis.

```
val bn_redc_loop_lemma (#t:limb_t) (#len:nat) (n:bignum t len)
  (mu:limb t) (i:nat{i ≤ len}) (a0:bignum t (len + len)) : Lemma
  (requires (1 + bn_v n ∗ v mu) % pow2 (bits t) = 0)
  (ensures
   (let (c, a) = bn_redc_loop n mu i (uint #t 0, a0) in
    let res = bn_v a + v c ∗ pow2 (bits t ∗ (len + i)) in
    res % bn_v n = bn_v a0 % bn_v n ∧ res % pow2 (bits t ∗ i) = 0 ∧
    res ≤ bn_v a0 + bn_v n ∗ (pow2 (bits t ∗ i) − 1)))
```

A division by $r = 2^{\text{bits t} \,*\, \text{len}}$ is simple since we need to skip the first len elements of the resulting bignum of the bn_redc_loop call. The function is invoked with a modulus n of length len, a constant mu, a length len, and a tuple containing an initial value for the carry and a bignum a of length (len + len).

```
let bn_redc_loop_div_r #t #len (n:bignum t len) (mu:limb t) (a:bignum t (len + len))
  : tuple2 (carry t) (bignum t len) =
  let (c0, a) = bn_redc_loop n mu len (uint #t 0, a) in
  c0, slice a len (len + len)
```

A final conditional subtraction can be done similarly to the modular reduction for addition from §5.6.1.

Word-based Montgomery reduction in Low* (stateful code). As we can see, we only need to store a length len, a modulus n of length len and the precomputed constants r2 and mu in the bn_mont_ctx structure. We keep the following invariant throughout the computations for a bignum r2 of length len and a limb mu:

```
(1 + bn_v n * v mu) % pow2 (bits t) = 0 ∧
bn_v r2 = pow2 (2 * bits t * len) % bn_v n
```

5.6.3 Almost Montgomery Multiplication

Almost Montgomery reduction [38, 63] loosens the conditions on a. It takes $a < r{\cdot}r$ and returns a result that is less than r. On the one hand keeping the invariant nat_mod r = x:nat{x < r} for a bignum slightly simplifies a final conditional subtraction, but on the other hand it makes modular addition and subtraction more inefficient. Almost Montgomery reduction can be used, for example, to perform a reduction of a 512-bit number modulo the order of the prime-order subgroup of the Curve25519 curve (§5.8.3).

The F* specifications for Montgomery reduction and Almost Montgomery reduction differ only in the final reduction modulo n. The former compares $(a + x \cdot n) \;/\; r$ with n and the latter with r.

```
let final_mont (n:pos) (res:nat{res < 2 * n}) : c:nat_mod n{c = res % n} =
  if n ≤ res then res − n else res

let final_almost_mont (n:pos) (r:pos{n < r}) (res:nat{res < r + n}) : c:nat_mod r{c % n = res % n} =
  if r ≤ res then res − n else res
```

The requirement $res < r + n$ for the final_almost_mont function comes from the following.

$$
\begin{aligned}
&(a + n \cdot x) \;/\; r = \\
&(a + n \cdot ((a \;\%\; r) \cdot n' \;\%\; r)) \;/\; r \le (r \cdot r - 1 + n \cdot (r - 1)) \;/\; r < (r \cdot r + r \cdot n) \;/\; r = r + n
\end{aligned}
$$

We can also avoid the final conditional subtraction by keeping the invariant nat_mod (2∗n) for a bignum. In this case, r has to be greater than $4 \cdot n$. However, this can have a negative impact on performance when we need to increase a length of a bignum by one.

$$
\begin{aligned}
&(a + n \cdot x) \;/\; r = \\
&(a + n \cdot ((a \;\%\; r) \cdot n' \;\%\; r)) \;/\; r \le (4 \cdot n \cdot n - 1 + n \cdot (r - 1)) \;/\; r < (r \cdot n + r \cdot n) \;/\; r = 2 \cdot n
\end{aligned}
$$

5.7 Verifying Exponentiation

Definitions. A *commutative monoid* $(X, \bullet)$ is a set X together with an operation $\bullet$, satisfying the following axioms

closure: for all a, b in X, $a \bullet b$ is also in X

associativity: for all a, b, c in X, $(a \bullet b) \bullet c = a \bullet (b \bullet c)$

identity element: there exists *one* in X such that for all a in X, $a \bullet \mathit{one} = \mathit{one} \bullet a = a$

commutativity: for all a, b in X, $a \bullet b = b \bullet a$

An *abelian group* $(X, \bullet)$ is constructed from a commutative monoid $(X, \bullet)$ and the $\bullet$ operation satisfies, in addition, the following axiom:

inverse element: for all a in X there exists a^{-1} in X such that $a \bullet a^{-1} = a^{-1} \bullet a = \mathit{one}$

Given a positive integer b and a commutative monoid $(X, \bullet)$, *exponentiation* is defined as a repeated application of the $\bullet$ operation to an element $a \in X$:

$$a^b = \underbrace{a \bullet a \bullet \ldots \bullet a}_{b \text{ times}}.$$

For $b = 0$, a^0 is defined as *one*.

When $(X, \bullet)$ is an abelian group, exponentiation for a negative integer b is defined as

$$a^b = (a^{-1})^{-b}, \text{ where } a^{-1} \text{ is the inverse element of } a.$$

Examples. As a first example, we can consider a repeated application of addition of integers which gives us multiplication. If we take X as $\mathbb{Z}$, the set of integers, together with an addition operation $+$, we can prove that $(\mathbb{Z}, +)$ is an abelian group with an identity element 0 and an inverse element of a is $-a$. Similarly, we can take X as $\mathbb{Z}_n$, the set of integers modulo n, together with a modular addition operation $+\%$ and construct an abelian group $(\mathbb{Z}_n, +\%)$ with an identity element 0 and an inverse element of a is $(n - a)$.

Another example is $(\mathbb{Z}, \cdot)$, the set of integers with a multiplication operation $\cdot$, that forms a commutative monoid with an identity element 1, but not an abelian group. However, $(\mathbb{Z}_n \setminus \{0\}, \cdot\%)$, the set of integers modulo prime n together with a modular multiplication operation $\cdot\%$, is an abelian group with an identity element 1 since there exists a multiplicative inverse of a, which can be computed as $a^{n-2} \;\%\; n$ (cf. Fermat's little theorem).

As a final example, we can take a set of points on an elliptic curve together with a point addition operation that forms an abelian group where the identity element is a point at infinity.

Exponential properties. There are different ways of rewriting powers that rely on the following properties.

$$x^n \bullet x^m = x^{n+m} \tag{5.1}$$

$$(x^n)^m = x^{n \cdot m} \tag{5.2}$$

$$x^n \bullet y^n = (x \bullet y)^n \tag{5.3}$$

Their correctness is derived from the definition of exponentiation and properties of the $\bullet$ operation.

$$x^n \bullet x^m = \underbrace{\underbrace{x \bullet x \bullet \ldots \bullet x}_{n \text{ times}} \bullet \underbrace{x \bullet x \bullet \ldots \bullet x}_{m \text{ times}}}_{(n+m) \text{ times}} = x^{n+m}$$

$$(x^n)^m = \underbrace{x^n \bullet x^n \bullet \ldots \bullet x^n}_{m \text{ times}} = \underbrace{\underbrace{x \bullet x \bullet \ldots \bullet x}_{n \text{ times}} \bullet \ldots \bullet \underbrace{x \bullet x \bullet \ldots \bullet x}_{n \text{ times}}}_{(n \cdot m) \text{ times}} = x^{n \cdot m}$$

$$x^n \bullet y^n = \underbrace{x \bullet x \bullet \ldots \bullet x}_{n \text{ times}} \bullet \underbrace{y \bullet y \bullet \ldots \bullet y}_{n \text{ times}} = \underbrace{(x \bullet y) \bullet (x \bullet y) \bullet \ldots \bullet (x \bullet y)}_{n \text{ times}} = (x \bullet y)^n$$

Methods for computing an exponentiation. The naive method of carrying out an exponentiation a^b requires $(b-1)$ "multiplications" for $b > 0$. In order to compute an exponentiation in a more efficient way, we distinguish three types of methods:

— generic methods, when both a and b can vary (e.g., left-to-right binary method)

— fixed base methods, when a is fixed (e.g., fixed-base comb method [81])

— fixed exponent methods, when b is fixed (e.g., addition-chain method).

In this thesis, we focus on *generic methods for computing an exponentiation with a non-negative exponent b.*

Specifying exponentiation in F* (high-level specification). A commutative monoid can be defined as a cm structure containing an operation mul, satisfying associativity and commutativity, and an identity element one:

```
type cm (t:Type) = {
  one: t;
  mul: t → t → t;
  lemma_mul_assoc: a:t → b:t → c:t → Lemma (mul (mul a b) c == mul a (mul b c));
  lemma_mul_comm: a:t → b:t → Lemma (mul a b == mul b a);
  lemma_one: a:t → Lemma (mul a one == a) }
```

The exponentiation is defined as a recursive total function pow that takes as input a commutative monoid k, an element a of type t and a non-negative power b and returns the result of the same type t.

```
let rec pow (#t:Type) (k:cm t) (a:t) (b:nat) : t =
  if b = 0 then k.one else k.mul a (pow k a (b − 1))
```

The base case (when $\mathsf{b} = 0$) returns an identity element one. The recursive case of the pow function invokes a mul operation with a and the result of the recursive call on the input k, a and index $(\mathsf{b} - 1)$.

The exponential properties can be proved by induction on n or m.

```
val lemma_pow_add (#t:Type) (k:cm t) (x:t) (n m:nat) :
  Lemma (k.mul (pow k x n) (pow k x m) == pow k x (n + m))

val lemma_pow_mul (#t:Type) (k:cm t) (x:t) (n m:nat) :
  Lemma (pow k (pow k x n) m == pow k x (n * m))

val lemma_pow_mul_base (#t:Type) (k:cm t) (x y:t) (n:nat) :
  Lemma (k.mul (pow k x n) (pow k y n) == pow k (k.mul x y) n)
```

5.7.1 Binary Method

Right-to-left binary method. A first method for carrying out an exponentiation a^b is based on the following equality that uses a binary representation of an exponent $b = (b_{n-1} \ldots b_2 b_1 b_0)_2$:

$$a^b = (a^{2^{n-1}})^{b_{n-1}} \bullet \ldots \bullet (a^{2^2})^{b_2} \bullet (a^2)^{b_1} \bullet a^{b_0}.$$

It is easy to see that the correctness of the formula is derived from properties (5.1) and (5.2):

$$(a^{2^{n-1}})^{b_{n-1}} \bullet \ldots \bullet (a^{2^2})^{b_2} \bullet (a^2)^{b_1} \bullet a^{b_0} = a^{b_{n-1} \cdot 2^{n-1} + \ldots + b_2 \cdot 2^2 + b_1 \cdot 2 + b_0} = a^{(b_{n-1} \ldots b_2 b_1 b_0)_2} = a^b.$$

The algorithm goes over the exponent b starting from the least significant bit and proceeds bit-by-bit to the most significant one. At each iteration i, it computes both values $acc_i = a^{(b_i \ldots b_2 b_1 b_0)_2}$ and $c_i = a^{2^{i+1}}$:

$$acc_i = a^{(b_i \ldots b_2 b_1 b_0)_2} = (\underbrace{a^{2^i}}_{c_{i-1}})^{b_i} \bullet \underbrace{(a^{2^{i-1}})^{b_{i-1}} \bullet \ldots \bullet (a^{2^2})^{b_2} \bullet (a^2)^{b_1} \bullet a^{b_0}}_{= a^{(b_{i-1} \ldots b_2 b_1 b_0)_2} = acc_{i-1}} = (c_{i-1})^{b_i} \bullet acc_{i-1}$$

$$c_i = a^{2^{i+1}} = a^{2^i} \bullet a^{2^i} = c_{i-1} \bullet c_{i-1}$$

The initial values of acc_0 and c_0 are *one* and a, respectively.

Example. Let us compute a^{71} using the right-to-left binary method for an exponent $b = (71)_{10} = (1000111)_2$.

i		0	1	2	3	4	5	6	
b_i		1	1	1	0	0	0	1	
$a^{(b_i \ldots b_0)_2}$	*one*	$a \bullet one$	$a^2 \bullet a$	$a^4 \bullet a^3$	a^7	a^7	a^7	$a^{64} \bullet a^7$	a^{71}
$a^{2^{i+1}}$	a	$(a)^2$	$(a^2)^2$	$(a^4)^2$	$(a^8)^2$	$(a^{16})^2$	$(a^{32})^2$	$(a^{64})^2$	a^{128}

It takes only 7 "squarings" and 4 "multiplications" to get the result, which is significantly less than in the naive method.

Left-to-right binary method. A second method for performing an exponentiation a^b uses a binary representation of an exponent $b = (b_{n-1} \dots b_2 b_1 b_0)_2$ and Horner's rule:

$$a^b = (((\dots(one^2 \bullet a^{b_{n-1}})^2 \dots)^2 \bullet a^{b_2})^2 \bullet a^{b_1})^2 \bullet a^{b_0}.$$

The correctness of the formula follows from properties (5.1) and (5.2):

$$\begin{aligned}&(((\dots(one^2 \bullet a^{b_{n-1}})^2 \dots)^2 \bullet a^{b_2})^2 \bullet a^{b_1})^2 \bullet a^{b_0}\\ &= a^{((\dots(b_{n-1}\cdot 2+\dots)\cdot 2+b_2)\cdot 2+b_1)\cdot 2+b_0} = a^{b_{n-1}\cdot 2^{n-1}+\dots+b_2\cdot 2^2+b_1\cdot 2+b_0} = a^{(b_{n-1}\dots b_2 b_1 b_0)_2} = a^b.\end{aligned}$$

The algorithm starts with the most significant bit of the exponent b and moves bit-by-bit towards the least significant one. At each iteration i, it computes $acc_i = a^{(b_{n-1}\dots b_{n-1-i})_2}$:

$$acc_i = a^{(b_{n-1}\dots b_{n-1-i})_2} = \underbrace{((\dots(one^2 \bullet a^{b_{n-1}})^2 \dots)^2 \bullet a^{b_{n-1-(i-1)}})}_{=a^{(b_{n-1}\dots b_{n-1-(i-1)})_2}=acc_{i-1}}{}^2 \bullet a^{b_{n-1-i}} = (acc_{i-1})^2 \bullet a^{b_{n-1-i}}.$$

The initial value of acc_0 is *one*.

Example. Let us consider the same example, a^{71}, but now computing the result using the left-to-right binary method.

i		0	1	2	3	4	5	6	
b_{n-1-i}		1	0	0	0	1	1	1	
$a^{(b_{n-1}\dots b_{n-1-i})_2}$	*one*	$(one)^2 \bullet a$	$(a)^2$	$(a^2)^2$	$(a^4)^2$	$(a^8)^2 \bullet a$	$(a^{17})^2 \bullet a$	$(a^{35})^2 \bullet a$	a^{71}

If we unroll the first iteration of the loop in the algorithm and simplify it, it takes 6 "squarings" and 3 "multiplications" to get the result. Therefore, we get one less "squaring" and one less "multiplication" in comparison with the right-to-left binary method. Also, there is no need for the extra variable $c_i = a^{2^{i+1}}$.

Complexity. Both binary methods, right-to-left and left-to-right, require n "squarings" and x "multiplications", where x is the number of 1's in the binary representation of the exponent (the Hamming weight).

Specifying and verifying left-to-right binary method in F* (high-level specification). One way to compute $acc_i = a^{(b_{n-1}\dots b_{n-1-i})_2}$ for the left-to-right binary method is to check whether the $(n-1-i)$-th bit of the exponent b is set. If so, we multiply $(acc_{i-1})^2$ by a. Otherwise, we return $(acc_{i-1})^2$ as the result, since $(acc_{i-1})^2 \bullet a^0 = (acc_{i-1})^2 \bullet one = (acc_{i-1})^2$. The problem with this method is that it is not constant-time, as it performs a "multiplication" depending on the secret value b_{n-1-i}. In order to have an algorithm resistant to timing attacks, we can employ the Montgomery ladder [89] which always performs one "squaring" and one "multiplication" per iteration.

Let us first specify a function get_ith_bit that takes as input a non-negative number $b < 2^{bBits}$ and returns the i-th bit of b.

```
let get_ith_bit (bBits:nat) (b:nat{b < pow2 bBits}) (i:nat{i < bBits}) =
  b / pow2 i % 2
```

Then, we define a signature for a cswap function that takes two elements, r0 and r1, of type t and returns a tuple (r1, r0) if sw = 1 or a tuple (r0, r1) otherwise.

```
val cswap (#t:Type) (sw:nat{sw ≤ 1}) (r0 r1:t) : Pure (tuple2 t t) (requires ⊤)
  (ensures (λ (r2, r3) → (r2, r3) == (if sw = 1 then (r1, r0) else (r0, r1))))
```

The function cswap can be implemented in a way similar to a constant-time selection function from §5.6.1.

Finally, we introduce a function exp_mont_ladder that expects a commutative monoid k, an element a of type t, an exponent b < 2^{bBits} and returns the result equal to pow k a b, as stated in the lemma exp_mont_ladder_lemma.

```
let rec exp_ml_loop (#t:Type) (k:cm t) (a:t) (bBits:nat) (b:nat{b < pow2 bBits})
  (i:nat{i ≤ bBits}) : tuple2 t t =
  if i = 0 then (k.one, a)
  else begin
    let (acc_i1, a_acc_i1) = exp_ml_loop k a bBits b (i − 1) in
    let bit = get_ith_bit bBits b (bBits − i) in
    let (r0, r1) = cswap bit acc_i1 a_acc_i1 in
    let (r0, r1) = (k.mul r0 r0, k.mul r1 r0) in
    let (acc_i, a_acc_i) = cswap bit r0 r1 in
    (acc_i, a_acc_i) end

let exp_mont_ladder (#t:Type) (k:cm t) (a:t) (bBits:nat) (b:nat{b < pow2 bBits}) : t =
  let (acc, _) = exp_ml_loop k a bBits b bBits in acc

val exp_mont_ladder_lemma (#t:Type) (k:cm t) (a:t) (bBits:nat) (b:nat{b < pow2 bBits}) :
  Lemma (exp_mont_ladder k a bBits b == pow k a b)
```

The proof of the lemma exp_mont_ladder_lemma follows from the following induction hypothesis for the total recursive exp_ml_loop function.

```
val exp_ml_loop_lemma (#t:Type) (k:cm t) (a:t) (bBits:nat) (b:nat{b < pow2 bBits}) (i:nat{i ≤ bBits}) :
  Lemma (let (acc_i, a_acc_i) = exp_ml_loop k a bBits b i in
    acc_i == pow k a (b / pow2 (bBits − i)) ∧ a_acc_i == k.mul a acc_i)
```

The base case of the exp_ml_loop function returns a tuple (k.one, a) which corresponds to $(a^0, a \bullet a^0)$. The recursive case can be proved as follows.

Induction Hypothesis. $acc_{i-1} = a^{(b_{n-1} \dots b_{n-1-(i-1)})_2} \wedge a\_acc_{i-1} = a \bullet acc_{i-1}$

Case 1. $b_{n-1-i} = 1 \Rightarrow acc_i = acc_{i-1} \bullet a\_acc_{i-1} \wedge a\_acc_i = a\_acc_{i-1} \bullet a\_acc_{i-1}$

Case 2. $b_{n-1-i} = 0 \Rightarrow acc_i = acc_{i-1} \bullet acc_{i-1} \wedge a\_acc_i = a\_acc_{i-1} \bullet acc_{i-1}$

In both cases, we have the following $acc_i = a^{(b_{n-1} \dots b_{n-1-i})_2} \wedge a\_acc_i = a \bullet acc_i$ which concludes our proof.

5.7.2 Fixed-window Method

Let us recap the left-to-right binary method for carrying out an exponentiation a^b that uses a binary representation for an exponent $b = (b_{n-1} \dots b_2 b_1 b_0)_2$:

$$a^b = (((\dots(one^2 \bullet a^{b_{n-1}})^2 \dots)^2 \bullet a^{b_2})^2 \bullet a^{b_1})^2 \bullet a^{b_0}.$$

A generalisation of this method is to use a base-2^w representation for non-negative integer w and exponent $b = (b_{m-1} \dots b_2 b_1 b_0)_{2^w}$:

$$a^b = (((\dots(one^{2^w} \bullet a^{b_{m-1}})^{2^w} \dots)^{2^w} \bullet a^{b_2})^{2^w} \bullet a^{b_1})^{2^w} \bullet a^{b_0}.$$

Similar to the binary method, the correctness of the formula is based on properties (5.1) and (5.2). The algorithm handles the exponent b digit-by-digit in base-2^w starting from the most significant digit to the least significant one. At each iteration i, it computes $acc_i = a^{(b_{m-1} \dots b_{m-1-i})_{2^w}}$:

$$\begin{aligned} acc_i &= a^{(b_{m-1} \dots b_{m-1-i})_{2^w}} \\ &= \underbrace{((\dots(one^{2^w} \bullet a^{b_{m-1}})^{2^w} \dots)^{2^w} \bullet a^{b_{m-1-(i-1)}})}_{=a^{(b_{m-1} \dots b_{m-1-(i-1)})_{2^w}} = acc_{i-1}}{}^{2^w} \bullet a^{b_{m-1-i}} = (acc_{i-1})^{2^w} \bullet a^{b_{m-1-i}}. \end{aligned}$$

The initial value of acc_0 is *one*.

The calculations for the algorithm can be split into two parts. The first is to store all the values for a^0, a^1, …, a^{2^w-1} in a look-up table. The second part is a *for* loop over the exponent b, where at each iteration i, the value $acc_i = (acc_{i-1})^{2^w} \bullet a^{b_{m-1-i}}$ is computed. The value $(acc_{i-1})^{2^w}$ can be calculated as w "squarings":

$$\left.\begin{aligned} &acc_{i-1}^{2^0} \bullet acc_{i-1}^{2^0} = acc_{i-1}^{2^1} \\ &acc_{i-1}^{2^1} \bullet acc_{i-1}^{2^1} = acc_{i-1}^{2^2} \\ &acc_{i-1}^{2^2} \bullet acc_{i-1}^{2^2} = acc_{i-1}^{2^3} \\ &\dots \\ &acc_{i-1}^{2^{w-1}} \bullet acc_{i-1}^{2^{w-1}} = acc_{i-1}^{2^w} \end{aligned}\right\} w \text{ times}$$

The other value $a^{b_{m-1-i}}$ is loaded from the precomputed look-up table. Note that access to the table has to be done in a constant-time way. This can be implemented using the bn_mask_select and eq_mask functions from §5.6.1 and §2.2, respectively.

```
let rec table_select_constanttime (#t:limb_t) (#len:nat)
  (table_len:range_t t) (table:lseq (limb t) (table_len ∗ len))
  (i:limb t{v i < table_len}) (j:nat{j ≤ table_len}) (acc:bignum t len) :
  Tot (bignum t len) (decreases (table_len − j)) =
  if j = table_len then acc
  else begin
    let table_j = slice table (j ∗ len) (j ∗ len + len) in
    let acc = bn_mask_select (eq_mask i (uint #t j)) table_j acc in
    table_select_constanttime table_len table i (j + 1) acc end
```

Example. Let us calculate a^{71} using the fixed-window method for $w = 3$ and an exponent $b = (71)_{10} = (1000111)_2 = (107)_{2^3}$.

i		0	1	2	
b_{m-1-i}		1	0	7	
$a^{(b_{m-1}\dots b_{m-1-i})_{2^w}}$	one	$(one)^{2^3} \bullet a^1$	$(a^1)^{2^3} \bullet a^0$	$(a^8)^{2^3} \bullet a^7$	a^{71}

If we unroll the first iteration of the for loop in the algorithm and simplify it, it takes 6 "squarings", 3 look-ups and 2 "multiplications" to get the result.

Complexity. The fixed-window method requires the same number of "squarings" as for the Montgomery ladder, but it reduces the number of "multiplications" to m (from $w \cdot m$):

$$w \cdot m \ \times \ \text{"squarings"} + m \ \times \ (\text{"multiplications"} + \text{a look-up table access}).$$

Possible Optimizations. When an inverse operation is fast, we can use a signed fixed-window method that halves in size the precomputed table. Namely, we change the representation of b_i from $0 \le b_i < 2^w$ to $2^{w-1} \le b_i < 2^{w-1}$ and compute a^{b_i} for a negative b_i as a look-up a^{-b_i} followed by the inverse operation, $(a^{-b_i})^{-1}$.

Double fixed-window method. In many signature verification algorithms, one needs to carry out a double-exponentiation $a^b \bullet c^d$. One way of doing it is to individually compute a^b and c^d and then find their "product". A more efficient way is to interleave these computations so that the number of "squarings" is significantly reduced.

As in the fixed-window method, we use a base-2^w representation for the exponents $b = (b_{m-1} \dots b_2 b_1 b_0)_{2^w}$ and $d = (d_{m-1} \dots d_2 d_1 d_0)_{2^w}$. For simplicity, we assume that they both have the same number of digits m. Knowing property (5.3), we can prove that the following formula for carrying out $a^b \bullet c^d$ is correct.

$$\begin{aligned}
a^b &= (((\dots(one^{2^w} \bullet a^{b_{m-1}})^{2^w} \dots)^{2^w} \bullet a^{b_2})^{2^w} \bullet a^{b_1})^{2^w} \bullet a^{b_0} \\
c^d &= (((\dots(one^{2^w} \bullet c^{d_{m-1}})^{2^w} \dots)^{2^w} \bullet c^{d_2})^{2^w} \bullet c^{d_1})^{2^w} \bullet c^{d_0} \\
a^b \bullet c^d &= (((\dots(one^{2^w} \bullet a^{b_{m-1}} \bullet c^{d_{m-1}})^{2^w} \dots)^{2^w} \bullet a^{b_2} \bullet c^{d_2})^{2^w} \bullet a^{b_1} \bullet c^{d_1})^{2^w} \bullet a^{b_0} \bullet c^{d_0}
\end{aligned}$$

The algorithm simultaneously processes both exponents b and d, starting from the most significant digit and moves to the least significant one. At each iteration i, the value $acc_i = a^{(b_{m-1}\dots b_{m-1-i})_{2^w}} \bullet c^{(d_{m-1}\dots d_{m-1-i})_{2^w}}$ is computed:

$$\begin{aligned}
acc_i &= a^{(b_{m-1}\dots b_{m-1-i})_{2^w}} \bullet c^{(d_{m-1}\dots d_{m-1-i})_{2^w}} \\
&= \underbrace{((\dots(one^{2^w} \bullet a^{b_{m-1}} \bullet c^{d_{m-1}})^{2^w} \dots)^{2^w} \bullet a^{b_{m-1-(i-1)}} \bullet c^{d_{m-1-(i-1)}})}_{=a^{(b_{m-1}\dots b_{m-1-(i-1)})_{2^w}} \bullet c^{(d_{m-1}\dots d_{m-1-(i-1)})_{2^w}} = acc_{i-1}}{}^{2^w} \bullet a^{b_{m-1-i}} \bullet c^{d_{m-1-i}} \\
&= (acc_{i-1})^{2^w} \bullet a^{b_{m-1-i}} \bullet c^{d_{m-1-i}}.
\end{aligned}$$

Example. Let us carry out $a^{71} \bullet c^{120}$ using the double fixed-window method for $w = 3$ and the exponents $b = (71)_{10} = (1000111)_2 = (107)_{2^3}$ and $d = (120)_{10} = (1111000)_2 = (170)_{2^3}$.

i		0	1	2	
b_{m-1-i}		1	0	7	
d_{m-1-i}		1	7	0	
$a^{(b_{m-1} \ldots b_{m-1-i})_{2^w}} \bullet c^{(d_{m-1} \ldots d_{m-1-i})_{2^w}}$	one	$(one)^{2^3} \bullet a^1 \bullet c^1$	$(a^1 \bullet c^1)^{2^3} \bullet a^0 \bullet c^7$	$(a^8 \bullet c^{15})^{2^3} \bullet a^7 \bullet c^0$	$a^{71} \bullet c^{120}$

If we unroll the first iteration of the for loop in the algorithm and simplify it, it takes 6 "squarings", 6 look-ups and 5 "multiplications" to get the result. So, we save 6 "squarings" compared to multiplying the results of calling the fixed-window method twice for a^{71} and c^{120}.

Complexity. The double fixed-window method reduces the number of "squarings" to $w \cdot m$ (from $2 \cdot w \cdot m$) and keeps the same number of "multiplications":

$$w \cdot m \;\times\; \text{“squarings”} + 2 \cdot m \;\times\; (\text{“multiplications”} + \text{a look-up table access}).$$

We also save one "multiplication" as we do not need to compute the "product" of a^b and c^d.

Possible Optimizations. Implementation of a signature verification algorithm does not need to be constant-time, so access to the precomputed table can be done in a variable-time way, i.e., the value located at index b_i or d_i can be immediately returned. A more efficient implementation for such case can be obtained using the sliding-window method [74] or window NAF method [117].

5.7.3 Modular Exponentiation

Given the definition of exponentiation from §5.7, we can define a modular exponentiation as an integer a raised to the b-th power and then taken modulo n.

$$a^b \ \% \ n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{b \text{ times}} \ \% \ n.$$

While this definition may be handy in proofs, it is rather inefficient in practice, especially when the exponent or base size can reach several thousand bits. A more efficient way is to define modular exponentiation as the repeated modular multiplication of the base $0 \leq a < n$:

$$\textsf{pow\_mod}: a^b \ \% \ n = (\ldots((a \cdot a) \ \% \ n \cdot a) \ \% \ n \cdot \ldots \cdot a) \ \% \ n.$$

As explained in §5.5, with the help of Montgomery arithmetic, we can avoid an expensive division operation when computing a reduction modulo an odd n. Therefore, we can define Montgomery exponentiation pow_mont as the repeated application of Montgomery multiplication to the base a in Montgomery form and prove that this function, together with the composition of conversion functions to and from Montgomery form, results in the modular exponentiation pow_mod.

$$\begin{tikzcd}
aM = a \cdot r \;\%\; n \arrow[r, "\mathsf{pow\_mont}", "r \cdot d \;\%\; n = 1"'] & (aM \cdot d)^b \cdot r \;\%\; n \arrow[d, "\mathsf{from\_mont}"] \\
a \arrow[u, "\mathsf{to\_mont}"] \arrow[r, "\mathsf{pow\_mod}"'] & a^b \;\%\; n
\end{tikzcd}$$

Specifying modular exponentiation in F* (high-level specification). In order to define exponentiation modulo a positive integer n, we instantiate a cm type with the identity element one = 1 % n and the monoid operation mul = mul_mod and prove the required properties such as commutativity and associativity of mul_mod.

```
let mul_mod (#n:pos) (a b:nat_mod n) : nat_mod n = (a ∗ b) % n

let mk_nat_mod_cm (n:pos) : cm (nat_mod n) =
  { one = 1 % n; mul = mul_mod; (∗ omitted: properties about mul_mod ∗) }

let pow_mod (#n:pos) (a:nat_mod n) (b:nat) : nat_mod n =
  pow (mk_nat_mod_cm n) a b
```

Given a commutative monoid mk_nat_mod_cm, we can obtain various implementations of modular exponentiation with no additional proofs. For example, the implementation of Montgomery ladder (§5.7.1) can be obtained as follows.

```
let pow_mod_mont_ladder (n:pos) (a:nat_mod n) (bBits:nat) (b:nat{b < pow2 bBits}) : nat_mod n =
  let k = mk_nat_mod_cm n in
  let res = exp_mont_ladder k a bBits b in
  exp_mont_ladder_lemma k a bBits b;
  assert (res == pow k a b);
  res
```

Still, we use the naive method pow_mod to prove, for example, the equivalence between Montgomery exponentiation and modular exponentiation.

Similarly, we define Montgomery exponentiation as the repeated application of Montgomery multiplication mul_mont with the identity element r % n (= to_mont n r d 1).

```
let mul_mont (n r:pos) (d:int{r ∗ d % n = 1}) (aM bM:nat_mod n) = redc n r d (aM ∗ bM)

let mk_nat_mont_cm (n r:pos) (d:int{r ∗ d % n = 1}) : cm (nat_mod n) =
  { one = r % n; mul = mul_mont n r d; (∗ omitted: properties about mul_mont ∗) }

let pow_mont (n r:pos) (d:int{r ∗ d % n = 1}) (aM:nat_mod n) (b:nat) : nat_mod n =
  pow (mk_nat_mont_cm n r d) aM b
```

The correspondence between Montgomery and modular exponentiations can be proved by induction on the exponent b with the following induction hypothesis.

```
val pow_mont_is_pow_mod (n r:pos) (d:int{r ∗ d % n = 1}) (aM:nat_mod n) (b:nat) :
  Lemma (pow_mont n r d aM b == pow_mod #n (aM ∗ d % n) b ∗ r % n)
```

5.7.4 Elliptic Curve Scalar Multiplication

Elliptic curve scalar multiplication is the repeated application of point addition to a point on the curve. Here, we consider the twisted Edwards curve "edwards25519" standardized in IETF RFC7448 [77]:

$$E = \{(x, y) \in \mathbb{Z}_p \times \mathbb{Z}_p \text{ s.t. } -x^2 + y^2 = 1 + d \cdot x^2 \cdot y^2\}, \text{ where}$$
$$d = -\frac{121665}{121666} \in \mathbb{Z}_p \text{ is a non-square in } \mathbb{Z}_p \text{ and } p = 2^{255} - 19.$$

The set of points on edwards25519 together with the following point addition in affine coordinates forms an abelian group:

$$\mathsf{aff\_point\_add}\text{: } (x_1, y_1) + (x_2, y_2) = (\frac{x_1 \cdot y_2 + x_2 \cdot y_1}{1 + d \cdot x_1 \cdot x_2 \cdot y_1 \cdot y_2}, \frac{y_1 \cdot y_2 + x_1 \cdot x_2}{1 - d \cdot x_1 \cdot x_2 \cdot y_1 \cdot y_2}) = (x_3, y_3).$$

The identity element is $(0, 1)$. The inverse element of (x, y) is $(-x, y)$. As d is a non-square in $\mathbb{Z}_p$, the denominators $1 + d \cdot x_1 \cdot x_2 \cdot y_1 \cdot y_2$ and $1 - d \cdot x_1 \cdot x_2 \cdot y_1 \cdot y_2$ are non-zero for all points (x_1, y_1) and (x_2, y_2) on the curve.

Point addition for edwards25519 in extended twisted Edwards coordinates. One way to avoid computing a modular multiplicative inverse for point addition in affine coordinates is to use extended twisted Edwards coordinates [69]. In this system, each point (x, y) on the curve is represented as (X, Y, Z, T), which corresponds to the affine point $(\frac{X}{Z}, \frac{Y}{Z})$ with $T = \frac{X \cdot Y}{Z}$ and Z is non-zero.

Point addition of (X_1, Y_1, Z_1, T_1) and (X_2, Y_2, Z_2, T_2) in extended twisted Edwards coordinates can be computed as follows.

$$A \leftarrow (Y_1 - X_1) \cdot (Y_2 - X_2),\ B \leftarrow (Y_1 + X_1) \cdot (Y_2 + X_2),\ C \leftarrow 2 \cdot d \cdot T_1 \cdot T_2,$$
$$D \leftarrow 2 \cdot Z_1 \cdot Z_2,\ E \leftarrow B - A,\ F \leftarrow D - C,\ G \leftarrow D + C,\ H \leftarrow B + A,$$
$$X_3 \leftarrow E \cdot F,\ Y_3 \leftarrow G \cdot H,\ T_3 \leftarrow E \cdot H,\ Z_3 \leftarrow F \cdot G$$
$$\mathsf{ext\_point\_add}\text{: } (X_1, Y_1, Z_1, T_1) + (X_2, Y_2, Z_2, T_2) = (X_3, Y_3, Z_3, T_3) \tag{5.4}$$

The correspondence between point addition in extended twisted Edwards coordinates and affine coordinates is as follows.

$$\mathsf{to\_aff}\ (\mathsf{ext\_point\_add}\ (X_1, Y_1, Z_1, T_1)\ (X_2, Y_2, Z_2, T_2)) ==$$
$$\mathsf{aff\_point\_add}\ (\mathsf{to\_aff}\ (X_1, Y_1, Z_1, T_1))\ (\mathsf{to\_aff}\ (X_2, Y_2, Z_2, T_2)).$$

For presentation purposes, we skip showing formulas for point doubling in affine coordinates and their efficient implementation in extended twisted Edwards coordinates.

Scalar multiplication for edwards25519. As expected, we define scalar multiplication aff_scalar_mul in affine coordinates as the repeated application of the aff_point_add function to the curve point $P = (x, y)$. To compute it efficiently, we introduce the ext_scalar_mul function that invokes the ext_point_add function instead of aff_point_add. The ext_scalar_mul function, together with the composition of conversion functions from and to affine coordinates, results in a scalar multiplication aff_scalar_mul.

$$\begin{tikzcd}
(x, y, 1, x \cdot y) \arrow[r, "\text{ext\_scalar\_mul}"] & (X_1, Y_1, Z_1, T_1) \arrow[d, "\text{to\_aff}"] \\
P = (x, y) \arrow[u, "\text{to\_ext}"] \arrow[r, "\text{aff\_scalar\_mul}"'] & {[n]P = (\frac{X_1}{Z_1}, \frac{Y_1}{Z_1})}
\end{tikzcd}$$

As a prime p is known in advance, a multiplicative inverse of Z_1 modulo p, Z_1^{p-2} % p, can be computed using the addition-chain exponentiation method. Note that to_ext is defined in such way that to_aff (to_ext (x,y)) = (x, y), as an initial point P is usually given in affine coordinates.

Specifying scalar multiplication for edwards25519 in F* (high-level specification). The elliptic curve edwards25519 is birationally equivalent to Curve25519 and is defined over the same finite field $\mathbb{Z}_p$, where $p = 2^{255} - 19$. Therefore, we can share both specification and implementation for finite-field arithmetic operations between the two elliptic curves.

```
let prime : pos = pow2 255 - 19
let elem = nat_mod prime
let ( +% ) (x y:elem) = (x + y) % prime
let ( -% ) (x y:elem) = (x - y) % prime
let ( *% ) (x y:elem) = (x * y) % prime
let ( /% ) (x y:elem) = x *% pow_mod y (prime - 2)
```

Next, we define a type aff_point_c representing a point in affine coordinates on the elliptic curve edwards25519.

```
let aff_point = elem & elem
let d : elem = 37095705934669439343138083508754565189542113879843219016388785533085940283555
let on_curve ((x, y):aff_point) = y *% y -% x *% x = 1 +% d *% (x *% x) *% (y *% y)
let aff_point_c = p:aff_point{on_curve p}
```

Finally, we instantiate a cm type with the identity element one = (0, 1) and the monoid operation mul = aff_point_add. The aff_scalar_mul function corresponds to the definition of scalar multiplication in affine coordinates.

```
let aff_point_add ((x1, y1):aff_point_c) ((x2, y2):aff_point_c) : aff_point_c =
  let x3 = (x1 *% y2 +% y1 *% x2) /% (1 +% d *% (x1 *% x2) *% (y1 *% y2)) in
  let y3 = (y1 *% y2 +% x1 *% x2) /% (1 -% d *% (x1 *% x2) *% (y1 *% y2)) in
  (x3, y3) (* omitted: proof that (x3, y3) is on curve *)

let mk_ed25519_cm: cm aff_point_c =
  { one = (0, 1); mul = aff_point_add; (* omitted: properties about aff_point_add *) }

let aff_scalar_mul (n:nat) (p:aff_point_c) : aff_point_c =
  pow mk_ed25519_cm p n
```

Specifying monoid operation replacement with a concrete implementation in F* (high-level specification). As discussed before, point addition in extended twisted Edwards coordinates is more efficient than in affine ones as it does not require a modular multiplicative inverse. Consequently, we encode the formulas from (5.4) and prove their correspondence to the point addition aff_point_add in affine coordinates.

```
let ext_point = elem & elem & elem & elem
let to_aff ((_X, _Y, _Z, _T):ext_point) : aff_point = _X /% _Z, _Y /% _Z
let is_ext ((_X, _Y, _Z, _T):ext_point) = _T = _X *% _Y /% _Z ∧ _Z ≠ 0
let ext_point_c = p:ext_point{is_ext p ∧ on_curve (to_aff p)}

val ext_point_add (p q:ext_point_c) : r:ext_point_c{to_aff r == aff_point_add (to_aff p) (to_aff q)}
```

Instead of proving all the properties about the ext_point_add function required for constructing a commutative monoid, we introduce a concrete_ops type reflecting the mapping from our efficient implementation of the arithmetic operations, “multiplication” c_mul and “squaring” c_sqr, to the commutative monoid operation mul.

```
type to_cm (t:Type) = {
  a_spec: Type;
  comm_monoid: cm a_spec;
  refl: x:t → a_spec; }

type concrete_ops (t:Type) = {
  to: to_cm t;
  c_one: one:t{to.refl one == to.comm_monoid.one};
  c_mul: x:t → y:t → xy:t{to.refl xy == to.comm_monoid.mul (to.refl x) (to.refl y)};
  c_sqr: x:t → xx:t{to.refl xx == to.comm_monoid.mul (to.refl x) (to.refl x)}; }
```

As before, we can obtain any implementation of exponentiation by providing the corresponding instance k of the concrete_ops type. Each of these implementations guarantees that the transformed result k.to.refl r is the same as we would perform an exponentiation pow on a commutative monoid k.to.comm_monoid, an element k.to.refl a and an exponent b. For example, the concrete_pow function has the following signature.

```
val concrete_pow: #t:Type → k:concrete_ops t → a:t → b:nat → Pure t (requires ⊤)
  (ensures λ r → k.to.refl r == pow k.to.comm_monoid (k.to.refl a) b)
```

For edwards25519 scalar multiplication, we instantiate the concrete_ops type with to.refl = to_aff, to.comm_monoid = mk_ed25519_cm, c_mul = ext_point_add and c_one = (0, 1, 1, 0):

```
let ext_scalar_mul (n:nat) (p:ext_point_c): r:ext_point_c{to_aff r = aff_scalar_mul n (to_aff p)} =
  concrete_pow mk_ed25519_concrete_ops p n
```

As a result, we immediately obtain the functional correctness proof for the ext_scalar_mul function.

5.8 Verifying Applications

5.8.1 Finite-Field Diffie-Hellman

In order to provide forward secrecy, TLS 1.3 [109] offers named finite field and elliptic curve groups for key exchange. Here, we consider Finite-Field Diffie-Hellman Ephemeral (FFDHE) groups with parameters defined in IETF RFC7919 [61].

Let us briefly recap the FFDHE key exchange. First, both parties need to agree on the parameters, a large prime number p ($2048-8192$ bits) and a generator g ($=2$). Then, each party generates a secret key $1 < sk < p-1$, computes the public key pk as modular exponentiation $g^{sk} \% p$ and sends the result pk to the other party.

Client		Server
generates sk_C		generates sk_S
$pk_C = g^{sk_C} \% p$		$pk_S = g^{sk_S} \% p$
	$\xrightarrow{pk_C}$	
	$\xleftarrow{pk_S}$	
$ss_C = pk_S^{sk_C} \% p$		$ss_S = pk_C^{sk_S} \% p$

When the other party's public key pk_O is received, each party computes the final modular exponentiation $pk_O^{sk} \% p$ if $1 < pk_O < p-1$. Due to the properties of modular arithmetic and exponentiation, both parties obtain the same result, the so-called shared secret:

$$\begin{aligned}
&ss_C = pk_S^{sk_C} \% p = (g^{sk_S} \% p)^{sk_C} \% p = g^{sk_S \cdot sk_C} \% p \\
&ss_S = pk_C^{sk_S} \% p = (g^{sk_C} \% p)^{sk_S} \% p = g^{sk_C \cdot sk_S} \% p \\
&\Rightarrow ss_C = ss_S.
\end{aligned}$$

The FFDHE key exchange uses modular exponentiation to compute the public key and shared secret. In both cases, the exponent is a secret value, so a constant-time implementation for modular exponentiation is required. Note that to speed up the computations, a fixed-base method can be applied to perform the first exponentiation, $g^{sk} \% p$.

5.8.2 RSA-PSS

The RSA-PSS probabilistic signature scheme is standardized in IETF RFC8017 [90] and is one of the signature algorithms included in TLS 1.3 [109]. As any RSA-based cryptosystem, it uses an RSA key pair, consisting of integers n, e, and d such that

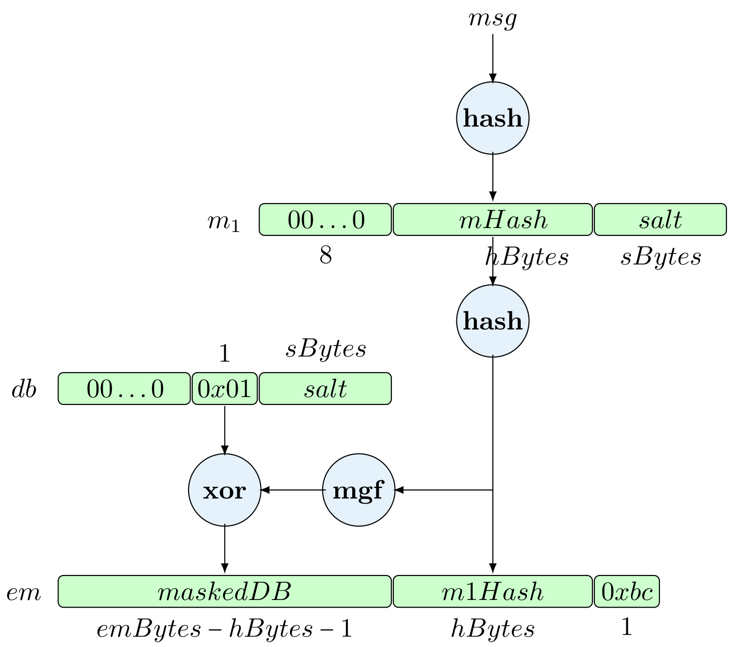
$$\forall\ 0 \leq m < n,\ m^{e \cdot d} \% n = m,\ \text{where}$$

— a modulus n is a product of two large prime numbers, p and q

— a public exponent e is chosen such that $1 < e < \phi(n) = (p-1) \cdot (q-1)$ and e and $\phi(n)$ are coprime

— a secret exponent d is the multiplicative inverse of e modulo $\phi(n)$, i.e., $e \cdot d \% \phi(n) = 1$.

Let us first recall a textbook RSA signature algorithm. A signing algorithm **rsa-sign** takes a secret key $sk = (n, e, d)$ and a message $0 \leq m < n$ and yields the signature s, calculated as a modular exponentiation $m^d \% n$. A signature verification **rsa-verify** algorithm checks whether a signature $s < n$ is valid for a given message m under the signer's public key $pk = (n, e)$.



em is encoded as a big-endian integer
$em = em \ \% \ 2^{emBits}$, where $emBits = modBits - 1 \iff$
if $emBits \ \% \ 8 > 0$ then $em.[0] = em.[0] \ \% \ 2^{emBits \ \% \ 8}$

Figure 5.11 – Encoding operation for RSA-PSS.

rsa-sign $(sk, m) \to s$	**rsa-verify** $(pk, m, s) \to$ valid/invalid
$sk = (n, e, d)$	$pk = (n, e)$
$s = m^d \ \% \ n$	$m' = s^e \ \% \ n =? = m$

For a valid signature s, the **rsa-verify** function can successfully validate the equality:

$$m' = s^e \ \% \ n = (m^d \ \% \ n)^e \ \% \ n = m^{d \cdot e} \ \% \ n = m.$$

RSA-PSS combines the above algorithm with the encoding method **pss-encode** depicted in Figure 5.11.

rsa-pss-sign $(sk, msg) \to s$	**rsa-pss-verify** $(pk, msg, s) \to$ valid/invalid
modBits is the size of the modulus n, i.e., $2^{modBits-1} < n < 2^{modBits}$	
$sk = (n, e, d)$	$pk = (n, e)$
$em =$ **pss-encode**$(msg, modBits - 1)$	$em' = s^e \ \% \ n$
$s = em^d \ \% \ n$	**pss-verify**$(msg, em', modBits - 1)$

The **pss-encode** function takes an octet string *msg* as input and produces a big-endian integer *em* less than $2^{modBits-1}$ by construction, i.e., $em < n$. It is parametrized by the choice of a hash function **hash**, a mask generation function **mgf** and a length *sBytes* of a random value *salt*. The **pss-verify** function checks whether em' is a valid encoding of a message *msg* by restoring *salt* from em' (relying on the properties of the **xor** function) and comparing $m1Hash'$ with the recomputed value $m1Hash$ for *msg*.

In both algorithms, the main arithmetic operation is an exponentiation modulo a large number n, which is 2048 bits and larger. In the **rsa-pss-sign** function, the exponent d is a large secret integer, so a constant-time implementation is required. In the **rsa-pss-verify** function, the exponent e is usually a relatively small public value (e is often chosen as $2^{16}+1$, which takes only 17 bits); thus, a simple binary method can be used.

5.8.3 Ed25519

The Ed25519 signature scheme is the Edwards-curve digital signature algorithm (EdDSA) using the twisted Edwards curve “edwards25519” birationally equivalent to Curve25519. Both EdDSA and Ed25519 are standardized in IETF RFC8032 [70].

Ed25519 uses keys and signatures significantly smaller than RSA3072, while providing the same level of security. The secret key *sk* is a 256-bit string, and the public key *pk* is a point on the elliptic curve “edwards25519” encoded as a 32-byte string (255 bits for y-coordinate and 1 bit for the sign of x-coordinate). The signature $(R \Vert s)$ is a 64-byte string that stores a point on the curve R followed by a little-endian integer s modulo q, where the prime $q = 2^{252} + 27742317777372353535851937790883648493$ is the order of the prime-order subgroup of the elliptic curve group defined in §5.7.4.

ed25519-secret-to-public	**ed25519-sign**	**ed25519-verify**
$(sk) \to pk$	$((sk, pk), msg) \to (R \Vert s)$	$(pk, msg, (R \Vert s)) \to$ valid/invalid
$(h_0, h_1, \ldots, h_{511}) = \mathbf{SHA512}(sk)$ $a = 2^{254} + \sum_{i=3}^{253} 2^i \cdot h_i$ $pk = [a]G$	$(h_0, h_1, \ldots, h_{511}) = \mathbf{SHA512}(sk)$ $a = 2^{254} + \sum_{i=3}^{253} 2^i \cdot h_i$ $r = \mathbf{SHA512}(h_{256} \Vert \ldots \Vert h_{511} \Vert msg) \ \% \ q$ $R = [r]G$ $h = \mathbf{SHA512}(R \Vert pk \Vert msg) \ \% \ q$ $s = (r + a \cdot h) \ \% \ q$	$h = \mathbf{SHA512}(R \Vert pk \Vert msg) \ \% \ q$ $[s]G =? = R + [h]pk$

The **ed25519-secret-to-public** function derives the public key *pk* from the secret key *sk*. The **ed25519-sign** function takes as input secret key *sk*, public key *pk* and octet string *msg* and outputs a signature $(R \Vert s)$. $\Vert$ denotes the concatenation of two strings. The **ed25519-verify** function checks whether a signature $(R \Vert s)$ is valid for a message *msg* under the signer’s public key *pk*, using the cofactorless verification equation. The difference with the cofactored one is discussed in [41].

In Ed25519, the main arithmetic operations are scalar multiplication and reduction modulo q. Both **ed25519-secret-to-public** and **ed25519-sign** functions perform scalar multiplication with the fixed base point G and a secret scalar, so a constant-time implementation is required. The **ed25519-verify** function computes a double scalar multiplication with fixed base point G and variable point *pk* if we rewrite the verification equation as follows: $[s]G - [h]pk =? = R$. Reduction modulo q can be calculated using the Barrett reduction [24] or Almost Montgomery reduction from §5.6.3.

	Our library	Fiat-Crypto [56]	WhyMP [84]
Description			
Verification tool	F*	Coq	Why3
Constant-time code	✓	✓	
Bignum representation	packed and unpacked (*)		2^{64}
Bignum size	arbitrary	fixed, ~ 500 bits	arbitrary
Integer Arithmetic			
Addition/Subtraction	✓		✓
Multiplication	✓		✓
Squaring	✓		
Modular Arithmetic			
Addition/Subtraction	✓	✓	
Multiplication	✓	✓	
Squaring	✓	✓	
Exponentiation	✓		✓
ECC			
Scalar multiplication	✓		

Table 5.1 – Coverage of the algorithms in portable verified bignum libraries.
(*) Our "Exponentiation" module is independent of the underlying bignum representation, whereas integer and modular arithmetic uses a packed representation, i.e., 2^{32} and 2^{64}.

5.9 Evaluation

5.9.1 Bignum Library Features

Table 5.1 reports the coverage of the algorithms required for RSA-like public-key and elliptic-curve cryptography from the following three projects: Fiat-Crypto [56], WhyMP [84] and our newly developed library.

Fiat-Crypto is a compiler that generates correct-by-construction C code of field arithmetic operations for a given prime of several hundred bits in size [1]. The compiler supports packed and unpacked representations for large numbers and provides modular-reduction optimizations such as Solinas and Montgomery arithmetic algorithms. All the formal reasoning is done in Coq.

WhyMP is an arbitrary-precision integer arithmetic library formally verified in Why3 and extracted to C. It uses a packed 64-bit representation for bignums and provides an efficient implementation for integer addition, subtraction, multiplication, division, square root, and modular exponentiation that mimics the algorithms from the GNU Multiple Precision Arithmetic Library (GMP). WhyMP verifies the sliding-window method [74] using Montgomery arithmetic for modular exponentiation and two variants of Toom-Cook [121, 45] for multiplication. However, the generated C code is not constant-time.

5.9.2 Run-Time Performance

We benchmarked each algorithm on a Dell XPS13 laptop with an Intel Kaby Lake i7-7560 running 64-bit Ubuntu Linux. All code was compiled with GCC-9 and CLANG-13 with flags `-O3`

[1] `https://github.com/mit-plv/fiat-crypto/issues/851`

Algorithm	Function	Exponentiation a^b	b is secret	Exponentiation methods (*)	Modulus	Modulus size in bits
FFDHE	secret-to-public shared-secret	g^{sk} % p pk^{sk} % p	✓ ✓	fixed base generic	p is a prime	2048 – 8192
RSA-PSS	sign verify	m^d % n s^e % n	✓	generic generic	$n = p \cdot q$, p and q are primes	2048 and larger
Ed25519	secret-to-public sign verify	$[a]G$ $[r]G$ $[s]G - [h]pk$	✓ ✓	fixed base fixed base double exp + fixed base	p is a prime $p = 2^{255} - 19$	255

Table 5.2 – Exponentiation in FFDHE, RSA-PSS and Ed25519
(*) Our library does not provide efficient methods for fixed-base exponentiation

Implementation	Compiler	2048	3072	4096	6144	8192
openssl-asm	gcc-9	6785	21509	50414	173646	411168
gmp-asm	gcc-9	8554	27121	62724	207042	486562
openssl-no-mulx	gcc-9	10613	34773	82075	279073	670069
hacl-star	clang-13	15969	51940	116838	381314	878264
openssl-portable	clang-13	39055	113443	263119	828745	1862540
gmp-portable	gcc-9	47283	149781	425988	1442425	3388961

Table 5.3 – KBENCH9000 Benchmarks for constant-time modular exponentiation a^b % n, where a, b and n are bignums of the same length. Measurements are in cycles (thousands) for input lengths ranging from 2048 to 8192 bits, obtained as the median of 10000 runs.

`-march=native -mtune=native`. We report the number of CPU cycles per operation obtained as the median of 10000 runs using a user-space version of KBENCH9000 [52]. Table 5.2 summarizes the requirements for exponentiation needed for both RSA-PSS and Ed25519 signature schemes and Finite-Field Diffie-Hellman.

We start from constant-time modular generic exponentiation used to obtain the RSA-PSS signature and shared secret in FFDHE. Figure 5.3 measures the performance of our implementation against (1) OpenSSL [2] compiled with assembly enabled (openssl-asm) and disabled (`no-asm`, openssl-portable), with the BMI2 and ADX extensions disabled [3] (openssl-no-mulx); (2) GNU Multiple Precision Arithmetic Library [4] (GMP) with assembly enabled (gmp-asm) and disabled (`-disable-assembly`, gmp-portable). The results show that our code is 2-3× faster than the portable C implementations but 2-3× slower than the targeted assembly implementations. To close the gap, we aim to replace some of the arithmetic functions, e.g., multiplication and squaring, with a verified assembly from the Vale [36, 59] project as soon as they become available.

Table 5.4 reports on the performance of our RSA-PSS and Ed25519 signature algorithms. For the RSA-PSS signature verification algorithm, we choose the public exponent e as $2^{16}+1$ for all key sizes ranging from 2048 to 8192 bits. For this algorithm, our portable C implementation is 2× slower than the targeted assembly implementation from OpenSSL. We also compared our Ed25519 code with fast C implementation from LibSodium [5]. Since our library does not yet support efficient methods for fixed-base exponentiation, our code is 2.38× and 1.17× slower than

[2] `https://github.com/openssl/openssl`, commit b756626

[3] `https://www.openssl.org/docs/manmaster/man3/OPENSSL_ia32cap.html`

[4] `https://gmplib.org`, version 6.2.1

[5] `https://github.com/jedisct1/libsodium`, commit 89943bd

		RSA-PSS										Ed25519	
		2048		3072		4096		6144		8192			
Implementation	Compiler	sign	verify	sign	verify	sign	verify	sign	verify	sign	verify	sign	verify
HACL*	gcc-9	22089	168	69223	340	140932	581	442459	1199	1013702	2066	236	246
	clang-13	16157	141	51596	301	117922	494	378420	1053	876962	1807	212	277
OpenSSL (asm)	gcc-9	6818	88	25103	156	55394	251	173675	510	411552	869	-	-
	clang-13	6796	88	22867	158	50375	249	173675	541	415362	874	-	-
LibSodium	gcc-9	-	-	-	-	-	-	-	-	-	-	89	210
	clang-13	-	-	-	-	-	-	-	-	-	-	89	211

Table 5.4 – KBENCH9000 Benchmarks for RSA-PSS and Ed25519. Measurements are in cycles (thousands) for processing a 1KB message, obtained as the median of 10000 runs.

Algorithm	High-level specification	Low-level specification+proofs	Low* code	Compiled C code
Exponentiation	863		1322	-
Bignum (mod arith)	7044		4049	2610
FFDHE	558	85	463	837
RSA-PSS	338	817	1236	777
Ed25519	122	1243	2758	1788
BignumQ		1884		
Total	12954		9828	6012

Table 5.5 – Coding and Verification Effort in lines of code and proof counted with `cloc`, discarding comments.

the signing and verifying algorithms, respectively, in LibSodium.

5.9.3 Verification Effort

Table 5.5 measures our coding and verification effort in lines of code and proof for our bignum library and applications built on top of it. All the algorithms except Ed25519 were newly developed.

We divide our verified bignum library into two parts, “Exponentiation” and “Bignum (mod arith)”. The former is independent of the underlying representation and is used to obtain implementations for scalar multiplication and modular exponentiation. The latter contains modular arithmetic operations and uses a packed representation in radix-2^{32} or radix-2^{64} for a bignum. It took the author several months to verify the bignum library.

In order to use modular exponentiation, one needs to either prove or provide additional checks for the following: (1) the modulus is an odd number greater than one; (2) the base is less than a modulus; (3) the given size of both modulus and exponent is correct. The above explains the proof overhead for RSA-PSS and FFDHE (for the latter, the constants p and g are shared between F* specification and Low* implementation). It took the author a few days to verify FFDHE and RSA-PSS.

For Ed25519, we do not include lines of code and proofs for SHA-512 and the field arithmetic library shared with Curve25519. Also, we separate the specialized BignumQ library (1884 lines), that implements arithmetic modulo the prime q, from the main development as it can be replaced with the generic bignum library with almost no additional proofs. To obtain a more efficient implementation for scalar multiplication, we ended up proving the correspondence between point addition in affine and extended twisted Edwards coordinates (1243 lines). This proof took the author several days.

5.10 Conclusions

We have extended the open-source verified cryptographic library HACL* with the generic bignum library and the RSA-PSS and FFDHE primitives used in TLS 1.3. For each algorithm, we have proved the following: memory safety, functional correctness against a high-level F* specification, and secret independence. To the best of our knowledge, our bignum library is the first formally verified *constant-time* library for *large* numbers (2048 – 8192 bits).

Our bignum library has been deployed in ElectionGuard[6]. Other deployments, such as in Mozilla's NSS cryptographic library (RSA-PSS), the Tezos blockchain (specialized bignum library) and hacspec[7] (generic bignum library), are left for the future.

We plan to enhance our bignum library with new algorithms such as Barrett reduction [24] and the fixed-base comb method [81] for exponentiation. Following our work on EverCrypt (Chapter 3) and HACL×N (Chapter 4), we aim to optimize our code by combining Low* with Vale [36, 59] (verified assembly) for some of the arithmetic functions and, independently, developing a vectorized bignum library.

[6] `https://github.com/microsoft/electionguard-cpp`, commit a9956fc

[7] `https://github.com/hacspec/hacspec`

Chapter 6

Conclusions

In this thesis, we formally verified several of the most important standardized cryptographic algorithms widely used today. We enhanced the open-source cryptographic library HACL* by more cryptographic primitives and fast implementations, as shown in Table 6.1.

Our verified scalar and vectorized implementations are compiled from one *generic* implementation written in Low* and significantly reduce the performance gap in comparison with the fastest hand-optimized assembly. Our multi-platform library of vectorized implementations, which can be compiled to platforms that support vector instructions (128-bit, 256-bit, and 512-bit vectors), includes the ChaCha20 encryption algorithm, the Poly1305 one-time MAC, and the SHA-2 and Blake2 families of hash functions. These new implementations were integrated into HACL* and made available through the API of the EverCrypt cryptographic provider that dynamically dispatches, based on runtime configuration, the most efficient implementation for the platform the code is running on. We also built verified vectorized implementations of HPKE and ChaCha20-Poly1305 for each supported platform.

Our verified portable and mixed assembly-C implementations for Curve25519 are compiled from one *generic* platform-independent code written in Low* that uses a shared interface between our two implementations for the low-level arithmetic. The first is written in Low* and generates portable C code that uses a radix-2^{51} representation. The second relies on verified Vale assembly and uses a radix-2^{64} representation. In 2019, to the best of our knowledge, our mixed assembly-C implementation of Curve25519 was the fastest verified or unverified implementation, whereas our portable C implementation was the second fastest verified implementation. Lately, several research projects [43, 68, 93] have explored various efficient techniques for the vectorization of the Montgomery ladder. Our verified vectorized bignum library that we built for Poly1305 can be used to verify these implementations.

Our generic bignum library compiles into 32-bit and 64-bit portable C implementations. The library supports the schoolbook algorithms for addition, subtraction and multiplication, along with more efficient algorithms for multiplication, namely, squaring and Karatsuba multiplication. We also expose an API for Montgomery arithmetic to efficiently compute a sequence of modular arithmetic operations to amortize the cost of converting from and to Montgomery form, for instance, in modular exponentiation. Our verified implementations of modular exponentiation and scalar multiplication are compiled from one *generic* implementation written in Low*. Our library was used to obtain verified implementations of the RSA-PSS signature scheme and Finite-

	Portable	Arm A64	Intel x64			
Algorithm	**C code**	**Neon**	**AVX**	**AVX2**	**AVX512**	**Vale**
AEAD						
ChaCha20-Poly1305	✓ [127] (+)	✓	✓	✓	✓	
AES-GCM						✓ [36]
Hashes						
SHA-224,256	✓ [127] (+)	✓	✓	✓	✓	✓ [36]
SHA-384,512	✓ [127] (+)	✓	✓	✓	✓	
Blake2s, Blake2b	✓	✓	✓	✓		
SHA3-224,256,384,512	✓					
HMAC and HKDF						
HMAC (SHA-2,Blake2)	✓ [127]	✓	✓	✓	✓	
HKDF (SHA-2,Blake2)	✓ [127]	✓	✓	✓	✓	
Elliptic Curve Cryptography						
Curve25519	✓ [127] (+)					✓
Ed25519	✓ [127] (+)					
P-256	✓					
Finite Field Cryptography						
FFDH	✓					
RSA-PSS	✓					
High-level APIs						
Box	✓ [127]					
HPKE	✓	✓	✓	✓	✓	✓
Post-Quantum Cryptography						
FrodoKEM	✓					

Table 6.1 – Supported algorithms and platforms for HACL*-v2 (2021).
Implementations marked with a (+) replaced prior C implementations from [127] (2017). The portable and platform-specific C implementations are composed with platform-specific Intel assembly code from Vale [36] (verified against the same specifications) to build the EverCrypt provider (Chapter 3). Vale assembly relies on AES-NI for AES-GCM, SHA-EXT for SHA-2, and ADX+BMI2 instructions for Curve25519.

Field Diffie-Hellman, with our code for constant-time modular exponentiation being 2-3× faster than the state-of-the-art C implementations in GMP and OpenSSL, but 2-3× slower than their targeted implementations. We also improved the performance of Ed25519 by using the fixed-window and double fixed-window methods for scalar multiplication.

Limitations of our work. While our guiding principle in this work was to *verify once* but *compile* and *specialize* many times, writing fast, secure, and correct code still requires considerable manual effort. As demonstrated in the USUBA [85] and Fiat-Crypto [56] projects, automatic generation of constant-time formally-verified cryptographic code is feasible (so far) for a small portion of the algorithm. For example, field arithmetic operations for a given prime of several hundred bits in size (excluding modular exponentiation or scalar multiplication) or a bitsliced implementation of a block cipher (excluding modes of operation) can be obtained using their tools.

On one hand, obtaining verified vectorized implementations from one generic implementation for platforms that support 128-bit, 256-bit, and 512-bit vector instructions reduces the programming and verification effort. On the other hand, compiled C code is not always as fast as the state-of-the-art hand-tuned assembly. For example, a more efficient technique [33], that halves the number of modular reductions, exists for computing polynomial evaluation in parallel that our library does not support.

In addition to memory safety and functional correctness, all our code is verified to be constant-time with respect to the assumptions discussed in §2.2. For example, our code is not constant-time on platforms where integer multiplication is variable-time. While our approach protects against certain timing side-channel attacks that rely on branching or memory access, it does not address other side-channel attacks, such as Differential Fault Analysis (DFA) or Differential Power Analysis (DPA). In the post-Spectre world, the constant-time programming model must also account for speculative execution in modern CPUs, as discussed in [27, 40], to provide formal guarantees of the absence of timing leaks in the code. We leave it for future work.

Future directions of our work. In this thesis, we built a formally-verified cryptographic library that supports many modern widely-used classical cryptographic primitives. The library can be further improved with regard to speed and supported platforms and algorithms.

In 2021, NIST announced the finalist candidates for lightweight and post-quantum cryptographic standards and is now preparing a future call for algorithms to be considered for multi-party threshold cryptographic standards. Many of these algorithms introduce new constructions, and, as with classical cryptography, the need for their constant-time and highly-optimized functionally-correct code is dire. For example, [66] exploits timing leakage in the ciphertext comparison step of the Fujisaki-Okamoto transformation in the official reference implementation of FrodoKEM to conduct a key-recovery timing attack.

We propose to use our methodology to verify the implementations of these standards. As a first case study, we have built verified portable C implementations for all versions of FrodoKEM, a lattice-based key encapsulation mechanism whose security relies on the hardness of the Learning with Errors problem. For this project, we can also reuse, for instance, the multiple input parallelism pattern from §4.3.2 to obtain a 4-way vectorized implementation of the SHA-3 hash function used in the official optimized implementation of FrodoKEM.

Bibliography

[1] curve25519-donna: Implementations of a fast Elliptic-Curve Diffie-Hellman primitive. `https://github.com/agl/curve25519-donna`.

[2] eBACS: ECRYPT Benchmarking of Cryptographic Systems – SUPERCOP. `https://bench.cr.yp.to/supercop.html`.

[3] OpenSSL. `https://github.com/openssl/openssl/commit/e8fb288`.

[4] Project Wycheproof. `https://github.com/google/wycheproof`.

[5] Signal Protocol Library for Java/Android. `https://github.com/signalapp/libsignal-protocol-java`.

[6] The sodium crypto library (libsodium). `https://github.com/jedisct1/libsodium`.

[7] Federal Information Processing Standards Publication 180-4: Secure hash standard (SHS). `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf`, 2015.

[8] BLAKE2b NEON suffers poor performance on ARMv8/Aarch64 with Cortex-A57. `https://github.com/weidai11/cryptopp/issues/367`, 2017.

[9] On the dangers of Intel's frequency scaling. `https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/`, 2017.

[10] WebAssembly 128-bit packed SIMD Extension. `https://github.com/WebAssembly/simd/blob/master/proposals/simd/SIMD.md`, 2020.

[11] Reynald Affeldt. On construction of a library of formally verified low-level arithmetic functions. In *Innovations in Systems and Software Engineering*, pages 59–77. Springer, 2013.

[12] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra Monads for Free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 515–529. ACM, 2017.

[13] Amal Ahmed, Deepak Garg, Cătălin Hriţcu, and Frank Piessens. Secure Compilation (Dagstuhl Seminar 18201). *Dagstuhl Reports*, pages 1–30, 2018.

[14] 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 Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1807–1823. ACM Press, October / November 2017.

[15] 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*, pages 965–982. IEEE Computer Society Press, May 2020.

[16] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Vincent Laporte, and Tiago Oliveira. Certified compilation for cryptography: Extended x86 instructions and constant-time verification. In Karthikeyan Bhargavan, Elisabeth Oswald, and Manoj Prabhakaran, editors, *INDOCRYPT 2020*, volume 12578 of *LNCS*, pages 107–127. Springer, Heidelberg, December 2020.

[17] Thorsten Altenkirch and Conor McBride. Generic Programming Within Dependently Typed Programming. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*. Springer, 2003.

[18] Andrew W. Appel. Verified Software Toolchain. In *Proceedings of the European Conference on Programming Languages and Systems (ESOP/ETAPS)*, pages 1–17. Springer, 2011.

[19] Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 1–31. ACM, 2015.

[20] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: Simpler, smaller, fast as MD5. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *ACNS 13*, volume 7954 of *LNCS*, pages 119–135. Springer, Heidelberg, June 2013.

[21] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. Cryptology ePrint Archive, Report 2019/1393, 2019. `https://eprint.iacr.org/2019/1393`.

[22] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. The Messaging Layer Security (MLS) Protocol (IETF Internet-Draft). `https://tools.ietf.org/html/draft-ietf-mls-protocol-12`, 2021.

[23] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption (IRTF Internet-Draft). `https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hpke-12`, 2021.

[24] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 311–323. Springer, Heidelberg, August 1987.

[25] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 1267–1279. ACM Press, November 2014.

[26] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. Cryptology ePrint Archive, Report 2019/926, 2019. `https://eprint.iacr.org/2019/926`.

[27] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. High-Assurance Cryptography in the Spectre Era. In *IEEE Symposium on Security and Privacy (SP)*, pages 788–805, 2021.

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

[29] David Benjamin. poly1305-x86.pl produces incorrect output. `https://mta.openssl.org/pipermail/openssl-dev/2016-March/006161`, 2016.

[30] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of OpenSSL HMAC. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 207–221. USENIX Association, August 2015.

[31] Daniel J. Bernstein. The poly1305-AES message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *FSE 2005*, volume 3557 of *LNCS*, pages 32–49. Springer, Heidelberg, February 2005.

[32] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Heidelberg, April 2006.

[33] Daniel J. Bernstein and Peter Schwabe. NEON crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 320–339. Springer, Heidelberg, September 2012.

[34] John Black, Martin Cochran, and Trevor Highland. A study of the MD5 attacks: Insights and improvements. In Matthew J. B. Robshaw, editor, *FSE 2006*, volume 4047 of *LNCS*, pages 262–277. Springer, Heidelberg, March 2006.

[35] Sascha Böhme and Tjark Weber. Fast LCF-Style Proof Reconstruction for Z3. In *Proceedings of Interactive Theorem Proving*, pages 179–194. Springer, 2010.

[36] 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 Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 917–934. USENIX Association, August 2017.

[37] Joppe W. Bos and Peter L. Montgomery. Montgomery arithmetic from a software perspective. Cryptology ePrint Archive, Report 2017/1057, 2017. `https://eprint.iacr.org/2017/1057`.

[38] Richard P. Brent and Paul Zimmermann. *Modern computer arithmetic.* Cambridge University Press, 2010.

[39] Billy Bob Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. Practical realisation and elimination of an ECC-related software bug attack. In Orr Dunkelman, editor, *CT-RSA 2012*, volume 7178 of *LNCS*, pages 171–186. Springer, Heidelberg, February / March 2012.

[40] Sunjay Cauligi, Craig Disselkoen, Klaus V. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-Time Foundations for the New Spectre Era.

In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 913–926. ACM, 2020.

[41] Konstantinos Chalkias, François Garillot, and Valeria Nikolaenko. Taming the many EdDSAs. Cryptology ePrint Archive, Report 2020/1244, 2020. `https://eprint.iacr.org/2020/1244`.

[42] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. Verifying Curve25519 Software. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 299–309. ACM, 2014.

[43] Hao Cheng, Johann Großschädl, Jiaqi Tian, Peter B. Rønne, and Peter Y.A. Ryan. High-throughput elliptic curve cryptography using AVX2 vector instructions. In *Selected Areas in Cryptography*, 2020.

[44] Tung Chou. Sandy2x: New Curve25519 speed records. In Orr Dunkelman and Liam Keliher, editors, *SAC 2015*, volume 9566 of *LNCS*, pages 145–160. Springer, Heidelberg, August 2016.

[45] Stephen A. Cook and Stål O. Aanderaa. On the minimum computation time of functions. *Transactions of the American Mathematical Society*, pages 291–314, 1969.

[46] Ivan Damgård. A design principle for hash functions. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 416–427. Springer, Heidelberg, August 1990.

[47] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *2017 IEEE Symposium on Security and Privacy*, pages 463–482. IEEE Computer Society Press, May 2017.

[48] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. A security model and fully verified implementation for the IETF QUIC record layer. Cryptology ePrint Archive, Report 2020/114, 2020. `https://eprint.iacr.org/2020/114`.

[49] Peter Dettman. Problems with field arithmetic. `https://github.com/armfazh/rfc7748_precomputed/issues/5`, 2018.

[50] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. Constructing Semantic Models of Programs with the Software Analysis Workbench. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 56–72. Springer, 2016.

[51] Jason A. Donenfeld. WireGuard: Next Generation Kernel Network Tunnel. In *NDSS*, pages 1–12, 2017.

[52] Jason A. Donenfeld. kBench9000 - simple kernel land cycle counter. `https://git.zx2c4.com/kbench9000/about/`, 2018.

[53] Jason A. Donenfeld. new 25519 measurements of formally verified implementations. `http://moderncrypto.org/mail-archive/curves/2018/000972.html`, 2018.

[54] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. Cryptology ePrint Archive, Report 2015/343, 2015. https://eprint.iacr.org/2015/343.

[55] Morris J. Dworkin. Recommendation for Block Cipher Modes of Operation: Methods and Techniques. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf, 2001.

[56] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy*, pages 1202–1219. IEEE Computer Society Press, May 2019.

[57] Kathleen Fisher, John Launchbury, and Raymond Richards. The HACMS Program: Using Formal Methods to Eliminate Exploitable Bugs. In *Philosophical Transactions A, Math Phys Eng Sci.*, 2017.

[58] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 328–343. ACM, 2017.

[59] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A verified, efficient embedding of a verifiable assembly language. In *Proceedings of the ACM on Programming Languages*, pages 1–30. ACM, 2019.

[60] Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Signed cryptographic program verification with typed CryptoLine. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1591–1606. ACM Press, November 2019.

[61] Daniel Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS) (IETF RFC 7919). https://tools.ietf.org/html/rfc7919, 2016.

[62] Shay Gueron. Intel® Advanced Encryption Standard (AES) New Instructions Set. https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf, 2010.

[63] Shay Gueron. Efficient software implementations of modular exponentiation. *Journal of Cryptographic Engineering*, 2(1):31–43, May 2012.

[64] Shay Gueron and Vlad Krasnov. Simultaneous hashing of multiple messages. Cryptology ePrint Archive, Report 2012/371, 2012. https://eprint.iacr.org/2012/371.

[65] Sean Gulley, Vinodh Gopal, Kirk Yap, Wajdi Feghali, Jim Guilford, and Gil Wolrich. Intel® SHA Extensions. https://www.intel.com/content/dam/develop/external/us/en/documents/intel-sha-extensions-white-paper-402097.pdf, 2013.

[66] Qian Guo, Thomas Johansson, and Alexander Nilsson. A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 359–386. Springer, Heidelberg, August 2020.

[67] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, 2014.

[68] Huseyin Hisil, Berkan Egrice, and Mert Yassi. Fast 4 way vectorized ladder for the complete set of montgomery curves. Cryptology ePrint Archive, Report 2020/388, 2020. `https://eprint.iacr.org/2020/388`.

[69] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards curves revisited. In Josef Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 326–343. Springer, Heidelberg, December 2008.

[70] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA) (IETF RFC 8032). `https://tools.ietf.org/html/rfc8032`, 2017.

[71] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. Mtac2: typed tactics for backward reasoning in Coq. In *Proceedings of the ACM on Programming Languages*, pages 1–31. ACM, 2018.

[72] Anatolii Alekseevich Karatsuba and Yu P. Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, pages 293–294. Russian Academy of Sciences, 1962.

[73] Anatolii Alexeevich Karatsuba. The complexity of computations. In *Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation*, pages 169–183. Providence, RI: American Mathematical Society, 1995.

[74] Cetin K. Koç. Analysis of sliding window techniques for exponentiation. *Computers & Mathematics with Applications*, 30(10):17–24, 1995.

[75] Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF) (IETF RFC 5869). `https://tools.ietf.org/html/rfc5869`, 2010.

[76] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191. ACM, 2014.

[77] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security (IETF RFC 7748). `https://tools.ietf.org/html/rfc7748`, 2016.

[78] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 348–370. Springer, 2010.

[79] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert – A Formally Verified Optimizing Compiler. In *Embedded Real Time Software and Systems (ERTS)*, 2016.

[80] Gaëtan Leurent and Thomas Peyrin. SHA-1 is a shambles - first chosen-prefix collision on SHA-1 and application to the PGP web of trust. Cryptology ePrint Archive, Report 2020/014, 2020. `https://eprint.iacr.org/2020/014`.

[81] Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 95–107. Springer, Heidelberg, August 1994.

[82] Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying arithmetic in cryptographic C programs. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 552–564. IEEE, 2019.

[83] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hriţcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *28th European Symposium on Programming (ESOP)*, pages 30–59. Springer, 2019.

[84] Guillaume Melquiond and Raphaël Rieu-Helft. WhyMP, a formally verified arbitrary-precision integer library. In *Proceedings of the 45th International Symposium on Symbolic and Algebraic Computation*, pages 352–359, 2020.

[85] Darius Mercadier. *Usuba, Optimizing Bitslicing Compiler*. PhD thesis, Sorbonne Université (France), 2020.

[86] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 218–238. Springer, Heidelberg, August 1990.

[87] Microsoft. Cryptographic API: Next generation - Windows Applications. `https://docs.microsoft.com/en-us/windows/win32/seccng/cng-portal`.

[88] Peter L. Montgomery. Modular multiplication without trial division. In *Mathematics of computation*, pages 519–521, 1985.

[89] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. In *Mathematics of computation*, pages 243–264, 1987.

[90] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS#1: RSA Cryptography Specifications Version 2.2 (IETF RFC 8017). `https://tools.ietf.org/html/rfc8017`, 2016.

[91] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. ŒUf: Minimizing the Coq Extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 172–185. ACM, 2018.

[92] Magnus O. Myreen and Gregorio Curello. Proof pearl: A verified bignum implementation in x86-64 machine code. In *International Conference on Certified Programs and Proofs*, pages 66–81. Springer, 2013.

[93] Kaushik Nath and Palash Sarkar. Efficient 4-way vectorizations of the montgomery ladder. Cryptology ePrint Archive, Report 2020/378, 2020. `https://eprint.iacr.org/2020/378`.

[94] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols (IETF RFC 7539). `https://tools.ietf.org/html/rfc7539`, 2015.

[95] Thomaz Oliveira, Julio Cesar López-Hernández, Hüseyin Hisil, Armando Faz-Hernández, and Francisco Rodríguez-Henríquez. How to (pre-)compute a ladder - improving the per-

formance of X25519 and X448. In Carlisle Adams and Jan Camenisch, editors, *SAC 2017*, volume 10719 of *LNCS*, pages 172–191. Springer, Heidelberg, August 2017.

[96] OpenSSL. Chase overflow bit on x86 and ARM platforms. `https://github.com/openssl/openssl/commit/dc3c506`.

[97] OpenSSL. Don't break carry chains. `https://github.com/openssl/openssl/commit/4b8736a`.

[98] OpenSSL. Don't loose 59-th bit. `https://github.com/openssl/openssl/commit/bbe9769`.

[99] OpenSSL. Vulnerabilities. `https://www.openssl.org/news/vulnerabilities.html`.

[100] OpenSSL Team. OpenSSL. `http://www.openssl.org/`.

[101] Erdinc Ozturk, James Guilford, Vinodh Gopal, and Wajdi Feghali. New Instructions Supporting Large Integer Arithmetic on Intel® Architecture Processors. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf`, 2012.

[102] Adam Petcher and Greg Morrisett. The Foundational Cryptography Framework. In *Principles of Security and Trust*, pages 53–72. Springer, 2015.

[103] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. HACLxN: Verified generic SIMD crypto (for all your favourite platforms). In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 899–918. ACM Press, November 2020.

[104] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying Arithmetic Assembly Programs in Cryptographic Primitives. In *Conference on Concurrency Theory (CONCUR)*, pages 4:1–4:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[105] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally verified cryptographic web applications in WebAssembly. In *2019 IEEE Symposium on Security and Privacy*, pages 1256–1274. IEEE Computer Society Press, May 2019.

[106] Jonathan Protzenko and Son Ho. Zero-cost meta-programmed stateful functors in F*. `https://arxiv.org/pdf/2102.01644.pdf`, 2021.

[107] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy*, pages 983–1002. IEEE Computer Society Press, May 2020.

[108] 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*. In *Proceedings of the ACM on Programming Languages (PACMPL)*, pages 17:1–17:29. ACM, 2017.

[109] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3 (IETF RFC 8446). https://tools.ietf.org/html/rfc8446, 2018.

[110] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. Encrypted Server Name Indication for TLS 1.3 (IETF Internet-Draft). https://tools.ietf.org/html/draft-ietf-tls-esni-13, 2020.

[111] Markku-Juhani Saarinen and Jean-Philippe Aumasson. The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC) (IETF RFC 7693). https://tools.ietf.org/html/rfc7693, 2015.

[112] Marc Schoolderman, Jonathan Moerman, Sjaak Smetsers, and Marko van Eekelen. Efficient Verification of Optimized Code: Correct High-speed X25519. https://eprint.iacr.org/2021/415.pdf, 2021.

[113] Peter Schwabe, Benoît Viguier, Timmy Weerwag, and Freek Wiedijk. A Coq proof of the correctness of X25519 in TweetNaCl. https://cryptojedi.org/papers/tweetverif-20210208.pdf, 2021.

[114] Robert Seacord. Implement abstract data types using opaque types. https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87151966, 2018.

[115] Tim Sheard and Simon Peyton Jones. Template Meta-Programming for Haskell. pages 60–75. ACM, 2002.

[116] Laurent Simon, David Chisnall, and Ross Anderson. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *IEEE European Symposium on Security and Privacy*, pages 1–15. IEEE, 2018.

[117] Jerome A. Solinas. Efficient arithmetic on Koblitz curves. In *Towards a Quarter-Century of Public Key Cryptography*, pages 125–179. Springer, 2000.

[118] 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 *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270. ACM, 2016.

[119] The Coq Development Team. The Coq Proof Assistant. http://coq.inria.fr.

[120] Aaron Tomb. Automated Verification of Real-World Cryptographic Implementations. In *IEEE Security and Privacy*, pages 26–33. IEEE, 2016.

[121] Andrei L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, pages 714–716, 1963.

[122] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1973–1987. ACM Press, October / November 2017.

[123] Tao Xie, Fanbao Liu, and Dengguo Feng. Fast collision attack on MD5. Cryptology ePrint Archive, Report 2013/170, 2013. https://eprint.iacr.org/2013/170.

[124] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294. ACM, 2011.

[125] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedTLS HMAC-DRBG. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2007–2020. ACM Press, October / November 2017.

[126] Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. A verified extensible library of elliptic curves. In Michael Hicks and Boris Köpf, editors, *CSF 2016 Computer Security Foundations Symposium*, pages 296–309. IEEE Computer Society Press, 2016.

[127] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1789–1806. ACM Press, October / November 2017.

Appendices

A Secret Independence

We restate and discuss the *Secret Independence for Hybrid Low*/Vale programs* theorem proven by Fromherz et al. [59].

Constant-time properties of Low* programs are proven using a syntactic, type-based analysis for secure information flow. In particular, Low* programmers can designate specific inputs in their programs as secret and give them abstract types to force them to be manipulated only through a given interface of constant-time primitives, e.g., only additions, multiplications, and bitwise operations, but no comparisons or divisions. A meta-theorem about Low* establishes that such well-typed programs produce execution traces (sequences of instructions, including branches taken and memory addresses accessed) that are independent of their secret inputs.

In contrast, Vale programs are analyzed for leakage using a taint analysis programmed and proven correct with F*. The taint analysis is proven to only accept programs whose execution traces (which also includes the sequences of instructions executed and the memory addresses they access) do not depend on inputs that are designated as secret.

To compose the two properties, Fromherz et al. [59] define an extended Low* syntax and semantics, which includes trace-instrumented atomic computation steps for the Vale program fragments executed. This extended language has an execution semantics that produces traces that are concatenations of Low* and Vale execution traces. The theorem (stated below) shows that the two notions of secret independence compose well, establishing that well-typed Low* programs that call into leakage-free Vale programs produce combined traces that are also secret independent.

The theorem is a bisimulation between runs of a pair of related, well-typed Vale-extended Low* runtime configurations (H_1, e_1) and (H_2, e_2), showing that they either both step to related configurations maintaining their invariants and producing identical traces, or they are both terminal configurations.

Theorem: Secret Independence. Given well-typed extended Low* configurations (H_1, e_1) and (H_2, e_2), where $\Gamma \vdash (H_1, e_1) : \tau$, $\Gamma \vdash (H_2, e_2) : \tau$, $H_1 \equiv_\Gamma H_2$ and $e_1 \equiv_\Gamma e_2$, and a secret-independent implementation of the secret interface P_s, either both the configurations cannot reduce further, or $\exists\, \Gamma' \supseteq \Gamma$ s.t. $P_s \vdash (H_1, e_1) \rightarrow^+_{\ell_1} (H'_1, e'_1)$, $P_s \vdash (H_2, e_2) \rightarrow^+_{\ell_2} (H'_2, e'_2)$, $\Gamma' \vdash (H'_1, e'_1) : \tau$, $\Gamma' \vdash (H'_2, e'_2) : \tau$, $\ell_1 = \ell_2$, $H'_1 \equiv_{\Gamma'} H'_2$, and $e'_1 \equiv_{\Gamma'} e'_2$,

B Code snippets for C and header files from EverCrypt

Figures B.1 and B.2 illustrate the C code extracted from EverCrypt.

C Analysis of Vulnerabilities in OpenSSL's Cryptographic Provider

In Table C.1, we summarize all 24 vulnerabilities reported against OpenSSL's cryptographic provider, `libcrypto`, between January 2016 and July 2019 [99]. Each row explains what the vulnerability affected and which of EverCrypt's properties (memory safety, functional correctness,

```
// From the C implementation of hashes.
static void Hash__hash__256(uint8__t *input, uint32__t input__len, uint8__t *dst)
{
  uint32__t s[8U] = {
    0x6a09e667U, 0xbb67ae85U, 0x3c6ef372U, 0xa54ff53aU,
    0x510e527fU, 0x9b05688cU, 0x1f83d9abU, 0x5be0cd19U
  };
  uint32__t blocks__n = input__len / (uint32__t)64U;
  uint32__t blocks__len = blocks__n * (uint32__t)64U;
  uint8__t *blocks = input;
  uint32__t rest__len = input__len - blocks__len;
  uint8__t *rest = input + blocks__len;
  bool has__shaext1 = AutoConfig2__has__shaext();
  if (has__shaext1) {
    uint64__t n1 = (uint64__t)blocks__n;
    uint64__t scrut = sha256__compress(s, blocks, n1,
      Hash__Core__SHA2__Constants__k224__256);
  }
  else {
    Hash__SHA2__compress__many__256(s, blocks, blocks__n);
  }
  Hash__compress__last__256(s, (uint64__t)blocks__len, rest, rest__len);
  Hash__Core__SHA2__finish__256(s, dst);
}
```

Figure B.1 – A representative snippet of the C code generated for EverCrypt. This is a specialized instance of the generic Merkle-Damgård construction for a compression function that multiplexes between Low* and Vale.

```
// This type is shared with specifications.
#define Hash_SHA2_224 0
#define Hash_SHA2_256 1
#define Hash_SHA2_384 2
#define Hash_SHA2_512 3
#define Hash_SHA1 4
#define Hash_MD5 5
typedef uint8_t Hash_hash_alg;

// Interface for hashes, block-aligned input data
struct Hash_state_s_s;
typedef struct Hash_state_s_s Hash_state_s;

Hash_state_s *Hash_create_in(Hash_hash_alg a);
void Hash_init(Hash_state_s *s);
void Hash_compress(Hash_state_s *s, uint8_t *block1);
void Hash_compress_many(Hash_state_s *s, uint8_t *blocks, uint32_t len1);
void Hash_compress_last(Hash_state_s *s, uint8_t *last1, uint64_t total_len);
void Hash_finish(Hash_state_s *s, uint8_t *dst);
void Hash_free(Hash_state_s *s);
void Hash_copy(Hash_state_s *s_src, Hash_state_s *s_dst);

void Hash_hash(Hash_hash_alg a, uint8_t *dst, uint8_t *input, uint32_t len1);

// Incremental interface for hashes, arbitrary input data,
// relies on an internal array (not covered in the thesis).
typedef struct Hash_Incremental_state_s {
  Hash_state_s *hash_state;
  uint8_t *buf;
  uint64_t total_len;
} Hash_Incremental_state;

Hash_Incremental_state Hash_Incremental_create_in(Hash_hash_alg a);
Hash_Incremental_state Hash_Incremental_compress(
  Hash_hash_alg a, Hash_Incremental_state s, uint8_t *data, uint32_t len1);
void Hash_Incremental_finish(Hash_hash_alg a, Hash_Incremental_state s, uint8_t *dst);

// Specialized versions of HMAC
void HMAC_compute_sha1(uint8_t *mac, uint8_t *key,
  uint32_t keylen, uint8_t *data, uint32_t datalen);
void HMAC_compute_sha2_256(uint8_t *mac, uint8_t *key,
  uint32_t keylen, uint8_t *data, uint32_t datalen);
void HMAC_compute_sha2_384(uint8_t *mac, uint8_t *key,
  uint32_t keylen, uint8_t *data, uint32_t datalen);
void HMAC_compute_sha2_512(uint8_t *mac, uint8_t *key,
  uint32_t keylen, uint8_t *data, uint32_t datalen);

// Agile HMAC
bool HMAC_is_supported_alg(Hash_hash_alg x);
void HMAC_compute(Hash_hash_alg a, uint8_t *mac,
  uint8_t *key, uint32_t keylen, uint8_t *data, uint32_t datalen);
```

Figure B.2 – A representative subset of the EverCrypt API. This file is taken as-is from the output of our toolchain (i.e., from the `.h` file). The file was edited only to remove some module name prefixes to make the code more compact.

CVE	Severity	Vulnerability	Broken property	Prevented?
2019-1543	Low	improper IV handling	functional correctness	✓
2018-5407	Low	EC multiplication timing leak	side-channel resistance	✓
2018-0734	Low	bignum timing leak	side-channel resistance	✓
2018-0735	Low	bignum allocation leak	side-channel resistance	✓
2018-0737	Low	bignum timing leak	side-channel resistance	✓
2018-0733	Moderate	incorrect hand-written assembly	functional correctness	✓
2017-3738	Low	buffer overflow	functional correctness	✓
2017-3736	Moderate	carry propagation bug	functional correctness	✓
2017-3733	High	inconsistent agility parameter	functional correctness	✓
2017-3732	Moderate	carry propagation bug	functional correctness	✓
2017-3731	Moderate	out of bounds access	memory safety	✓
2016-7055	Low	carry propagation bug	functional correctness	✓
2016-7054	High	incorrect memset	memory safety	✓
2016-6303	Low	integer overflow	functional correctness	✓
2016-2178	Low	cache timing leak	side-channel resistance	✓
2016-2177	Low	undefined behavior	memory safety	✓
2016-2107	High	missing bounds check	functional correctness	✓
2016-2106	Low	integer overflow	functional correctness	✓
2016-2105	Low	integer overflow	functional correctness	✓
2016-0705	Low	double free	memory safety	✓
2016-0704	Moderate	key recovery	side-channel resistance	✓
2016-0703	High	key recovery	side-channel resistance	✓
2016-0702	Low	cache timing leak	side-channel resistance	✓
2016-0701	High	load unsafe DH primes	misconfiguration	×

Table C.1 – Recent CVEs in `libcrypto` that would have been prevented by EverCrypt's methodology.

or timing-attack resistance) would have prevented it. The list covers both `libcrypto` algorithms that have been re-implemented in EverCrypt, and algorithms that, if implemented in EverCrypt, would be immune to those vulnerabilities. The only vulnerability EverCrypt's methodology might not have prevented is 2016-0701, in which OpenSSL added a feature to allow the loading of Diffie-Hellman primes from X9.42-style parameter files. The library trusted these primes to be safe, rather than checking them for safety.

D Performance Benchmarks

This appendix presents our performance measurements using two benchmarking frameworks across several machines:

Tables D.1 & D.4: a Dell XPS13 laptop, with an Intel Core i7-7560U (Kaby Lake, AVX2) CPU, running 64-bit Ubuntu Linux 18.04,

Tables D.2 & D.5: a Dell Precision workstation, with an Intel Xeon Gold 5122 (AVX512) CPU, running 64-bit Ubuntu Linux 18.04,

Tables D.3 & D.6: a Raspberry PI 3B+ single-board computer, with a Broadcom BCM2837B0 Cortex-A53 (64-bit, NEON) CPU, running 64-bit Ubuntu Linux 18.04,

Table D.7: an Amazon EC2 `t3.large` instance, with an Intel Xeon Platinum 8259CL (AVX512) CPU, running 64-bit Amazon Linux 2,

Table D.8: an Amazon EC2 `c5.metal` instance, with an Intel Xeon Platinum 8275CL (AVX512) CPU, running 64-bit Amazon Linux 2,

Table D.9: an Amazon EC2 `a1.metal` instance, with an Amazon Graviton Cortex-A72 (64-bit, Neon) CPU, running 64-bit Amazon Linux 2,

Table D.10: an Amazon EC2 `m6g.metal` instance, with an Amazon Graviton2 Cortex-A76 (64-bit, Neon) CPU, running 64-bit Amazon Linux 2.

SUPERCOP. We downloaded supercop-20200417.tar.xz [1] and installed it on all seven machines above. We configured SUPERCOP to use the default GCC and CLANG compilers installed on each machine (typically gcc-7 and clang-7) and we also installed the latest versions of these compilers (typically gcc-9 and clang-9). SUPERCOP evaluated each algorithm for all compilers under a variety of optimization flags, with the best performance usually achieved by the combination: `-O3 -march=native -mtune=native`. We report numbers for the best compiler combination.

To the existing implementations in SUPERCOP, we added: (1) Jasmin Intel assembly code [2], including verified scalar x86 code, (unverified) AVX, and verified AVX2; (2) Blake2 reference source code package [3], including scalar, NEON, and AVX code; (3) OpenSSL, compiled from the latest source in the OpenSSL repository [4] with both assembly enabled and disabled (`no-asm`)

For each algorithm, we set the input size (`TUNE_BYTES`) parameter to 16384 bytes. For Poly1305, we modified the benchmarking code to measure just a single call to the Poly1305 MAC function (the original SUPERCOP measured two calls, one for MACing and one for verification.) The rest of SUPERCOP was left unchanged.

We then ran SUPERCOP which tested and measured all the implementations it could compile on each platform. For example, on the Graviton, it ignores all the Intel assembly implementations. We removed some redundant implementations from SUPERCOP (e.g., many similar variants of Blake2 with identical perfomance). Finally, we post-processed the results with a script to obtain the tables shown below, adding implementation author names for clarity.

KBENCH9000. We downloaded the kernel benchmarking suite KBENCH9000 [5]. We exten-

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

[2] `https://github.com/tfaoliveira/libjc`, commit da8dab6

[3] `https://github.com/BLAKE2/BLAKE2`, commit 997fa5b

[4] `https://github.com/openssl/openssl`, commit b756626

[5] `https://git.zx2c4.com/kbench9000/about/`, commit baf706e

sively used this benchmarking suite for our own code (which runs in the Linux kernel), but some of the other implementations we wanted to measure (notably OpenSSL) could not be run in the kernel without significant modifications. Consequently, we ported this benchmarking suite to work in user-space, along with a script that turns off Turbo-Boost and HyperThreading and then runs the benchmark on a single core. We then measured each algorithm for input lengths ranging from 1024 bytes to 32768 bytes, and for each length we pick the median measurement from 100000 runs. As a sanity check, we compared the performance numbers for our own code between the kernel and user-space versions of KBENCH9000 and the figures were indistinguishable, which gives us more confidence in these measurements.

In addition to our own code and Jasmin, Blake2 (reference) and OpenSSL, we added calls to the LibSodium library [6]. We then ran these measurements on the three machines we owned: the Dell XPS13 laptop, the Xeon workstation, and the Raspberry Pi 3B+.

ChaCha20 and Poly1305. On our Intel laptop, which supports AVX2 but not AVX512, our vectorized AVX2 code for ChaCha20 and Poly1305 is 4.8× and 4.3× faster than portable code. On the Xeon workstation, the speedup for our AVX512 code grows to 10.3× for ChaCha20 and 5.9× for Poly1305. On the Raspberry PI, the speedups are more modest: 1.7× for ChaCha20, and 1.4× fo Poly1305.

Among the other implementations we measured, Jasmin had the fastest ChaCha20 and Poly1305 AVX2 implementations. For inputs of 16384 bytes, our code was 3-6% slower than Jasmin, but the difference is significantly greater for smaller inputs, where Jasmin uses specialized code, but our implementation still uses generic vectorization. For medium-to-large inputs, the speed difference is because of the manual assembly-level instruction interleaving in the Jasmin code. By mimicking this interleaving in our C code, we were able to get closer to Jasmin's performance, but we decided not to use this optimization because it obfuscates the structure of the code and because it is unclear whether such low-level optimizations will still be effective on future platforms.

This speed difference disappears entirely on the Xeon workstation, where our ChaCha20 and Poly1305 implementations are uniformly the fastest among all the code we tested, matching the performance of the fastest AVX512 implementation in SUPERCOP. Interestingly, even our AVX2 code catches up to Jasmin's AVX2 on the AVX-512 machine, where the manual instruction interleaving appears to offer less benefit. OpenSSL also includes AVX512 code that we believe is at least as fast as ours but this code appears to be disabled on our Xeon workstation (and on the Amazon Xeon instances we tested) because of frequency scaling issues with AVX-512 [9], and we could not find an easy way to re-enable this code on our (first generation) AVX512 machines. We carefully inspected the OpenSSL AVX-512 code, and we expect that it should be at least as fast as our code. We intend to test it more thoroughly on newer AVX512 processors with Integer Fused Multiply Accumulate (IFMA) instructions enabled.

On the Raspberry Pi, the fastest implementation we found was hand-optimized assembly from OpenSSL, which was 16% faster than our ChaCha20, and 2.1× faster than our Poly1305. Our Poly1305 code gets closer to OpenSSL on newer ARMv8 chips; e.g., it is 47% slower than OpenSSL on the Amazon Graviton2. On inspecting the OpenSSL Poly1305 code, we found that

[6] `https://github.com/jedisct1/libsodium`, commit 89943bd

the main difference is that it was making use of efficient multiply-with-accumulate instructions available in ARM NEON (but not on Intel). We intend to extend our vector libraries to support these instructions in the future.

Blake2s and Blake2b. Compared to our portable C code, our 128-bit vectorized code for Blake2s offers a modest speedup on Intel machines: 1.29× on the laptop, 1.47× on Xeon. Our 256-bit vectorized code for Blake2b offers even smaller speedups: 1.13× on the laptop, 1.27× on Xeon. These measurements match the speedups we have observed for other Blake2 implementations. If the effect of vectorization seems less pronounced than for ChaCha20 and Poly1305, it is perhaps because the portable C code for Blake2 is already very fast, and easy to optimize for modern C compilers.

On all the ARM64 chips, however, we see a surprising *performance loss* for vectorized code compared to portable C. This is a known issue on ARM CPUs where the latency of vector shift instructions (used extensively in hash functions like Blake2 and SHA-2) is quite high [8]. Consequently, for hash functions, vectorization on the cheap ARMv8 CPUs we measured does not appear to provide many benefits. However, on higher-end ARM devices, like the Apple A9, and on upcoming ARM servers, we expect that vectorized code will reap significant benefits.

The fastest implementations of Blake2 we found were written by Samuel Neves for the BLAKE2 team. Our vectorized code is about 10% slower than this implementation on both the laptop and workstation. This difference is because Neves' implementation uses AVX2 instructions to implement the Blake2 message permutation table in code, whereas our generic vectorized code uses load instructions that are available on all platforms.

Multi-Buffer SHA-2. Our multi-buffer SHA-2 implementation offers a large speedup over portable code on Intel platforms, but as with Blake2, is not effective on the ARM devices we tested. Our 8-way SHA-256 implementation is 4.6× faster (per input) than portable code on AVX2, and 6.7× faster on AVX512. Our 4-way SHA-512 implementation is 2.6× faster than portable code on AVX2, our 8-way SHA-512 is and 5.1× faster on AVX512.

On all platforms, the fastest other SHA-2 implementations are from OpenSSL, which relies on vector instructions to speed up message scheduling and uses native SHA instruction (SHA-EXT) when available. OpenSSL also includes a multi-buffer assembly implementation, but only for SHA-256 (not the other variants), and only for Intel platforms. For SHA-256 on our AVX2 laptop, the hand-written multi-buffer OpenSSL assembly code is 9% faster than HACL×N when processing 8 inputs in parallel. However, on AVX512, our code leapfrogs OpenSSL by a significant margin (36%) and is the fastest implementation we tested. Furthermore, for SHA-224, SHA-384, and SHA-512, ours are the only multi-buffer implementations and hence are the fastest implementations in our benchmarks.

On ARM, our 4-way vectorized SHA-256 code is 22% faster than our scalar code and 17% faster than OpenSSL. This is far less than the speedups obtained on AVX2 and AVX512, and this is because of the poor shift/rotate performance on NEON. Some Intel and ARM processors support native SHA-2 instructions, and using these instructions can provide much better performance than vectorization. On Amazon Graviton, for instance, OpenSSL assembly uses hardware SHA instructions and is by far the fastest implementation.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	dolbeau/amd64-avx2	C	AVX2	clang-11	0.75
	jasmin/avx2	assembly	AVX2	gcc-8	0.75
	openssl	assembly	AVX2	clang-11	0.75
	hacl-star/vec256	C	AVX2	gcc-8	0.77
	dolbeau/generic-gccsimd256	C	AVX2	clang-10	0.87
	goll-gueron	C	AVX2	gcc-8	0.90
	krovetz/avx2	C	AVX2	gcc-8	1.00
	jasmin/avx	assembly	AVX	gcc-9	1.44
	hacl-star/vec128	C	AVX	gcc-8	1.50
	dolbeau/generic-gccsimd128	C	AVX	clang-11	1.57
	krovetz/vec128	C	SSSE3	gcc-9	1.71
	bernstein/e/amd64-xmm6	assembly	SSE2	clang-11	1.83
	jasmin/ref	assembly		gcc-9	3.62
	hacl-star/scalar	C		gcc-8	3.73
	openssl-portable	C		clang-11	4.10
	bernstein/e/ref	C		gcc-9	4.10
Poly1305	openssl	assembly	AVX2	clang-11	0.35
	jasmin/avx2	assembly	AVX2	clang-11	0.35
	hacl-star/vec256	C	AVX2	clang-11	0.37
	moon/avx2/64	assembly	AVX2	clang-10	0.37
	jasmin/avx	assembly	AVX	clang-10	0.56
	moon/sse2/64	assembly	SSE2	clang-11	0.58
	moon/avx/64	assembly	AVX	clang-10	0.60
	jasmin/ref3	assembly		gcc-9	0.65
	hacl-star/vec128	C	AVX	clang-10	0.72
	openssl-portable	C		clang-11	1.19
	hacl-star/scalar	C		gcc-9	1.59
	bernstein/amd64	assembly	SSE2	gcc-8	1.65
	bernstein/53	C		gcc-8	1.79
Blake2b	neves/avx2	C	AVX2	clang-11	2.02
	neves/avxicc	assembly	AVX	clang-10	2.12
	moon/avx2/64	assembly	AVX2	clang-10	2.20
	moon/avx/64	assembly	AVX	gcc-9	2.21
	hacl-star/vec256	C	AVX2	clang-11	2.26
	neves/regs	C		gcc-9	2.34
	blake2-reference/sse	C	AVX	gcc-8	2.51
	blake2-reference/ref	C		gcc-9	2.52
	hacl-star/scalar	C		gcc-8	2.56
	neves/ref	C		gcc-8	2.72
Blake2s	neves/xmm	C	AVX	clang-11	3.06
	neves/avxicc	assembly	AVX	clang-11	3.07
	blake2-reference/sse	C	AVX	clang-11	3.07
	moon/ssse3/64	assembly	SSSE3	gcc-9	3.29
	hacl-star/vec128	C	AVX	gcc-9	3.34
	moon/avx/64	assembly	AVX	clang-11	3.48
	moon/sse2/64	assembly	SSE2	gcc-8	3.81
	neves/regs	C		gcc-9	4.01
	blake2-reference/ref	C		gcc-8	4.28
	hacl-star/scalar	C		gcc-9	4.32
	neves/ref	C		gcc-9	4.33
SHA-256	openssl/sha256-mb8	asssembly	AVX2	clang-11	1.49 (11.92 / 8)
	hacl-star/sha256-mb8	C	AVX2	gcc-9	1.62 (12.93 / 8)
	openssl/sha256-mb4	asssembly	AVX	clang-11	2.84 (11.36 / 4)
	hacl-star/sha256-mb4	C	AVX	clang-10	3.14 (12.58 / 4)
	openssl	assembly	AVX2	clang-11	4.83
	sphlib-small	C		gcc-9	7.29
	sphlib	C		gcc-9	7.33
	hacl-star/scalar	C		gcc-9	7.41
	openssl-portable	C		clang-11	10.16
SHA-512	**hacl-star/sha512-mb4**	C	AVX2	clang-10	1.95 (7.81 / 4)
	openssl	assembly	AVX2	clang-11	3.25
	sphlib	C		gcc-8	4.84
	sphlib-small	C		gcc-9	4.98
	hacl-star/scalar	C		gcc-8	5.06
	openssl-portable	C		clang-10	5.83

Table D.1 – SUPERCOP Benchmarks on Dell XPS13 with Intel Kaby Lake i7-7560U processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-8, gcc-9, clang-10, and clang-11.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	**hacl-star/vec512**	C	AVX512	gcc-9	0.56
	dolbeau/amd64-avx2	C	AVX512	clang-10	0.56
	openssl	assembly	AVX2	clang-10	0.77
	hacl-star/vec256	C	AVX2	gcc-7	0.84
	dolbeau/generic-gccsimd256	C	AVX2	clang-10	0.99
	jasmin/avx2	assembly	AVX2	clang-10	1.12
	krovetz/avx2	C	AVX2	gcc-9	1.37
	hacl-star/vec128	C	AVX	gcc-9	1.53
	dolbeau/generic-gccsimd128	C	AVX	clang-10	1.79
	krovetz/vec128	C	SSSE3	clang-10	1.99
	jasmin/avx	assembly	AVX	clang-10	2.21
	bernstein/e/amd64-xmm6	assembly	SSE2	gcc-9	2.81
	jasmin/ref	assembly		gcc-9	5.57
	hacl-star/scalar	C		gcc-9	5.74
	bernstein/e/ref	C		gcc-9	5.97
	openssl-portable	C		clang-10	6.00
Poly1305	**hacl-star/vec512**	C	AVX512	gcc-9	0.39
	jasmin/avx2	assembly	AVX2	clang-6	0.51
	openssl	assembly	AVX2	gcc-9	0.52
	hacl-star/vec256	C	AVX2	gcc-9	0.52
	moon/avx2/64	assembly	AVX2	gcc-7	0.57
	jasmin/avx	assembly	AVX	clang-10	0.87
	moon/avx/64	assembly	AVX	gcc-7	0.88
	moon/sse2/64	assembly	SSE2	clang-10	0.89
	jasmin/ref3	assembly		clang-10	0.97
	hacl-star/vec128	C	AVX	gcc-9	1.04
	openssl-portable	C		gcc-7	1.85
	hacl-star/scalar	C		gcc-9	2.31
	bernstein/amd64	assembly		gcc-9	2.53
	bernstein/53	C		gcc-9	2.73
Blake2b	neves/avx2	C	AVX2	clang-10	2.84
	blake2-reference/sse	C	AVX	clang-10	2.98
	hacl-star/vec256	C	AVX2	clang-10	3.13
	neves/avxicc	assembly	AVX	gcc-9	3.26
	moon/avx2/64	assembly	AVX2	clang-10	3.39
	moon/avx/64	assembly	AVX	gcc-9	3.40
	neves/regs	C		gcc-9	3.61
	blake2-reference/ref	C		gcc-9	3.88
	neves/ref	C		gcc-7	3.97
	hacl-star/scalar	C		gcc-9	3.97
Blake2s	neves/xmm	C	AVX	clang-6	4.11
	blake2-reference/sse	C	AVX	clang-6	4.12
	hacl-star/vec128	C	AVX	gcc-7	4.52
	neves/avxicc	assembly	AVX	gcc-9	4.72
	moon/ssse3/64	assembly	SSSE3	gcc-9	5.06
	moon/avx/64	assembly	AVX	gcc-9	5.21
	moon/sse2/64	assembly	SSE2	gcc-9	5.85
	neves/regs	C		gcc-9	6.17
	blake2-reference/ref	C		gcc-9	6.45
	neves/ref	C		gcc-9	6.57
	hacl-star/scalar	C		gcc-9	6.63
SHA-256	**hacl-star/sha256-mb8**	C	AVX2	gcc-9	1.69 (13.53 / 8)
	openssl/sha256-mb8	asssembly	AVX2	gcc-9	2.29 (18.31 / 8)
	hacl-star/sha256-mb4	C	AVX	gcc-9	3.22 (12.90 / 4)
	openssl/sha256-mb4	asssembly	AVX	clang-10	4.36 (17.46 / 4)
	openssl	assembly	AVX2	gcc-9	7.43
	sphlib-small	C		gcc-7	11.04
	sphlib	C		gcc-7	11.25
	hacl-star/scalar	C		gcc-9	11.36
	openssl-portable	C		gcc-9	15.35
SHA-512	**hacl-star/sha512-mb8**	C	AVX512	clang-10	1.44 (11.49 / 8)
	hacl-star/sha512-mb4	C	AVX2	gcc-9	2.07 (8.29 / 4)
	openssl	assembly	AVX2	gcc-9	4.99
	sphlib	C		gcc-9	6.72
	sphlib-small	C		gcc-9	6.75
	hacl-star/scalar	C		gcc-7	7.38
	openssl-portable	C		clang-10	9.12

Table D.2 – SUPERCOP Benchmarks on Dell Precision Workstation with Intel Xeon Gold 5122 processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-7, gcc-9, clang-6, and clang-10.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	openssl	assembly	NEON	clang	4.49
	hacl-star/vec128	C	NEON	gcc	5.19
	dolbeau/arm-neon	C	NEON	clang	5.50
	krovetz/vec128	C	NEON	gcc	6.22
	dolbeau/generic-gccsimd128	C	NEON	clang	7.01
	hacl-star/scalar	C		gcc	8.69
	openssl-portable	C		gcc	8.84
	bernstein/e/ref	C		gcc	9.08
Poly1305	openssl	assembly	NEON	clang	1.50
	hacl-star/vec128	C	NEON	clang	3.11
	openssl-portable	C		clang	3.57
	hacl-star/scalar	C		clang	4.20
	bernstein/53	C		gcc	4.95
Blake2b	neves/regs	C		gcc	6.02
	blake2-reference/ref	C		gcc	6.70
	hacl-star/scalar	C		clang	6.99
	neves/ref	C		gcc	7.35
	blake2-reference/neon	C	NEON	gcc	10.27
Blake2s	neves/regs	C		gcc	9.80
	blake2-reference/ref	C		gcc	10.70
	blake2-reference/neon	C	NEON	clang	11.31
	neves/ref	C		gcc	11.31
	hacl-star/scalar	C		gcc	11.42
	hacl-star/vec128	C	NEON	gcc	15.30
SHA-256	**hacl-star/sha256-mb4**	C	NEON	gcc	12.92 (51.66 / 4)
	openssl	assembly	NEON	clang	15.09
	hacl-star/scalar	C		clang	15.70
	sphlib-small	C	NEON	gcc	15.97
	sphlib	C	NEON	gcc	16.40
	openssl-portable	C		gcc	19.85
SHA-512	openssl	assembly	NEON	gcc	9.77
	openssl-portable	C		gcc	10.07
	hacl-star/scalar	C		gcc	11.27
	sphlib	C	NEON	gcc	12.40
	sphlib-small	C	NEON	gcc	12.40

Table D.3 – SUPERCOP Benchmarks on Raspberry Pi 3B+, with a Broadcom BCM2837B0 quad-core Cortex-A53 (ARMv8) @ 1.4GHz. Implementations are compiled with gcc-9 and clang-9.

Algorithm	Implementation	Compiler	1024	2048	4096	8192	16384	32768
ChaCha20	jasmin/avx2	gcc-9	1.21	1.18	1.17	1.16	1.16	1.16
	openssl-assembly	gcc-9	1.24	1.19	1.17	1.16	1.17	1.17
	libsodium	gcc-9	1.34	1.28	1.25	1.24	1.24	1.23
	hacl-star/vec256	gcc-9	1.38	1.29	1.25	1.23	1.28	1.27
	hacl-star/vec128	gcc-9	2.38	2.34	2.32	2.31	2.37	2.36
	hacl-star/scalar	gcc-9	6.23	6.18	6.16	6.15	6.15	6.15
	openssl-portable	clang-9	6.23	6.20	6.18	6.17	6.17	6.16
Poly1305	openssl-assembly	clang-9	0.75	0.63	0.57	0.54	0.52	0.51
	jasmin/avx2	clang-9	0.67	0.59	0.55	0.53	0.52	0.52
	hacl-star/vec256	clang-9	0.85	0.72	0.63	0.61	0.57	0.57
	libsodium	clang-9	1.23	1.10	1.04	1.00	0.99	0.99
	hacl-star/vec128	clang-9	1.29	1.20	1.17	1.16	1.14	1.13
	openssl-portable	gcc-9	1.99	1.94	1.91	1.90	1.89	1.89
	hacl-star/scalar	gcc-9	2.50	2.44	2.41	2.39	2.38	2.39
Blake2b	libsodium	clang-9	3.34	3.22	3.15	3.12	3.11	3.10
	hacl-star/vec256	clang-9	3.71	3.63	3.60	3.58	3.57	3.58
	reference-avx	gcc-9	4.12	4.09	4.02	3.99	3.98	3.97
	hacl-star/scalar	gcc-9	4.23	4.22	4.16	4.13	4.11	4.11
	openssl-portable	clang-9	6.42	5.31	4.75	4.49	4.36	4.29
Blake2s	reference-avx	clang-9	4.97	4.96	4.90	4.87	4.86	4.85
	hacl-star/vec128	gcc-9	5.42	5.36	5.34	5.32	5.32	5.35
	hacl-star/scalar	gcc-9	7.03	6.93	6.89	6.86	6.85	6.86
	openssl-portable	gcc-9	8.96	7.87	7.33	7.07	6.94	6.95
SHA-256	**hacl-star/mb8**	clang-9	2.74	2.64	2.60	2.57	2.56	2.56
	hacl-star/mb4	gcc-9	5.31	5.14	5.05	5.00	4.98	4.98
	openssl-assembly	gcc-9	8.38	8.04	7.86	7.78	7.74	7.73
	libsodium	gcc-9	12.60	12.14	11.89	11.76	11.70	11.66
	hacl-star/scalar	gcc-9	12.62	12.15	11.93	11.80	11.74	11.72
	openssl-portable	clang-9	17.30	16.73	16.43	16.27	16.19	16.15
SHA-512	**hacl-star/mb4**	clang-9	3.50	3.29	3.18	3.13	3.11	3.10
	openssl-assembly	gcc-9	6.03	5.59	5.36	5.25	5.20	5.18
	libsodium	clang-9	8.62	8.01	7.72	7.56	7.49	7.45
	hacl-star/scalar	gcc-9	8.66	8.08	7.80	7.66	7.59	7.56
	openssl-portable	clang-9	10.48	9.82	9.50	9.31	9.22	9.18

Table D.4 – KBENCH9000 Benchmarks on Dell XPS13 with Intel Kaby Lake i7-7560U processor running 64-bit Ubuntu Linux. All implementations are compiled with gcc-9 and clang-9. Measurements are in cycles/byte, for input lengths ranging from 1024 bytes to 32768 bytes, obtained as the median of 100000 runs.

Algorithm	Implementation	Compiler	1024	2048	4096	8192	16384	32768
ChaCha20	**hacl-star/vec512**	gcc-9	0.68	0.61	0.58	0.56	0.56	0.56
	openssl-assembly	gcc-9	0.89	0.83	0.81	0.80	0.79	0.79
	hacl-star/vec256	gcc-9	0.98	0.93	0.90	0.89	0.88	0.88
	jasmin/avx2	gcc-9	1.20	1.17	1.16	1.15	1.15	1.15
	libsodium	gcc-9	1.26	1.21	1.18	1.17	1.16	1.16
	hacl-star/vec128	gcc-9	1.63	1.60	1.58	1.58	1.58	1.57
	hacl-star/scalar	gcc-9	6.19	6.15	6.12	6.12	6.11	6.11
	openssl-portable	gcc-9	6.23	6.19	6.17	6.17	6.16	6.16
Poly1305	**hacl-star/vec512**	gcc-9	0.94	0.65	0.51	0.43	0.40	0.38
	jasmin/avx2	gcc-9	0.67	0.59	0.55	0.53	0.52	0.51
	openssl-assembly	clang-9	0.75	0.63	0.57	0.54	0.52	0.51
	hacl-star/vec256	gcc-9	0.82	0.66	0.58	0.54	0.52	0.51
	libsodium	clang-9	1.14	1.01	0.95	0.92	0.91	0.90
	hacl-star/vec128	gcc-9	1.27	1.16	1.11	1.09	1.07	1.06
	openssl-portable	gcc-9	1.97	1.93	1.92	1.89	1.88	1.88
	hacl-star/scalar	gcc-9	2.49	2.45	2.41	2.39	2.38	2.39
Blake2b	reference-avx	clang-9	3.23	3.14	3.09	3.07	3.06	3.05
	libsodium	gcc-9	3.34	3.22	3.18	3.15	3.14	3.14
	hacl-star/vec256	clang-9	3.40	3.36	3.33	3.32	3.31	3.31
	hacl-star/scalar	gcc-9	4.21	4.14	4.11	4.10	4.09	4.09
	openssl-portable	gcc-9	5.88	5.06	4.62	4.40	4.30	4.29
Blake2s	reference-avx	clang-9	4.60	4.53	4.50	4.49	4.48	4.48
	hacl-star/vec128	gcc-9	4.77	4.71	4.69	4.67	4.67	4.69
	openssl-portable	gcc-9	8.33	7.46	7.02	6.82	6.71	6.73
	hacl-star/scalar	gcc-9	6.90	6.86	6.85	6.83	6.82	6.84
SHA-256	**hacl-star/mb8**	gcc-9	1.87	1.80	1.76	1.75	1.74	1.74
	hacl-star/mb4	gcc-9	3.55	3.43	3.36	3.33	3.32	3.31
	openssl-assembly	clang-9	8.38	8.04	7.85	7.76	7.72	7.71
	libsodium	clang-9	12.57	12.10	11.85	11.73	11.67	11.63
	hacl-star/scalar	gcc-9	12.51	12.10	11.88	11.76	11.71	11.69
	openssl-portable	clang-9	16.92	16.39	16.11	15.96	15.88	15.85
SHA-512	**hacl-star/mb8**	clang-9	1.72	1.61	1.56	1.53	1.52	1.52
	hacl-star/mb4	gcc-9	2.40	2.25	2.18	2.14	2.12	2.11
	openssl-assembly	gcc-9	6.06	5.58	5.33	5.22	5.17	5.14
	libsodium	gcc-9	8.55	8.00	7.72	7.57	7.50	7.47
	hacl-star/scalar	gcc-9	8.59	8.05	7.79	7.65	7.58	7.55
	openssl-portable	gcc-9	10.63	9.95	9.63	9.45	9.37	9.32

Table D.5 – KBENCH9000 Benchmarks on Dell Precision workstation with Intel(R) Xeon(R) Gold 5122 CPU @ 3.60GHz processor running 64-bit Ubuntu Linux. All implementations are compiled with gcc-9 and clang-9. Measurements are in cycles/byte, for input lengths ranging from 1024 bytes to 32768 bytes, obtained as the median of 100000 runs.

Algorithm	Implementation	Compiler	1024	2048	4096	8192	16384	32768
ChaCha20	openssl-assembly	clang	4.59	4.53	4.50	4.49	4.50	4.55
	hacl-star/vec128	gcc	5.42	5.32	5.27	5.25	5.27	5.32
	openssl-portable	clang	8.88	8.84	8.82	8.82	8.84	8.94
	hacl-star/scalar	gcc	8.91	8.86	8.84	8.83	8.87	8.99
	libsodium	clang	9.33	9.25	9.21	9.21	9.25	9.37
Poly1305	openssl-assembly	gcc	1.97	1.72	1.59	1.53	1.50	1.51
	hacl-star/vec128	clang	3.48	3.30	3.21	3.17	3.15	3.16
	openssl-portable	gcc	3.74	3.65	3.61	3.58	3.57	3.59
	hacl-star/scalar	gcc	4.61	4.52	4.48	4.46	4.45	4.47
	libsodium	gcc	5.35	5.27	5.23	5.21	5.20	5.22
Blake2b	openssl-portable	gcc	11.33	8.71	7.39	6.74	6.43	6.30
	hacl-star/scalar	gcc	7.12	7.01	6.95	6.93	6.92	6.96
	libsodium	gcc	7.60	7.42	7.34	7.29	7.28	7.33
	reference-neon	gcc	11.13	10.96	10.87	10.82	10.81	10.91
Blake2s	openssl-portable	gcc	14.92	12.60	11.44	10.89	10.63	10.56
	hacl-star/scalar	gcc	11.59	11.51	11.48	11.46	11.47	11.57
	reference-neon	gcc	11.83	11.68	11.61	11.57	11.58	11.66
	hacl-star/vec128	gcc	16.58	16.52	16.49	16.49	16.61	16.64
SHA-256	**hacl-star/mb4**	gcc	13.68	13.23	13.01	12.99	13.08	13.00
	openssl-assembly	gcc	16.22	15.58	15.26	15.10	15.15	15.12
	hacl-star/scalar	gcc	17.49	16.92	16.64	16.51	16.58	16.54
	libsodium	gcc	19.43	18.68	18.31	18.13	18.22	18.19
	openssl-portable	clang	21.01	20.25	19.88	19.70	19.79	19.73
SHA-512	openssl-assembly	gcc	11.10	10.37	10.01	9.83	9.76	9.82
	openssl-portable	gcc	11.45	10.70	10.33	10.14	10.08	10.12
	hacl-star/scalar	gcc	12.70	11.94	11.55	11.36	11.28	11.34
	libsodium	gcc	13.73	12.76	12.27	12.03	11.94	11.98

Table D.6 – KBENCH9000 Benchmarks on Raspberry Pi 3B+, with a Broadcom BCM2837B0 quad-core Cortex-A53 (ARMv8) @ 1.4GHz running 64-bit Ubuntu Linux. All implementations are compiled with gcc-9 and clang-9. Measurements are in cycles/byte, for input lengths ranging from 1024 bytes to 32768 bytes, obtained as the median of 100000 runs.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	**hacl-star/vec512**	C	AVX512	gcc-9	0.52
	dolbeau/amd64-avx2	C	AVX512	clang	0.52
	openssl	assembly	AVX2	gcc-9	0.64
	hacl-star/vec256	C	AVX2	gcc-9	0.71
	jasmin/avx2	assembly	AVX2	gcc-9	0.93
	dolbeau/generic-gccsimd256	C	AVX2	gcc-9	0.94
	krovetz/avx2	C	AVX2	gcc-9	1.14
	hacl-star/vec128	C	AVX	gcc-9	1.27
	dolbeau/generic-gccsimd128	C	AVX	gcc-9	1.51
	jasmin/avx	assembly	AVX	gcc-9	1.83
	krovetz/vec128	C	SSSE3	clang	1.88
	bernstein/e/amd64-xmm6	assembly	SSE2	gcc-9	2.33
	jasmin/ref	assembly		clang-9	4.62
	hacl-star/scalar	C		gcc-9	4.76
	bernstein/e/ref	C		gcc-9	4.95
	openssl-portable	C		gcc-9	4.98
Poly1305	**hacl-star/vec512**	C	AVX512	gcc-9	0.40
	jasmin/avx2	assembly	AVX2	gcc-9	0.49
	openssl	assembly	AVX2	gcc-9	0.49
	hacl-star/vec256	C	AVX2	gcc-9	0.49
	moon/avx2/64	assembly	AVX2	clang-9	0.53
	jasmin/avx	assembly	AVX	gcc-9	0.72
	moon/avx/64	assembly	AVX	gcc-9	0.77
	jasmin/ref3	assembly		gcc-9	0.80
	moon/sse2/64	assembly	SSE2	gcc-9	0.86
	hacl-star/vec128	C	AVX	gcc-9	0.88
	openssl-portable	C		gcc-9	1.53
	hacl-star/scalar	C		gcc-9	1.92
	bernstein/amd64	assembly		gcc-9	2.20
	bernstein/53	C		gcc-9	2.51
Blake2b	neves/avx2	C	AVX2	clang-9	2.60
	neves/avxicc	assembly	AVX	gcc-9	2.70
	moon/avx/64	assembly	AVX	gcc-9	2.82
	blake2-reference/sse	C	AVX	clang-9	2.83
	neves/regs	C		gcc-9	2.99
	hacl-star/vec256	C	AVX2	gcc-9	2.99
	blake2-reference/ref	C		gcc-9	3.21
	moon/avx2/64	assembly	AVX2	clang-9	3.23
	hacl-star/scalar	C		gcc-9	3.29
	neves/ref	C		gcc-9	3.34
Blake2s	blake2-reference/sse	C	AVX	clang	3.33
	neves/xmm	C	AVX	clang	3.37
	hacl-star/vec128	C	AVX	gcc-9	3.76
	neves/avxicc	assembly	AVX	gcc-9	3.91
	moon/ssse3/64	assembly	SSSE3	gcc-9	4.20
	moon/avx/64	assembly	AVX	gcc-9	4.32
	moon/sse2/64	assembly	SSE2	gcc-9	4.85
	neves/regs	C		gcc-9	5.11
	blake2-reference/ref	C		gcc-9	5.35
	neves/ref	C		gcc-9	5.45
	hacl-star/scalar	C		gcc-9	5.49
SHA-256	**hacl-star/sha256-mb8**	C	AVX2	gcc-9	1.40 (11.21 / 8)
	hacl-star/sha256-mb4	C	AVX	gcc-9	2.68 (10.70 / 4)
	openssl	assembly	AVX2	clang-9	6.23
	sphlib-small	C		gcc	9.15
	sphlib	C		gcc-9	9.34
	hacl-star/scalar	C		gcc-9	9.43
	openssl-portable	C		gcc-9	12.73
SHA-512	**hacl-star/sha512-mb8**	C	AVX512	clang	1.39 (11.11 / 8)
	hacl-star/sha512-mb4	C	AVX2	gcc-9	1.72 (6.89 / 4)
	openssl	assembly	AVX2	gcc-9	4.19
	sphlib	C		gcc-9	5.63
	sphlib-small	C		gcc-9	5.64
	hacl-star/scalar	C		gcc-9	6.19
	openssl-portable	C		gcc-9	7.56

Table D.7 – SUPERCOP Benchmarks on Amazon EC2 `t3.large` instance with Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-7, clang-7, gcc-9, and clang-9.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	**hacl-star/vec512**	C	AVX512	gcc-9	0.44
	dolbeau/amd64-avx2	C	AVX512	clang-9	0.44
	openssl	assembly	AVX2	gcc-9	0.61
	hacl-star/vec256	C	AVX2	gcc-9	0.67
	dolbeau/generic-gccsimd256	C	AVX2	clang-9	0.81
	jasmin/avx2	assembly	AVX2	gcc-9	0.89
	krovetz/avx2	C	AVX2	gcc-9	1.08
	hacl-star/vec128	C	AVX	gcc-9	1.21
	dolbeau/generic-gccsimd128	C	AVX	clang-9	1.44
	krovetz/vec128	C	SSSE3	clang-9	1.61
	jasmin/avx	assembly	AVX	gcc-9	1.74
	bernstein/e/amd64-xmm6	assembly	SSE2	gcc-9	2.22
	jasmin/ref	assembly		gcc-9	4.40
	hacl-star/scalar	C		gcc-9	4.54
	bernstein/e/ref	C		gcc-9	4.72
	openssl-portable	C		gcc-9	4.75
Poly1305	**hacl-star/vec512**	C	AVX512	gcc-9	0.31
	jasmin/avx2	assembly	AVX2	gcc-9	0.41
	openssl	assembly	AVX2	clang	0.41
	hacl-star/vec256	C	AVX2	gcc-9	0.41
	moon/avx2/64	assembly	AVX2	gcc	0.46
	jasmin/avx	assembly	AVX	gcc-9	0.69
	moon/avx/64	assembly	AVX	gcc-9	0.71
	moon/sse2/64	assembly	SSE2	gcc-9	0.72
	jasmin/ref3	assembly		gcc-9	0.76
	hacl-star/vec128	C	AVX	gcc-9	0.82
	openssl-portable	C		gcc-9	1.46
	hacl-star/scalar	C		gcc-9	1.83
	bernstein/amd64	assembly		gcc-9	2.00
	bernstein/53	C		gcc-9	2.14
Blake2b	neves/avx2	C	AVX2	clang-9	2.74
	moon/avx2/64	assembly	AVX2	clang-9	2.75
	hacl-star/vec256	C	AVX2	clang-9	2.85
	blake2-reference/sse	C	AVX	clang-9	3.06
	moon/avx/64	assembly	AVX	clang-9	3.28
	neves/avxicc	assembly	AVX	clang-9	3.35
	hacl-star/scalar	C		clang-9	3.85
	neves/regs	C		clang-9	4.48
	blake2-reference/ref	C		clang-9	4.58
	neves/ref	C		clang-9	4.68
Blake2s	blake2-reference/sse	C	AVX	clang-9	3.53
	moon/ssse3/64	assembly	SSSE3	clang-9	4.01
	hacl-star/vec128	C	AVX	clang-9	4.05
	neves/xmm	C	AVX	clang	4.11
	neves/avxicc	assembly	AVX	clang-9	4.15
	moon/avx/64	assembly	AVX	clang	4.59
	moon/sse2/64	assembly	SSE2	clang	5.02
	hacl-star/scalar	C		gcc-9	5.68
	neves/regs	C		clang	5.73
	blake2-reference/ref	C		gcc-9	6.22
	neves/ref	C		gcc-9	6.99
SHA-256	**hacl-star/sha256-mb8**	C	AVX2	gcc-9	1.33 (10.60 / 8)
	hacl-star/sha256-mb4	C	AVX	gcc-9	2.55 (10.20 / 4)
	openssl	assembly	AVX2	gcc-9	6.26
	sphlib	C		clang-9	9.78
	hacl-star/scalar	C		clang	9.81
	sphlib-small	C		clang-9	9.93
	openssl-portable	C		clang	12.14
SHA-512	**hacl-star/sha512-mb8**	C	AVX512	clang	1.18 (9.43 / 8)
	hacl-star/sha512-mb4	C	AVX2	clang	1.75 (6.99 / 4)
	openssl	assembly	AVX2	clang-9	3.98
	hacl-star/scalar	C		clang	6.39
	sphlib	C		clang-9	6.67
	openssl-portable	C		clang-9	7.40
	sphlib-small	C		clang-9	7.45

Table D.8 – SUPERCOP Benchmarks on Amazon EC2 `c5.metal` instance with Intel(R) Xeon(R) Platinum 8275CL CPU @ 2.50GHz processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-7, clang-7, gcc-9, and clang-9.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	openssl	assembly	NEON	clang	4.40
	hacl-star/vec128	C	NEON	gcc	5.10
	dolbeau/arm-neon	C	NEON	gcc	5.16
	krovetz/vec128	C	NEON	clang	5.79
	dolbeau/generic-gccsimd128	C	NEON	clang	5.87
	hacl-star/scalar	C		gcc	5.95
	openssl-portable	C		gcc	8.43
	bernstein/e/ref	C		clang	8.90
Poly1305	openssl	assembly	NEON	clang	1.16
	hacl-star/vec128	C	NEON	clang	1.98
	openssl-portable	C		clang	3.08
	bernstein/53	C		clang	3.74
	hacl-star/scalar	C		gcc	5.13
Blake2b	neves/regs	C		gcc	5.46
	blake2-reference/ref	C		gcc	5.78
	hacl-star/scalar	C		gcc	5.95
	neves/ref	C		gcc	6.21
	blake2-reference/neon	C	NEON	clang	11.63
Blake2s	neves/regs	C		gcc	9.10
	blake2-reference/ref	C		gcc	9.34
	hacl-star/scalar	C		gcc	9.78
	neves/ref	C		gcc	10.06
	blake2-reference/neon	C	NEON	clang	17.15
	hacl-star/vec128	C	NEON	gcc	19.15
SHA-256	openssl	assembly	SHA-EXT	clang	2.01
	hacl-star/sha256-mb4	C	NEON	clang	10.12 (40.46 / 4)
	sphlib-small	C	NEON	clang	12.08
	hacl-star/scalar	C		gcc	12.15
	sphlib	C	NEON	gcc	12.31
	openssl-portable	C		gcc	14.58
SHA-512	openssl	assembly	NEON	gcc	7.28
	openssl-portable	C		gcc	7.75
	hacl-star/scalar	C		gcc	7.93
	sphlib-small	C	NEON	clang	9.81
	sphlib	C	NEON	clang	9.82

Table D.9 – SUPERCOP Benchmarks on Amazon EC2 `a1.metal` instance with Amazon Graviton1 Cortex-A72 @ 2.3GHz, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-7 and clang-7.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	openssl	assembly	NEON	gcc	2.36
	hacl-star/vec128	C	NEON	gcc	2.95
	dolbeau/arm-neon	C	NEON	gcc	3.16
	krovetz/vec128	C	NEON	gcc	3.58
	hacl-star/scalar	C		gcc	3.66
	dolbeau/generic-gccsimd128	C	NEON	gcc	3.74
	openssl-portable	C		gcc	5.78
	bernstein/e/ref	C		clang	5.98
Poly1305	openssl	assembly	NEON	clang	1.05
	hacl-star/vec128	C	NEON	clang	1.54
	openssl-portable	C		gcc	2.82
	bernstein/53	C		gcc	2.93
	hacl-star/scalar	C		gcc	5.07
Blake2b	neves/regs	C		clang	3.78
	blake2-reference/ref	C		gcc	3.82
	hacl-star/scalar	C		gcc	3.98
	neves/ref	C		gcc	3.99
	blake2-reference/neon	C	NEON	clang	7.83
Blake2s	neves/regs	C		gcc	6.26
	blake2-reference/ref	C		gcc	6.45
	neves/ref	C		gcc	6.54
	hacl-star/scalar	C		gcc	6.60
	hacl-star/vec128	C	NEON	clang	10.44
	blake2-reference/neon	C	NEON	clang	11.16
SHA-256	openssl	assembly	SHA-EXT	gcc	1.57
	hacl-star/sha256-mb4	C	NEON	clang	6.52 (26.09 / 4)
	sphlib-small	C	NEON	clang	9.76
	sphlib	C	NEON	gcc	10.16
	hacl-star/scalar	C		gcc	10.44
	openssl-portable	C		gcc	11.72
SHA-512	openssl	assembly	NEON	gcc	6.03
	openssl-portable	C		gcc	6.31
	hacl-star/scalar	C		gcc	7.00
	sphlib-small	C	NEON	clang	7.96
	sphlib	C	NEON	clang	7.99

Table D.10 – SUPERCOP Benchmarks on Amazon EC2 `m6g.metal` instance with Amazon Graviton2 Cortex-A76 @ 2.3GHz, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-7 and clang-7.

E Measuring the Impact of Compiler Optimizations

To estimate the impact of compiler optimizations on our code, we reran SUPERCOP with just the HACL×N algorithms on the Intel Kaby Lake Core i7-7560U laptop and the Intel Xeon Gold 5122 Workstation, enabling a variety of compiler options: GCC-9 at optimization levels O0, O1, O2, O3; CLANG-9 at optimization levels O0, O1, O2, O3; and CompCert 3.7 at optimization level O.

The results are depicted in Table E.1 and E.2. In general, GCC-9 performs slightly better than CLANG-9 on our code. Optimization level O0 disables all optimizations, so, as expected, our code is 10-50× slower than code at O3 for both CLANG and GCC. The main performance improvements kick in at O1. Notably, for all algorithms except ChaCha20, O3 does not provide any improvement over O2 (in some cases, our measurements for O2 are even better than those for O3). This means that the performance of our code mainly relies on the well-understood stable compiler optimizations that are enabled at O2 in GCC and CLANG, not on the potentially dangerous optimizations in O3.

To further dig down into the precise optimizations that were used in optimizing our code, we systematically measured our code by turning off each optimization and checking if the resulting assembly code was changed. Our ChaCha20 code triggers 23 out of 95 optimizations available at O1, 38 out of 135 optimizations at O2, and 50 out of 151 optimizations at O3. Of these optimizations, the ones that make the most difference appear to be (various flavors of) forward constant propagation, dead store elimination, loop unrolling, and function inlining.

Finally, we measure the performance of CompCert for our scalar implementations (CompCert does not support SIMD instructions). We find that the performance of CompCert falls between O0 and O1. This is consistent with measurements in prior work [26], since CompCert only implements some of the optimizations of O1 and does not have any of the tree-based optimizations that GCC relies on. We hope that future improvements in CompCert will close this gap, and upcoming support for SIMD [16] will allow us to compile all our implementations via this verified compiler.

Algorithm	Implementation	clang-9				gcc-9				ccomp
		-O0	-O1	-O2	-O3	-O0	-O1	-O2	-O3	-O
ChaCha20	hacl-star/scalar	45.71	6.82	6.25	6.46	41.94	4.36	4.12	3.88	6.97
	hacl-star/vec128	31.10	1.87	1.72	1.60	30.11	1.69	1.67	1.50	
	hacl-star/vec256	25.78	1.02	0.92	0.90	30.22	1.25	1.23	0.77	
Poly1305	hacl-star/scalar	8.78	1.87	1.68	1.64	8.47	1.81	1.59	1.51	2.49
	hacl-star/vec128	10.52	0.75	0.71	0.71	10.01	0.96	0.90	0.84	
	hacl-star/vec256	5.60	0.45	0.36	0.36	4.83	0.45	0.42	0.40	
Blake2b	hacl-star/scalar	109.09	3.09	2.75	2.75	60.85	2.62	2.59	2.59	8.13
	hacl-star/vec256	54.29	2.73	2.26	2.25	46.71	2.63	2.35	2.51	
Blake2s	hacl-star/scalar	107.80	5.21	4.71	4.71	98.79	4.35	4.32	4.32	13.42
	hacl-star/vec128	57.38	5.00	3.66	3.66	54.65	3.69	3.49	3.37	
SHA-256	hacl-star/scalar	48.44	8.79	8.07	8.08	35.89	7.57	7.40	7.45	12.96
	hacl-star/sha256-mb4	50.21	3.33	3.16	3.16	52.24	3.26	3.22	3.14	
	hacl-star/sha256-mb8	48.73	2.11	2.12	1.61	36.29	1.78	1.78	1.63	
SHA-512	hacl-star/scalar	34.67	5.47	4.99	4.99	23.97	4.86	4.81	4.85	8.21
	hacl-star/sha512-mb4	59.02	2.02	1.96	1.96	50.48	2.05	2.01	2.00	

Table E.1 – SUPERCOP Benchmarks on Dell XPS13 with Intel Kaby Lake i7-7560U processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-9, clang-9, and CompCert3.7.

Algorithm	Implementation	clang-9				gcc-9				ccomp
		-O0	-O1	-O2	-O3	-O0	-O1	-O2	-O3	-O
ChaCha20	hacl-star/scalar	73.09	12.23	8.24	8.23	65.71	6.67	6.50	5.96	11.26
	hacl-star/vec128	27.59	2.61	2.11	1.89	27.86	1.88	1.86	1.54	
	hacl-star/vec256	15.52	1.40	1.00	1.00	16.06	1.19	1.12	0.86	
	hacl-star/vec512	8.65	0.88	0.72	0.73	9.04	0.58	0.55	0.56	
Poly1305	hacl-star/scalar	12.38	2.83	2.59	2.59	12.76	2.62	2.32	2.31	3.84
	hacl-star/vec128	16.22	1.44	1.29	1.29	15.65	1.19	1.06	1.06	
	hacl-star/vec256	8.71	0.71	0.69	0.69	8.47	0.63	0.54	0.53	
	hacl-star/vec512	5.16	0.45	0.47	0.47	4.84	0.40	0.39	0.40	
Blake2b	hacl-star/scalar	98.20	5.51	4.49	4.50	94.05	4.04	3.97	3.97	11.95
	hacl-star/vec256	47.25	4.56	3.32	3.38	47.48	3.68	3.39	3.63	
Blake2s	hacl-star/scalar	156.55	9.38	7.62	7.65	156.85	6.78	6.64	6.64	20.50
	hacl-star/vec128	70.99	7.96	5.08	5.08	69.81	4.99	4.66	4.59	
SHA-256	hacl-star/scalar	75.04	13.16	12.42	12.41	56.12	11.56	11.37	11.39	20.36
	hacl-star/sha256-mb4	47.52	3.52	3.51	3.51	46.98	3.22	3.22	3.22	
	hacl-star/sha256-mb8	34.02	1.86	1.88	1.85	28.97	1.68	1.69	1.69	
SHA-512	hacl-star/scalar	52.30	8.47	7.92	7.92	36.94	7.45	7.46	7.46	12.64
	hacl-star/sha512-mb4	41.07	2.34	2.32	2.30	35.16	2.07	2.08	2.07	
	hacl-star/sha512-mb8	66.79	1.50	1.51	1.51	44.26	2.33	2.35	2.31	

Table E.2 – SUPERCOP Benchmarks on Dell Precision Workstation with Intel Xeon Gold 5122 processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-9, clang-9, and CompCert3.7.

List of Figures

List of Tables

RÉSUMÉ

Beaucoup de logiciels critiques d'un point de vue sécurité ont besoin d'implémentations d'algorithmes cryptographiques sûres et performantes. Pour répondre à ce besoin, des bibliothèques cryptographiques généralistes comme OpenSSL incluent des douzaines d'implémentations hybrides C/Assembleurs pour chaque primitive, chacune de ces implémentations étant spécialisée et optimisée pour une plateforme répandue. La plupart des optimisations présentes dans ces implémentations exploitent le parallélisme Single Instruction, Multiple Data (SIMD) qui nécessite de changer de manière significative la structure du code, de telle sorte qu'il ne ressemble plus que très peu à l'algorithme scalaire originel.

Cependant, et malgré des années de conception et d'implémentation méticuleuse, des bugs et attaques continuent d'être trouvées dans ces bases de code optimisées. Ces bugs et attaques incluent des erreurs de sûreté mémoire, des fuites d'information par attaque temporelle et des bugs de correction fonctionnelle. La probabilité de détecter ces bugs par des méthodes de test est extrêmement faible, mais ils peuvent quand même être exploités pour conduire une attaque. Nous défendons l'utilisation de méthodes formelles afin de prouver mathématiquement l'absence de tels bugs dans les implémentations. À l'inverse d'autres approches comme le test ou l'audit, l'utilisation de méthodes formelles apporte des garanties forte concernant la sûreté mémoire du code, sa correction fonctionnelle par rapport à une spécification haut-niveau, et enfin son indépendance par rapport aux données secrètes (constant-time) qui protège contre certaines attaques temporelles par canaux auxiliaires. Le défi est ici de vérifier des centaines de milliers de lignes de code extrêmement optimisées ; nous avons donc besoin d'une approche systématique à ce problème.

Cette dissertation présente une nouvelle approche pour la construction d'une bibliothèque cryptographique multi-plateforme formellement vérifiée qui compile du code générique vérifié écrit en F* vers du code C optimisé pour différentes plateformes, que l'on peut également composer avec de l'assembleur vérifiée. Notre approche réduit l'effort de programmation et de vérification en partageant le code entre les implémentations d'un même algorithme pour différentes plateformes, et entre les implémentations de primitives cryptographiques différentes.

Notre bibliothèque incorpore les résultats de trois projets qui sont chacun décrit dans leur propre chapitre. Premièrement, nous montrons comment écrire des implémentations de la courbe elliptique Curve25519 en C pur (pour la portabilité) et en un mélange de C et d'assembleur. Deuxièmement, nous montrons comment écrire des implémentations vérifiées et vectorisées de l'algorithme de chiffrement Chacha20, de l'algorithme d'authentification à usage unique Poly13015, et des familles d'algorithmes de hash SHA-2 et Blake2, pour des plateformes qui supportent des vecteurs SIMD de 128, 256 et 512 bits. Troisièmement, nous montrons comment écrire des implémentations vérifiées en C pur (pour la portabilité) pour les schémas de signature RSA-PSS et Ed25519 pour l'échange de clés en champ fini Diffie-Hellman sur des groupes standards. Ainsi, nous développons une nouvelle version de la bibliothèque cryptographique open-source HACL*, qui inclue par exemple désormais les premières implémentations vérifiées de RSA-PSS et les premières implémentations vectorisées vérifiées sur ARM Neon et AVX512.

MOTS CLÉS

Vérification Formelle, Cryptographie, Standards, Implémentations, SIMD, Arithmétique Multiprécision

ABSTRACT

Many security-critical applications need efficient and secure implementations of cryptographic algorithms. To address this demand, general-purpose cryptographic libraries like OpenSSL include dozens of mixed assembly-C implementations for each primitive, highly optimized for multiple popular platforms. Most of these optimizations exploit single-instruction, multiple-data (SIMD) parallelism that significantly changes the structure of the code, making it no longer resemble the original scalar algorithm.

However, despite years of careful design and implementation, bugs and attacks continue to be found in such optimized code. These include memory safety errors, timing leaks, and functional correctness bugs. The probability of catching such bugs by testing is extremely low, but they still can be exploited to conduct an attack. We advocate the use of formal verification to mathematically prove the absence of such implementation bugs. Unlike other approaches, such as testing and auditing, it provides strong guarantees that code is memory safe, functionally correct against its high-level specification, and secret independent ("constant-time") to protect against certain timing side-channel attacks. The challenge is to verify hundreds of thousands of lines of highly-optimized code, and, hence, a more systematic approach is needed.

This thesis presents a new approach towards building a formally verified high-performance multi-platform cryptographic library that compiles generic verified code written in F* to optimized C code for different platforms, composable with verified assembly. Our approach reduces programming and verification effort by sharing code between implementations of an algorithm for different platforms and between implementations of different cryptographic primitives.

Our library incorporates the results of three projects, each described in its own chapter. First, we show how to write verified portable C and mixed assembly-C implementations for the Curve25519 elliptic curve. Second, we show how to write verified vectorized implementations for platforms that support 128-bit, 256-bit, and 512-bit SIMD vector instructions for the ChaCha20 encryption algorithm, the Poly1305 one-time MAC, and the SHA-2 and Blake2 families of hash algorithms. Third, we show how to write verified portable C implementations for the RSA-PSS and Ed25519 signature schemes and for the Finite-Field Diffie-Hellman key exchange over standard groups. Hence, we developed a new version of the open-source cryptographic library HACL*, which now includes the first verified implementations of, for example, RSA-PSS and the first verified vectorized implementations on ARM Neon and AVX512.

KEYWORDS

Formal Verification, Cryptography, Standards, Implementations, SIMD, Multi-precision Arithmetic