A Verified Extensible Library of Elliptic Curves

Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, Karthikeyan Bhargavan

A Verified Extensible Library of Elliptic Curves

Jean Karim Zinzindohoué
ENPC/INRIA
jean-karim.zinzindohoue@inria.fr

Evmorfia-Iro Bartzia
INRIA
irobartzia@gmail.com

Karthikeyan Bhargavan
INRIA
karthikeyan.bhargavan@inria.fr

***Abstract*—In response to increasing demand for elliptic curve cryptography, and specifically for curves that are free from the suspicion of influence by the NSA, new elliptic curves such as Curve25519 and Curve448 are currently being standardized, implemented, and deployed in major protocols such as Transport Layer Security. As with all new cryptographic code, the correctness of these curve implementations is of concern, because any bug or backdoor in this code can potentially compromise the security of important Internet protocols. We present a principled approach towards the verification of elliptic curve implementations by writing them in the dependently-typed programming language F* and proving them functionally correct against a readable mathematical specification derived from a previous Coq development. A key technical innovation in our work is the use of *templates* to write and verify arbitrary precision arithmetic once and for all for a variety of Bignum representations used in different curves. We also show how to use abstract types to enforce a coding discipline that mitigates side-channels at the source level. We present a verified F* library that implements the popular curves Curve25519, Curve448, and NIST-P256, and we show how developers can add new curves to this library with minimal programming and verification effort.**

I. Verifying Cryptographic Libraries

The security of important Internet protocols, such as Transport Layer Security (TLS), crucially relies on cryptographic constructions implemented in software and hardware. Any bug or backdoor in these implementations can be catastrophic for security. Yet, although the precise computational security of composite constructions and protocols has been widely studied using formal tools [1]–[3], the correctness of implementations of the underlying cryptographic primitives have received far less attention from the formal verification community.

For symmetric primitives, such as block ciphers and hash functions, the algorithm *is* the specification. Hence, verifying a block cipher implementation amounts to proving the equivalence between a concrete program written for some platform and an abstract program given in the standard specification. Practitioners commonly believe that a combination of careful code inspection and comprehensive testing is enough to ensure functional correctness for such primitives, but that the greater challenge is preventing side-channels such as timing leaks that have led to many recent attacks (e.g. see [4]). However, separating these concerns is not always possible or desirable, and researchers have shown that formal approaches can be effective both for verifying subtle performance optimizations in source programs [5], and for detecting side-channels in assembly code [6]. Furthermore, such code-based guarantees can also be formally linked to high-level cryptographic proofs [7].

For asymmetric primitives, such as RSA encryption, finite-field Diffie-Hellman, or elliptic curves, the gap between specification and code can be significantly large. Abstractly, such primitives compute well-defined mathematical functions in some finite field, whereas concretely, their implementations manipulate arrays of bytes that represent arbitrary precision integers (called *bignums*). Furthermore, since asymmetric primitives are typically much slower and can form the bottleneck in a cryptographic protocol, most implementations incorporate a range of subtle performance optimizations that further distance the code from the mathematical specification. Consequently, even for small prime fields, comprehensive testing is ineffective for guaranteeing the correctness of asymmetric primitive implementations, leading to bugs in even well-vetted cryptographic libraries [8], [9]. Even worse, asymmetric primitives are often used with long-term keys, hence any bug that leaks the underlying key material can be disastrous.

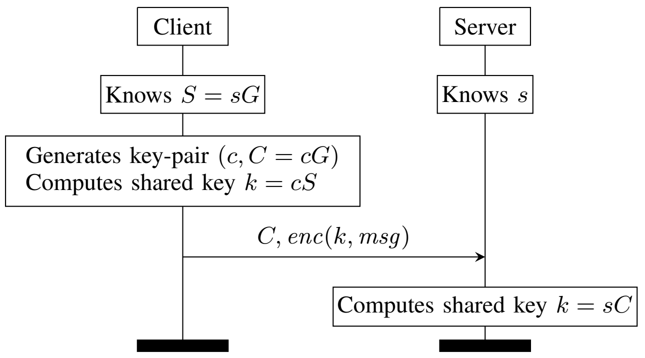
Recent trends in protocol design indicate a shift towards the use of elliptic curves in preference to older asymmetric primitives. This is partly due to concerns about mass surveillance, which means that non-forward-secret primitives such as RSA encryption are no longer considered sufficient. Moreover, finite-field Diffie-Hellman computations are both slow and vulnerable to precomputation [10], two limitations that do not apply to elliptic curves, as far as currently known.

Cryptographic libraries such as OpenSSL already implement dozens of standardized elliptic curves. However, concerns about backdoors in NIST standards [11] have led to the standardization of new elliptic curves such as Curve25519 and Curve448 [12], and implementations of these relatively new curves are currently being developed and widely deployed. The verification of these fresh implementations of new elliptic curves was presented as an open challenge from practitioners to academics at the Real World Cryptography workshop in 2015 [13]. Verifying libraries with multiple curves is particularly difficult because each curve implements its own optimized field arithmetic code, and so two elliptic curves do not share much code between them. This is in direct contrast to RSA and finite-field Diffie-Hellman libraries that are written once and for all and remain stable thereafter.

In this paper, we take up this challenge and show how to write a library of elliptic curves that can be mechanically verified using a dependent type system. To better explain our approach and the design of our verified library, we first consider a motivating example.

A. Example: Elliptic Curve Diffie-Hellman Key Exchange

One of the main applications of elliptic curves in cryptographic protocols is to implement a Diffie-Hellman key-exchange. Figure 1 illustrates a simple elliptic curve Diffie-Hellman protocol inspired by the QUIC transport protocol [14]


Client
Server
Knows $S = sG$
Knows s
Generates key-pair $(c, C = cG)$
Computes shared key $k = cS$
$C, enc(k, msg)$
Computes shared key $k = sC$

```
let send sPub msg =
  let cPriv,cPubBytes = ECDH.keygen () in
  let k = ECDH.shared_secret cPriv sPub in
  (cPub, AEAD.encrypt k msg)

let receive sPriv (cPubBytes,encmsg) =
  if ECDH.is_on_curve cPubBytes then
    let cPub = ECDH.to_spoint cPubBytes in
    let k = ECDH.shared_secret sPriv cPub in
    AEAD.decrypt k msg
  else failwith "error"
```

Fig. 1. An ephemeral-static elliptic curve Diffie-Hellman (ECDH) key exchange sending a single encrypted message (msg). The client and server are assumed to have agreed upon a curve with generator G. The server's static public key S is known to everyone. The client is anonymous. Scalar multiplication (e.g. sG) can be read as a classic Diffie-Hellman modular exponentiation (G^x). Sample F* code for the client and server is shown on the right.

and the 0-RTT mode of TLS 1.3 [15]. The right side of the figure displays example client-server code for this protocol in an ML-like language called F* (e.g. the miTLS library [16] implements all of the TLS protocol in this style.)

An anonymous client wishes to send a secret message msg to a public server. We assume that both have agreed upon a curve (e.g. Curve25519) with public generator G, and we assume that the client has previously retrieved the server's static public key S from some trusted source. In the elliptic-curve setting S is computed as the scalar multiplication sG, where s is the server's private key.

To send the message, the client generates an ephemeral private-public key-pair (c, C), where $C = cG$, computes a Diffie-Hellman shared secret $k = cS$, uses it to encrypt the message, and sends the encrypted message along with its public key C. The server first verifies that C is a valid point on the curve, then computes $k = sC$, decrypts the message, and returns an (unencrypted) error message if it fails.

The main goal of the protocol is that the message msg must remain a secret between the client and the server. This property relies on the secrecy of the private keys, the strength of the encryption scheme, and on some variant of the computational Diffie-Hellman assumption for the group of points on the elliptic curve. However, even if we store our keys securely and choose strong cryptographic parameters, there are many ways our implementation can go wrong.

For example, if the implementation of ECDH.shared_secret is buggy, then the generated key k may be leaked to the attacker, leading to msg being leaked. Alternatively, if the server forgets to call ECDH.is_on_curve to verify the client's public key, or if this check is incorrectly implemented, then the server becomes vulnerable to a small subgroup attack that can reveal the server's private key s [17]. Furthermore, if the time taken by ECDH.shared_secret is not constant in its inputs, it may leak the input keys (c or s) or the output shared secret (k) to an observant network attacker. Finally, if the application code is careless, it may reveal s or msg to the adversary, by accidentally writing them to a public file, for example.

In all these cases, the secret message is leaked to the adversary, and in some cases, even the server's long-term key may be compromised. Consequently, it is essential to verify that the ECDH library is functionally correct, that it does not leak its secret inputs via side-channels, and that its security guarantees are preserved by the application code.

B. Towards a Verified Elliptic Curve Library

The most relevant prior work on verifying an elliptic curve implementation appears in [18], which shows how an implementation of Curve25519 in the qhasm language [19] can be formally verified to be functionally correct using a combination of an SMT solver and the Coq theorem prover. This result is particularly impressive because it applies to highly optimized code written in a low-level assembly language.

Inspired by [18], we propose to build a verified library that consists of multiple elliptic curves that maximally share code, so that the verification effort of adding a new curve can be minimized. However, the approach used in [18] is not particularly well-suited for this purpose. As a low-level programming language, qhasm does not lend itself to modular, incremental, proof-friendly development. The qhasm code for each curve is completely different, so the proof effort would have to be repeated from scratch for a new curve. Even for the same curve, if the underlying bignum representation is changed, say in order to optimize for a different platform, the proof would have to be redone.

To mitigate side-channel leaks via timing and memory accesses, cryptographic code, both in qhasm and in other programming languages, is typically written according to an informal coding discipline that can be summarized as *no branching on secrets* and *no table lookups at secret indices*. This discipline is not currently enforced in qhasm code, and although prior work shows how it can be formally enforced for assembly programs [6], extending qhasm with such sound static analyses can be hard, since it lacks a formal semantics.

We propose an alternative approach that explores a different trade-off between runtime performance and ease of verification. We develop our library of elliptic curves in a high-level programming language with a well-defined formal semantics. We use semi-automated tools to verify the functional correctness of our code, and we enforce a source-level coding discipline that mitigates side-channels. Furthermore, by using the same language and verification technique for both cryptographic libraries and protocol code, we can safely embed our library into larger verified protocol stacks.

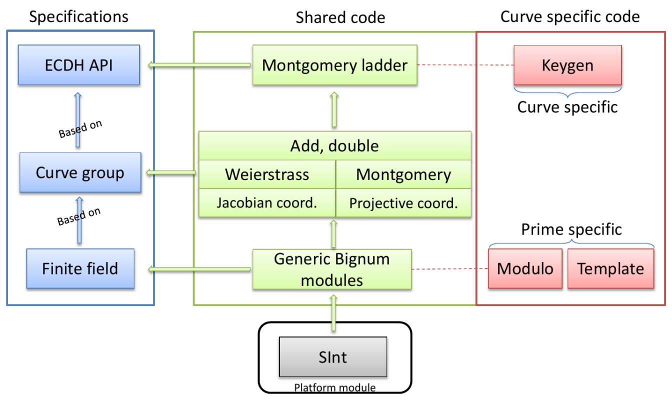


Fig. 2. Architecture of our verified elliptic curve library

C. *Our Approach and Contributions*

We write a new elliptic curve library in F*, a dependently-typed variant of the ML programming language [20]. The library provides a typed API that encodes the mathematical specification of elliptic curves. Each curve in the library is proved to satisfy this specification by typing. Our curve code is stateful and implements the same algorithmic optimizations as state-of-the-art elliptic curve implementations. In particular, we implement a bignum library that allows each curve to choose its own unpacked bignum representation (called a *template*) and obtain verified field arithmetic for free, except for a few curve-specific functions that need to be implemented and verified separately. Mitigations against side-channels are systematically enforced throughout the library by treating secrets as opaque bytestrings whose values cannot be inspected.

Figure 2 illustrates the architecture of our library. The modules on the left constitute the library API encoding the mathematical specifications of finite fields, elliptic curves, and ECDH as F* interfaces. A user of the library only needs to inspect these interfaces to understand the functionality provided by the library. F* typechecking guarantees that each elliptic curve implementation meets these interfaces. Our F* definitions closely correspond to the Coq definitions for elliptic curves developed in prior work [21]. However, although we rely on close syntactic proximity between F* and Coq, we do not present a formal proof of this correspondence.

The library is designed to share as much code as possible between different curve implementations. The modules in the middle contain shared code, including a generic bignum library implementing field arithmetic, and generic implementations of Weierstrass and Montgomery curves, using Jacobian and Projective coordinates, respectively. The equivalence between the optimized implementation of the curve operations in those coordinate systems and the ones in standard Weierstrass form is verified by a SAGE script. We do not prove the formal correspondence between the SAGE script and our F* and Coq definitions; we assume that the mathematical theory of integers is consistent between these three systems.

All this code relies on a platform module SInt that encapsulates machine integers as an abstract type and implements low-level constant-time operations on them. The other modules cannot access the concrete contents of these integers and hence cannot branch on them or use them as array indices.

To extend the library with a new curve, a programmer must write and verify the modules on the right, including the curve parameters, a bignum representation (*template*), and an optimized *modulo* function for the underlying prime. Some curves may also need a specialized key generation function.

Our main contribution is the first verified elliptic curve library that covers three popular curves—Curve25519, Curve448, and NIST-P256—and that can be easily extended with new curves. We also present a verified generic bignum library, which is of independent interest. Our use of F* allows for more generic code and thus more sharing, while keeping the implementation effort reasonable. Furthermore, our library can be readily incorporated into larger verified cryptographic applications written in F*, such as miTLS.

Conversely, by writing our library in a high-level programming language, we introduce a significant gap between the verified source code and the compiled executable, which leads to performance penalties and requires additional trust assumptions on the compiler and runtime libraries. Our code is about 100 times slower than optimized C code. Furthermore, any side-channel mitigations enforced in our source code may not be preserved in the compiled code. While both these limitations may be addressed by careful compiler design (see e.g. [22]) we leave these improvements for future work.

Outline: Section II gives some background and shows how elliptic curves are specified in F*. Section III presents the SInt module and shows how it prevents overflows and enforces a coding discipline that mitigates side-channels. Section IV describes our generic bignum library. Section V presents our three elliptic curve implementations. Section VI analyzes our results. Section VII summarizes related work.

II. Specifying Elliptic Curves in F*

A. *Elliptic curves, briefly*

We present some basic mathematical definitions for elliptic curves, which may be helpful to better understand the coming sections. For more details, we refer the reader to the substantial mathematical literature on elliptic curves (e.g. [23]) and to previous formal developments [21].

An elliptic curve is a special case of a projective algebraic curve, defined as follows. Let K be a field of characteristic different from 2 and 3 — K is typically a prime field $\mathbb{Z}/p\mathbb{Z}$ for some large prime p. An *elliptic curve* $\mathcal{E}$ is defined by a Weierstrass equation of the form:

$$\mathcal{E}_W : y^2 = x^3 + ax + b$$

where a and b are in K and the curve has no singularity, that is, the discriminant $\Delta(a,b) = 4a^3 + 27b^2$ is equal to 0.

Figure 3 depicts an elliptic curve. The set of points of $\mathcal{E}_W$ is formed by the solutions (x, y) of the equation augmented by a distinguished point $\mathcal{O}$ called the *point at infinity*. This set can be equipped with an abelian group structure by giving the following geometrical definition to the sum operator.

Let P and Q be points on the curve $\mathcal{E}_W$ and l be the line that goes through P and Q. If $Q = P$ then l is the tangent to the curve at P and in that case the sum $P +_{ec} P$

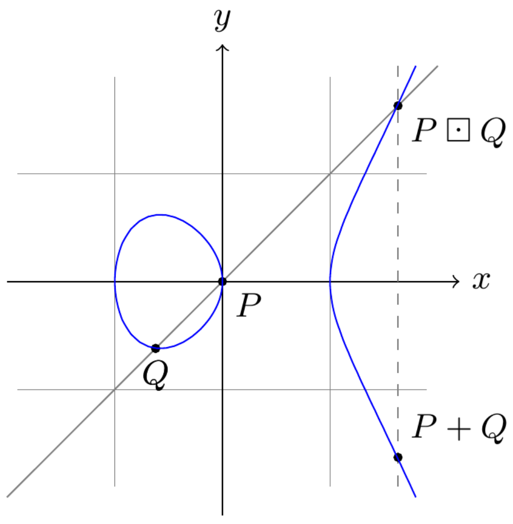


Fig. 3. An elliptic curve and its group sum operator

is called doubling. l intersects E at a third point (counting multiplicities), denoted by $P \boxdot Q$. The sum $P +_{ec} Q$ is the opposite of $P \boxdot Q$, obtained by taking the symmetric of $P \boxdot Q$ with respect to the x axis. We complete the definition of $+_{ec}$ as follows:

- $\mathcal{O}$ is the neutral element:
  $\forall P.\, P +_{ec} \mathcal{O} = \mathcal{O} +_{ec} P = P$,
- the opposite of a point $P = (x_P, y_P)$ (resp. $\mathcal{O}$) is $(x_P, -y_P)$ (resp. $\mathcal{O}$), and
- if three points are collinear, their sum is equal to $\mathcal{O}$.

This geometrical definition can be translated into an algebraic setting, which yields polynomial formulas for addition, doubling, scalar multiplication etc. For certain choices of curve parameters and point representations, these computations can be made significantly more efficient. For example, some curves, such as Curve25519, have equations that can be written in the so-called Montgomery form:

$$\mathcal{E}_M : By^2 = x^3 + Ax^2 + x$$

where A, $B \in K$, K is of characteristic different from 2, $A \neq -2, 2$, $B \neq 0$, and $B(A^2 - 4) \neq 0$.

In particular, if the characteristic of the field K is different from 3 then the following function ψ maps a Montgomery curve to an equivalent Weierstrass curve $v^2 = u^3 + au + b$, where $a = \frac{3-A^2}{3B^2}$ and $b = \frac{2A^3-9A}{27B^3}$:

$$\psi : \mathcal{E}_M \to \mathcal{E}_W : (x, y) \mapsto (u, v) = \left(\frac{x}{B} + \frac{A}{3B}, \frac{y}{B}\right)$$

Both these curve representations harbor the same abelian group structure although the sum operation is implemented by different polynomial formulas. The key difference is that the Montgomery formulas are easier to implement efficiently, and in a way that resists side-channels.

The rest of this paper presents verified implementations of both Weierstrass curves (NIST-P256) and Montgomery curves (Curve25519, Curve448) in their different algebraic settings.

B. F* Syntax, briefly

All the code and specifications in this paper are written in F*, an ML-like functional programming language with a dependent-type system designed to aid program verification. We assume the reader's familiarity with usual ML syntax and types. Here, we briefly explain F*-specific features used in this paper. For full details on F*, see [20] and the online tutorial.[1]

The F* type system includes refinement types of the form v:t{Phi(x)} which represent values v of type t for whom the first-order logical formula Phi(v) holds. To verify that an expression has this type, the F* typechecker queries an external SMT solver to discharge the formula. Types can be indexed by values, by types, and by predicates. For example, the type- and value-indexed type larray 'a (len:int) may represent arrays containing len elements of type 'a. Both the SMT encoding and the SMT solver Z3 [2] are trusted to be correct.

F* also provides a lattice of computation effects that appear in function types of the form f:Type → Effect Type. The Tot effect specifies that the function is total: it is has no side-effects and it always terminates. The GTot effect applies to total *ghost* functions that are computationally irrelevant and erased by the compiler. Only Tot and GTot functions can appear in types and formulas. Lemmas are a subclass of Tot functions that return unit refined with a logical theorem. That theorem must first be proven, and can then be invoked within code to aid in the verification of more complex properties.

The ST effect specifies stateful, exception-free functions that modify an implicit *heap* modeled as a map from references to values. Functions with the ST effect are written with explicit stateful pre- and post-conditions of the form **requires** (**fun** h0 → Pre) and **ensures** (**fun** h0 r h1 → Post), where h0 refers to the heap when the function is called, h1 is the heap when the function returns, and r is the return value.

When no effect is specified, a function has the default ALL effect, which means that the function may throw exceptions, modify state, or fail to terminate.

The F* syntax allows the definition of new infix operators, and in the code listings in this paper we prettyprint some functions and operators to improve readability. For instance, pow2 n is shown as 2^n, pow x n as x^n, FStar.Prims.nat as $\mathbb{N}$, `forall` as $\forall$, <> as $\neq$, etc.

C. Elliptic Curves in F*

The proof that an elliptic curve implements an abelian group is not trivial, but we can rely on an existing formalization and mechanized proof in Coq for a general elliptic curve theory [21]. We carefully transcribe the Coq definitions used in that work as an F* interface and reflect their Coq theorems as F* assumptions. Figure 4 displays the Curve interface in F*. For reference, we show a fragment of the corresponding Coq theory in Appendix A. While there is no formal link between F* and Coq, we impose an informal discipline whereby all unverified elliptic curve assumptions in F* must be justified by a corresponding theorem in Coq.

[1]https://www.fstar-lang.org/#tutorial
[2]https://github.com/Z3Prover/z3

```
type AbelianGroup (#a:Type) (zero:a) (opp:a → Tot a)
                          (add:a → a → Tot a) =
  (∀ x y z. add (add x y) z = add x (add y z)) // Associative
  ∧ (∀ x y. add x y = add y x) // Commutative
  ∧ (∀ x. add x zero = x) // Neutral element
  ∧ (∀ x. add x (opp x) = zero) // Inverse

(* Field elements, parameters of the equation *)
val a: felem
val b: felem
val is_weierstrass_curve: unit → Lemma
   (4 +* a$^3$ ^+ 27 +* b$^2$ ≠ zero ∧ characteristic ≠ 2 ∧
  characteristic ≠ 3)

type affine_point =
   | Inf | Finite: x:felem → y:felem → affine_point

let on_curve p = is_Inf p || (is_Finite p &&
  (let x, y = get_x p, get_y p in y$^2$ = (x$^3$ ^+ a ^* x ^+ b)))
type CurvePoint (p:affine_point) = b2t(on_curve p)

let neg' p = if is_Inf p then Inf
  else Finite (Finite.x p) (−(Finite.y p))

let add' p1 p2 =
   if not(on_curve p1) then Inf
   else if not(on_curve p2) then Inf
   else if is_Inf p1 then p2
   else if is_Inf p2 then p1
   else (
      let x1 = get_x p1 in let x2 = get_x p2 in
      let y1 = get_y p1 in let y2 = get_y p2 in
      if x1 = x2 then (
         if y1 = y2 && y1 ≠ zero then (
            let lam = ((3 +* (x1$^2$) ^+ a) ^/ (2 +* y1)) in
            let x = ((lam$^2$) ^− (2 +* x1)) in
            let y = ((lam ^* (x1 ^− x)) ^− y1) in
            Finite x y
         ) else (...)))

(* Type of points on the curve *)
type celem = p:affine_point{CurvePoint p}
val neg: celem → Tot celem
val neg_lemma: p:celem → Lemma (neg p = neg' p)
val add: p:celem → q:celem → Tot celem
val add_lemma: p:celem → q:celem → Lemma (add p q = add' p q)

val ec_group_lemma:
   unit → Lemma (AbelianGroup #celem Inf neg add)

(* EC multiplication of a point by a scalar *)
val smul : ℕ → celem → Tot celem
let smul n p = match n with
   | 0 → Inf | _ → add p (smul (n−1))
```

Fig. 4. An Elliptic Curve Specification in F*

In Figure 4, the AbelianGroup predicate gives a textbook definition of an abelian group equipped with a neutral element zero, an opposite function opp and addition operator add. The curve definition assumes a finite field K represented by elements of type felem. Points on the plane are of type affine_point: they can either be Inf, the point at infinity, or a pair of felem coordinates.

A point that verifies the curve equation satisfies on_curve and can be given the refined type celem, denoting curve elements. We define two operations over curve points: a negation function neg and an internal group operation add. The ec_group_lemma says that Inf, neg, and add together form an abelian group structure. Hence, we can define scalar multiplication as repeated addition over the curve.

We rely on the proofs in [21] to justify three assumptions in this F* interface:

- neg_lemma: the opposite of a point on the curve is on the curve;
- add_lemma: the result of the addition of two points on the curve is also on the curve;
- ec_group_lemma: the elliptic curve is an abelian group.

The rest of this paper shows how we implement this Curve interface with different elliptic curves. We did not display the Field interface that defines operations on felems, but we will show how standard (modular) field arithmetic is implemented by our bignum library. But first, we describe a platform module that implements arithmetic on machine words.

III. Secrecy-Preserving Integer Arithmetic

Our library relies on a platform-specific module SInt that implements arithmetic and bitwise operations on machine integers and bytes. We carefully design the typed interface of this module to abstract away from the underlying machine representations. This serves several purposes. First, our elliptic curve and bignum libraries can be platform-agnostic for the most part, allowing us to support both 32-bit and 64-bit architectures with minimal changes to the code. Second, we enrich the SInt interface with pre-conditions that guarantee that applications using this module cannot cause overflows. Consequently, all operations preserve the natural mapping from machine integers to mathematical integers. Third, we consistently treat all machine integers as potentially secret by encapsulating them within abstract types. Therefore, our elliptic curve code (or even other applications of SInt) cannot compare secret machine integers or convert them to concrete types like bytes or booleans. Hence, typing enforces a side-channel mitigation discipline that prevents programs from branching on secret values or accessing arrays at secret indices.

The rest of this section describes the interface of SInt but we do not prescribe any implementation. SInt may be implemented in F*, OCaml, C, or even in assembly. We assume that the implementation correctly realizes the mathematical operations specified in the interface and that all operations are constant-time (i.e. their execution time is independent of the inputs).

A. *Platform-specific Types for Machine Words*

The SInt module defines machine words of three different sizes: a byte is a word of 8 bits, a limb is a word of SIZE bits where SIZE is a constant specific to a certain platform (typically 32 or 64), and a wide is a word of 2*SIZE bits. All three types are abstract, so the only way for an application to manipulate values of these types is through operations and functions provided by the SInt interface.

For example, SInt defines several ways of constructing abstract limbs. We can use the zero_limb and one_limb constants, or we can convert native F* integers to limb, as long as they are of the right size. We can also inter-convert between byte, limb, and wide, as long as the source value fits into the target word. Once we obtain abstract limbs, SInt defines the following operations on them:

- arithmetic operations : add_limb, mul_limb, sub_limb, mod_limb, neg_limb and div_limb;
- bitwise operators: log_and_limb, log_xor_limb, log_or_limb, log_not_limb, shift_left_limb, shift_right_limb;
- masking operators: eq_limb which returns a mask of all ones if the two inputs are equal, a mask of all zeros if not, and gte_limb which returns a mask of ones if the first argument is greater than or equal to the second one, of all zeros otherwise.

Notably, there are no functions that can extract concrete types from abstract limbs, and the F* typechecker statically enforces that we cannot (say) compare two limbs and obtain a boolean.

B. Mapping machine words to mathematical integers

In order to prove mathematical properties about code that uses SInt, we need to map each abstract type in SInt to its corresponding mathematical value. In our setting, we only consider unsigned words, and we map each word to the (positive) integer value of its binary representation. This mapping is defined as a ghost function v from an abstract platform type sint to mathematical integers ($\mathbb{Z}$):

```
val v: sint → GTot ℤ
```

Using the type sint and the evaluation function v, we can then formally specify our three platform types as follows:

```
val bitsize: x:ℤ → n:ℕ → GTot bool
let bitsize x n = (x ≥ 0 && x < 2^n)

type usint (n:ℕ) = x:sint{bitsize (v x) n}
type byte = usint 8
type limb = usint SIZE
type wide = usint (2*SIZE)
```

The indexed type usint is parameterized by a positive integer representing the number of bits in the underlying machine word representation. Hence, the usint n type represents unsigned integers of n bits that map to mathematical integers between 0 and $2^n - 1$ (inclusive).

C. Arithmetic Operations and Overflow Prevention

The SInt interface also specifies the semantics of each operation on platform types in terms of mathematical integers, using the v mapping. We do not model modular machine arithmetic, where values wrap around when they exceed the word size. Instead, we conservatively treat overflows as irrecoverable errors and require that applications using SInt must guarantee that such overflow errors will never occur.

For example, we define addition on limbs as follows:

```
val add_limb: x:limb → y:limb{v x + v y < 2^SIZE} →
              Tot (z:limb{v z = v x + v y})
```

The precondition states that the function can only be called on values x, y for which the F* typechecker can statically prove that v x + v y $< 2^{SIZE}$, meaning that there will be no overflow. This specification is more restrictive than usual definitions. Overflows are forbidden by the type system rather than having undefined or wrapping semantics. For instance, an alternative could have been to use a more relaxed constraint:

```
val add_limb: x:limb → y:limb →
        Tot (z:limb{v x + v y < 2^SIZE ⟹ v z = v x + v y})
```

Here, the addition operator can be called on any input values, but the value of the output will only be defined as long as the system could prove that no overflow occurred. We choose the former style as it is more likely to catch programming errors early. Furthermore, we find that the F* typechecker can typically easily prove these overflow preconditions by using the SMT solver, so we do not find the more restrictive interface to be particularly onerous.

D. Secrecy with Abstract Types and Constant-Time Operations

The SInt interface has no functions that allow the abstract machine words to be converted to concrete F* types like integers or booleans. Note that the v function has the GTot effect and hence cannot be used in concrete code; it appears only in specifications and annotations.

Hence, by construction, F* code cannot branch on the values of machine words, or use any value derived from a machine word as an index to access a memory location. Consequently, an F* program that represents a secret as a limb is forced to enforce source-level side-channel mitigations. Although we do not formally prove any side-channel prevention theorem, we note that our types enforce the same rules that have been shown to guarantee system-level non-interference in previous work [6]. In a sense, we are using the sint abstract type to control information flow between secret machine words and public F* types. This technique of using parametricity and type abstraction to provide noninterference has been well-studied in various settings(e.g. see [24]).

In our setting, programs cannot directly compare secret machine words, but we can still use masking to safely implement conditional arithmetic computations. For example, the eq_limb function is meant to return a limb containing all zeros if the two arguments are unequal:

```
val eq_limb: x:limb → y:limb →
  Tot (z:limb{(v z=0∧v x≠v y) ∨ (v z=2^SIZE − 1∧v x=v y)})
```

This function can be implemented in constant-time using bitwise operators. For example, the C code for this function on a 32-bit platform could be written as follows:

```
unsigned int eq_limb(unsigned int x, unsigned int y){
  unsigned int a; int b;
  a = ~(x ^ y);
  a &= a << 16;
  a &= a << 8;
  a &= a << 4;
  a &= a << 2;
  a &= a << 1;
  b = ((int)a) >> 31;
  (unsigned int)b;}
```

Name	Prime	Templ.	Type	Coord.	SAGE
Curve25519	$2^{255} - 19$	$i \to 51$	Montg.	Projective	[25]
Curve448	$2^{448} - 2^{224} - 1$	$i \to 56$	Montg.	Projective	[25]
NIST-P256	$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$	$i \to 32$	Weierst.	Jacobian	[26], [27]

Fig. 5. Specific settings for the implemented elliptic curves: the prime for the underlying field, the template used, the curve type, Montgomery or Weierstrass, the coordinate system and the addition equations proved correct via the SAGE software.

This function can then be used to implement conditional swapping that is used, for example, in the Montgomery ladder implementing scalar multiplication in our elliptic curve library.

More generally, we enforce a discipline where application code must cleanly separate public byte arrays (bytes) that are native to F* from secret byte arrays (sbytes) that are defined in terms of SInt: bytes can be converted to sbytes but not vice versa. We assume that the lengths of secret byte arrays are public, which is true for our example code, We can then guarantee by typing that our elliptic curve library does not leak any secrets, and even protects them against side-channels, at least in the source language. Once compiled to machine code, additional side-channel leaks may appear, but we leave their detection or prevention using techniques like [6] for future work.

IV. A VERIFIED GENERIC BIGNUM LIBRARY

Elliptic curves are built on top of a prime field, and to be robust against current attackers, they have to provide more than a hundred bits of security, meaning that the prime has to be at least two hundreds bits long: on the curves we implement, Curve25519, NIST-P256 and Curve448 those are respectively 255, 256 and 448 bits long. As no widely deployed platform today handles natively such large integers, we have to rely on a bignum library. This library being the core of the elliptic curve implementation, it needs two crucial properties. The first one is efficiency. If it is slow then the whole curve implementation, which uses tuples of bignums to represent points will suffer from it. The second one is resistance to side channel attacks, in particular timing attacks. In the previous section we detailed how we provide that for machine words. Relying on this construction, we present how to implement a modular bignum library that satisfies those requirements. Although the library's current performances are still far from hand-optimized C code, we implement low-level optimizations that, as F* extraction mechanism improves, will allow us to gradually close the gap with existing implementations.

While it fits the elliptic curve implementation needs, it is more general and can be adapted to any other cryptographic primitives requiring a constant-time modular API.

Our implementation and proofs rely on those three features:

1) We present *templates* to provide a generic encoding for unpacked representations of bignums, thus making the code parametric and general purpose;
2) The functional correctness of all arithmetic operations in the library is proven against the finite field specifications exposed in the Field module thanks to a mapping between the bignum encoding and the mathematical field,
3) The modular structure of the code allows for efficient prime specific functions to be plugged in the code without breaking the proof. This allows prime specific functions to be implemented securely without loss of performances and at a low incremental cost now that the generic code has already been written.

A. Representation and Templates

1) Motivations: When implementing a bignum library one has two possibilities to represent the data: either going for a *packed* or an *unpacked* representation. The packed representation is such that the bignum is stored in an array of words which are fully used: the mathematical value of the bignum is intuitively the concatenation of all the words of the array in big-endian representation. Hence when computing on two such values, carries must be taken into account if overflows occur. While this is satisfying and efficient in a completely general context, to ensure constant-time execution carries have to be systematic, which leads to an inefficient library as YF Chen and al. [18] showed for the Curve25519 case.

Therefore in our setting it becomes more efficient to represent bignums as unpacked arrays. It means that when in its normal form, some of the most significant bits of each of its word are left empty. As an example, an unpacked representation for a bignum of 448 bits (as used in Curve448) is an array of 8 64-bits words in which only the first 56 bits would be used, when the packed representation only requires 7 words of 64-bits. Now because the unpacked representation has some additional space, several operations may take place on those words without having to worry about carries and yet without overflowing. For instance a 64-bits word can store the addition of 256 56-bits values before overflowing. Hence, even though the unpacked representation is not as memory efficient as the packed one, when carefully used it provides more efficient algorithms in a constant-time setting. Popular implementations of curves such as NIST-P256 or Curve25519 indeed use such unpacked representations.

*2) Templates: template*s are used to encode the representation of bignums. Their type definition is the following:

```
type template = ℕ → Tot ℕ*
```

It represents the weight of each limbs, that is the number of bits they should be encoded on when the bignum is in its normal form. While the word size is given by the size parameter n passed to the platform usint n type, the number of bits concretely used is given by the template. If $\forall i \in \mathbb{N}, t(i) = n$, the representation is packed. Our library uses such a representation for serialized bignums for instance, represented as arrays of byte. In that case, since all the bits of the bytes are used, the resulting template is $t_{bytes} : i \to 8$ (recall that **type** byte = usint 8).

If $\forall i \in \mathbb{N}, t(i) < n$ then the representation is unpacked, which is the case of the internal representation of all bignums in our library when deserialized. We will only consider unpacked representations here.

3) Evaluating Bignums: The template tells us how to evaluate the mathematical value of a bignum. Given b a bignum, $i \in \mathbb{N}^*$:

$$w(b,n) = \sum_{i=0}^{n-1} b.t(i)$$

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

where b.t is the template associated to the bignum b. Intuitively, the representation of a bignum with a certain template corresponds to writing it in the template's base. Just as the hexadecimal representation of a number corresponds to writing it in base 2^4, the template $t_{56} : n \rightarrow 56$ used for Curve448 for instance corresponds to the equivalent base 2^{56}. Generally although we will consider mostly constant templates in this work, more sophisticated ones could be used, like $t_{26/25} : n \rightarrow 26 - (n\%2)$ for Curve25519 on a 32-bits platform[3].

The eval function is the inverse of the decomposition of the integer on a base: from the concrete representation it computes back the mathematical value. Note that for it to be correct, the concrete value of the words need not be smaller that the template indicates. It only means that the eval function is surjective: two encodings may represent the same integer. Intuitively $w(b,i)$ computes the weight of the value stored in the i-th limb of the bignum b from its template, while $eval(b,i,len)$ is the weighted sum of the len first limbs of the same bignum.

We encode bignums, templates and the eval function in F*:

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

val w : t:template → n:ℕ → GTot ℕ
let rec w t n = match n with
  | 0 → 0 | _ → t (n−1) + w t (n−1)

val eval : h:heap → #size:ℕ* → b:biginteger size{ live h b } →
           n:ℕ{ n ≤ getLength h b } → GTot ℕ
let rec eval h #size b n = match n with
  | 0 → 0
  | _ → pow2 (w b.t (n−1)) ∗ v (getValue h b (n−1)) + eval h b (n−1)
```

The w and eval functions are as presented above. The eval function takes an extra parameter h which is a heap representing a memory state. Indeed as the **array** type used to represent the bignum's data is mutable, the specification has to take the memory state into account, passed implicitly in the ST effect.

Also, similarly to the v function presented earlier for the sint type, the eval function which is purely aimed at specification is ghost and will be erased by the compiler. It shall never be executed and thus has no impact on performance nor secrecy. This is statically enforced by F*: since eval calls to v the verifier will refuse to typecheck any other effect that GTot.

B. Verifying Generic Bignum Operations

1) A Stateful Specification: We now have a way to abstractly evaluate our bignums: all bignums are mapped via their template to the mathematical value they encode. But writing a computationally intensive library such as a cryptographic library in the pure fragment is extremely inefficient. Indeed, as there are no side effects, in the pure setting all new values are fresh, leading to extremely inefficient code. For instance updating a word in a bignum would result into issuing a fresh copy of the whole data. Hence, we chose to make our library stateful, using mutable data types, in particular the native F* **array** type.

There are several ways to encode the memory state in F* [20], but here we chose to use the ST effect with a single heap because it has enough granularity for what we need to write. In this setting, the implicit heap object is encoded as a map from references to objects, which can be updated through stateful ST computations.

As in our setting the bignums' data is represented as a mutable **array**, the stateful encoding of the code is crucial to the functional correctness proof. The listing below presents the specification of a limb to limb addition operation which takes two bignums a and b as input and sums inplace the content of each of their limbs, storing the result in the corresponding limbs of the input a.

```
type IsSum (h0:heap) (h1:heap) (a:bigint{...}) (b:bigint{...}) (ctr:ℕ) =
  (∀ (i:ℕ). (i≥ctr ∧ i<l) ⟹ (v (get h1 a i) = v (get h0 a i) + v (get h0 b i)))
  // l is a parameter associated to the prime (e.g. 8 for curve448)

val sum:
  a:bigint → b:bigint{Similar a b} → ST unit
    (requires (fun h → Normalized h a ∧ Normalized h b))
    (ensures (fun h0 u h1 →
      Normalized h0 a ∧ Normalized h0 b
      ∧ Normalized h1 b
      ∧ (live h1 a) ∧ (modifies !{getRef a} h0 h1)
      ∧ (getLength h1 a = getLength h0 a)
      ∧ (IsSum h0 h1 a b 0)
      ∧ (eval h1 a l = eval h0 a l + eval h0 b l)))
```

The code being parameterizable, let us assume that we are using Curve448 specific values: the template for Curve448 is t_{448}, the canonical length of a bignum is 8 words (the l variable) and the word size SIZE is 64 bits. Properties starting with a capital letter are predicates. As we are in the ST effect, the pre and post-conditions are parametrized by the heap. The Normalized predicate holds for both bignums a and b in the pre-state h.

```
type Normalized (h:heap) (#size:ℕ*) (b:biginteger size) =
  live h b ∧ getLength h b ≥ norm_length
  ∧ (∀ (n:ℕ). n < getLength h b ⟹
          bitsize (v (getValue h b n)) (getTemplate b n))
```

It specifies that the bignum's data is canonically formatted under the template's unpacked representation. In our example it requires the bignum's array to be a live reference to a memory block, to be at least 8 limbs long, each limbs to be both greater than or equal to 0 and less than 2^{56} and the associated template to be t_{448}. The refinement on b, the Similar predicate

```
type Similar (#size_a:ℕ*) (a:biginteger size_a)
             (#size_b:ℕ*) (b:biginteger size_b) =
  (getTemplate a = getTemplate b) ∧ (getRef a ≠ getRef b)
```

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

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

2) Proving Functional Correctness: Given those properties, the verification system guarantees that the post-condition holds in the resulting memory state h1. The **modifies** clause states that only a's data is modified through the course of the function. Given the separation condition between a and b enforced by the Similar predicate, that implies that b is left untouched by sum, and is therefore still Normalized. Now, as the reference a has been modified, we need to indicate and prove that it still exists and points to valid data when the function returns. That is what the live condition gives us. This liveness condition is also a prerequisite to be allowed to express logical properties on a's data in the h1 environment, such as the fact that the length of the underlying array has been left unchanged.

Next are the key properties. First the IsSum predicate expresses what was computed by the function: the l first limbs of a in state h1 contain the sum of the respective a and b limbs in state h0. Knowing that the bignums were Normalized in the initial h0 state, it allows for further proofs on the size of the limbs of a in h1 as it keeps track of the ranges of possible values in each limb. That is necessary to prove the absence of overflows in the next steps.

Second, the equality on the eval function is what guarantees the functional correctness of the sum function. Indeed we show that the result of the sum function maps to a mathematical integer which is the (integer) sum of the mappings of the inputs. In a second step, after computing the modulo function, the modular addition with be proven correct not with regard to the integer arithmetic, but with regard to finite field $\mathbb{Z}/p\mathbb{Z}$.

Here is a flavor of how a step of the computation is proven:

```
type NotModified (h0:heap) (h1:heap) (a:bigint...) (ctr:ℕ) =
  (∀ (i:ℕ). ((i ≠ ctr ∧ i < getLength h0 a) ⟹
                      getValue h1 a i == getValue h0 a i))

val sum_index:
  a:bigint → b:bigint{Similar a b} → ctr:ℕ{ctr≤l} →
  ST unit
    (requires (fun h → (live h a) ∧ (live h b)
      ∧ (l ≤ getLength h a ∧ l ≤ getLength h b)
      ∧ (∀ (i:ℕ). (i ≥ ctr ∧ i < l) ⟹
          (v (get h a i) + v (get h b i) < 2^SIZE)) ))
    (ensures (fun h0 _ h1 →
      (live h0 a) ∧ (live h0 b) ∧ (live h1 a)
      ∧ (l ≤ getLength h0 a) ∧ (modifies !{getRef a} h0 h1)
      ∧ (getLength h0 a = getLength h1 a)
      ∧ (IsSum h0 h1 a b ctr) ∧ (NotModified2 h0 h1 a ctr)))
let rec sum_index a b ctr =
  match l − ctr with
  | 0 → ()
  | _ →
    let ai = index_limb a ctr in let bi = index_limb b ctr in
    let z = add_limb ai bi in
    upd_limb a ctr z;
    sum_index a b (ctr+1)
```

Given that the functions called the in body of sum_index have been properly specified, F* is able to automatically prove that if the **requires** clause is satisfied, then the **ensures** clause will hold when the function returns.

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

After proving by induction the following lemma:

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

which is based on the IsSum predicate, we prove the equality on the eval functions in the sum function post-condition.

Calling this lemma after the sum_index function allows us to prove the sum complete specification. The concrete code has to provide some additional intermediate lemmas to help the prover and make it more efficient and more flexible to amend the code without breaking the proof, but these are the key steps.

C. Prime Specific Code

We implement constant-time modular arithmetic on bignums in a given prime field. Therefore the modulo operation is crucial to allow us to efficiently run these computations. In modern cryptography, and in particular for the curves we are considering, the primes have been carefully chosen so as to allow for efficient constant time modulo reductions. Still, the way these reduction operations can be efficiently implemented depends on the value of each prime and cannot be parametrized. Thus these steps have to be re-implemented by a programmer when extending the library with a new primitive.

Inspired from existing libraries, we implement five distinct functions relying on the prime value:

1) freduce_degree takes a bignum a of size 2∗l−1 where l is the length in the canonical unpacked representation, and returns a bignum b of size l of mapping to the same value in the field;
2) freduce_coefficient will proceed to two carry passes on the bignum b and return a Normalized c bignum encoding the same field element as b;
3) freduce_complete will take a Normalized bignum c and process it into d such that there exists a bijection between d's encoding and the elements of the prime field they encode;
4) An inversion function crecip, which from a normalized bignum encoding the field element e computes the normalized encoding of $\frac{1}{a} = e^{p-2}$;
5) Additionally add_big_zero which takes a normalized bignum f and adds of multiple of the prime to it such that it prevents underflow in subtractions.

This last function is necessary in our setting where we only use unsigned integers to represent the limbs of the bignums. Our specification of the subtraction function on sint does not allow for underflow, so one always has to prove that $a \geq b$ before computing $a - b$. Hence when subtracting two bignums limb to limb, it is necessary to have that $\forall i \in \mathbb{N}, i < l \implies a[i] \geq b[i]$. To get this property while not modifying the encoded values of the bignums, we add to the bignum a a multiple of the prime, typically $2p$ or $4p$ encoded in such a way that $\forall i \in \mathbb{N}, i < l \implies p_{multiple}[i] \geq 2^{56}$ in our example, and then compute the value of a' such that $\forall i \in \mathbb{N}. i < l \implies a'[i] = a[i] + p_{multiple}[i]$ minus b limb to limb. As we encode the prime multiple to meet those specific constraints on each of its limbs, we cannot do it generically and the programmer has to provide such an encoding. We give examples for the primes of Curve25519, Curve448 and NIST-P256.

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

1) Functionally Correct API: For the internal intermediate computations the first two reduction functions are sufficient to implement functionally correct modular arithmetic. Indeed, a Normalized array will meet all prerequisite for future computations so there is no need to reduce it further once that predicate is satisfied. Nevertheless, given that our bignum library encodes prime field elements, the returned encoding must be unique for each element of the prime field. And a Normalized bignum does not satisfy this condition. Sticking to the example of Curve448, the prime has value $p_{448} = 2^{448} - 2^{224} - 1$. The unpacked representation guarantees the uniqueness of the representation of integer values between 0 and $2^{448} - 1$ included, which means that values greater than or equal to p_{448} and less than 2^{448} have two valid encodings. Our last freduce_complete function takes care of this issue making sure that returned values are in bijection with the prime field. However, as it is costly to implement in constant time and not required for the correctness of the internal computations, freduce_complete will only be computed once, when serializing data.

The modular addition operation of bignums of limbs is eventually exposed as

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

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

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

V. Verified curves

A. Three Code Sharing Implementations

To illustrate the extensibility feature of our library we implement three popular curves. While Curve25519 and Curve448 are very closely related as far as their structure is concerned [12], NIST-P256 is quite different, since it is a Weierstrass curve while the other two are Montgomery curves (see II). In the rest of this section we detail our verification approach and why it is largely independent from the curves.

B. Two Coordinate Systems

1) Different Settings for Montgomery and Weierstrass Curves: From the bignum code which encodes the finite field arithmetic the curve relies on, we need to build the concrete data structures representing curve points alongside with the curve addition operation. The specification defines only one addition operation on the curve, but practically the equations are different whether the operation occurs on the same point or on two different points. Hence, in the following we distinguish between the operation on the same point which we call doubling, and addition on distinct points.

These operations are central to the library and since they will be intensively used, they need to be computed in a minimal number of operations. Therefore curves from different families will use different settings. Yet both cases must link to the same theory on Weierstrass curves, with the generic addition and doubling equations.

The elliptic curve addition consists in a series of operations on the coordinates of the input points as presented in II. Among those, the inversion operation in the finite field is by far the most expensive. Unfortunately the generic adding and doubling formulas involve several divisions. To tackle this issue and improve performances, standard elliptic curve implementations use a different coordinate system than the affine system in which the curve theory is built.

Two coordinate systems typically provide a better structure for these computations: the Jacobian coordinate system and the Projective coordinate system. Both have in common that they include an additional Z coordinate, thus going for the 3-coordinates system instead of a 2-coordinates system. Each affine point maps to a class of Jacobian or Projective points which allows the adding and doubling formulas to be computed purely using additions, multiplications and subtractions. The absence of division by far compensates for the cost of having an extra coordinate.

Unfortunately, while the Jacobian coordinate system is the best suited for computations on standard Weierstrass curves such as NIST-P256, Montgomery curves such as Curve25519 or Curve448 are designed to be more efficient in the Projective system. Indeed, in Projective coordinates, there is no need for these curves to carry the y_{proj} coordinate because the other two are independent from it. Furthermore, only the x_{affine} coordinate needs to be eventually used as the shared secret and it can be recomputed from only x_{proj} and z_{proj}. Incidentally, it removes the need to check that the point is on the curve.

We provide code for both coordinate systems. In practice the differences appear only in two places: to serialize the result, computing the affine coordinates back from either the Projective or Jacobian point the equations are different. And the adding and doubling formulas themselves which are different in both coordinate systems.

2) Efficient Adding and Doubling: We implemented equations available at [25]–[27], and used in popular existing implementation for the curves we consider. These equations also come with scripts for the SAGE mathematics software which show that in the specific setting of the curve they are used for, the optimized equations are indeed computing the addition and the double of the inputs as specified in the Curve module.

To quickly illustrate how we proceed, let us carry on the example of Curve448, a Montgomery curve for which efficient computation uses Projective coordinates. Let $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$ two points in Projective coordinates and let g be the function that transforms a Projective point to an affine one, $g(X, Y, Z) = (X/Z, Y/Z)$. Then, let $g(P_1) = (x_1, y_1) = (X_1/Z_1, Y_1/Z_1)$ and $g(P_2) = (x_2, y_2) = (X_2/Z_2, Y_2/Z_2)$ be the corresponding points (of P_1 and P_2) in affine representation.

We link our code with the generic addition and doubling formulas using the *SAGE* mathematics software to prove that, provided the curve characteristics, $g(P_1+P_2) = g(P_1)+g(P_2)$. There is not formal proof that SAGE is correct, nor that the SAGE scripts properly map to our code. However the equations are simple enough to get reasonable confidence there is not implementation error between the equations verified in SAGE and those implemented in the F* specification. Given that the script only proves arithmetic equality over simple equations in a finite field, we trust the SAGE software to be correct for that verification.

C. The Montgomery ladder

1) The Specification: To implement and prove the scalar multiplication, we use the *Montgomery ladder* algorithm. Although it may not be the most efficient algorithm, it is meant to be constant time which why it is used in the popular elliptic curves implementations. Here is some pseudo code for the algorithm:

```
let montgomery_ladder n q =
  let p = ref p∞ in
  let pq = ref q in
  for i = 1 to size(n) do
    if n[size(n)−i] = 0 then (
      pq := !p + !pq;
      p := 2 ∗ !p
    ) else (
      p := !p + !pq;
      pq := 2 ∗ !pq
    );
  return p
```

We consider that the scalar n is an array of bits, while q is the encoding of a point. The algorithm iterates the same step for each bit of the input scalar. Whatever the value of that bit is, it always computes an addition operation and a doubling operation. Relying on the fact that those operations are constant-time whatever the values of those points are, both cases are expected to be indistinguishable. Our implementation uses masking to enforce a single execution path through the whole ladder computation. To remove conditional branching, the input values are conditionally swapped using masking:

```
val pointOf: heap → p:point → Tot Curve.celem

val swap_conditional: a:point → b:point{Distinct a b} →
    s:limb{v s = 2^SIZE−1 ∨ v s = 0 } → ST unit
    (requires (fun h → (OnCurve h a ∧ OnCurve h b) ))
    (ensures (fun h0 _ h1 → OnCurve h0 a ∧ OnCurve h0 b
      ∧ OnCurve h1 a ∧ OnCurve h1 b
      ∧ (v s = 0 ⟹ (pointOf h1 a == pointOf h0 a
                      ∧ pointOf h1 b == pointOf h0 b))
      ∧ (v s = 2^SIZE − 1 ⟹ (pointOf h1 a == pointOf h0 b
                      ∧ pointOf h1 b == pointOf h0 a))))
```

Now, once we have a properly specified and implemented addAndDouble function as needed in the algorithm,

```
val addAndDouble: two_p:point → two_p_plus_q:point →
  p:point → p_plus_q:point → q:point → ST unit
  (requires (fun h → Live h two_p ∧ Live h two_p_plus_q
    ∧ OnCurve h p ∧ OnCurve h p_plus_q ∧ OnCurve h q))
  (ensures (fun h0 _ h1 →
    Live h0 two_p ∧ Live h0 two_p_plus_q
    ∧ OnCurve h0 p ∧ OnCurve h0 p_plus_q ∧ OnCurve h0 q
    ∧ OnCurve h1 two_p ∧ OnCurve h1 two_p_plus_q
    ∧ Live h1 p ∧ Live h1 p_plus_q ∧ OnCurve h1 q
    ∧ (modifies !{two_p,two_p_plus_q,p,p_plus_q} h0 h1)
    ∧ (pointOf h1 two_p ==
            Curve.cadd (pointOf h0 p) (pointOf h0 p))
    ∧ (pointOf h1 two_p_plus_q ==
            Curve.cadd (pointOf h0 p) (pointOf h0 p_plus_q))))
```

implementing a step of the Montgomery ladder is straightforward. The proof only relies on the fact that the *addition* is an additive law on an abelian group. Details about the internal algorithms, data or group structures do not matter and left hidden. Indeed the definition of the addAndDouble function above states that the result maps to the sum and the double of the inputs values where the operation is defined with regard to the elliptic curve group. These operations are assumed to satisfy the additive group law properties thanks to the Coq development and the scalar multiplication using the ladder relies only on those.

The Montgomery ladder proofs uses the following predicate through the iterations of its step:

```
type NtimesQ (n:ℕ) (q:celem) (h:heap) (p:point) (p':point) =
  OnCurve h p ∧ OnCurve h p' ∧ pointOf h p == n +∗ q
  ∧ pointOf h p' == (n+1) +∗ q
```

Let us assume that we are at the i-th iteration of the ladder step, in the multiplication of the point q by the scalar s. Then the predicate NtimesQ n q hi p p' holds where

$$n = s/(size(s) - i + 1)$$

and hi is the memory state of the program before the i-th step is computed.

We show that NtimesQ (2∗n+bi) q hi' p p' holds after the i-th step is run. Here bi is the value (0 or 1) of the i-th most signed bit of the scalar s, and hi' is the memory state of the program after the i-th step has been computed. Given that

$$2n + bi = s/(size(s) - i),$$

we get by induction that after the $size(s)$-th iteration the predicate NtimesQ s q hs p p' holds. In other words, in the post-state hs, p is the proper encoding of the scalar multiplication of q by s.

Hence the Montgomery ladder specification in F*:

```
val montgomery_ladder: res:point →
  n:serialized{Distinct2 n res} →
  q:point{Distinct2 n q ∧ Distinct res q} → ST unit
    (requires (fun h →
      Live h res ∧ Serialized h n ∧ OnCurve h q))
    (ensures (fun h0 _ h1 → Live h0 res
      ∧ Serialized h0 n ∧ OnCurve h0 q ∧ OnCurve h1 res
      ∧ modifies (refs res) h0 h1
      ∧ pointOf h1 res = valueOfB h0 n +* (pointOf h0 q)))
```

The specification here is already close to the top-level scalar multiplication. The serialized type is that of bignum which data is encoded on a sbyte array. The Distinct and Distinct2 predicates are merely memory separation conditions required on the different manipulated object to prove that editing one will not affect the others.

We prove that, provided that the input encodes a point on the curve, the result of the scalar multiplication using the Montgomery ladder is the proper encoding of the expected elliptic curve point. The missing part between this point and the top-level API is only the serializing and deserializing step.

D. Finalizing the Implementation

Eventually we implement the specific formatting required by curve specifications. For instance the RFC 7748 *Elliptic curves for security* [12] specifies that for Curve448, one has to format the secret scalar as follows before computing (in pseudo-code):

```
def decodeScalar448(k):
        k_list = [ord(b) for b in k]
        k_list[0] &= 252
        k_list[55] |= 128
        return decodeLittleEndian(k_list, 448)
```

Such functions are simple and have to be written on a curve-by-curve basis so we leave placeholders for programmer to implement these in order to have a clean, RFC compliant elliptic curve implementation.

E. The ECDH API

Finally, from the scalar multiplication we can extract a top-level ECDH API providing the key functions for a Diffie-Hellman exchange as presented in the first section: generation of fresh keys, validation of a given point and computation of the shared secret from the elliptic curve scalar multiplication:

```
type sbytes = seq UInt.byte
val to_sbytes: bytes → Tot sbytes
val ghost_bytes: sbytes → GTot bytes

val pointOf: b:bytes{length b = 2*Parameters.length} →
              GTot Curve.affine_point
val valueOf: sbytes → GTot ℕ

type OnCurve (b:bytes) = length b = 2*Parameters.length
  ∧ Curve.CurvePoint (pointOf b)

let (pg:bytes{OnCurve pg}) = Parameters.generator
let (+*) = Curve.smul

val is_on_curve:p:bytes → Tot (b:bool(b  ⟹  OnCurve p))

val keygen: unit → ST (sbytes * bytes)
      (requires (fun h → True))
      (ensures (fun h0 (sk,pk) h1 → (modifies !{} h0 h1)
        ∧ (pointOf pk = valueOf sk +* pointOf pg)))

val shared_secret:sk:bytes →
  p:sbytes{OnCurve(ghost_bytes p)} →
  St (q:sbytes{pointOf (ghost_bytes q) =
                          valueOf sk +* pointOf (ghost_bytes p)})
```

TABLE I. CODE SIZE, VERIFICATION EFFORT, AND PERFORMANCE

Function	Specs	Code	Annotations	Verification	Computation
Math	150	0	10	-	
Zsum	22	11	311	8s	$0.3\mu s$
Zmul	55	11	1144	57m	$5\mu s$
Modulo (25519)	65	60	551	15m11s	$9\mu s$
Ladder	13	40	527	2m50s	20ms

This API displays the core requirements and properties the user should be concerned with.

In particular the fact that this shared_secret function operates only on secret values which must have been validated (via the attached OnCurve predicate) prevents all chances of small subgroup attacks. It also provides guaranties about the secrecy of those values inside as well as outside of our implementation since the returned value is also typed secret, and functional correctness using the mathematical definition of curve.

VI. Results and Discussion

We have developed a library of verified elliptic curves in F* that implements three popular curves. The full library currently consists of about 5800 lines of code and it continues to evolve as we add new curves and refactor existing code for efficiency and to simplify and speed up our proofs. We plan to make the library available as an open-source third-party contribution distributed with the F* language.

A. Verification

Table I quantifies the size, verification effort, and runtime performance for key components in our library. The Field and Curve modules which contain all the mathematical specifications are grouped together under the Math line. The Zsum and Zmul functions correspond to standard (non-modular) addition and multiplication in $\mathbb{Z}$, that is without calls to the curve-specific reduction functions. We then display results for the Modulo function in Curve25519 and the Montgomery Ladder implementation that is common between different curves.

In F*, code and specification are quite intertwined and separating them is tricky. We distinguish between F* specifications, concrete code and proof-related annotations using the following discipline: we call specification only the types, function declarations and function bodies that are to be exposed to other modules in an interface. The number of concrete *code* lines corresponds to the number of lines in ML let declarations without any kind of annotation. The rest of the code, consisting of lemmas, subgoals and auxiliary functions that are helpful for the proof, are treated as the annotation burden.

In order to extend the library with each new curve, only the modulo function, the curve parameter values, and key formatting functions have to be written and verified. Compared to the 5800 lines of code in the full library, curve specific code and annotation amount to only about 600 lines per curve. Hence, the additional effort of adding new curves is quite modest.

B. Performance

We have made no effort to optimize our code for performance, since our focus has been on verification so far. Still, Table I shows the execution times for our code when it is run using the F* to OCaml backend. In comparison to equivalent C code, our implementation is two orders of magnitude slower. For instance, the C code for Curve25519 takes 0.18ms for scalar multiplication, whereas our code takes 20ms. This large gap may be somewhat surprising since our F* code implements all the algorithmic optimizations present in the C implementation. However, we pay a large penalty due to the use of boxed machine integers and boxed arrays in the OCaml code extracted from F*. We expect these numbers to improve as the OCaml extraction back-end of F* is optimized to better handle loops and tail recursion.

C. Limitations and Future Work

Typechecking in F* is automated by relying on an external SMT solver, but verifying complicated modules can take a long time, and sometimes the SMT solver is highly sensitive to small changes in the code. Consequently, to improve the verification time and to make it more predictable, we used a significant number of annotations throughout our code, which makes our code more verbose and less readable. While some of these annotations are necessary to guide the solver through complex mathematical proofs, proof automation could certainly be improved on certain aspects of the code, such as overflow and array bounds checks.

Our verification results rely on a large trusted computing base (TCB) including the F* typechecker, the OCaml runtime, and various platform libraries. Moreover, we do not have a formally verified link between our Coq and F* definitions, nor between SAGE and F*. Furthermore, while we enforce side-channel mitigations in the source code, they do not apply to compiled executables. In future work, we may be able to reduce this TCB by verifying and certifying different elements of this verification architecture.

A major barrier against the more widespread adoption of our library is its poor runtime performance compared to optimized C and assembly code. We are working on a new backend for F* that would translate cryptographic code directly to C. We observe that our concrete curve code is mostly written in a restricted subset of F* that is quite similar to C, even though its specification uses advanced features. Indeed, after an erasure phase which removes the ghost code and specifications, the resulting code is first-order and tail-recursive. We are building a verified compiler from this subset of F* to C. Early experiments indicate that the generated C code offers an order-of-magnitude improvement over OCaml, which would significantly narrow the performance gap between our code and other popular elliptic curve implementations.

VII. Related work

Attacks on asymmetric cryptography are more common in the literature than proofs of functional correctness. As we discussed in Section I-B, the closest prior verification work on elliptic curves is [18], which targets low-level hand-optimized *qhasm* [19] code for Curve25519 [28]. Our two approaches make different trade-offs, in that our library is extensible with minimal additional verification effort, whereas they focus on verifying a highly-performant implementation of one curve. In a separate line of work, the *Ironclad* [29] crypto library also provides security guarantees for SHA, HMAC and RSA at the assembly level, but they have not verified elliptic curves so far, and they do not enforce side-channel mitigations.

Other verification efforts have targeted symmetric cryptography. [5] used the Coq proof assistant to prove that a legacy *SHA-256* implementation written in C was correct with regard to its specification and [30] showed that the OpenSSL implementation of *HMAC* using *SHA-256* implements its specifications correctly and provides the expected cryptographic guarantees. [31] provides a tool to verify that a cryptographic implementation matches a high-level specification, and this tool has been used to verify block ciphers and hash functions.

Several works design and implement formal side-channel analyses for cryptographic code. [6] show how to use information flow analysis on target assembly code to prove the absence of certain side-channels. [22] shows how high-level side-channel mitigations can be provably compiled down to machine code through a verified compiler. Most recently, [7] shows how to prove cryptographic security, functional correctness, and side-channel protection for a complex cryptographic construction all the way from high-level cryptographic definitions down to assembly code, using a combination of several different verification tools.

Finally, a number of works address the problem of verifying the security of complex cryptographic constructions, protocols, and their implementations [1]–[3]. These works are complementary to our approach in that we verify the cryptographic primitives that are used (and assumed to be correct) within these projects. Indeed, our library already implements the two most popular curves, Curve25519 and NIST-P256, that are used in mainstream cryptographic protocols such as TLS 1.3. In ongoing work, we are integrating our elliptic curve library within the new version of miTLS, written in F*.

Acknowledgements

This research received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 683032 - CIRCUS)

References

[1] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Formal certification of code-based cryptographic proofs," pp. 90–101, 2009.

[2] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *Advances in Cryptology (CRYPTO)*, 2011, pp. 71–90.

[3] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella-Béguelin, "Proving the tls handshake secure (as it is)," in *Advances in Cryptology (CRYPTO)*, 2014, pp. 235–255.

[4] N. J. Al Fardan and K. G. Paterson, "Lucky thirteen: Breaking the tls and dtls record protocols," in *IEEE Symposium on Security and Privacy (Oakland)*, 2013, pp. 526–540.

[5] A. W. Appel, "Verification of a cryptographic primitive: Sha-256," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 37, no. 2, p. 7, 2015.

[6] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1267–1279.

[7] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC," *IACR Cryptology ePrint Archive*, vol. 2015, p. 1241, 2015. [Online]. Available: http://eprint.iacr.org/2015/1241

[8] B. B. Brumley, M. Barbosa, D. Page, and F. Vercauteren, "Practical realisation and elimination of an ecc-related software bug attack," in *Topics in Cryptology–CT-RSA 2012*. Springer, 2012, pp. 171–186.

[9] OpenSSL, "Bignum squaring may produce incorrect results (cve-2014-3570)," Jan. 2015, https://www.openssl.org/news/secadv/20150108.txt.

[10] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect forward secrecy: How diffie-hellman fails in practice," in *ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 5–17.

[11] S. Checkoway, R. Niederhagen, A. Everspaugh, M. Green, T. Lange, T. Ristenpart, D. J. Bernstein, J. Maskiewicz, H. Shacham, and M. Fredrikson, "On the practical exploitability of dual ec in tls implementations," in *USENIX Security Symposium*, 2014, pp. 319–335.

[12] A. Langley and M. Hamburg, "Elliptic curves for security," IETF RFC 7748, 2016.

[13] E. Kasper, "We ♡ SSL," Real World Cryptography, 2015.

[14] A. Langlet and W.-T. Chang, "QUIC Crypto," https://www.chromium.org/quic, 2015.

[15] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," IETF Internet Draft, 2016.

[16] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub, "Implementing tls with verified cryptographic security," in *IEEE Symposium on Security & Privacy (Oakland)*, 2013, pp. 445–462.

[17] T. Jager, J. Schwenk, and J. Somorovsky, "Practical invalid curve attacks on tls-ecdh," in *Computer Security–ESORICS 2015*. Springer, 2015, pp. 407–425.

[18] Y.-F. Chen, C.-H. Hsu, H.-H. Lin, P. Schwabe, M.-H. Tsai, B.-Y. Wang, B.-Y. Yang, and S.-Y. Yang, "Verifying curve25519 software," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 299–309.

[19] D. J. Bernstein, "qhasm software package (2007)," *URL: http://cr. yp. to/qhasm. html. Citations in this document*, vol. 4.

[20] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in F*," in *43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, Jan. 2016, pp. 256–270. [Online]. Available: https://www.fstar-lang.org/papers/mumon/

[21] E.-I. Bartzia and P.-Y. Strub, "A formal library for elliptic curves in the coq proof assistant," in *Interactive Theorem Proving*. Springer, 2014, pp. 77–92.

[22] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations," in *ACM Conference on Computer and Communications Security (CCS)*, 2013, pp. 1217–1230.

[23] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.

[24] W. J. Bowman and A. Ahmed, "Noninterference for free," in *ACM International Conference on Functional Programming (ICFP)*, 2015, pp. 101–113.

[25] hyperelliptic.org, "SAGE script for montgomery curves addition and doubling," http://www.hyperelliptic.org/EFD/g1p/auto-montgom-xz.html#ladder-ladd-1987-m-2.

[26] ——, "SAGE script for weierstrass curves addition," http://hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-3.html#addition-add-2007-bl.

[27] ——, "SAGE script for weierstrass curves doubling," http://hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-3.html#doubling-dbl-2001-b.

[28] D. J. Bernstein, "Curve25519: new diffie-hellman speed records," in *Public Key Cryptography-PKC 2006*. Springer, 2006, pp. 207–228.

[29] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad apps: End-to-end security via automated full-system verification," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 165–181.

[30] L. Beringer, A. Petcher, Q. Y. Katherine, and A. W. Appel, "Verified correctness and security of openssl hmac," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 207–221.

[31] G. Inc, "Cryptol."

APPENDIX

We list a few definitions in the elliptic curve theory formalized and mechanically verified in [21]. The syntax uses *SsReflect* notation.[4] Our primary goal for including this listing is to note its similarity to the F* specification in Figure 4.

```
Record ecuFieldMixins (K:fieldType): Type :=
  Mixin { _: 2 != 0; _: 3 != 0 }.
Record ecuType :=
  {A:K; B:K; _:4 ∗ A^3 + 27 ∗ B^2 != 0}.

Inductive point := EC_Inf | EC_In of K & K.
Notation "(x,␣y)" := (EC_In x y).

Definition oncurve (p : point) :=
  if p is (x, y) then y^2 == x^3 + A ∗ x + B else true.
Inductive ec : Type := EC p of oncurve p.

Definition neg (p : point) :=
  if p is (x, y) then (x, −y) else EC_Inf.

Definition add (p1 p2 : point) :=
  let p1 := if oncurve p1 then p1 else EC_Inf in
  let p2 := if oncurve p2 then p2 else EC_Inf in
    match p1, p2 with
    | EC_Inf, _ => p2
    | _, EC_Inf => p1
    | (x1, y1), (x2, y2) =>
        if x1 == x2 then ... else
        let s := (y2 − y1) / (x2 − x1) in
        let xs := s^2 − x1 − x2 in
        (xs, − s ∗ (xs − x1) − y1) end.

Lemma addO (p q : point): oncurve (add p q).
Definition addec (p1 p2 : ec) : ec := EC p1 p2 (addO p1 p2).

scalar_multiplication (n:nat) (p:point K) = p ∗+ n.
```

[4]See http://ssr.msr-inria.inria.fr/