A Coq proof of the correctness of X25519 in TweetNaCl

Abstract

We formally prove that the C implementation of the X25519 key-exchange protocol in the TweetNaCl library is correct. We prove both that it correctly implements the protocol from Bernstein's 2006 paper, as standardized in RFC 7748, as well as the absence of undefined behavior like arithmetic overflows and array out of bounds errors. We also formally prove, based on the work of Bartzia and Strub, that X25519 is mathematically correct, i.e., that it correctly computes scalar multiplication on the elliptic curve Curve25519.

The proofs are all computer-verified using the Coq theorem prover. To establish the link between C and Coq we use the Verified Software Toolchain (VST).

1 Introduction

The Networking and Cryptography library (NaCl) [9] is an easy-to-use, high-security, high-speed cryptography library. It uses specialized code for different platforms, which makes it rather complex and hard to audit. TweetNaCl [10] is a compact re-implementation in C of the core functionalities of NaCl and is claimed to be *"the first cryptographic library that allows correct functionality to be verified by auditors with reasonable effort"* [10]. The original paper presenting TweetNaCl describes some effort to support this claim, for example, formal verification of memory safety, but does not actually prove correctness of any of the primitives implemented by the library.

One core component of TweetNaCl (and NaCl) is the key-exchange protocol X25519 presented by Bernstein in [7]. This protocol is standardized in RFC 7748 and used by a wide variety of applications [22] such as SSH, Signal Protocol, Tor, Zcash, and TLS to establish a shared secret over an insecure channel. The X25519 key-exchange protocol is an x-coordinate-only elliptic-curve Diffie–Hellman key exchange using the Montgomery curve $E : y^2 = x^3 + 486662x^2 + x$ over the field $\mathbb{F}_{2^{255}-19}$. Note that originally, the name "Curve25519" referred to the key-exchange protocol, but Bernstein suggested to rename the protocol to X25519 and to use the name Curve25519 for the underlying elliptic curve [8]. We use this updated terminology in this paper.

Contribution of this paper. In short, in this paper we provide a computer-verified proof that the X25519 implementation in TweetNaCl matches the mathematical definition of the function given in [7, Sec. 2]. This proof is done in three steps:

We first formalize RFC 7748 [24] in Coq [23].

In the second step we prove equivalence of the C implementation of X25519 to our RFC formalization. This part of the proof uses the Verifiable Software Toolchain (VST) [1] to establish a link between C and Coq. VST uses separation logic [20, 30] to show that the semantics of the program satisfy a functional specification in Coq. To the best of our knowledge, this is the first time that VST is used in the formal proof of correctness of an implementation of an asymmetric cryptographic primitive.

In the last step we prove that the Coq formalization of the RFC matches the mathematical definition of X25519 as given in [7, Sec. 2]. We do this by extending the Coq library for elliptic curves [4] by Bartzia and Strub to support Montgomery curves, and in particular Curve25519.

To our knowledge, this verification effort is the first to not just connect a low-level implementation to a higher-level implementation (or "specification"), but to prove correctness all the way up to the mathematical definition in terms of scalar multiplication on an elliptic curve. As a consequence, the result of this paper can readily be used in mechanized proofs of higher-level protocols that work with the mathematical definition of X25519. Also, connecting our formalization of the RFC to the mathematical definition significantly increases trust into the correctness of the formalization and reduces the effort of manual auditing of that formalization.

Related work. The field of computer-aided cryptography, i.e., using computer-verified proofs to strengthen our trust into cryptographic constructions and cryptographic software, has seen massive progress in the recent past. This progress, the state of the art, and future challenges have recently been compiled in a SoK paper by Barbosa, Barthe, Bhargavan,

Blanchet, Cremers, Liao, and Parno [3]. This SoK paper, in Section III.C, also gives an overview of verification efforts of X25519 software. What all the previous approaches have in common is that they prove correctness by verifying that some low-level implementation matches a higher-level specification. This specification is kept in terms of a sequence of finite-field operations, typically close to the pseudocode in RFC 7748.

There are two general approaches to establish this link between low-level code and higher-level specification: Synthesize low-level code from the specification or write the low-level code by hand and prove that it matches the specification.

The X25519 implementation from the Evercrypt project [28] uses a low-level language called Vale that translates directly to assembly and proves equivalence to a high-level specification in F*. In [32], Zinzindohoué, Bartzia, and Bhargavan describe a verified extensible library of elliptic curves in F* [29]. This served as ground work for the cryptographic library HACL* [33] used in the NSS suite from Mozilla. The approach they use is a combination of proving and synthesising: A fairly low-level implementation written in Low* is proven to be equivalent to a high-level specification in F*. The Low* code is then compiled to C using an unverified and thus trusted compiler called Kremlin.

Coq not only allows verification but also synthesis [13]. Erbsen, Philipoom, Gross, and Chlipala make use of it to have correct-by-construction finite-field arithmetic, which is used to synthesize certified elliptic-curve crypto software [15, 16, 27]. This software suite is now being used in BoringSSL [17].

All of these X25519 verification efforts use a clean-slate approach to obtain code and proofs. Our effort targets existing software; we are aware of only one earlier work verifying existing X25519 software: In [12], Chen, Hsu, Lin, Schwabe, Tsai, Wang, Yang, and Yang present a mechanized proof of two assembly-level implementations of the core function of X25519. Their proof takes a different approach from ours. It uses heavy annotation of the assembly-level code in order to "guide" a SAT solver; also, it does not cover the full X25519 functionality and does not make the link to the mathematical definition from [7].

Finally, in terms of languages and tooling the work closest to what we present here is the proof of correctness of OpenSSL's implementations of HMAC [6], and mbedTLS' implementations of HMAC-DRBG [31] and SHA-256 [2]. As those are all symmetric primitives without the rich mathematical structure of finite field and elliptic curves the actual proofs are quite different.

Reproducing the proofs. To maximize reusability of our results we place the code of our formal proof presented in this paper into the public domain. It is available at `https://anonfile.com/B9x43aZ6n4/coq-verif-tweetnacl.tar.gz` with instructions of how to compile and verify our proof. A description of the content of the code archive is provided in Appendix C.

Organization of this paper. Section 2 gives the necessary background on Curve25519 and X25519 implementations and a brief explanation of how formal verification works. Section 3 provides our Coq formalization of X25519 as specified in RFC 7748 [24]. Section 4 provides the specifications of X25519 in TweetNaCl and some of the proof techniques used to show the correctness with respect to RFC 7748 [24]. Section 5 describes our extension of the formal library by Bartzia and Strub and the correctness of X25519 implementation with respect to Bernstein's specifications [8]. Finally in Section 6 we discuss the trusted code base of our proofs and conclude with some lessons learned about TweetNaCl and with sketching the effort required to extend our work to other elliptic-curve software.

Figure 1 shows a graph of dependencies of the proofs. C source files are represented by .C while .V corresponds to Coq files.

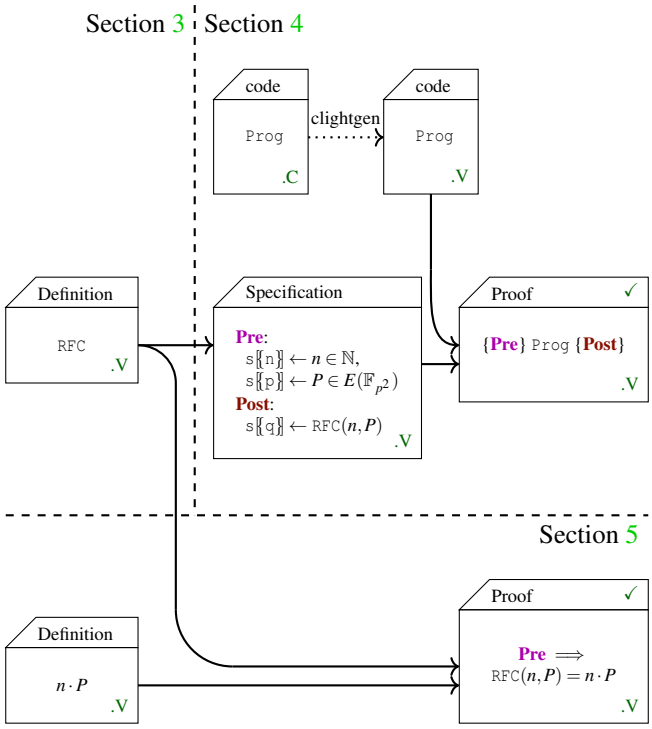

Figure 1: Structure of the proof.

2 Preliminaries

In this section, we first give a brief summary of the mathematical background on elliptic curves. We then describe X25519 and its implementation in TweetNaCl. Finally, we provide a brief description of the formal tools we use in our proofs.

2.1 Arithmetic on Montgomery curves

Definition 2.1 *Given a field $\mathbb{K}$, and $a, b \in \mathbb{K}$ such that $a^2 \neq 4$ and $b \neq 0$, $M_{a,b}$ is the Montgomery curve defined over $\mathbb{K}$ with equation*

$$M_{a,b} : by^2 = x^3 + ax^2 + x.$$

Definition 2.2 *For any algebraic extension $\mathbb{L} \supseteq \mathbb{K}$, we call $M_{a,b}(\mathbb{L})$ the set of $\mathbb{L}$-rational points, defined as*

$$M_{a,b}(\mathbb{L}) = \{O\} \cup \{(x,y) \in \mathbb{L} \times \mathbb{L} \mid by^2 = x^3 + ax^2 + x\}.$$

Here, the additional element O denotes the point at infinity.

Details of the formalization can be found in Section 5.1.2.

For $M_{a,b}$ over a finite field $\mathbb{F}_p$, the parameter b is known as the "twisting factor". For $b' \in \mathbb{F}_p \backslash \{0\}$ and $b' \neq b$, the curves $M_{a,b}$ and $M_{a,b'}$ are isomorphic via $(x,y) \mapsto (x, \sqrt{b/b'} \cdot y)$.

Definition 2.3 *When b'/b is not a square in $\mathbb{F}_p$, $M_{a,b'}$ is a* quadratic twist *of $M_{a,b}$, i.e., a curve that is isomorphic over $\mathbb{F}_{p^2}$ [14].*

Points in $M_{a,b}(\mathbb{K})$ can be equipped with a structure of an abelian group with the addition operation $+$ and with neutral element the point at infinity O. For a point $P \in M_{a,b}(\mathbb{K})$ and a positive integer n we obtain the scalar product

$$n \cdot P = \underbrace{P + \cdots + P}_{n \text{ times}}.$$

In order to efficiently compute the scalar multiplication we use an algorithm similar to square-and-multiply: the Montgomery ladder where the basic operations are differential addition and doubling [26].

We consider x-coordinate-only operations. Throughout the computation, these x-coordinates are kept in projective representation $(X : Z)$, with $x = X/Z$; the point at infinity is represented as $(1 : 0)$. See Section 5.1.3 for more details. We define the operation:

$$\texttt{xDBL\&ADD} : (x_{Q-P}, (X_P : Z_P), (X_Q : Z_Q)) \mapsto ((X_{2\cdot P} : Z_{2\cdot P}), (X_{P+Q} : Z_{P+Q}))$$

In the Montgomery ladder, the arguments P and Q of `xDBL&ADD` are swapped depending of the value of the k^{th} bit. We use a conditional swap `CSWAP` to change the arguments of the above function while keeping the same body of the loop. Given a pair (P_0, P_1) and a bit b, `CSWAP` returns the pair (P_b, P_{1-b}).

By using the differential addition and doubling operations we define the Montgomery ladder computing a x-coordinate-only scalar multiplication (see Algorithm 1).

Algorithm 1 Montgomery ladder for scalar mult.

Input: x-coordinate x_P of a point P, scalar n with $n < 2^m$
Output: x-coordinate x_Q of $Q = n \cdot P$

```
Q = (X_Q : Z_Q) ← (1 : 0)
R = (X_R : Z_R) ← (x_P : 1)
for k := m down to 1 do
    (Q,R) ← CSWAP((Q,R), k^th bit of n)
    (Q,R) ← xDBL&ADD(x_P, Q, R)
    (Q,R) ← CSWAP((Q,R), k^th bit of n)
end for
return X_Q/Z_Q
```

2.2 The X25519 key exchange

From now on let $\mathbb{F}_p$ be the field with $p = 2^{255} - 19$ elements. We consider the elliptic curve E over $\mathbb{F}_p$ defined by the equation $y^2 = x^3 + 486662x^2 + x$. For every $x \in \mathbb{F}_p$ there exists a point P in $E(\mathbb{F}_{p^2})$ such that x is the x-coordinate of P.

The core of the X25519 key-exchange protocol is a scalar-multiplication function, which we will also refer to as X25519. This function receives as input two arrays of 32 bytes each. One of them is interpreted as the little-endian encoding of a non-negative 256-bit integer n (see 3). The other is interpreted as the little-endian encoding of the x-coordinate $x_P \in \mathbb{F}_p$ of a point in $E(\mathbb{F}_{p^2})$, using the standard mapping of integers modulo p to elements in $\mathbb{F}_p$.

The X25519 function first computes a scalar n' from n by setting bits at position 0, 1, 2 and 255 to `0`; and at position 254 to `1`. This operation is often called "clamping" of the scalar n. Note that $n' \in 2^{254} + 8\{0, 1, \ldots, 2^{251} - 1\}$. X25519 then computes the x-coordinate of $n' \cdot P$.

RFC 7748 [24] standardize the X25519 Diffie–Hellman key-exchange algorithm. Given the base point B where $X_B = 9$, each party generates a secret random number s_a (respectively s_b), and computes X_{P_a} (respectively X_{P_b}), the x-coordinate of $P_A = s_a \cdot B$ (respectively $P_B = s_b \cdot B$). The parties exchange X_{P_a} and X_{P_b} and compute their shared secret $s_a \cdot s_b \cdot B$ with X25519 on s_a and X_{P_b} (respectively s_b and X_{P_a}).

2.3 TweetNaCl specifics

As its name suggests, TweetNaCl aims for code compactness (*"a crypto library in 100 tweets"*). As a result it uses a few defines and typedefs to gain precious bytes while still remaining human-readable.

```
#define FOR(i,n) for (i = 0;i < n;++i)
#define sv static void
typedef unsigned char u8;
typedef long long i64;
```

TweetNaCl functions take pointers as arguments. By convention the first one points to the output array `o`. It is then followed by the input arguments.

Due to some limitations of VST, indexes used in `for` loops have to be of type `int` instead of `i64`. We changed the code

to allow our proofs to carry through. We believe this does not affect the correctness of the original code. A complete diff of our modifications to TweetNaCl can be found in Appendix A.

2.4 X25519 in TweetNaCl

We now describe the implementation of X25519 in TweetNaCl.

Arithmetic in $\mathbb{F}_{2^{255}-19}$. In X25519, all computations are performed in $\mathbb{F}_p$. Throughout the computation, elements of that field are represented in radix 2^{16}, i.e., an element A is represented as $(a_0,\ldots,a_{15})$, with $A=\sum_{i=0}^{15} a_i 2^{16i}$. The individual "limbs" a_i are represented as 64-bit `long long` variables:

```
typedef i64 gf[16];
```

The conversion from the input byte array to this representation in radix 2^{16} is done with the `unpack25519` function.

The radix-2^{16} representation in limbs of 64 bits is highly redundant; for any element $A \in \mathbb{F}_{2^{255}-19}$ there are multiple ways to represent A as $(a_0,\ldots,a_{15})$. This is used to avoid or delay carry handling in basic operations such as Addition (`A`), subtraction (`Z`), multiplication (`M`) and squaring (`S`). After a multiplication, limbs of the result `o` are too large to be used again as input. Two calls to `car25519` at the end of `M` takes care of the carry propagation.

Inverses in $\mathbb{F}_{2^{255}-19}$ are computed with `inv25519`. This function uses exponentiation by $p-2=2^{255}-21$, computed with the square-and-multiply algorithm.

`sel25519` implements a constant-time conditional swap (`CSWAP`) by applying a mask between two fields elements.

Finally, the `pack25519` function converts the internal redundant radix-2^{16} representation to a unique byte array representing an integer in $\{0,\ldots,p-1\}$ in little-endian format. This function is considerably more complex as it needs to convert to a *unique* representation, i.e., also fully reduce modulo p and remove the redundancy of the radix-2^{16} representation.

The C definitions of those functions are available in Appendix A.

The Montgomery ladder. With these low-level arithmetic and helper functions defined, we can now turn our attention to the core of the X25519 computation: the `crypto_scalarmult` API function of TweetNaCl, which is implemented through the Montgomery ladder.

```
1  int crypto_scalarmult(u8 *q,
2                        const u8 *n,
3                        const u8 *p)
4  {
5    u8 z[32];
6    i64 r;
7    int i;
8    gf x,a,b,c,d,e,f;
9    FOR(i,31) z[i]=n[i];
10   z[31]=(n[31]&127)|64;
11   z[0]&=248;
12   unpack25519(x,p);
13   FOR(i,16) {
14     b[i]=x[i];
15     d[i]=a[i]=c[i]=0;
16   }
17   a[0]=d[0]=1;
18   for(i=254;i>=0;--i) {
19     r=(z[i>>3]>>(i&7))&1;
20     sel25519(a,b,r);
21     sel25519(c,d,r);
22     A(e,a,c);
23     Z(a,a,c);
24     A(c,b,d);
25     Z(b,b,d);
26     S(d,e);
27     S(f,a);
28     M(a,c,a);
29     M(c,b,e);
30     A(e,a,c);
31     Z(a,a,c);
32     S(b,a);
33     Z(c,d,f);
34     M(a,c,_121665);
35     A(a,a,d);
36     M(c,c,a);
37     M(a,d,f);
38     M(d,b,x);
39     S(b,e);
40     sel25519(a,b,r);
41     sel25519(c,d,r);
42   }
43   inv25519(c,c);
44   M(a,a,c);
45   pack25519(q,a);
46   return 0;
47 }
```

Note that lines 10 & 11 represent the "clamping" operation.

2.5 Coq, separation logic, and VST

Coq [23] is an interactive theorem prover based on type theory. It provides an expressive formal language to write mathematical definitions, algorithms, and theorems together with their proofs. It has been used in the proof of the four-color theorem [18] and it is also the system underlying the CompCert formally verified C compiler [25]. Unlike systems like F* [29], Coq does not rely on an SMT solver in its trusted code base. It uses its type system to verify the applications of hypotheses, lemmas, and theorems [21].

Hoare logic is a formal system which allows reasoning about programs. It uses triples such as

$$\{\textbf{Pre}\}\ \texttt{Prog}\ \{\textbf{Post}\}$$

where **Pre** and **Post** are assertions and `Prog` is a fragment of code. It is read as "when the precondition **Pre** is met, executing `Prog` will yield postcondition **Post**". We use compositional rules to prove the truth value of a Hoare triple. For example, here is the rule for sequential composition:

$$\text{Hoare-Seq}\ \frac{\{P\}C_1\{Q\} \qquad \{Q\}C_2\{R\}}{\{P\}C_1;C_2\{R\}}$$

Separation logic is an extension of Hoare logic which allows reasoning about pointers and memory manipulation. This logic enforces strict conditions on the memory shared such

as being disjoint. We discuss this limitation further in Section 4.1.

The Verified Software Toolchain (VST) [11] is a framework which uses separation logic (proven correct with respect to CompCert semantics) to prove the functional correctness of C programs. The first step consists of translating the source code into Clight, an intermediate representation used by CompCert. For such purpose one uses the parser of CompCert called `clightgen`. In a second step one defines the Hoare triple representing the specification of the piece of software one wants to prove. Then using VST, one uses a strongest postcondition approach to prove the correctness of the triple. This can be seen as a forward symbolic execution of the program.

3 Formalizing X25519 from RFC 7748

In this section we present our formalization of RFC 7748 [24].

The specification of X25519 in RFC 7748 is formalized by the function `RFC` in Coq.

More specifically, we formalized X25519 with the following definition:

```
Definition RFC (n: list Z) (p: list Z) : list Z :=
  let k := decodeScalar25519 n in
  let u := decodeUCoordinate p in
  let t := montgomery_rec_swap
    255  (* iterate 255 times  *)
    k    (* clamped n          *)
    1    (* x2                 *)
    u    (* x3                 *)
    0    (* z2                 *)
    1    (* z3                 *)
    0    (* dummy              *)
    0    (* dummy              *)
    u    (* x1                 *)
    0    (* previous bit = 0   *) in
  let a := get_a t in
  let c := get_c t in
  let o := ZPack25519 (Z.mul a (ZInv25519 c))
  in encodeUCoordinate o.
```

In this definition `montgomery_rec_swap` is a generic ladder instantiated with integers and defined as follows:

```
Fixpoint montgomery_rec_swap (m : nat) (z : T')
(a: T) (b: T) (c: T) (d: T) (e: T) (f: T) (x: T) (swap:Z) :
(* a:    x2                  *)
(* b:    x3                  *)
(* c:    z2                  *)
(* d:    z3                  *)
(* e:    temporary  var      *)
(* f:    temporary  var      *)
(* x:    x1                  *)
(* swap: previous bit value  *)
(T * T * T * T * T * T * T) :=
match m with
| S n ⇒
  let r := Getbit (Z.of_nat n) z in
    (* k_t = (k >> t) & 1                              *)
  let swap := Z.lxor swap r in
    (* swap ^= k_t                                     *)
  let (a, b) := (Sel25519 swap a b, Sel25519 swap b a) in
    (* (x2 , x3 ) = cswap(swap, x2 , x3 )              *)
  let (c, d) := (Sel25519 swap c d, Sel25519 swap d c) in
    (* (z2 , z3 ) = cswap(swap, z2 , z3 )              *)
  let e := a + c in (* A  = x2 + z2                     *)
  let a := a - c in (* B  = x2 - z2                     *)
  let c := b + d in (* C  = x3 + z3                     *)
  let b := b - d in (* D  = x3 - z3                     *)
  let d := e^2   in (* AA = A^2                        *)
  let f := a^2   in (* BB = B^2                        *)
  let a := c * a in (* CB = C * B                      *)
  let c := b * e in (* DA = D * A                      *)
  let e := a + c in (* x3 = (DA + CB)^2                *)
  let a := a - c in (* z3 = x1 * (DA - CB)^2           *)
  let b := a^2   in (* z3 = x1 * (DA - CB)^2           *)
  let c := d - f in (* E  = AA - BB                    *)
  let a := c * C_121665 in
                    (* z2  = E * (AA + a24 * E)        *)
  let a := a + d in (* z2  = E * (AA + a24 * E)        *)
  let c := c * a in (* z2  = E * (AA + a24 * E)        *)
  let a := d * f in (* x2  = AA * BB                   *)
  let d := b * x in (* z3  = x1 * (DA - CB)^2          *)
  let b := e^2   in (* x3  = (DA + CB)^2               *)
  montgomery_rec_swap n z a b c d e f x r
    (* swap = k_t                                      *)

| 0%nat ⇒
  let (a, b) := (Sel25519 swap a b, Sel25519 swap b a) in
    (* (x2 , x3 ) = cswap(swap, x2 , x3 )              *)
  let (c, d) := (Sel25519 swap c d, Sel25519 swap d c) in
    (* (z2 , z3 ) = cswap(swap, z2 , z3 )              *)
  (a,b,c,d,e,f)
end.
```

The comments in the ladder represent the text from the RFC which our formalization matches perfectly. In order to optimize the number of calls to `CSWAP` (defined in Section 2.1) the RFC uses an additional variable to decide whether a conditional swap is required or not.

Later in our proof we use a simpler description of the ladder (`montgomery_rec`) which follows strictly the Algorithm 1 and prove those ladder equivalent.

"To implement the X25519(k, u) [...] functions (where k is the scalar and u is the u-coordinate), first decode k and u and then perform the following procedure, which is taken from [curve25519] and based on formulas from [montgomery]. All calculations are performed in GF(p), i.e., they are performed modulo p." [24]

Operations used in the Montgomery ladder of `RFC` are performed on integers (See Appendix B.2). The reduction modulo $2^{255} - 19$ is deferred to the very end as part of the `ZPack25519` operation.

We now turn our attention to the decoding and encoding of the byte arrays. We define the little-endian projection to integers as follows.

Definition 3.1 *Let `ZofList` : $\mathbb{Z} \to$ `list` $\mathbb{Z} \to \mathbb{Z}$, a function given n and a list l returns its little endian decoding with radix 2^n.*

```
Fixpoint ZofList {n:Z} (a:list Z) : Z :=
  match a with
  | [] ⇒ 0
  | h :: q ⇒ h + 2^n * ZofList q
  end.
```

The encoding from integers to bytes is defined in a similar way:

Definition 3.2 *Let* `ListofZ32` : $\mathbb{Z} \to \mathbb{Z} \to$ `list` $\mathbb{Z}$, *given n and a returns a's little-endian encoding as a list with radix* 2^n.

```
Fixpoint ListofZn_fp {n:Z} (a:Z) (f:nat) : list Z :=
match f with
  | 0%nat ⇒ []
  | S fuel ⇒ (a mod 2^n) :: ListofZn_fp (a/2^n) fuel
end.

Definition ListofZ32 {n:Z} (a:Z) : list Z :=
  ListofZn_fp n a 32.
```

In order to increase the trust in our formalization, we prove that `ListofZ32` and `ZofList` are inverse to each other.

```
Lemma ListofZ32_ZofList_Zlength: forall (l:list Z),
  Forall (λ x ⇒ 0 ≤ x < 2^n) l →
  Zlength l = 32 →
  ListofZ32 n (ZofList n l) = l.
```

With those tools at hand, we formally define the decoding and encoding as specified in the RFC.

```
Definition decodeScalar25519 (l: list Z) : Z :=
  ZofList 8 (clamp l).

Definition decodeUCoordinate (l: list Z) : Z :=
  ZofList 8 (upd_nth 31 l (Z.land (nth 31 l 0) 127)).

Definition encodeUCoordinate (x: Z) : list Z :=
  ListofZ32 8 x.
```

In the definition of `decodeScalar25519`, `clamp` is taking care of setting and clearing the selected bits as stated in the RFC and described in Section 2.2.

4 Proving equivalence of X25519 in C and Coq

In this section we prove the following theorem:

The implementation of X25519 in TweetNaCl (`crypto_scalarmult`) matches the specifications of RFC 7748 [24] (`RFC`).

More formally:

```
Theorem body_crypto_scalarmult:
  (* VST boiler plate. *)
  semax_body
    (* Clight translation of TweetNaCl. *)
    Vprog
    (* Hoare triples for function calls. *)
    Gprog
    (* function we verify. *)
    f_crypto_scalarmult_curve25519_tweet
    (* Our Hoare triple, see below. *)
    crypto_scalarmult_spec.
```

Using our formalization of RFC 7748 (Section 3) we specify the Hoare triple before proving its correctness with VST (4.1). We provide an example of equivalence of operations over different number representations (4.2).

4.1 Applying the Verifiable Software Toolchain

We now turn our focus to the formal specification of `crypto_scalarmult`. We use our definition of X25519 from the RFC in the Hoare triple and prove its correctness.

Specifications. We show the soundness of TweetNaCl by proving a correspondence between the C version of TweetNaCl and the same code as a pure Coq function. This defines the equivalence between the Clight representation and our Coq definition of the ladder (`RFC`).

```
Definition crypto_scalarmult_spec :=
DECLARE _crypto_scalarmult_curve25519_tweet
WITH
  v_q: val, v_n: val, v_p: val, c121665:val,
  sh : share,
  q : list val, n : list Z, p : list Z
(*-----------------------------------------*)
PRE [ _q OF (tptr tuchar),
     _n OF (tptr tuchar),
     _p OF (tptr tuchar) ]
PROP (writable_share sh;
      Forall (λ x ↦ 0 ≤ x < 2^8) p;
      Forall (λ x ↦ 0 ≤ x < 2^8) n;
      Zlength q = 32; Zlength n = 32;
      Zlength p = 32)
LOCAL(temp _q v_q; temp _n v_n; temp _p v_p;
      gvar __121665 c121665)
SEP  (sh ⟦ v_q ⟧ ↤(uch32)⊢ q;
      sh ⟦ v_n ⟧ ↤(uch32)⊢ mVI n;
      sh ⟦ v_p ⟧ ↤(uch32)⊢ mVI p;
      Ews ⟦ c121665 ⟧ ↤(lg16)⊢ mVI64 c_121665)
(*-----------------------------------------*)
POST [ tint ]
PROP (Forall (λ x ↦ 0 ≤ x < 2^8) (RFC n p);
      Zlength (RFC n p) = 32)
LOCAL(temp ret_temp (Vint Int.zero))
SEP  (sh ⟦ v_q ⟧ ↤(uch32)⊢ mVI (RFC n p);
      sh ⟦ v_n ⟧ ↤(uch32)⊢ mVI n;
      sh ⟦ v_p ⟧ ↤(uch32)⊢ mVI p;
      Ews ⟦ c121665 ⟧ ↤(lg16)⊢ mVI64 c_121665
```

In this specification we state preconditions like:

`PRE`: `_p OF (tptr tuchar)`
The function `crypto_scalarmult` takes as input three pointers to arrays of unsigned bytes (`tptr tuchar`) `_p`, `_q` and `_n`.

`LOCAL`: `temp _p v_p`
Each pointer represent an address `v_p`, `v_q` and `v_n`.

`SEP`: `sh ⟦ v_p ⟧ ↤(uch32)⊢ mVI p`
In the memory share `sh`, the address `v_p` points to a list of integer values `mVI p`.

`PROP`: `Forall (λ x ↦ 0 ≤ x < 2^8) p`
In order to consider all the possible inputs, we assume each element of the list `p` to be bounded by 0 included and 2^8 excluded.

`PROP`: `Zlength p = 32`
We also assume that the length of the list `p` is 32. This defines the complete representation of `u8[32]`.

As postcondition we have conditions like:

POST: tint
The function crypto_scalarmult returns an integer.

LOCAL: temp ret_temp (Vint Int.zero)
The returned integer has value 0.

SEP: sh {{ v_q }}↤(uch32)⊢ mVI (RFC n p)
In the memory share sh, the address v_q points to a list of integer values mVI (RFC n p) where RFC n p is the result of the crypto_scalarmult of n and p.

PROP: Forall (λ x ↦ 0 ≤ x < 2^8) (RFC n p)
PROP: Zlength (RFC n p) = 32
We show that the computation for RFC fits in u8[32].

crypto_scalarmult computes the same result as RFC in Coq provided that inputs are within their respective bounds: arrays of 32 bytes.

The correctness of this specification is formally proven in Coq as Theorem body_crypto_scalarmult.

Memory aliasing. The semicolon in the SEP parts of the Hoare triples represents the *separating conjunction* (often written as a star), which means that the memory shares of q, n and p do not overlap. In other words, we only prove correctness of crypto_scalarmult when it is called without aliasing. But for other TweetNaCl functions, like the multiplication function M(o,a,b), we cannot ignore aliasing, as it is called in the ladder in an aliased manner.

In VST, a simple specification of this function will assume that the pointer arguments point to non-overlapping space in memory. When called with three memory fragments (o, a, b), the three of them will be consumed. However assuming this naive specification when M(o,a,a) is called (squaring), the first two memory areas (o, a) are consumed and VST will expect a third memory section (a) which does not *exist* anymore. Examples of such cases are illustrated in Figure 2. As a result, a function must either have multiple specifications or specify which aliasing case is being used. The first option would require us to do very similar proofs multiple times for a same function. We chose the second approach: for functions with 3 arguments, named hereafter o, a, b, we define an additional parameter k with values in $\{0,1,2,3\}$:

- if $k = 0$ then o and a are aliased.
- if $k = 1$ then o and b are aliased.
- if $k = 2$ then a and b are aliased.
- else there is no aliasing.

In the proof of our specification, we do a case analysis over k when needed. This solution does not cover all the possible cases of aliasing over 3 pointers (*e.g.*, o = a = b) but it is enough to satisfy our needs.

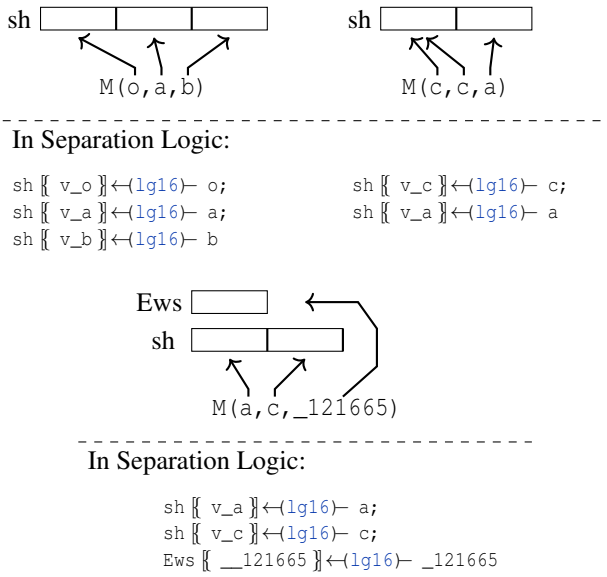

Figure 2: Aliasing and Separation Logic

Improving speed. To make the verification the smoothest, the Coq formal definition of the function should be as close as possible to the C implementation. Optimizations of such definitions are often counter-productive as they increase the amount of proofs required for *e.g.*, bounds checking, loop invariants.

In order to further speed-up the verification process, to prove the specification crypto_scalarmult, we only need the specification of the subsequently called functions (*e.g.*, M). This provide with multiple advantages: the verification by Coq can be done in parallel and multiple users can work on proving different functions at the same time.

4.2 Number representation and C implementation

As described in Section 2.3, numbers in gf are represented in 2^{16} and we use a direct mapping to represent that array as a list integers in Coq. However, in order to show the correctness of the basic operations, we need to convert this number to an integer. We reuse the mapping ZofList : $\mathbb{Z} \to$ list $\mathbb{Z} \to \mathbb{Z}$ from Section 3 and define a notation where n is 16, placing us with a radix of 2^{16}.

```
Notation "ℤ16.lst A" := (ZofList 16 A).
```

To facilitate working in $\mathbb{Z}_{2^{255}-19}$, we define the $:GF$ notation.

```
Notation "A :GF" := (A mod (2^255 - 19)).
```

Later in Section 5.2.1, we formally define $\mathbb{F}_{2^{255}-19}$ as a field. Equivalence between operations in $\mathbb{Z}_{2^{255}-19}$ (*i.e.*, under $:GF$) and in $\mathbb{F}_{2^{255}-19}$ are easily proven.

Using these two definitions, we prove intermediate lemmas such as the correctness of the multiplication Low.M where

`Low.M` replicates the computations and steps done in C.

Lemma 4.1 *`Low.M` correctly implements the multiplication over* $\mathbb{Z}_{2^{255}-19}$.

And specified in Coq as follows:

```
Lemma mult_GF_Zlength :
  forall (a:list Z) (b:list Z),
  Zlength a = 16 →
  Zlength b = 16 →
   (ℤ16.lst (Low.M a b)) :GF =
   (ℤ16.lst a * ℤ16.lst b) :GF.
```

However for our purpose, simple functional correctness is not enough. We also need to define the bounds under which the operation is correct, allowing us to chain them, guaranteeing us the absence of overflow.

Lemma 4.2 *if all the values of the input arrays are constrained between* -2^{26} *and* 2^{26}*, then the output of `Low.M` will be constrained between* -38 *and* $2^{16}+38$.

And seen in Coq as follows:

```
Lemma M_bound_Zlength :
  forall (a:list Z) (b:list Z),
  Zlength a = 16 →
  Zlength b = 16 →
  Forall (λ x ⇒ -2^26 < x < 2^26) a →
  Forall (λ x ⇒ -2^26 < x < 2^26) b →
    Forall (λ x ⇒ -38 ≤ x < 2^16 + 38) (Low.M a b).
```

Using reflection (chapter 15 in [13]), we prove the functional correctness of the multiplicative inverse over $\mathbb{Z}_{2^{255}-19}$.

Lemma 4.3 *`Inv25519` computes an inverse in* $\mathbb{Z}_{2^{255}-19}$.

This statement is formalized as

```
Corollary Inv25519_Zpow_GF : forall (g:list Z),
  length g = 16 →
  ℤ16.lst (Inv25519 g) :GF  =
    (pow (ℤ16.lst g) (2^255 − 21)) :GF.
```

By using each function `Low.M`; `Low.A`; `Low.Sq`; `Low.Zub`; `Unpack25519`; `clamp`; `Pack25519`; `Inv25519`; `car25519`; `montgomery_rec`, we defined in Coq `Crypto_Scalarmult` and with VST proved it matches the exact behavior of X25519 in TweetNaCl.

By proving that each function `Low.M`; `Low.A`; `Low.Sq`; `Low.Zub`; `Unpack25519`; `clamp`; `Pack25519`; `Inv25519`; `car25519` behave over `list Z` as their equivalent over `Z` with `:GF` (in $\mathbb{Z}_{2^{255}-19}$), we prove that given the same inputs `Crypto_Scalarmult` performs the same computation as `RFC`.

```
Lemma Crypto_Scalarmult_RFC_eq :
  forall (n: list Z) (p: list Z),
  Zlength n = 32 →
  Zlength p = 32 →
  Forall (λ x ⇒ 0 ≤ x ∧ x < 2 ^ 8) n →
  Forall (λ x ⇒ 0 ≤ x ∧ x < 2 ^ 8) p →
  Crypto_Scalarmult n p = RFC n p.
```

Using this equality, we can directly replace `Crypto_Scalarmult` in our specification by `RFC`, proving that TweetNaCl's X25519 implementation respect RFC 7748.

5 Proving that X25519 in Coq matches the mathematical model

In this section we prove the following informal theorem:

The implementation of X25519 in TweetNaCl computes the $\mathbb{F}_p$*-restricted x-coordinate scalar multiplication on* $E(\mathbb{F}_{p^2})$ *where* p *is* $2^{255}-19$ *and* E *is the elliptic curve* $y^2 = x^3 + 486662x^2 + x$.

More precisely, we prove that our formalization of the RFC matches the definitions of Curve25519 by Bernstein:

```
Theorem RFC_Correct: forall (n p : list Z)
  (P:mc curve25519_Fp2 _mcuType),
  Zlength n = 32 →
  Zlength p = 32 →
  Forall (λ x ⇒ 0 ≤ x ∧ x < 2 ^ 8) n →
  Forall (λ x ⇒ 0 ≤ x ∧ x < 2 ^ 8) p →
  Fp2 _x (decodeUCoordinate p) = P#x0 →
  RFC n p =
    encodeUCoordinate
      ((P *+ (Z.to_nat (decodeScalar25519 n))) _x0).
```

We first review the work of Bartzia and Strub [4] (5.1.1). We extend it to support Montgomery curves (5.1.2) with homogeneous coordinates (5.1.3) and prove the correctness of the ladder (5.1.4). We discuss the twist of Curve25519 (5.2.1) and explain how we deal with it in the proofs (5.2.2).

5.1 Formalization of elliptic Curves

Figure 3 presents the structure of the proof of the ladder's correctness.

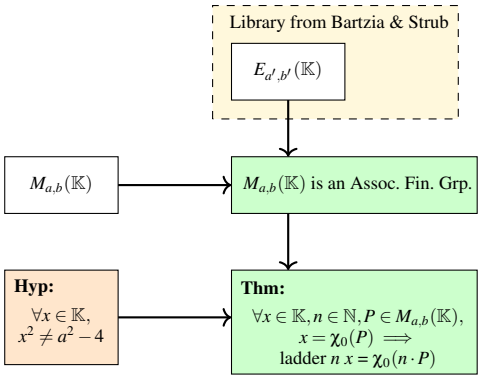

Figure 3: Overview of the proof of Montgomery ladder's correctness

We consider elliptic curves over a field $\mathbb{K}$. We assume that the characteristic of $\mathbb{K}$ is neither 2 or 3.

Definition 5.1 *Given a field* $\mathbb{K}$*, using an appropriate choice of coordinates, an elliptic curve* E *is a plane cubic algebraic curve defined by an equation* $E(x,y)$ *of the form:*

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where the a_i's are in $\mathbb{K}$ and the curve has no singular point (i.e., no cusps or self-intersections). The set of points defined over $\mathbb{K}$, written $E(\mathbb{K})$, is formed by the solutions (x,y) of E together with a distinguished point O called point at infinity:

$$E(\mathbb{K}) = \{(x,y) \in \mathbb{K} \times \mathbb{K} \mid E(x,y)\} \cup \{O\}$$

5.1.1 Short Weierstraß curves

This equation $E(x,y)$ can be reduced into its short Weierstraß form.

Definition 5.2 *Let $a \in \mathbb{K}$ and $b \in \mathbb{K}$ such that*

$$\Delta(a,b) = -16(4a^3 + 27b^2) \neq 0.$$

The elliptic curve $E_{a,b}$ is defined by the equation:

$$y^2 = x^3 + ax + b.$$

$E_{a,b}(\mathbb{K})$ is the set of all points $(x,y) \in \mathbb{K}^2$ satisfying the $E_{a,b}$ along with an additional formal point O, "at infinity". Such a curve does not have any singularity.

In this setting, Bartzia and Strub defined the parametric type `ec` which represent the points on a specific curve. It is parameterized by a `K : ecuFieldType` — the type of fields which characteristic is not 2 or 3 — and `E : ecuType` — a record that packs the curve parameters a and b along with the proof that $\Delta(a,b) \neq 0$.

```
Inductive point := EC_Inf | EC_In of K * K.
Notation "(| x, y |)" := (EC_In x y).
Notation "∞" := (EC_Inf).

Record ecuType :=
  { A : K; B : K; _ : 4 * A^3 + 27 * B^2 ≠ 0}.
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.
```

Points on an elliptic curve are equipped with the structure of an abelian group.

- The negation of a point $P = (x,y)$ is defined by reflection over the x axis $-P = (x,-y)$.
- The addition of two points P and Q is defined as the negation of the third intersection point of the line passing through P and Q, or tangent to P if $P = Q$.
- O is the neutral element under this law: if 3 points are collinear, their sum is equal to O.

These operations are defined in Coq as follows (where we omit the code for the tangent case):

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

Definition add (p1 p2 : point) :=
  match p1 , p2 with
    | ∞ , _ ⇒ p2
    | _ , ∞ ⇒ 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.
```

The value of `add` is proven to be on the curve (with coercion):

```
Lemma add0 (p q : ec): oncurve (add p q).

Definition addec (p1 p2 : ec) : ec :=
  EC p1 p2 (add0 p1 p2 )
```

5.1.2 Montgomery curves

Speedups can be obtained by switching to homogeneous coordinates and other forms than the Weierstraß form. We consider the Montgomery form [26].

Definition 5.3 *Let $a \in \mathbb{K}\backslash\{-2,2\}$, and $b \in \mathbb{K}\backslash\{0\}$. The elliptic curve $M_{a,b}$ is defined by the equation:*

$$by^2 = x^3 + ax^2 + x,$$

$M_{a,b}(\mathbb{K})$ is the set of all points $(x,y) \in \mathbb{K}^2$ satisfying the $M_{a,b}$ along with an additional formal point O, "at infinity".

Similar to the definition of `ec`, we defined the parametric type `mc` which represents the points on a specific Montgomery curve. It is parameterized by a `K : ecuFieldType` — the type of fields which characteristic is not 2 or 3 — and `M : mcuType` — a record that packs the curve parameters a and b along with the proofs that $b \neq 0$ and $a^2 \neq 4$.

```
Record mcuType :=
  { cA : K; cB : K; _ : cB ≠ 0; _ : cA^2 ≠ 4}.
Definition oncurve (p : point K) :=
if p is (| x, y |)
  then cB * y^2 == x^3 + cA * x^2 + x
  else true.
Inductive mc : Type := MC p of oncurve p.

Lemma oncurve_mc: forall p : mc, oncurve p.
```

We define the addition on Montgomery curves in a similar way as for the Weierstraß form.

```
Definition add (p1 p2 : point K) :=
  match p1 , p2 with
    | ∞, _ ⇒ p2
    | _, ∞ ⇒ p1
    | (|x1 , y1 |), (|x2 , y2 |) ⇒
      if x1 == x2
      then if (y1 == y2 ) && (y1 ≠ 0)
           then ... else ∞
      else
        let s := (y2 - y1 ) / (x2 - x1 ) in
        let xs := s^2 * cB - cA - x1 - x2 in
          (| xs , - s * (xs - x1 ) - y1 |)
  end.
```

And again we prove the result is on the curve:

```
Lemma add0 (p q : mc) : oncurve (add p q).
Definition addmc (p1 p2 : mc) : mc :=
  MC p1 p2 (add0 p1 p2 )
```

We then define a bijection between a Montgomery curve and its short Weierstraß form. In this way we get associativity of addition on Montgomery curves from the corresponding property for Weierstraß curves.

Lemma 5.4 *Let $M_{a,b}$ be a Montgomery curve, define*

$$a' = \frac{3-a^2}{3b^2} \quad \text{and} \quad b' = \frac{2a^3-9a}{27b^3}.$$

then $E_{a',b'}$ is an elliptic curve, and the mapping $\varphi : M_{a,b} \mapsto E_{a',b'}$ defined as:

$$\varphi(O_M) = O_E$$
$$\varphi((x,y)) = \left(\frac{x}{b} + \frac{a}{3b}, \frac{y}{b}\right)$$

is an isomorphism between elliptic curves.

```
Definition ec_of_mc_point p :=
  match p with
  | ∞ ⇒ ∞
  | (|x, y|) ⇒ (|x/b + a/(3 * b), y/b|)
  end.
Lemma ec_of_mc_point_ok p :
  oncurve M p →
  ec.oncurve E (ec_of_mc_point p).

Definition ec_of_mc p :=
  EC (ec_of_mc_point_ok (oncurve_mc p)).

Lemma ec_of_mc_bij : bijective ec_of_mc.
```

5.1.3 Projective coordinates

In a projective plane, points are represented with triples $(X : Y : Z)$, with the exception of $(0 : 0 : 0)$. Scalar multiples are representing the same point, *i.e.*, for all $\lambda \neq 0$, the triples $(X : Y : Z)$ and $(\lambda X : \lambda Y : \lambda Z)$ represent the same point. For $Z \neq 0$, the projective point $(X : Y : Z)$ corresponds to the point $(X/Z, Y/Z)$ on the affine plane. Likewise the point (X,Y) on the affine plane corresponds to $(X : Y : 1)$ on the projective plane.

Using fractions as coordinates, the equation for a Montgomery curve $M_{a,b}$ becomes:

$$b\left(\frac{Y}{Z}\right)^2 = \left(\frac{X}{Z}\right)^3 + a\left(\frac{X}{Z}\right)^2 + \left(\frac{X}{Z}\right)$$

Multiplying both sides by Z^3 yields:

$$bY^2Z = X^3 + aX^2Z + XZ^2$$

With this equation we can additionally represent the "point at infinity". By setting $Z = 0$, we derive $X = 0$, giving us the "infinite point" $(0 : 1 : 0)$.

By restricting the parameter a of $M_{a,b}(\mathbb{K})$ such that $a^2 - 4$ is not a square in $\mathbb{K}$, we ensure that $(0,0)$ is the only point with a y-coordinate of 0.

Hypothesis 5.5 *$a^2 - 4$ is not a square in $\mathbb{K}$.*

```
Hypothesis mcu_no_square : forall x : K, x^2  ≠ a^2  - 4.
```

We define χ and χ_0 to return the x-coordinate of points on a curve.

Definition 5.6 *Let χ and χ_0:*
– $\chi : M_{a,b}(\mathbb{K}) \to \mathbb{K} \cup \{\infty\}$
such that $\chi(O) = \infty$ and $\chi((x,y)) = x$.
– $\chi_0 : M_{a,b}(\mathbb{K}) \to \mathbb{K}$
such that $\chi_0(O) = 0$ and $\chi_0((x,y)) = x$.

Using projective coordinates we prove the formula for differential addition.

Lemma 5.7 *Let $M_{a,b}$ be a Montgomery curve such that $a^2 - 4$ is not a square in $\mathbb{K}$, and let $X_1, Z_1, X_2, Z_2, X_4, Z_4 \in \mathbb{K}$, such that $(X_1, Z_1) \neq (0,0)$, $(X_2, Z_2) \neq (0,0)$, $X_4 \neq 0$ and $Z_4 \neq 0$. Define*

$$X_3 = Z_4((X_1 - Z_1)(X_2 + Z_2) + (X_1 + Z_1)(X_2 - Z_2))^2$$
$$Z_3 = X_4((X_1 - Z_1)(X_2 + Z_2) - (X_1 + Z_1)(X_2 - Z_2))^2,$$

then for any point P_1 and P_2 in $M_{a,b}(\mathbb{K})$ such that $X_1/Z_1 = \chi(P_1), X_2/Z_2 = \chi(P_2)$, and $X_4/Z_4 = \chi(P_1 - P_2)$, we have $X_3/Z_3 = \chi(P_1 + P_2)$.
Remark: *These definitions should be understood in $\mathbb{K} \cup \{\infty\}$. If $x \neq 0$ then we define $x/0 = \infty$.*

Similarly we also prove the formula for point doubling.

Lemma 5.8 *Let $M_{a,b}$ be a Montgomery curve such that $a^2 - 4$ is not a square in $\mathbb{K}$, and let $X_1, Z_1 \in \mathbb{K}$, such that $(X_1, Z_1) \neq (0,0)$. Define*

$$c = (X_1 + Z_1)^2 - (X_1 - Z_1)^2$$
$$X_3 = (X_1 + Z_1)^2 (X_1 - Z_1)^2$$
$$Z_3 = c\left((X_1 + Z_1)^2 + \frac{a-2}{4} \times c\right),$$

then for any point P_1 in $M_{a,b}(\mathbb{K})$ such that $X_1/Z_1 = \chi(P_1)$, we have $X_3/Z_3 = \chi(2P_1)$.

With Lemma 5.7 and Lemma 5.8, we are able to compute efficiently differential additions and point doublings using projective coordinates.

5.1.4 Scalar multiplication algorithms

By taking Algorithm 1 and replacing `xDBL&ADD` by a combination of the formulae from Lemma 5.7 and Lemma 5.8, we define a ladder `opt_montgomery` (in which $\mathbb{K}$ has not been fixed yet).

This gives us the theorem of the correctness of the Montgomery ladder.

Theorem 5.9 *For all $n, m \in \mathbb{N}$, $x \in \mathbb{K}$, $P \in M_{a,b}(\mathbb{K})$, if $\chi_0(P) = x$ then `opt_montgomery` returns $\chi_0(n \cdot P)$*

```
Theorem opt_montgomery_ok (n m: nat) (x : K) :
  n < 2^m →
  forall (p : mc M), p#x0 = x →
  opt_montgomery n m x = (p *+ n)#x0.
```

The definition of `opt_montgomery` is similar to `montgomery_rec_swap` that was used in `RFC`. We proved their equivalence, and used it in our final proof of `Theorem RFC_Correct`.

5.2 Curves, twists and extension fields

Figure 4 gives a high-level view of the proofs presented here.

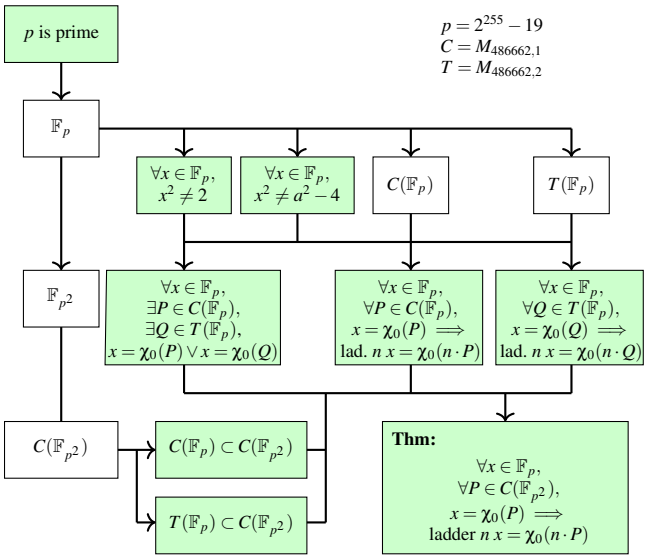

Figure 4: Proof dependencies for the correctness of X25519.

To be able to use the Theorem 5.9 we need to satisfy hypothesis 5.5:

$$\forall x \in \mathbb{K},\ x^2 \neq a^2 - 4.$$

However there always exists $x \in \mathbb{F}_{p^2}$ such that $x^2 = a^2 - 4$, preventing the use Theorem 5.9 with $\mathbb{K} = \mathbb{F}_{p^2}$.

We first study Curve25519 and one of its quadratic twists Twist25519, both defined over $\mathbb{F}_p$.

5.2.1 Curves and twists

We define $\mathbb{F}_p$ as the numbers between 0 and $p = 2^{255} - 19$. We create a `Zmodp` module to encapsulate those definitions.

```
Module Zmodp.
Definition betweenb x y z := (x ≤? z) && (z <? y).
Definition p := locked (2^255 - 19).
Fact Hp_gt0 : p > 0.
Inductive type := Zmodp x of betweenb 0 p x.

Lemma Z_mod_betweenb (x y : Z) :
  y > 0 → betweenb 0 y (x mod y).
Definition pi (x : Z) : type :=
  Zmodp (Z_mod_betweenb x Hp_gt0).
Coercion repr (x : type) : Z :=
  let: @Zmodp x _ := x in x.
```

We define the basic operations $(+, -, \times)$ with their respective neutral elements $(0, 1)$ and prove Lemma 5.10.

Lemma 5.10 *$\mathbb{F}_p$ is a field.*

For $a = 486662$, by using the Legendre symbol we prove that $a^2 - 4$ and 2 are not squares in $\mathbb{F}_p$.

```
Fact a_not_square : forall x: Zmodp.type,
  x^2  ≠ (Zmodp.pi 486662)^2  - 4.

Fact two_not_square : forall x: Zmodp.type,
  x^2  ≠ 2.
```

We now consider $M_{486662,1}(\mathbb{F}_p)$ and $M_{486662,2}(\mathbb{F}_p)$, one of its quadratic twists.

Definition 5.11 *We instantiate `opt_montgomery` in two specific ways:*
– Curve25519_F p(n,x) for $M_{486662,1}(\mathbb{F}_p)$.
– Twist25519_F p(n,x) for $M_{486662,2}(\mathbb{F}_p)$.

With Theorem 5.9 we derive the following two lemmas:

Lemma 5.12 *For all $x \in \mathbb{F}_p$, $n \in \mathbb{N}$, $P \in \mathbb{F}_p \times \mathbb{F}_p$, such that $P \in M_{486662,1}(\mathbb{F}_p)$ and $\chi_0(P) = x$.*

$$Curve25519\_F\,p(n,x) = \chi_0(n \cdot P)$$

Lemma 5.13 *For all $x \in \mathbb{F}_p$, $n \in \mathbb{N}$, $P \in \mathbb{F}_p \times \mathbb{F}_p$ such that $P \in M_{486662,2}(\mathbb{F}_p)$ and $\chi_0(P) = x$.*

$$Twist25519\_F\,p(n,x) = \chi_0(n \cdot P)$$

As the Montgomery ladder does not depend on b, it is trivial to see that the computations done for points in $M_{486662,1}(\mathbb{F}_p)$ and in $M_{486662,2}(\mathbb{F}_p)$ are the same.

```
Theorem curve_twist_eq: forall n x,
  curve25519_Fp_ladder n x = twist25519_Fp_ladder n x.
```

Because 2 is not a square in $\mathbb{F}_p$, it allows us split $\mathbb{F}_p$ into two sets.

Lemma 5.14 *For all x in $\mathbb{F}_p$, there exists y in $\mathbb{F}_p$ such that*

$$y^2 = x \quad \vee \quad 2y^2 = x$$

For all $x \in \mathbb{F}_p$, we can compute $x^3 + ax^2 + x$. Using Lemma 5.14 we can find a y such that (x, y) is either on the curve or on the quadratic twist:

Lemma 5.15 *For all $x \in \mathbb{F}_p$, there exists a point P in $M_{486662,1}(\mathbb{F}_p)$ or in $M_{486662,2}(\mathbb{F}_p)$ such that the x-coordinate of P is x.*

```
Theorem x_is_on_curve_or_twist:
  forall x : Zmodp.type,
  (exists (p : mc curve25519_mcuType), p#x0 = x) ∨
  (exists (p' : mc twist25519_mcuType), p'#x0 = x).
```

5.2.2 Curve25519 over $\mathbb{F}_{p^2}$

The quadratic extension $\mathbb{F}_{p^2}$ is defined as $\mathbb{F}_p[\sqrt{2}]$ by [7]. The theory of finite fields already has been formalized in the Mathematical Components library, but this formalization is rather abstract, and we need concrete representations of field elements here. For this reason we decided to formalize a definition of $\mathbb{F}_{p^2}$ ourselves.

We can represent $\mathbb{F}_{p^2}$ as the set $\mathbb{F}_p \times \mathbb{F}_p$, representing polynomials with coefficients in $\mathbb{F}_p$ modulo X^2-2. In a similar way as for $\mathbb{F}_p$ we use a module in Coq.

```
Module Zmodp2 .
Inductive type :=
  Zmodp2  (x: Zmodp.type) (y:Zmodp.type).

Definition pi (x: Zmodp.type * Zmodp.type) : type :=
  Zmodp2  x.1 x.2.
Coercion repr (x: type) : Zmodp.type*Zmodp.type :=
  let: Zmodp2  u v := x in (u, v).
Definition mul (x y: type) : type :=
  pi ((x.1 * y.1) + (2 * (x.2 * y.2)),
      (x.1 * y.2) + (x.2 * y.1)).
```

We define the basic operations $(+,-,\times)$ with their respective neutral elements 0 and 1. Additionally we verify that for each element of in $\mathbb{F}_{p^2}\backslash\{0\}$, there exists a multiplicative inverse.

Lemma 5.16 *For all $x \in \mathbb{F}_{p^2}\backslash\{0\}$ and $a,b \in \mathbb{F}_p$ such that $x=(a,b)$,*

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

As in $\mathbb{F}_p$, we define $0^{-1}=0$ and prove Lemma 5.17.

Lemma 5.17 *$\mathbb{F}_{p^2}$ is a commutative field.*

We then specialize the basic operations in order to speed up the verification of formulas by using rewrite rules:

$$(a,0)+(b,0)=(a+b,0) \qquad (a,0)\cdot(b,0)=(a\cdot b,0)$$
$$(a,0)^{-1}=(a^{-1},0) \qquad (0,a)^{-1}=(0,(2\cdot a)^{-1})$$

The injection $a \mapsto (a,0)$ from $\mathbb{F}_p$ to $\mathbb{F}_{p^2}$ preserves $0,1,+,-,\times$. Thus $(a,0)$ can be abbreviated as a without confusions.

We define $M_{486662,1}(\mathbb{F}_{p^2})$. With the rewrite rule above, it is straightforward to prove that any point on the curve $M_{486662,1}(\mathbb{F}_p)$ is also on the curve $M_{486662,1}(\mathbb{F}_{p^2})$. Similarly, any point on the quadratic twist $M_{486662,2}(\mathbb{F}_p)$ also corresponds to a point on the curve $M_{486662,1}(\mathbb{F}_{p^2})$. As direct consequence, using Lemma 5.15, we prove that for all $x \in \mathbb{F}_p$, there exists a point $P \in \mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$ on $M_{486662,1}(\mathbb{F}_{p^2})$ such that $\chi_0(P)=(x,0)=x$.

```
Lemma x_is_on_curve_or_twist_implies_x_in_Fp2 :
  forall (x:Zmodp.type),
    exists (p: mc curve25519_Fp2 _mcuType),
      p#x0 = Zmodp2 .Zmodp2  x 0.
```

We now study the case of the scalar multiplication and show similar proofs.

Definition 5.18 *Define the functions φ_c, φ_t and ψ*
– $\varphi_c : M_{486662,1}(\mathbb{F}_p) \mapsto M_{486662,1}(\mathbb{F}_{p^2})$
such that $\varphi((x,y)) = ((x,0),(y,0))$.
– $\varphi_t : M_{486662,2}(\mathbb{F}_p) \mapsto M_{486662,1}(\mathbb{F}_{p^2})$
such that $\varphi((x,y)) = ((x,0),(0,y))$.
– $\psi : \mathbb{F}_{p^2} \mapsto \mathbb{F}_p$
such that $\psi(x,y) = x$.

Lemma 5.19 *For all $n \in \mathbb{N}$, for all point $P \in \mathbb{F}_p \times \mathbb{F}_p$ on the curve $M_{486662,1}(\mathbb{F}_p)$ (respectively on the quadratic twist $M_{486662,2}(\mathbb{F}_p)$), we have:*

$$P \in M_{486662,1}(\mathbb{F}_p) \implies \varphi_c(n\cdot P) = n\cdot\varphi_c(P)$$
$$P \in M_{486662,2}(\mathbb{F}_p) \implies \varphi_t(n\cdot P) = n\cdot\varphi_t(P)$$

Notice that:

$$\forall P \in M_{486662,1}(\mathbb{F}_p),\ \ \psi(\chi_0(\varphi_c(P))) = \chi_0(P)$$
$$\forall P \in M_{486662,2}(\mathbb{F}_p),\ \ \psi(\chi_0(\varphi_t(P))) = \chi_0(P)$$

In summary for all $n \in \mathbb{N}$, $n < 2^{255}$, for any given point $P \in \mathbb{F}_p \times \mathbb{F}_p$ in $M_{486662,1}(\mathbb{F}_p)$ or $M_{486662,2}(\mathbb{F}_p)$, *Curve*25519_*Fp* computes the $\chi_0(n\cdot P)$. We proved that for all $P \in \mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$ such that $\chi_0(P) \in \mathbb{F}_p$ there exists a corresponding point on the curve or the twist over $\mathbb{F}_p$. We proved that for any point, on the curve or the twist we can compute the scalar multiplication by n and yield to the same result as if we did the computation in $\mathbb{F}_{p^2}$.

Theorem 5.20 *For all $n \in \mathbb{N}$, such that $n < 2^{255}$, for all $x \in \mathbb{F}_p$ and $P \in M_{486662,1}(\mathbb{F}_{p^2})$ such that $\chi_0(P) = x$, Curve*25519_*Fp*(n,x) *computes $\chi_0(n\cdot P)$.*

which is formalized in Coq as:

```
Theorem curve25519_Fp2 _ladder_ok:
  forall (n : nat) (x:Zmodp.type),
  (n < 2^255)%nat →
  forall (p  : mc curve25519_Fp2 _mcuType),
  p #x0 = Zmodp2 .Zmodp2  x 0 →
  curve25519_Fp_ladder n x = (p *+ n)#x0 /p.
```

We then prove the equivalence between of operations in $\mathbb{F}_{2^{255}-19}$ and $\mathbb{Z}_{2^{255}-19}$, in other words between `Zmodp` and `:GF`. This allows us to show that given a clamped value n and normalized x-coordinate of P, `RFC` gives the same results as *Curve*25519_*Fp*.

All put together, this finishes the proof of the mathematical correctness of X25519: the fact that the code in X25519, both in the RFC 7748 and in TweetNaCl versions, correctly computes multiplication in the elliptic curve.

6 Conclusion

Any formal system relies on a trusted base. In this section we describe our chain of trust.

Trusted Code Base of the proof. Our proof relies on a trusted base, i.e. a foundation of definitions that must be correct. One should not be able to prove a false statement in that system, *e.g.*, by proving an inconsistency.

In our case we rely on:

- **Calculus of Inductive Constructions**. The intuitionistic logic used by Coq must be consistent in order to trust the proofs. As an axiom, we assume that the functional extensionality is also consistent with that logic.

$$\forall x, f(x) = g(x) \implies f = g$$

```
Lemma f_ext: forall (A B:Type),
  forall (f g: A → B),
  (forall x, f(x) = g(x)) → f = g.
```

- **Verifiable Software Toolchain**. This framework developed at Princeton allows a user to prove that a Clight code matches pure Coq specification.

- **CompCert**. When compiling with CompCert we only need to trust CompCert's assembly semantics, as the compilation chain has been formally proven correct. However, when compiling with other C compilers like Clang or GCC, we need to trust that the CompCert's Clight semantics matches the C17 standard.

- **`clightgen`**. The tool making the translation from C to Clight, the first step of the CompCert compilation. VST does not support the direct verification of `o[i] = a[i] + b[i]`. This needs to be rewritten into:

```
aux1 = a[i]; aux2 = b[i];
o[i] = aux1 + aux2;
```

  The `-normalize` flag is taking care of this rewriting and factors out assignments from inside subexpressions.

- Finally, we must trust the **Coq kernel** and its associated libraries; the **Ocaml compiler** on which we compiled Coq; the **Ocaml Runtime** and the **CPU**. Those are common to all proofs done with this architecture [2, 23].

Corrections in TweetNaCl. As a result of this verification, we removed superfluous code. Indeed indexes 17 to 79 of the `i64 x[80]` intermediate variable of `crypto_scalarmult` were adding unnecessary complexity to the code, we removed them.

Peter Wu and Jason A. Donenfeld brought to our attention that the original `car25519` function carried a risk of undefined behavior if `c` is a negative number.

```
c=o[i]>>16;
o[i]-=c<<16; // c < 0 = UB !
```

We replaced this statement with a logical `and`, proved correctness, and thus solved this problem.

```
o[i]&=0xffff;
```

Aside from this modifications, all subsequent alterations to the TweetNaCl code—such as the type change of loop indexes (`int` instead of `i64`)—were required for VST to step through the code properly. We believe that those adjustments do not impact the trust of our proof.

We contacted the authors of TweetNaCl and expect that the changes described above will soon be integrated in a new version of the library.

Extending our work. The high-level definition (Section 5) can easily be ported to any other Montgomery curves and with it the proof of the ladder's correctness assuming the same formulas are used. In addition to the curve equation, the field $\mathbb{F}_p$ would need to be redefined as $p = 2^{255} - 19$ is hard-coded in order to speed up some proofs.

With respect to the C code verification (Section 4), the extension of the verification effort to Ed25519 would make directly use of the low-level arithmetic. The ladder steps formula being different this would require a high level verification similar to Theorem 5.9.

The verification of *e.g.*, X448 [19, 24] in C would require the adaptation of most of the low-level arithmetic (mainly the multiplication, carry propagation and reduction). Once the correctness and bounds of the basic operations are established, reproving the full ladder would make use of our generic definition.

A complete proof. We provide a mechanized formal proof of the correctness of the X25519 implementation in TweetNaCl from C up the mathematical definitions with a single tool. We first formalized X25519 from RFC 7748 [24] in Coq. We then proved that TweetNaCl's implementation of X25519 matches our formalization. In a second step we extended the Coq library for elliptic curves [4] by Bartzia and Strub to support Montgomery curves. Using this extension we proved that the X25519 from the RFC matches the mathematical definitions as given in [7, Sec. 2]. Therefore in addition to proving the mathematical correctness of TweetNaCl, we also increases the trust of other works such as [16, 33] which rely on RFC 7748.

References

[1] Andrew W. Appel. Verified software toolchain. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, volume 7226 of *LNCS*, page 2. Springer, 2012. https://doi.org/10.1007/978-3-642-28891-3_2.

[2] Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems*, 37(2):7:1–7:31, 2015. http://doi.acm.org/10.1145/2701415.

[3] 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.

[4] Evmorfia-Iro Bartzia and Pierre-Yves Strub. A formal library for elliptic curves in the Coq proof assistant. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, volume 8558 of *LNCS*, pages 77–92. Springer, 2014. https://hal.inria.fr/hal-01102288.

[5] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto1. ANSI/ISO C specification language, 2019. https://frama-c.com/download/acsl.pdf.

[6] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of OpenSSL HMAC. In *Proceedings of the 24th USENIX Security Symposium*, pages 207–221. USENIX Association, 2015. https://www.cs.cmu.edu/~kqy/resources/verified-hmac.pdf.

[7] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography – PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, 2006. http://cr.yp.to/papers.html#curve25519.

[8] Daniel J. Bernstein. 25519 naming. Posting to the CFRG mailing list, Aug 2008. https://www.ietf.org/mail-archive/web/cfrg/current/msg04996.html.

[9] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 159–176. Springer, 2012. http://cryptojedi.org/papers/#coolnacl.

[10] Daniel J. Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. TweetNaCl: A crypto library in 100 tweets. In Diego Aranha and Alfred Menezes, editors, *Progress in Cryptology – LATINCRYPT 2014*, volume 8895 of *LNCS*, pages 64–83. Springer, 2015. http://cryptojedi.org/papers/#tweetnacl.

[11] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning*, 61(1-4):367–422, 2018.

[12] 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. https://cryptojedi.org/papers/#verify25519.pdf.

[13] Adam Chlipala. An introduction to programming and proving with dependent types in Coq. *Journal of Formalized Reasoning*, 3(2):1–93, 2010. http://adam.chlipala.net/cpdt/.

[14] Craig Costello and Benjamin Smith. Montgomery curves and their arithmetic: The case of large characteristic fields. *Journal of Cryptographic Engineering*, 8(3), 2018. https://eprint.iacr.org/2017/212.

[15] Andres Erbsen. Crafting certified elliptic curve cryptography implementations in Coq. Master's thesis, Massachusetts Institute of Technology, 2017. http://adam.chlipala.net/theses/andreser_meng.pdf.

[16] Andres Erbsen, Jade Philipoom, Jason Gross, Rober Sloan, and Adam Chlipala. Systematic synthesis of elliptic curve cryptography implementations, 2016. https://people.csail.mit.edu/jgross/personal-website/papers/2017-fiat-crypto-pldi-draft.pdf.

[17] 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 73–90, 2019. https://people.csail.mit.edu/jgross/personal-website/papers/2019-fiat-crypto-ieee-sp.pdf.

[18] Georges Gonthier. Formal proof—the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008. https://www.ams.org/notices/200811/tx081101382p.pdf.

[19] Mike Hamburg. Ed448-goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015. https://eprint.iacr.org/2015/625.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. *ACM*, 12(10):576–580, 1969. http://doi.acm.org/10.1145/363235.363259.

[21] W. A. Howard. The formulæ-as-types notion of construction. In Philippe De Groote, editor, *The Curry-Howard Isomorphism*. Academia, 1995. https://www.cs.cmu.edu/~crary/819-f09/Howard80.pdf.

[22] Things that use Curve25519, 2019. https://ianix.com/pub/curve25519-deployment.html.

[23] The Coq Proof Assistant – Frequently Asked Questions. https://coq.inria.fr/faq.

[24] Adam Langley, Mike Hamburg, and Sean Turner. RFC 7748 – elliptic curves for security. https://tools.ietf.org/html/rfc7748.

[25] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009. http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf.

[26] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–243, 1987. http://links.jstor.org/sici?sici=0025-5718(198701)48:177<243:STPAEC>2.0.CO;2-3.

[27] Jade Philipoom. Correct-by-construction finite field arithmetic in Coq. Master's thesis, Massachusetts Institute of Technology, 2018. http://adam.chlipala.net/theses/jadep_meng.pdf.

[28] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cedric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph Wintersteiger, and Santiago Zanella-Beguelin. Evercrypt: A fast, verified, cross-platform cryptographic provider. Cryptology ePrint Archive, Report 2019/757, 2019. https://eprint.iacr.org/2019/757.

[29] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. In *Proceedings of the ACM on Programming Languages*, number 1 in ICFP, page 17. ACM, 2017. http://arxiv.org/abs/1703.00053.

[30] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, volume 17, pages 55–74. IEEE, 2002. http://www.cs.cmu.edu/~jcr/seplogic.pdf.

[31] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedtls hmac-drbg. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2007–2020, New York, NY, USA, 2017. ACM. https://doi.org/10.1145/3133956.3133974.

[32] Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. A verified extensible library of elliptic curves. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 296–309. IEEE, 2016. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7536383.

[33] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806. ACM, 2017. https://eprint.iacr.org/2017/536.pdf.

A The complete X25519 code from TweetNaCl

Verified C Code We provide below the code we verified.

```
#define FOR(i,n) for (i = 0;i < n;++i)
#define sv static void

typedef unsigned char u8;
typedef long long i64 __attribute__((aligned(8)));
typedef i64 gf[16];

sv set25519(gf r, const gf a)
{
  int i;
  FOR(i,16) r[i]=a[i];
}

sv car25519(gf o)
{
  int i;
  FOR(i,16) {
    o[(i+1)%16]+=(i<15?1:38)*(o[i]>>16);
    o[i]&=0xffff;
  }
}

sv sel25519(gf p,gf q,int b)
{
  int i;
  i64 t,c=~(b-1);
  FOR(i,16) {
    t= c&(p[i]^q[i]);
    p[i]^=t;
    q[i]^=t;
  }
}

sv pack25519(u8 *o,const gf n)
{
  int i,j,b;
  gf t,m={0};
  set25519(t,n);
  car25519(t);
  car25519(t);
  car25519(t);
  FOR(j,2) {
    m[0]=t[0]-0xffed;
    for(i=1;i<15;i++) {
      m[i]=t[i]-0xffff-((m[i-1]>>16)&1);
      m[i-1]&=0xffff;
    }
```

```
    m[15]=t[15]-0x7fff-((m[14]>>16)&1);
    b=(m[15]>>16)&1;
    m[14]&=0xffff;
    b=1-b;
    sel25519(t,m,b);
  }
  FOR(i,16) {
    o[2*i]=t[i]&0xff;
    o[2*i+1]=t[i]>>8;
  }
}

sv unpack25519(gf o, const u8 *n)
{
  int i;
  FOR(i,16) o[i]=n[2*i]+((i64)n[2*i+1]<<8);
  o[15]&=0x7fff;
}

sv A(gf o,const gf a,const gf b)
{
  int i;
  FOR(i,16) o[i]=a[i]+b[i];
}

sv Z(gf o,const gf a,const gf b)
{
  int i;
  FOR(i,16) o[i]=a[i]-b[i];
}

sv M(gf o,const gf a,const gf b)
{
  int i,j;
  i64 t[31];
  FOR(i,31) t[i]=0;
  FOR(i,16) FOR(j,16) t[i+j]+=a[i]*b[j];
  FOR(i,15) t[i]+=38*t[i+16];
  FOR(i,16) o[i]=t[i];
  car25519(o);
  car25519(o);
}

sv S(gf o,const gf a)
{
  M(o,a,a);
}

sv inv25519(gf o,const gf i)
{
  gf c;
  int a;
  set25519(c,i);
  for(a=253;a>=0;a--) {
    S(c,c);
    if(a!=2&&a!=4) M(c,c,i);
  }
  FOR(a,16) o[a]=c[a];
}

int crypto_scalarmult(u8 *q,const u8 *n,const u8 *p)
{
  u8 z[32];
  int r,i;
  gf x,a,b,c,d,e,f;
  FOR(i,31) z[i]=n[i];
  z[31]=(n[31]&127)|64;
  z[0]&=248;
  unpack25519(x,p);
  FOR(i,16) {
    b[i]=x[i];
    d[i]=a[i]=c[i]=0;
  }
  a[0]=d[0]=1;
  for(i=254;i>=0;--i) {
    r=(z[i>>3]>>(i&7))&1;
    sel25519(a,b,r);
    sel25519(c,d,r);
    A(e,a,c);
    Z(a,a,c);
    A(c,b,d);
    Z(b,b,d);
    S(d,e);
    S(f,a);
    M(a,c,a);
    M(c,b,e);
    A(e,a,c);
    Z(a,a,c);
    S(b,a);
    Z(c,d,f);
    M(a,c,_121665);
    A(a,a,d);
    M(c,c,a);
    M(a,d,f);
    M(d,b,x);
    S(b,e);
    sel25519(a,b,r);
    sel25519(c,d,r);
  }
  inv25519(c,c);
  M(a,a,c);
  pack25519(q,a);
  return 0;
}
```

Diff from TweetNaCl We provide below the diff between the original code of TweetNaCl and the code we verified.

```
--- tweetnacl.c
+++ tweetnaclVerifiableC.c
@@ -5,7 +5,7 @@
 typedef unsigned char u8;
 typedef unsigned long u32;
 typedef unsigned long long u64;
-typedef long long i64;
+typedef long long i64 __attribute__((aligned(8)));
 typedef i64 gf[16];
 extern void randombytes(u8 *,u64);

@@ -273,18 +273,16 @@
 sv car25519(gf o)
 {
   int i;
-  i64 c;
   FOR(i,16) {
-    o[i]+=(1LL<<16);
-    c=o[i]>>16;
-    o[(i+1)*(i<15)]+=c-1+37*(c-1)*(i==15);
-    o[i]-=c<<16;
+    o[(i+1)%16]+=(i<15?1:38)*(o[i]>>16);
+    o[i]&=0xffff;
   }
 }

 sv sel25519(gf p,gf q,int b)
 {
-  i64 t,i,c=~(b-1);
+  int i;
+  i64 t,c=~(b-1);
   FOR(i,16) {
     t= c&(p[i]^q[i]);
     p[i]^=t;
```

```
@@ -295,8 +293,8 @@
 sv pack25519(u8 *o,const gf n)
 {
   int i,j,b;
-  gf m,t;
-  FOR(i,16) t[i]=n[i];
+  gf t,m={0};
+  set25519(t,n);
   car25519(t);
   car25519(t);
   car25519(t);
@@ -309,7 +307,8 @@
     m[15]=t[15]-0x7fff-((m[14]>>16)&1);
     b=(m[15]>>16)&1;
     m[14]&=0xffff;
-    sel25519(t,m,1-b);
+    b=1-b;
+    sel25519(t,m,b);
   }
   FOR(i,16) {
     o[2*i]=t[i]&0xff;
@@ -353,7 +352,8 @@

 sv M(gf o,const gf a,const gf b)
 {
-  i64 i,j,t[31];
+  int i,j;
+  i64 t[31];
   FOR(i,31) t[i]=0;
   FOR(i,16) FOR(j,16) t[i+j]+=a[i]*b[j];
   FOR(i,15) t[i]+=38*t[i+16];
@@ -371,7 +371,7 @@
 {
   gf c;
   int a;
-  FOR(a,16) c[a]=i[a];
+  set25519(c,i);
   for(a=253;a>=0;a--) {
     S(c,c);
     if(a!=2&&a!=4) M(c,c,i);
@@ -394,8 +394,8 @@
 int crypto_scalarmult(u8 *q,const u8 *n,const u8 *p)
 {
   u8 z[32];
-  i64 x[80],r,i;
-  gf a,b,c,d,e,f;
+  int r,i;
+  gf x,a,b,c,d,e,f;
   FOR(i,31) z[i]=n[i];
   z[31]=(n[31]&127)|64;
   z[0]&=248;
@@ -430,15 +430,9 @@
     sel25519(a,b,r);
     sel25519(c,d,r);
   }
-  FOR(i,16) {
-    x[i+16]=a[i];
-    x[i+32]=c[i];
-    x[i+48]=b[i];
-    x[i+64]=d[i];
-  }
-  inv25519(x+32,x+32);
-  M(x+16,x+16,x+32);
-  pack25519(q,x+16);
+  inv25519(c,c);
+  M(a,a,c);
+  pack25519(q,a);
   return 0;
 }
```

As follow, we provide the explanations of the above changes to TweetNaCl's code.

- lines 7-8: We tell VST that `long long` are aligned on 8 bytes.
- lines 16-23: We remove the the undefined behavior as explained in Section 6.
- lines 29-31; lines 60-62: VST does not support `for` loops over `i64`, we convert it into an `int`.
- lines 39 & 41: We initialize `m` with `0`. This change allows us to prove the functional correctness of `pack25519` without having to deal with an array containing a mix of uninitialized and initialized values. A hand iteration over the loop trivially shows that no uninitialized values are used.
- lines 40 & 42; lines 70 & 71: We replace the `FOR` loop by `set25519`. The code is the same once the function is inlined. This small change is purely cosmetic but stays in the spirit of tweetnacl: keeping a small code size while being auditable.
- lines 50-52: VST does not allow computation in the argument before a function call. Additionally `clightgen` does not extract the computation either. We add this small step to allow VST to carry through the proof.
- lines 79-82: VST does not support `for` loops over `i64`, we convert it into an `int`.
  In the function calls of `sel25519`, the specifications requires the use of `int`, the value of `r` being either `0` or `1`, we consider this change safe.
- Lines 90-101: The `for` loop does not add any benefits to the code. By removing it we simplify the source and the verification steps as we do not need to deal with pointer arithmetic. As a result, `x` can be limited to only 16 `i64`, *i.e.*, `gf`.

B Coq definitions

B.1 Montgomery Ladder

Generic definition of the ladder:

```
(* Typeclass to encapsulate the operations *)
Class Ops (T T': Type) (Mod: T → T):=
{
  A   : T → T → T;          (* Add          *)
  M   : T → T → T;          (* Mult         *)
  Zub : T → T → T;          (* Sub          *)
```

```
  Sq  : T → T;                (* Square        *)
  C_0 : T;                    (* Constant 0    *)
  C_1 : T;                    (* Constant 1    *)
  C_121665: T;                (* const (a-2)/4 *)
  Sel25519: Z → T → T → T;    (* CSWAP         *)
  Getbit: Z → T' → Z;         (* ith bit       *)
}.

Local Notation "X + Y" := (A X Y) (only parsing).
Local Notation "X - Y" := (Zub X Y) (only parsing).
Local Notation "X * Y" := (M X Y) (only parsing).
Local Notation "X ^ 2 " := (Sq X) (at level 40,
  only parsing, left associativity).

Fixpoint montgomery_rec_swap (m : nat) (z : T')
(a: T) (b: T) (c: T) (d: T) (e: T) (f: T) (x: T) (swap:Z) :
(* a:    x2                  *)
(* b:    x3                  *)
(* c:    z2                  *)
(* d:    z3                  *)
(* e:    temporary  var      *)
(* f:    temporary  var      *)
(* x:    x1                  *)
(* swap: previous bit value  *)
(T * T * T * T * T * T * T) :=
match m with
| S n ⇒
  let r := Getbit (Z.of_nat n) z in
    (* k_t = (k >> t) & 1                              *)
  let swap := Z.lxor swap r in
    (* swap ^= k_t                                     *)
  let (a, b) := (Sel25519 swap a b, Sel25519 swap b a) in
    (* (x2 , x3 ) = cswap(swap, x2 , x3 )              *)
  let (c, d) := (Sel25519 swap c d, Sel25519 swap d c) in
    (* (z2 , z3 ) = cswap(swap, z2 , z3 )              *)
  let e := a + c in (* A  = x2 + z2                    *)
  let a := a - c in (* B  = x2 - z2                    *)
  let c := b + d in (* C  = x3 + z3                    *)
  let b := b - d in (* D  = x3 - z3                    *)
  let d := e^2    in (* AA = A^2                       *)
  let f := a^2    in (* BB = B^2                       *)
  let a := c * a in (* CB = C * B                      *)
  let c := b * e in (* DA = D * A                      *)
  let e := a + c in (* x3 = (DA + CB)^2                *)
  let a := a - c in (* z3 = x1 * (DA - CB)^2           *)
  let b := a^2    in (* z3 = x1 * (DA - CB)^2          *)
  let c := d - f in (* E  = AA - BB                    *)
  let a := c * C_121665 in
                    (* z2  = E * (AA + a24 * E)        *)
  let a := a + d in (* z2  = E * (AA + a24 * E)        *)
  let c := c * a in (* z2  = E * (AA + a24 * E)        *)
  let a := d * f in (* x2  = AA * BB                   *)
  let d := b * x in (* z3  = x1 * (DA - CB)^2          *)
  let b := e^2    in (* x3  = (DA + CB)^2              *)
  montgomery_rec_swap n z a b c d e f x r
    (* swap = k_t                                      *)

| 0%nat ⇒
  let (a, b) := (Sel25519 swap a b, Sel25519 swap b a) in
    (* (x2 , x3 ) = cswap(swap, x2 , x3 )              *)
  let (c, d) := (Sel25519 swap c d, Sel25519 swap d c) in
    (* (z2 , z3 ) = cswap(swap, z2 , z3 )              *)
  (a,b,c,d,e,f)
end.


Definition get_a (t:(T * T * T * T * T * T * T)) : T :=
match t with
  (a,b,c,d,e,f) ⇒ a
end.
Definition get_c (t:(T * T * T * T * T * T * T)) : T :=
match t with
  (a,b,c,d,e,f) ⇒ c
end.
```

B.2 RFC in Coq

Instantiation of the Class `Ops` with operations over $\mathbb{Z}$ and modulo $2^{255} - 19$.

```
Definition modP (x:Z) : Z :=
  Z.modulo x (Z.pow 2 255 - 19).

(* Encapsulate in a module. *)
Module Mid.
  (* shift to the right by n bits *)
  Definition getCarry (n: Z) (m: Z) : Z :=
    Z.shiftr m n.

  (* logical and with n ones *)
  Definition getResidue (n: Z)  (m: Z) : Z :=
    Z.land n (Z.ones n).

  Definition car25519 (n: Z) : Z  :=
    38 * getCarry 256 n +  getResidue 256 n.
  (* The carry operation is invariant under modulo *)
  Lemma Zcar25519_correct:
    forall (n: Z), n:GF = (Mid.car25519 n) :GF.

  (* Define Mid.A, Mid.M ... *)
  Definition A a b := Z.add a b.
  Definition M a b :=
    car25519 (car25519 (Z.mul a b)).
  Definition Zub a b := Z.sub a b.
  Definition Sq a := M a a.
  Definition C_0 := 0.
  Definition C_1 := 1.
  Definition C_121665 := 121665.
  Definition Sel25519 (b p q: Z) :=
    if (Z.eqb b 0) then p else q.

  Definition getbit (i:Z) (a: Z) :=
    if (Z.ltb a 0) then      (* a < 0*)
      0
    else if (Z.ltb i 0) then (* i < 0 *)
        Z.land a 1
    else                     (* 0 ≤ a & 0 ≤ i *)
        Z.land (Z.shiftr a i) 1.
End Mid.

(* Clamping *)
Definition clamp (n: list Z) : list Z :=
  (* set last 3 bits to 0 *)
  let x := nth 0 n 0 in
  let x' := Z.land x 248 in
  (* set bit 255 to 0 and bit 254 to 1 *)
  let t := nth 31 n 0 in
  let t' := Z.lor (Z.land t 127) 64 in
  (* update the list *)
  let n' := upd_nth 31 n t' in
    upd_nth 0 n' x'.

(* x^{p - 2} *)
Definition ZInv25519 (x: Z) : Z :=
  Z.pow x (Z.pow 2 255 - 21).

(* reduction modulo P *)
Definition ZPack25519 (n: Z) : Z :=
  Z.modulo n (Z.pow 2 255 - 19).


(* instantiate over Z *)
Instance Z_Ops : (Ops Z Z modP) := {}.
Proof.
  apply Mid.A.        (* instantiate +       *)
```

```
  apply Mid.M.        (* instantiate *       *)
  apply Mid.Zub.      (* instantiate -       *)
  apply Mid.Sq.       (* instantiate x^2     *)
  apply Mid.C_0.      (* instantiate Const 0 *)
  apply Mid.C_1.      (* instantiate Const 1 *)
  apply Mid.C_121665. (* instantiate (a-2)/4 *)
  apply Mid.Sel25519. (* instantiate CSWAP   *)
  apply Mid.getbit.   (* instantiate ith bit *)
Defined.

Definition decodeScalar25519 (l: list Z) : Z :=
  ZofList 8 (clamp l).

Definition decodeUCoordinate (l: list Z) : Z :=
  ZofList 8 (upd_nth 31 l (Z.land (nth 31 l 0) 127)).

Definition encodeUCoordinate (x: Z) : list Z :=
  ListofZ32 8 x.

(* instantiate montgomery_rec_swap with Z_Ops *)
Definition RFC (n: list Z) (p: list Z) : list Z :=
  let k := decodeScalar25519 n in
  let u := decodeUCoordinate p in
  let t := montgomery_rec_swap
    255  (* iterate 255 times  *)
    k    (* clamped n          *)
    1    (* x2                 *)
    u    (* x3                 *)
    0    (* z2                 *)
    1    (* z3                 *)
    0    (* dummy              *)
    0    (* dummy              *)
    u    (* x1                 *)
    0    (* previous bit = 0   *) in
  let a := get_a t in
  let c := get_c t in
  let o := ZPack25519 (Z.mul a (ZInv25519 c))
  in encodeUCoordinate o.
```

C Organization of the proof files

Requirements Our proofs requires the use of *Coq 8.8.2* for the proofs and *Opam 2.0* to manage the dependencies. We are aware that there exists more recent versions of Coq; VST; CompCert etc. however to avoid dealing with backward breaking compatibility we decided to freeze our dependencies.

Associated files The archive containing the proof is composed of two folders **packages** and **proofs**. It aims to be used at the same time as an *opam* repository to manage the dependencies of the proof and to provide the code.

The actual proofs can be found in the **proofs** folder in which the reader will find the directories **spec** and **vst**.

packages/ This folder makes sure that we are using the correct version of Verifiable Software Toolchain (version 2.0) and CompCert (version 3.2). Additionally it pins the version of the elliptic curves library by Bartzia and Strub and allows us to use the theorem of quadratic reciprocity.

proofs/spec/ In this folder the reader will find multiple levels of implementation of X25519.

- **Libs/** contains basic libraries and tools to help use reason with lists and decidable procedures.
- **ListsOp/** defines operators on list such as `ZofList` and related lemmas using *e.g.*, `Forall`.
- **Gen/** defines a generic Montgomery ladder which can be instantiated with different operations. This ladder is the stub for the following implementations.
- **High/** contains the theory of Montgomery curves, twists, quadratic extensions and ladder. It also proves the correctness of the ladder over $\mathbb{F}_{2^{255}-19}$.
- **Mid/** provides a list-based implementation of the basic operations `A`, `Z`, `M` ... and the ladder. It makes the link with the theory of Montgomery curves.
- **Low/** provides a second list-based implementation of the basic operations `A`, `Z`, `M` ... and the ladder. Those functions are proven to provide the same results as the ones in `Mid/`, however their implementation are closer to `C` in order facilitate the proof of equivalence with TweetNaCl code.
- **rfc/** provides our rfc formalization. It uses integers for the basic operations `A`, `Z`, `M` ... and the ladder. It specifies the decoding/encoding of/to byte arrays (seen as list of integers) as in RFC 7748.

proofs/vst/ Here the reader will find four folders.

- **c** contains the C Verifiable implementation of TweetNaCl. `clightgen` will generate the appropriate translation into Clight.
- **init** contains basic lemmas and memory manipulation shortcuts to handle the aliasing cases.
- **spec** defines as Hoare triple the specification of the functions used in `crypto_scalarmult`.
- **proofs** contains the proofs of the above Hoare triples and thus the proof that TweetNaCl code is sound and correct.

D Prior reviews

This paper has been submitted to IEEE Symposium on Security and Privacy 2020.

We thank and really appreciate the reviewers who took time to provide us with such detailed feedback.

D.1 S&P 2020 Review #582A

Overall merit: 3. Weak reject - The paper has flaws, but I will not argue against it.

===== Brief paper summary (2-3 sentences) =====

This paper presents a verification of the C implementation of the X25519 key-exchange protocol in the TweetNaCl library. Correctness properties including the C code implements the algorithm correctly are verified in Coq, using the Verified Software Toolchain (VST).

===== Strengths =====

+ This paper produces a fully formally verified crypto library.

+ This paper presents solid technical work.

===== Weaknesses =====

- The novelty of this paper is not well explained. It's unclear whether this paper has pushed the boundary in terms of what can be formally verified.

===== Detailed comments for the author(s) =====

I appreciate work on producing fully verified software. This paper presents another instance: X25519 protocol. The result here is very strong: it verified both safety properties and the correctness of the C implementation.

One main weakness of this paper is that it is hard to extract from the writing, what are the new discoveries in this exercise. Are there new proof techniques developed for this verification? Is there anything new about the way that loops are handled? Is the application of reflection challenging? Are there bugs found, in particular, in the mathematical operations?

Our answer:
The novelty of this work is not in the proof techniques. It lies in the assurance gained by the formalization of the X25519 from RFC 7748, and its correctness with respect to the theory of elliptic curves. Additionally it is the first time, a link from the C code up to the mathematical definitions of Curve25519 using a single tool is presented.

A related comment is that it is unclear from the paper whether now we can re-use proofs in this paper to verify mathematical operations of other protocols that use elliptic curve.

Our answer:
We believe that parts of our proofs are reusable either by their structure to give an intuition or by directly changing some values (for e.g. X448). The mathematical definitions are generic and may be instantiated over any fields (of characteristic neither 2 or 3). The Ladder may be instantiated over operations over different data structure for the underlying arithmetic, making it reusable. The low level operations *e.g.*, `A` are also reusable in the proof of ed25519.

In the Corrections in TweetNaCl paragraph, two things were discussed. It's unclear how and whether either one of them directly relate to the verification effort. It would be nice if they are. Then the story for the usefulness of the verification become stronger.

Our answer:
The switch from `i64` to `int` in `for` loops and the extraction of the computation outside of the function call in `pack25519` are required by VST but we believe those change have no impact on the verification effort.
The only modification which made the verification easier was the removal of the *dead code* at the end of `crypto_scalarmult`. Peter Wu and Jason A. Donenfeld had also noticed this part and informed us after we already fixed the code.

It would be nice if the authors comment on the verification effort: person month etc.

Our answer:
In addition to the time required to get familiar with research software, we faced a few bugs which we reported to the developers of VST to get them fixed. It is very hard to work with a tool without being involved in the development loop. Additionally newer versions often broke some of our proofs and it was often needed to adapt to the changes. As a result we do not believe the metric person-month to be a good representation of the verification effort.

Also, did the verified version replace the version in use in production software? If not, why not?

Our answer:
We contacted the authors of TweetNaCl and expect that the changes above mentioned will soon be integrated in a new version of the library, mostly for `car25519` and the simplification in `crypto_scalarmult`.

Section V is extremely dry. What are the high-level insights? Perhaps a picture representation of how the theorems and lemmas fit together would be a better way to represent this section.

Our answer:
We provided overviews of the proof at the beginning of sections V.1 and V.2. We hope this will give the reader an intuition of how lemmas fit together.

In summary, this paper is in need of significant improvements in the writing and providing detailed discussions of high-level intellectual contributions in this verification effort.

Minor

On page 6: "each elements" => no s

On page 7, the beginning of the page, a half sentence is dangling there.

On page 9: "each functions" => no s

D.2 S&P 2020 Review #582B

Overall merit: 4. Weak accept - While flawed, the paper has merit and we should consider accepting it.

===== Brief paper summary (2-3 sentences) =====

In this paper, the authors verify the C implementation of the X25519 elliptic curve key exchange in the TweetNaCl library, a short implementation of the NaCl library. The X25519 key exchange method is used in TLS 1.3, Signal, Tor, Zcash and SSH. Within TLS 1.3 (and probably also in other protocols), any implementation of X25519 may be used in an implementation of the standard. Thus, the present work contributes to a fully verified code-base for TLS 1.3 (and other crucial protocols)

The verification itself is practically significant. The paper makes a good effort to describe the approach, but I was, at times, a little disappointed at the description. Performing this type of verification yields substantial human understanding and enables the authors to carry out a similar approach for other primitives much more easily. I wish that the authors would extract their conceptual insights on the technique and communicate those to the readers. Regardless, I would very much like to see this work at S&P, but I would like to encourage the authors to de-emphasize technical details (even more) and spend more space of the body on conceptual insights (see comments below).

===== Detailed comments for the author(s) =====

I was confused that you did not seem have found any errors in an implementation. I can believe that the implementers of [10] are proficient implementers, but I would have expected some minor bugs/parsing issues to be found anyway. What is your explanation that you haven't found any issues?

Our answer:
The implementation of TweetNaCl is straight forward and simple, big number arithmetic is done without optimizations. It is easy to get an intuition of how the code work by reading it. The ladder is a direct mapping of RFC 7748. It does not aim for speed optimizations. This reduce significantly the complexity of the code compared to e.g., Fiat Crypto generated optimized C code.
While we did not find errors in the implementation with respect to functional correctness under the CompCert assumptions, we removed an undefined behavior by the C standard.

p.1: You discuss [7] as using heavy annotation of code, differently from your own code. However, as far as I understand, your Coq implementation must also be type-annotated. Thus, I imagine that the difference lies in whether SAT-solvers are used or not? This did not become very clear, because you mostly describe names of tool while leaving out the details of what these tools carry out conceptually.

Our answer:
SMT solvers strategies are in need of annotation, often written as parsable comment in the middle of the C code [5] in order to "guide" the proof. In our case we only need to annotate the beginning of the function and loop invariants which allows us to get tighter bounds.

p.2: Fig 1 is very nice. However, at this point, none of the notation has been explained to the reader, and I imagine that V stands for VST? It wasn't clear to me. Maybe, the caption of the figure could explain what the details inside the figures represent. In particular, the Figure seems to used a nice abstraction of properties that is not used elsewhere in the paper. I.e., elsewhere in the paper, there is usually a lot of code details.

Our answer:
".V" is the file extension for Coq files (as opposed to ".C" for C files). This allows us to emphasis that all our work is done in a single framework. We clarified this in the introduction.

p.2-4: Subsection A took some mathematical background. Obviously, not all of it could be covered, but the motivation for why certain content was presented and not others was not clear to me as a reader, which was quite frustrating. In Section B, it was even less clear why particular details were presented, e.g., the discussion of the computation of n' was completely unclear to me. Subsection D seemed a good idea, but was very tough to follow, too much content in short time. Subsection D seems to jump between implementation levels and discussing high-level arguments such as Fermat's little theorem.

Our answer:
We aim to give a quick view of how Montgomery curves are defined and remind the reader of the "twisting factor". The computation of n' is required for the security of X25519 as it reduces the computations to a prime-order group, we use this presentation to refer it later as "clamping". We modified Subsection D to omit all the code definitions of small functions to ease the reader experience. The C definitions are now provided in the appendix.

p.3: I very much liked the discussion of the Diff between TweetNaCl and your modifications! I imagine that many of them are conceptually interesting in that they make verification easier. I would have enjoyed reading a conceptual discussion of the differences. Instead, Appendix A just contains a Diff, and as a reader, I would need to perform the extraction of insights myself.

Our answer:
Due to page restrictions for IEEE S&P, including appendices, We had a condensed diff. We changed the format to a "diff -u" to provide the context of the applied changes. We also now describe the motivations of each changes below the said Diff.

p.5: CSWAP: Maybe use a backward reference to p.3 where CSWAP was defined, since in between, there was a lot of content, and relying on the reader's memory does not really work here (or at least, it did not work for me).

Our answer:
As we rewrote significantly section 2.4, the space between the definition and the use of CSWAP decreased. We added a backward reference to where it was originally defined.

p.7: I enjoyed the discussion on aliasing! This was the type of conceptual discussion I find interesting. I was confused about the mentioning that the case distinction k=0,1,2 does not cover "all cases" - all cases of what? And why is this distinction sufficient for the task at hand?

Our answer:
In that section we clarified the discussion of what would be an unnecessary aliasing case and why such case do not happen in TweetNaCl. For example, TweetNaCl never has function calls where the 3 pointers are aliased (`o = a = b`), as such we do not consider this aliasing case in the specifications of the function.

Most of the rest of the paper felt hard to read, because it read like a chronological enumeration of all steps. I think that it would be much more interesting if the paper highlights the interesting aspects of the verification.

Our answer:
We removed sections and details in order to simplify the reading experience. We also added figures in section V in order to provide the reader a mental image of how the lemmas, definitions and theorems work together.

Related work:

In reference [29], the order of authors is different than in the original publication. I think that you might want to discuss [29]in the Related work section, since it seems quite close. I was wondering whether you consider the approach in [29]as synthesis or verification, because to me, it seemed a mix/neither.

Our answer:
The bibtex was from DBLP. We fixed the order of the authors with respect to the original publication and contacted DBLP to update their files.
Figure 5 highlights the difference between Low* and our approach.

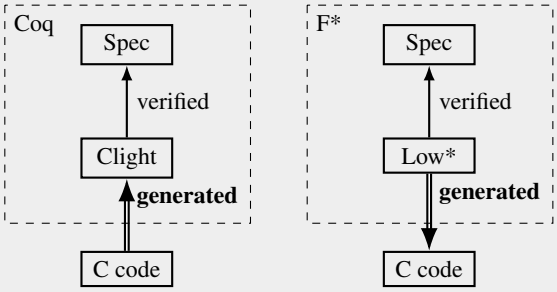

Figure 5: VST vs Low*

Low* generates C code, thus this is synthesis. We generate Clight from C code, thus this is verification. Because Low* also has a verification step with respect to some high-level specifications, we consider their process to be a mix of the two methods. Also to be noted that both VST and Low* allow the verification of memory safety.

Clarification in intro:

The intro states that this is, to the authors knowledge, the first use of Verifiable Software Toolchain (VST) in software verification of an implementation of an asymmetric cryptographic primitives. Does this mean that VST has been used for symmetric cryptographic primitives before? From the text, I wasn't sure whether this is careful quantification or not.

Our answer:
This is indeed careful quantification. VST has already been used to verify symmetric Cryptographic primitives (*e.g.*, SHA), as mentioned later by [6] and [2].

D.3 S&P 2020 Review #582C

Overall merit: 3. Weak reject - The paper has flaws, but I will not argue against it.

===== Brief paper summary (2-3 sentences) =====

This paper presents a formalization of the X25519 (Curve25519) elliptic curve Diffie-Hellman (ECDH) construction in Coq, and uses this formalization to verify the functional correctness of a C implementation of this construction taken from the TweetNaCl library. Going further than prior work, the authors also link their Coq specification to a mathematical theory of elliptic curves in Coq.

===== Strengths =====

Proofs of cryptographic algorithms like X25519 tend to mean different things in different communities: (a) one may prove that X25519 implements a group based on the mathematical theory of elliptic curves, (b) one may prove that an X25519 implementation meets its mathematical spec, or (c) one may prove that a protocol that uses X25519 is provably secure, based on some cryptographic assumption on the ECDH construction. It is rare to find papers that combine more than one of these proof levels, and this paper does a creditable job of linking (a) with (b), relying on the Coq framework to soundly glue these proofs together. Although a similar goal was considered in [32], this paper provides a more satisfying solution by not leaving any gaps between the formalizations, and by relying on a smaller trusted computing base.

===== Weaknesses =====

X25519 implementations have now been verifying in multiple papers using multiple verification frameworks, and [16] even uses Coq to verify a large portion of a C implementation of X25519 fully automatically. So, the main contribution here is that that the authors verify an existing popular implementation (TweetNaCl) without making many changes to the source code. Still, TweetNaCl is not necessarily the most complex (and certainly not the most efficient) implementation of X25519 that has been verified, which makes the novelty of this work hard to justify.

Our answer:
Indeed [16]'s C code of X25519 is verified. However the authors do not qualify their approach as verification but as synthesis. They do not use Coq to verify C code, instead they "synthesize" (generate) C code from verified refinements of the specification. Our approach goes from the C code up to the specification.

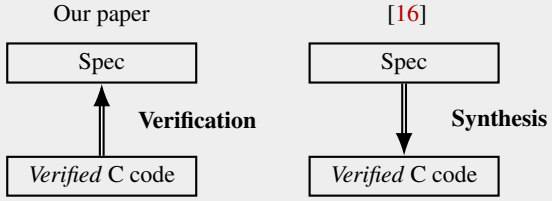

Figure 6: Our approach vs [16]

===== Detailed comments for the author(s) =====

I like the approach taken in this paper and I think this work has value and should be published. However, I am not sure the current presentation of the work works well.

The paper can be essentially divided into two components: the proof of TweetNaCl using VST, and the proof of equivalence between the Coq spec of RFC7748 and the mathematical theory of Curve25519. The details of both are interesting to formalists but the Coq details are quite hard to parse and distracting when trying to understand the text. This is a well-known problem for formal verification papers, and I don't have great advice except to suggest that the authors separate out the code fragments into figures and let the text focus on the high-level ideas of the code.

Our answer:
We removed a significant part of the Coq details, especially details about reflections in order to make the reading experience smooth. As suggested, we also added a few figures to give the reader an overview of how the proofs are linked together.

At the end of Section IV, I would have loved to see some high-level lessons on what the proof taught you, and what part of this proof would be reusable if you decided to verify Ed25519 or X448 or P-256, for example. This is particularly important for this work, since prior papers have shown how to share proof effort across multiple curves.

Our answer:
We added insights in Section 4.1 of how to improve working with VST. In section Section 6, we described in further details how our work can be extended to other curves. The TweetNaCl ed25519 implementation makes use of the same low level arithmetic as X25519, reusing our proofs. For example, `pow2523` is similar to `inv25519` and would allow us to reuse our reflection abstraction. X448 would reuse the ladder but the low level arithmetic would need to be proven again. P-256 is a different case as due to its short Weierstraß shape, this implies that our ladder is not fit for it. However using [4] and our proofs as example it is possible to prove a similar ladder as ours.

It also appears that the authors do not verify the constant-time guarantees of the TweetNaCl code. Could this be done in their framework?

Our answer:
Constant-timeness is not a property verifiable with VST. This is not verifiable with our framework.

Section V presents the proof of mathematical equivalence. It appears to me that this proof is completely independent of the TweetNaCl proof. Is that the case? Is there any lemma

from the first part that is needed in the second? More importantly, could you claim that your proof of equivalence also provides a justification for the implementations of Fiat Crypto or HACL*, hence adding value to those other projects in addition to your own?

Our answer:
Indeed those two proofs are completely independent. The only link between Section 4 and Section 5 is the definition of `RFC` and the preconditions on the inputs: 2 arrays of 32 bytes. We highlighted in Section 6 that our proof of correctness of X25519 in RFC 7748 increases the confidence in other implementations such as Fiat Crypto and HACL* (under the assumption they correctly formalized the RFC).

I was hoping to see where the "clamping" used in X25519 would appear in the formalization of Section V but was unable to spot it. Does clamping affect the proofs in any way? Is there a stronger property you could prove because of the clamping (e.g. membership in the prime-order group)? Do you already prove such a property?

Our answer:
The clamping by itself has no impact on the computations using the ladder. It is thus normal to not see it appear in Section 5 as our proofs work for any 255-bit n, We prove the correctness of X25519 in RFC 7748, we do not prove its security. The proof of a stronger property (e.g. membership in prime-order group) shall be done separately before being applied to Curve25519.

More generally, what about the proof of Section V is specific to X25519 and what is generic?

Our answer:
Everything in Section 5.1 is generic, anything left in Section 5.2 is specific to X25519.