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).

I. Introduction

The Networking and Cryptography library (NaCl) [1] 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 [2] 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"* [2]. 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 [3]. This protocol is standardized in RFC 7748 and used by a wide variety of applications [4] 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 [5]. We use this updated terminology in this paper.

Contribution of this paper. We provide a mechanized formal proof of the correctness of the X25519 implementation in TweetNaCl. This proof is done in three steps:

We first formalize RFC 7748 [3] in Coq [6].

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) [7] to establish a link between C and Coq. VST uses Separation logic [8], [9] 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 implementation matches the mathematical definition of X25519 as given in [10, Sec. 2]. This provides additional confidence that there are no transcription errors in our formalized version of X25519. We do this by extending the Coq library for elliptic curves [11] by Bartzia and Strub to support Montgomery curves, and in particular Curve25519. This extension may be of independent interest.

Related work. Two methodologies exist to get formal guarantees that software meets its specification. The first is to write a formal specification and then synthesize machine code by refinement; the second is to formalize a specification and then verify that some implementation satisfies it.

The synthesis approach was used by Zinzindohoué, Bartzia, and Bhargavan to build a verified extensible library of elliptic curves [12] in F* [13]. This served as ground work for the cryptographic library HACL* [14] used in the NSS suite from Mozilla.

Coq not only allows verification but also synthesis [15]. 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 [16]–[18]. This software suite is now being used in BoringSSL [19].

The verification approach has been used to prove the correctness of OpenSSL's implementations of HMAC [20] and SHA-256 [21]. In terms of languages and tooling, this work is closest to what we present here, but our work considers an asymmetric primitive and provides computer-verified proofs up to the mathematical definition of the group theory behind elliptic curves.

When considering the target of the verification effort, the closest work to this paper is presented in [22], which presents a mechanized proof of two assembly-level implementations of the core function of X25519. The proof in [22] takes a different approach from ours. It uses heavy annotation of code together with SAT solvers; also, it does not cover the full X25519 functionality and does not make the link to the mathematical definition from [10].

Reproducing the proofs. To maximize reusability of our results we placed the code of our formal proof presented in this paper into the public domain. It is available at https://cdn-09.anonfile.com/6fp8ta70n9/014c4f8c-1569921449/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 II gives the necessary background on Curve25519 and X25519 implementations and

a brief explanation of how formal verification works. Section III provides our Coq formalization of X25519 as specified in RFC 7748 [3]. Section IV provides the specifications of X25519 in TweetNaCl and some of the proof techniques used to show the correctness with respect to RFC 7748 [3]. Section V describes our extension of the formal library by Bartzia and Strub and the correctness of X25519 implementation with respect to Bernstein's specifications [5]. Finally in Section VI we discuss the trusted code base of our proofs. Figure 1 shows a graph of dependencies of the proofs.

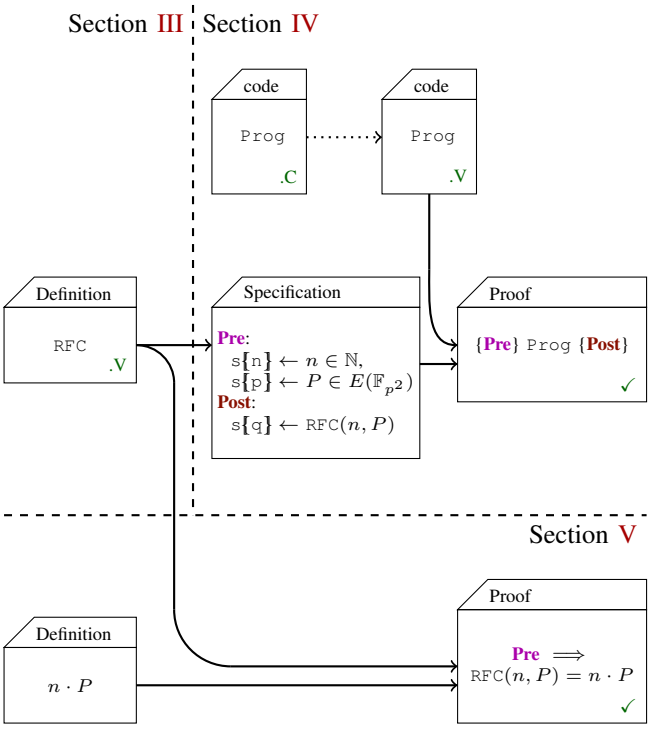

Fig. 1. Structure of the proof

II. 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.

A. Arithmetic on Montgomery curves

Definition II.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 II.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*

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

Here, the additional element $\mathcal{O}$ denotes the point at infinity.

Details of the formalization can be found in Section V-A2.

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 II.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 $\mathbb{F}_{p^2}$ [23].*

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 $\mathcal{O}$. 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 [24].

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 V-A3 for more details. We define two operations:

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

In the Montgomery ladder, the arguments of `xADD` and `xDBL` 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 ← xDBL(Q)
    R ← xADD(x_P, Q, R)
    (Q, R) ← CSWAP((Q, R), k^th bit of n)
end for
return X_Q/Z_Q
```

B. 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 integer n (see III-B). 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' by setting bit 255 of n to `0`, setting bit 254 to `1`, and setting the lower 3 bits to `0`. 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 [3] standardized 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 with X25519 on s_a and X_{P_b} (respectively s_b and X_{P_a}).

C. 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. It is then followed by the input arguments.

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

D. 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];
```

Conversion from the input byte array to this representation in radix 2^{16} is done as follows:

```
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;
}
```

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 carry handling in the implementations of addition (`A`) and subtraction (`Z`) in $\mathbb{F}_{2^{255}-19}$:

```
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];
}
```

Multiplications (`M`) also heavily exploit the redundancy of the representation to delay carry handling.

```
sv M(gf o,const gf a,const gf b) {
  i64 i,j,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);
}
```

After the multiplication, the limbs of the result `o` are too large to be used again as input. The two calls to `car25519` at the end of `M` propagate the carries through the limbs:

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

In order to simplify the verification of this function, we treat the last iteration of the loop $i = 15$ as a separate step.

Inverses in $\mathbb{F}_{2^{255}-19}$ are computed with `inv25519`. This function uses exponentiation by $2^{255} - 21$, computed with the square-and-multiply algorithm. Fermat's little theorem gives the correctness. Notice that in this case the inverse of 0 is defined as 0.

```
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];
}
```

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

```
sv sel25519(gf p,gf q,i64 b)
{
```

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

Finally, we need the `pack25519` function, which converts from the internal redundant radix-2^{16} representation to a unique byte array representing an integer in $\{0, \ldots, p-1\}$ in little-endian format.

```
sv pack25519(u8 *o,const gf n)
{
  int i,j;
  i64 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);
    m[14]&=0xffff;
    b=1-((m[15]>>16)&1);
    sel25519(t,m,b);
  }
  FOR(i,16) {
    o[2*i]=t[i]&0xff;
    o[2*i+1]=t[i]>>8;
  }
}
```

As we can see, this function is considerably more complex than the unpacking function. The reason is that 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 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.

```
int crypto_scalarmult(u8 *q,
                      const u8 *n,
                      const u8 *p)
{
  u8 z[32];
  i64 r;
  int 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;
}
```

E. Coq, separation logic, and the VST

Coq [6] 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 [25] and it is also the system underlying the CompCert formally verified C compiler [26]. Unlike systems like F* [13], 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 [27].

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 IV-A.

The Verified Software Toolchain (VST) [28] 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 the VST, one uses a strongest postcondition approach to prove the correctness of the triple. This approach can be seen as a forward symbolic execution of the program.

III. Formalizing X25519 from RFC 7748

In this section we present our formalization of RFC 7748 [3].

The specification of X25519 in RFC 7748 is formalized by `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
    255  (* iterate 255 times *)
    k    (* clamped n         *)
    1    (* x2                *)
    u    (* x3                *)
    0    (* z2                *)
    1    (* z3                *)
    0    (* dummy             *)
    0    (* dummy             *)
    u    (* x1                *) 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` is defined as follows:

```
Fixpoint montgomery_rec (m : nat) (z : T')
(a: T) (b: T) (c: T) (d: T) (e: T) (f: T) (x: T) :
(* a: x2              *)
(* b: x3              *)
(* c: z2              *)
(* d: z3              *)
(* e: temporary  var  *)
(* f: temporary  var  *)
(* x: x1              *)
(T * T * T * T * T * T) :=
match m with
| 0%nat ⇒  (a,b,c,d,e,f)
| S n ⇒
  let r := Getbit (Z.of_nat n) z in
    (* k_t = (k >> t) & 1                          *)
    (* swap ← k_t                                  *)
  let (a, b) := (Sel25519 r a b, Sel25519 r b a) in
    (* (x2, x3) = cswap(swap, x2, x3)              *)
  let (c, d) := (Sel25519 r c d, Sel25519 r 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          *)
  let (a, b) := (Sel25519 r a b, Sel25519 r b a) in
    (* (x2, x3) = cswap(swap, x2, x3)              *)
  let (c, d) := (Sel25519 r c d, Sel25519 r d c) in
    (* (z2, z3) = cswap(swap, z2, z3)              *)
  montgomery_rec n z a b c d e f x
end.
```

The definitions of the encoding and decoding functions are detailed in Section III-B.

RFC 7748 states that *"All calculations are performed in GF(p), i.e., they are performed modulo p."* Operations used in the Montgomery ladder of `RFC` are performed on integers (See Appendix B2). The reduction modulo $2^{255} - 19$ is deferred to the very end as part of the `ZPack25519` operation.

We briefly introduce our generic description of the Montgomery ladder (III-A). Then we describe our mapping between little-endian representations and integers (III-B) which we use to formalize the encoding and decoding.

A. A generic ladder

TweetNaCl implements X25519 with numbers represented as arrays. RFC 7748 defines X25519 over field elements. We show the equivalence between the different number representations. To simplify our proof, we define operations used in the ladder over generic types `T` and `T'`. Those types are later instantiated as list of integers, integers, natural numbers, or field elements.

Our formalization differs slightly from the RFC. Indeed in order to optimize the number of calls to `CSWAP` the RFC uses an additional variable to decide whether a conditional swap is required or not. Our description of the ladder follows strictly the shape of the exponent as described in Algorithm 1. This divergence is allowed by the RFC: *"Note that these formulas are slightly different from Montgomery's original paper. Implementations are free to use any correct formulas."* [3]. We later prove our ladder correct in that respect (Section V).

B. Integers and bytes

"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." [3]

In TweetNaCl, as described in Section II-C, elements of $\mathbb{F}_p$ are represented as big integers using radix 2^{16}. We use a direct mapping to represent such an array of limbs as a list of integers in Coq.

We define the little-endian projection to integers as follows.

Definition III.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.*

We define it in Coq as:

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

`ZofList` is used in the decoding of byte arrays and also as a base to verify operations over lists in TweetNaCl (see IV-B). The encoding from integers to bytes is defined in a similar way:

Definition III.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.*

We define it in Coq as:

```
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 this definition, `clamp` is taking care of setting and unsetting the selected bits as stated in the RFC and described in Section II-B.

IV. 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 [3] (`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 III) we specify the Hoare triple before proving its correctness with the VST (IV-A). We provide an example of equivalence of operations over different number representations (IV-B). Then, we describe efficient techniques used in some of our more complex proofs (IV-C).

A. 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 elements 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]`.

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 with Theorem body_crypto_scalarmult.

This specification (proven with VST) shows that crypto_scalarmult in C

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 the 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 the VST will expect a third memory section (a) which does not *exist* anymore. Examples of such cases are illustrated in Figure 2.

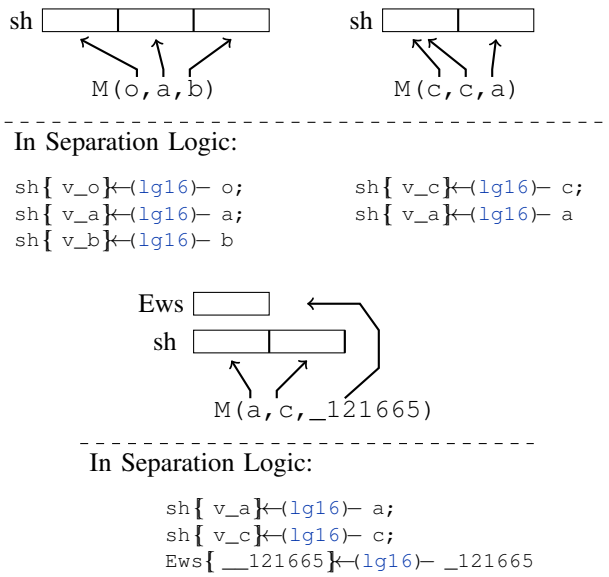

Fig. 2. Aliasing and Separation Logic

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 cases (*e.g.*, all arguments are aliased) but it is enough for our needs.

Verifying `for` loops. Final states of for loops are usually computed by simple recursive functions. However, we must define invariants which are true for each iteration step.

Assume that we want to prove a decreasing loop where indexes go from 3 to 0. Define a function $g : \mathbb{N} \to State \to State$ which takes as input an integer for the index and a state, then returns a state. It simulates the body of the for loop. Define the recursion: $f : \mathbb{N} \to State \to State$ which iteratively applies g with decreasing index:

$$f(i,s) = \begin{cases} s & \text{if } s = 0 \\ f(i-1, g(i-1,s)) & \text{otherwise} \end{cases}$$

Then we have:

$$f(4,s) = g(0,g(1,g(2,g(3,s))))$$

To prove the correctness of $f(4,s)$, we need to prove that intermediate steps $g(3,s)$; $g(2,g(3,s))$; $g(1,g(2,g(3,s)))$; $g(0,g(1,g(2,g(3,s))))$ are correct. Due to the computation order of recursive function, our loop invariant for $i \in \{0,1,2,3,4\}$ cannot use $f(i)$. To solve this, we define an auxiliary function with an accumulator such that given $i \in \{0,1,2,3,4\}$, it will compute the first i steps of the loop.

We then prove for the complete number of steps, the function with the accumulator and without returns the same result. We formalized this result in a generic way in Appendix B3.

Using this formalization, we prove that the 255 steps of the Montgomery ladder in C provide the same computations as in RFC.

B. Number representation and C implementation

As described in Section II-C, numbers in gf are represented in base 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 a full integer. We reuse the mapping ZofList : $\mathbb{Z} \to$ list $\mathbb{Z} \to \mathbb{Z}$ from Section III-B 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 V-B1, we formally define $\mathbb{F}_{2^{255}-19}$. 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 IV.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 IV.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).
```

C. Reflections, inversions and packing

We now turn our attention to the multiplicative inverse in $\mathbb{Z}_{2^{255}-19}$ and techniques to improve the verification speed of complex formulas.

Inversion in $\mathbb{Z}_{2^{255}-19}$. We define a Coq version of the inversion mimicking the behavior of `inv25519` (see below) over `list Z`.

```
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];
}
```

We specify this with 2 functions: a recursive `pow_fn_rev` to simulate the `for` loop and a simple `step_pow` containing the body.

```
Definition step_pow (a:Z)
  (c:list Z) (g:list Z) : list Z :=
  let c := Sq c in
    if a ≠? 2 && a ≠? 4
    then M c g
    else c.

Function pow_fn_rev (a:Z) (b:Z)
  (c: list Z) (g: list Z)
  {measure Z.to_nat a} : (list Z) :=
  if a ≤? 0
    then c
    else
      let prev := pow_fn_rev (a - 1) b c g in
        step_pow (b - a) prev g.
```

This `Function` requires a proof of termination. It is done by proving the well-foundedness of the decreasing argument: `measure Z.to_nat a`. Calling `pow_fn_rev` 254 times allows us to reproduce the same behavior as the Clight definition.

```
Definition Inv25519 (x:list Z) : list Z :=
  pow_fn_rev 254 254 x x.
```

Similarly we define the same function over $\mathbb{Z}$.

```
Definition step_pow_Z (a:Z) (c:Z) (g:Z) : Z :=
  let c := c * c in
  if a ≠? 2 && a ≠? 4
    then c * g
    else c.

Function pow_fn_rev_Z (a:Z) (b:Z) (c:Z) (g: Z)
  {measure Z.to_nat a} : Z :=
  if (a ≤? 0)
    then c
    else
      let prev := pow_fn_rev_Z (a - 1) b c g in
        step_pow_Z (b - a) prev g.

Definition Inv25519_Z (x:Z) : Z :=
  pow_fn_rev_Z 254 254 x x.
```

By using Lemma IV.1, we prove their equivalence in $\mathbb{Z}_{2^{255}-19}$.

Lemma IV.3 *The function* `Inv25519` *over list of integers computes the same result at* `Inv25519_Z` *over integers in* $\mathbb{Z}_{2^{255}-19}$*.*

This is formalized in Coq as follows:

```
Lemma Inv25519_Z_GF : forall (g:list Z),
  length g = 16 →
  (ℤ16.lst (Inv25519 g)) :GF =
  (Inv25519_Z (ℤ16.lst g)) :GF.
```

In TweetNaCl, `inv25519` computes an inverse in $\mathbb{Z}_{2^{255}-19}$. It uses Fermat's little theorem by raising to the power of $2^{255}-21$ with a square-and-multiply algorithm. The binary representation of $p-2$ implies that every step does a multiplications except for bits 2 and 4. To prove the correctness of the result we could use multiple strategies such as:

- Proving it is a special case of square-and-multiply algorithm applied to $2^{255}-21$.
- Unrolling the for loop step-by-step and applying the equalities $x^a \times x^b = x^{(a+b)}$ and $(x^a)^2 = x^{(2\times a)}$. We prove that the resulting exponent is $2^{255}-21$.

We use the second method because it is simpler. However, it requires us to apply the unrolling and exponentiation formulas 255 times. This could be automated in Coq with tacticals such as `repeat`, but it generates a proof object which will take a long time to verify.

Reflections. In order to speed up the verification we use a technique called "Reflection". It provides us with flexibility, *e.g.*, we don't need to know the number of times nor the order in which the lemmas needs to be applied (chapter 15 in [15]).

The idea is to *reflect* the goal into a decidable environment. We show that for a property P, we can define a decidable Boolean property P_{bool} such that if P_{bool} is `true` then P holds.

$$\mathit{reify\_P} : P_{bool} = \mathit{true} \implies P$$

By applying *reify_P* on P our goal becomes $P_{bool} = \mathit{true}$. We then compute the result of P_{bool}. If the decision goes well we are left with the tautology $\mathit{true} = \mathit{true}$.

With this technique we prove the functional correctness of the inversion over $\mathbb{Z}_{2^{255}-19}$.

Lemma IV.4 *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.
```

This reflection technique is also used where proofs requires some computing over a small and finite domain of variables to test e.g. for all i such that $0 \leq i < 16$. Using reflection we prove that we can split the for loop in `pack25519` into two parts.

```
for(i=1;i<15;i++) {
  m[i]=t[i]-0xffff-((m[i-1]>>16)&1);
  m[i-1]&=0xffff;
}
```

The first loop computes the subtraction, and the second applies the carries.

```
for(i=1;i<15;i++) {
  m[i]=t[i]-0xffff
}

for(i=1;i<15;i++) {
  m[i]=m[i]-((m[i-1]>>16)&1);
  m[i-1]&=0xffff;
}
```

This loop separation allows simpler proofs. The first loop is seen as the subtraction of $2^{255} - 19$. The resulting number represented in $\mathbb{Z}_{2^{255}-19}$ is invariant with the iteration of the second loop. This result in the proof that `pack25519` reduces modulo $2^{255} - 19$ and returns a number in base 2^8.

```
Lemma Pack25519_mod_25519 :
  forall (l:list Z),
  Zlength l = 16 →
  Forall (λ x ⇒ -2^62 < x < 2^62) l →
  ZofList 8 (Pack25519 l) =
  (ℤ16.lst l) mod (2^255 − 19).
```

By using each functions `Low.M`; `Low.A`; `Low.Sq`; `Low.Zub`; `Unpack25519`; `clamp`; `Pack25519`; `Inv25519`; `car25519`; `montgomery_rec`, we defined a Coq definition `Crypto_Scalarmult` mimicking the exact behavior of X25519 in TweetNaCl.

By proving that each functions `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.
```

This proves that TweetNaCl's X25519 implementation respect RFC 7748.

V. 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 [11] (V-A1). We extend it to support Montgomery curves (V-A2) with homogeneous coordinates (V-A3) and prove the correctness of the ladder (V-A4).

A. Formalization of elliptic Curves

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

Definition V.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 $\mathcal{O}$ called point at infinity:

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

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

Definition V.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 $\mathcal{O}$, "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 $P = Q$.
- $\mathcal{O}$ is the neutral element under this law: if 3 points are collinear, their sum is equal to $\mathcal{O}$.

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 addO (p q : ec): oncurve (add p q).

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

2) Montgomery curves: Speedups can be obtained by switching to homogeneous coordinates and other forms than the Weierstraß form. We consider the Montgomery form [24].

Definition V.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 $\mathcal{O}$, "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 the similar way as in 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 (again with coercion):

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

$$\varphi(\mathcal{O}_M) = \mathcal{O}_E$$
$$\varphi((x, y)) = \left(\frac{x}{b} + \frac{a}{3b}, \frac{y}{b}\right)$$

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.
```

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 V.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 V.6 *Let χ and χ_0:*
– $\chi : M_{a,b}(\mathbb{K}) \to \mathbb{K} \cup \{\infty\}$
such that $\chi(\mathcal{O}) = \infty$ and $\chi((x, y)) = x$.
– $\chi_0 : M_{a,b}(\mathbb{K}) \to \mathbb{K}$
such that $\chi_0(\mathcal{O}) = 0$ and $\chi_0((x, y)) = x$.

Using projective coordinates we prove the formula for differential addition (Lemma V.7).

Lemma V.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 V.8).

Lemma V.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 V.7 and Lemma V.8, we are able to compute efficiently differential additions and point doublings using projective coordinates.

4) Scalar multiplication algorithms: By taking Algorithm 1 and replacing `xDBL` and `xADD` with their respective formula (Lemma V.7 and Lemma V.8), we define a ladder `opt_montgomery` (in which $\mathbb{K}$ has not been fixed yet), similar to the one used in TweetNaCl. This definition is closely related to `montgomery_rec` that was used in the definition of `RFC`, and is easily proved to correspond to it. In Coq this correspondence proof is hidden in the proof of `RFC_Correct` shown above.

We prove its correctness for any point whose x-coordinate is not 0.

```
Lemma opt_montgomery_x :
  forall (n m : nat) (x : K),
  n < 2^m → x ≠ 0 →
  forall (p : mc M), p#x0 = x →
  opt_montgomery n m x = (p *+ n)#x0.
```

We can remark that for an input $x = 0$, the ladder returns 0.

```
Lemma opt_montgomery_0:
  forall (n m : nat), opt_montgomery n m 0 = 0.
```

Also $\mathcal{O}$ is the neutral element of $M_{a,b}(\mathbb{K})$.

```
Lemma p_x0_0_eq_0 : forall (n : nat) (p : mc M),
  p #x0 = 0 → (p *+ n) #x0 = 0%R.
```

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

Theorem V.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.
```

B. Curves, twists and extension fields

To be able to use the above theorem we need to satisfy hypothesis V.5: $a^2 - 4$ is not a square in $\mathbb{K}$:

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

There always exists $x \in \mathbb{F}_{p^2}$ such that $x^2 = a^2 - 4$, preventing the use Theorem V.9 with $\mathbb{K} = \mathbb{F}_{p^2}$.

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

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 V.10.

Lemma V.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 V.11 *We instantiate `opt_montgomery` in two specific ways:*
– $Curve25519\_Fp(n, x)$ for $M_{486662,1}(\mathbb{F}_p)$.
– $Twist25519\_Fp(n, x)$ for $M_{486662,2}(\mathbb{F}_p)$.

With Theorem V.9 we derive the following two lemmas:

Lemma V.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 $\chi_0(P) = x$. Given n and x, $Curve25519\_Fp(n, x) = \chi_0(n \cdot P)$.*

Lemma V.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 $\chi_0(P) = x$. Given n and x, $Twist25519\_Fp(n, x) = \chi_0(n \cdot P)$.*

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 V.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 V.14 we can find a y such that (x, y) is either on the curve or on the quadratic twist:

Lemma V.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).
```

2) Curve25519 over $\mathbb{F}_{p^2}$: The quadratic extension $\mathbb{F}_{p^2}$ is defined as $\mathbb{F}_p[\sqrt{2}]$ by [10]. 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 $\mathbb{F}_p \times \mathbb{F}_p$, in other words, the polynomial 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 V.16 *For all $x \in \mathbb{F}_{p^2} \backslash \{0\}$ and $a, b \in \mathbb{F}_p$ such that $x = (a, b)$,*

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

Similarly as in $\mathbb{F}_p$, we define $0^{-1} = 0$ and prove Lemma V.17.

Lemma V.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:

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

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 V.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 V.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 V.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:

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

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 $M_{486662,2}(\mathbb{F}_p)$, $Curve25519\_Fp$ 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 V.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 $\chi_0(P) = x$, $Curve25519\_Fp(n, x)$ 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 $Curve25519\_Fp$.

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.

VI. 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, because it 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. It is the first step of the compilation. VST does not support the direct verification of `o[i] = a[i] + b[i]`. This required us to rewrite the lines into:

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

  The trust of the proof relies on the trust of a correct translation from the initial version of *TweetNaCl* to *TweetNaClVerifiableC*. `clightgen` comes with `-normalize` flag which factors out function calls and assignments from inside subexpressions. The changes required for C code to make it verifiable are now minimal.
- 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 [6], [21].

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;
```

We believe that the type change of the loop index (`int` instead of `i64`) does not impact the trust of our proof.

A complete proof. We provide a mechanized formal proof of the correctness of the X25519 implementation in TweetNaCl. We first formalized X25519 from RFC 7748 [3] in Coq. Then we proved that TweetNaCl's implementation of X25519 matches our formalization. In a second step we extended the Coq library for elliptic curves [11] by Bartzia and Strub to support Montgomery curves. Using this extension we proved that the X25519 from the RFC and therefore its implementation in TweetNaCl matches the mathematical definitions as given in [10, Sec. 2].

References

[1] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *Progress in Cryptology – LATINCRYPT 2012*, ser. LNCS, A. Hevia and G. Neven, Eds., vol. 7533. Springer, 2012, pp. 159–176, http://cryptojedi.org/papers/#coolnacl. 1

[2] D. J. Bernstein, B. van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers, "TweetNaCl: A crypto library in 100 tweets," in *Progress in Cryptology – LATINCRYPT 2014*, ser. LNCS, D. Aranha and A. Menezes, Eds., vol. 8895. Springer, 2015, pp. 64–83, http://cryptojedi.org/papers/#tweetnacl. 1

[3] A. Langley, M. Hamburg, and S. Turner, "RFC 7748 – elliptic curves for security," https://tools.ietf.org/html/rfc7748. 1, 2, 3, 5, 6, 13

[4] "Things that use Curve25519," 2019, https://ianix.com/pub/curve25519-deployment.html. 1

[5] D. J. Bernstein, "25519 naming," Posting to the CFRG mailing list, Aug 2008, https://www.ietf.org/mail-archive/web/cfrg/current/msg04996.html. 1, 2

[6] "The Coq Proof Assistant – Frequently Asked Questions," https://coq.inria.fr/faq. 1, 4, 13

[7] A. W. Appel, "Verified software toolchain," in *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, ser. LNCS, A. Goodloe and S. Person, Eds., vol. 7226. Springer, 2012, p. 2, https://doi.org/10.1007/978-3-642-28891-3_2. 1

[8] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969, http://doi.acm.org/10.1145/363235.363259. 1

[9] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, vol. 17. IEEE, 2002, pp. 55–74, http://www.cs.cmu.edu/~jcr/seplogic.pdf. 1

[10] D. J. Bernstein, "Curve25519: new Diffie-Hellman speed records," in *Public Key Cryptography – PKC 2006*, ser. LNCS, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., vol. 3958. Springer, 2006, pp. 207–228, http://cr.yp.to/papers.html#curve25519. 1, 12, 13

[11] E.-I. Bartzia and P.-Y. Strub, "A formal library for elliptic curves in the Coq proof assistant," in *Interactive Theorem Proving*, ser. LNCS, G. Klein and R. Gamboa, Eds., vol. 8558. Springer, 2014, pp. 77–92, https://hal.inria.fr/hal-01102288. 1, 9, 13

[12] J. K. Zinzindohoue, E.-I. Bartzia, and K. Bhargavan, "A verified extensible library of elliptic curves," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. IEEE, 2016, pp. 296–309, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7536383. 1

[13] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, C. Hritcu, J. Protzenko, T. Ramananandro, A. Rastogi, N. Swamy, P. Wang, S. Z. Béguelin, and J. K. Zinzindohoué, "Verified low-level programming embedded in F*," in *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP. ACM, 2017, p. 17, http://arxiv.org/abs/1703.00053. 1, 4

[14] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1789–1806, https://eprint.iacr.org/2017/536.pdf. 1

[15] A. Chlipala, "An introduction to programming and proving with dependent types in Coq," *Journal of Formalized Reasoning*, vol. 3(2), pp. 1–93, 2010, http://adam.chlipala.net/papers/CpdtJFR/. 1, 8

[16] J. 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. 1

[17] A. 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. 1

[18] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Systematic synthesis of elliptic curve cryptography implementations," 2016, https://people.csail.mit.edu/jgross/personal-website/papers/2017-fiat-crypto-pldi-draft.pdf. 1

[19] ——, "Simple high-level code for cryptographic arithmetic – with proofs, without compromises," in *2019 IEEE Symposium on Security and Privacy*, 2019, pp. 73–90, https://people.csail.mit.edu/jgross/personal-website/papers/2019-fiat-crypto-ieee-sp.pdf. 1

[20] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel, "Verified correctness and security of OpenSSL HMAC," in *Proceedings of the 24th USENIX Security Symposium*. USENIX Association, 2015, pp. 207–221, https://www.cs.cmu.edu/~kqy/resources/verified-hmac.pdf. 1

[21] A. W. Appel, "Verification of a cryptographic primitive: SHA-256," *ACM Transactions on Programming Languages and Systems*, vol. 37, no. 2, pp. 7:1–7:31, 2015, http://doi.acm.org/10.1145/2701415. 1, 13

[22] Y.-F. Chen, C.-H. Hsu, H.-H. Lin, P. Schwabe, M.-H. Tsai, B.-Y. Wang, B.-Y. Yang, and S.-Y. Yang, "Verifying Curve25519 software," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 299–309, https://cryptojedi.org/papers/#verify25519.pdf. 1

[23] C. Costello and B. Smith, "Montgomery curves and their arithmetic: The case of large characteristic fields," *Journal of Cryptographic Engineering*, vol. 8, no. 3, 2018, https://eprint.iacr.org/2017/212. 2

[24] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Mathematics of Computation*, vol. 48, no. 177, pp. 243–243, 1987, http://links.jstor.org/sici?sici=0025-5718(198701)48:177<243:STPAEC>2.0.CO;2-3. 2, 10

[25] G. Gonthier, "Formal proof—the four-color theorem," *Notices of the AMS*, vol. 55, no. 11, pp. 1382–1393, 2008, https://www.ams.org/notices/200811/tx081101382p.pdf. 4

[26] X. Leroy, "A formally verified compiler back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 363–446, 2009, http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf. 4

[27] W. A. Howard, "The formulæ-as-types notion of construction," in *The Curry-Howard Isomorphism*, P. D. Groote, Ed. Academia, 1995, https://www.cs.cmu.edu/~crary/819-f09/Howard80.pdf. 4

[28] Q. Cao, L. Beringer, S. Gruetter, J. Dodds, and A. W. Appel, "VST-Floyd: A separation logic tool to verify correctness of C programs," *Journal of Automated Reasoning*, vol. 61, no. 1-4, pp. 367–422, 2018. 4

Appendix

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;
  i64 c;
  FOR(i,15) {
    o[(i+1)]+=o[i]>>16;
    o[i]&=0xffff;
  }
  o[0]+=38*(o[15]>>16);
  o[15]&=0xffff;
}

sv sel25519(gf p,gf q,i64 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;
  i64 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);
    m[14]&=0xffff;
    b=1-((m[15]>>16)&1);
    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], aux;
  FOR(i,31) t[i]= 0;
  FOR(i,16) {
    aux = a[i];
    FOR(j,16) t[i+j]+=aux*b[j];
  }
  FOR(i,15) t[i]+=(i64)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];
  i64 r;
  int 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.

```
8c8
< typedef long long i64;
---
@@ We tell VST that long long
@@ are aligned on 8 bytes.
> typedef long long i64
> __attribute__((aligned(8)));
277,281c277,279
<   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;
---
@@ We separate the loop iteration:
@@ 0-14 and 15 as the last.
@@ also simplify the carry propagation code.
>   FOR(i,15) {
>     o[(i+1)]+=o[i]>>16;
>     o[i]&=0xffff;
282a281,282
>   o[0]+=38*(o[15]>>16);
>   o[15]&=0xffff;
285c285
@@ b is a mask of 64 bits.
< sv sel25519(gf p,gf q,int b)
---
> sv sel25519(gf p,gf q,i64 b)
287c287,288
<   i64 t,i,c=~(b-1);
---
@@ For-loop indexes have to be int.
>   int i;
>   i64 t,c=~(b-1);
297,299c298,301
<   int i,j,b;
```

```
<   gf m,t;
<   FOR(i,16) t[i]=n[i];
---
@@ For-loop indexes have to be int.
@@ b is a 64 bit mask.
@@ Initialize m to simplify verification.
>   int i,j;
>   i64 b;
>   gf t,m={0};
>   set25519(t,n);
310d311
<     b=(m[15]>>16)&1;
312c313,314
<     sel25519(t,m,1-b);
---
@@ Computations in arguments
@@ are not allowed in VST.
>     b=1-(m[15]>>16)&1;
>     sel25519(t,m,b);
332c334
<   return d[0]&1;
---
@@ Force the casting.
>   return d[0]&(u8)1;
356,359c358,365
<   i64 i,j,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-loop indexes have to be int.
>   int i,j;
>   i64 t[31], aux;
>   FOR(i,31) t[i]= 0;
>   FOR(i,16) {
@@ introduce an auxiliary variable to
@@ simplify verification (loop invariants).
>     aux = a[i];
>     FOR(j,16) t[i+j]+=aux*b[j];
>   }
>   FOR(i,15) t[i]+=(i64)38*t[i+16];
374c380
<   FOR(a,16) c[a]=i[a];
---
@@ gain 5 bytes.
>   set25519(c,i);
397,398c403,405
<   i64 x[80],r,i;
<   gf a,b,c,d,e,f;
---
@@ x only needs gf.
@@ For-loop indexes have to be int.
>   i64 r;
>   int i;
>   gf x,a,b,c,d,e,f;
433,441c440,442
<   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);
---
@@ simplify
>   inv25519(c,c);
>   M(a,a,c);
>   pack25519(q,a);
```

B. Coq definitions

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 (m: nat) (z: T')
(a: T) (b: T) (c: T) (d: T) (e: T) (f: T) (x: T) :
(* a: x2              *)
(* b: x3              *)
(* c: z2              *)
(* d: z3              *)
(* e: temporary  var  *)
(* f: temporary  var  *)
(* x: x1              *)
(T * T * T * T * T * T) :=
match m with
| 0%nat ⇒ (a,b,c,d,e,f)
| S n ⇒
  let r := Getbit (Z.of_nat n) z in
    (* k_t = (k >> t) & 1                          *)
    (* swap ← k_t                                  *)
  let (a, b) := (Sel25519 r a b, Sel25519 r b a) in
    (* (x2, x3) = cswap(swap, x2, x3)              *)
  let (c, d) := (Sel25519 r c d, Sel25519 r 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 = A²                   *)
  let f := a ² in     (* BB = B²                   *)
  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           *)
  let (a, b) := (Sel25519 r a b, Sel25519 r b a) in
    (* (x2, x3) = cswap(swap, x2, x3)              *)
  let (c, d) := (Sel25519 r c d, Sel25519 r d c) in
    (* (z2, z3) = cswap(swap, z2, z3)              *)
  montgomery_rec n z a b c d e f x
end.

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

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 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
    255  (* iterate 255 times *)
    k    (* clamped n         *)
    1    (* x2                *)
    u    (* x3                *)
    0    (* z2                *)
    1    (* z3                *)
    0    (* dummy             *)
    0    (* dummy             *)
    u    (* x1                *) in
  let a := get_a t in
  let c := get_c t in
  let o := ZPack25519 (Z.mul a (ZInv25519 c))
  in encodeUCoordinate o.
```

3) Equivalence between For Loops:

```
Variable T: Type.
Variable g: nat → T → T.

Fixpoint rec_fn (n:nat) (s:T) :=
  match n with
  | 0 ⇒ s
  | S n ⇒ rec_fn n (g n s)
  end.

Fixpoint rec_fn_rev_acc (n:nat) (m:nat) (s:T) :=
  match n with
  | 0 ⇒ s
  | S n ⇒ g (m - n - 1) (rec_fn_rev_acc n m s)
  end.

Definition rec_fn_rev (n:nat) (s:T) :=
  rec_fn_rev_acc n n s.

Lemma Tail_Head_equiv :
  forall (n:nat) (s:T),
  rec_fn n s = rec_fn_rev n s.
```

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`.
- **High/** contains the theory of Montgomery curves, twists, quadratic extensions and ladder. It also proves the correctness of the ladder over $\mathbb{F}_{2^{255}-19}$.
- **Gen/** defines a generic Montgomery ladder which can be instantiated with different operations. This ladder is the stub for the following implementations.
- **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.