A Coq proof of the correctness of X25519 in TweetNaCl

Peter Schwabe
MPI-SP, Germany &
Radboud University, The Netherlands

Benoît Viguier
Radboud University,
The Netherlands

Timmy Weerwag
Radboud University &
Open University
of the Netherlands

Freek Wiedijk
Radboud University,
The Netherlands

***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, the 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.

Contributions 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 [3, Sec. 2]. This proof is done in three steps:

We first formalize RFC 7748 [6] in Coq [7].

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) [8] to establish a link between C and Coq. VST uses separation logic [9], [10] to show that the semantics of the program satisfies 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 [3, Sec. 2]. We do this by extending the Coq library for elliptic curves [11] 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 arguing about the security of cryptographic constructions on the more abstract level of operations in groups and related problems, like the elliptic-curve discrete-logarithm (ECDLP) or elliptic-curve Diffie-Hellman (ECDH) problem. 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 manually auditing the formalization.

The bigger picture of high-assurance crypto. This work fits into the bigger area of *high-assurance* cryptography, i.e., a line of work that applies techniques and tools from formal methods to obtain computer-verified guarantees for cryptographic software. Traditionally, high-assurance cryptography is concerned with three main properties of cryptographic software:

1) verifying **correctness** of cryptographic software, typically against a high-level specification;
2) verifying **implementation security** and in particular security against timing attacks; and
3) verifying **cryptographic security** notions of primitives and protocols through computer-checked reductions from some assumed-to-be-hard mathematical problem.

A recent addition to this triplet (or rather an extension of implementation security) is security also against attacks expoiting speculative execution; see, e.g., [12]. This paper targets only the first point and attempts to make results immediately usable for verification efforts of cryptographic security.

Verification of implementation security is probably equally important as verification of correctness, but working on the

C language level as we do in this paper is not helpful. To obtain guarantees of security against timing-attack we recommend verifying *compiled* code on LLVM level with, e.g., ct-verif [13], or even better on binary level with, e.g., BINSEC/REL [14].

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 [15]. 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 [16] uses a low-level language called Vale that translates directly to assembly and proves equivalence to a high-level specification in F*. In [17], Zinzindohoué, Bartzia, and Bhargavan describe a verified extensible library of elliptic curves in F* [18]. This served as ground work for the cryptographic library HACL* [19] used in the NSS suite from Mozilla. The approach they use is a combination of proving and synthesizing: 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 [20]. 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 [21], [22], [23]. This software suite is now being used in BoringSSL [24].

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 [25], 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 [3]. As a consequence, this work would not find bugs in any of the routines processing the scalar (like "clamping", see Section II-B), bugs in the the serialization routines or, maybe most importantly, bugs in the high-level specification that the code is verified against.

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 [26], and mbedTLS' implementations of HMAC-DRBG [27] and SHA-256 [28]. 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://doi.org/10.5281/zenodo.4439686 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 [6]. Section IV details the specifications of X25519 in TweetNaCl and some of the proof techniques used to show the correctness with respect to RFC 7748 [6]. Section V describes our extension of the formal library by Bartzia and Strub and the proof of correctness of the X25519 implementation with respect to Bernstein's specifications [5]. Finally in Section VI 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. In a nutshell, we formalize X25519 into a Coq function `RFC`, and we write a specification in separation logic with VST. The first step of CompCert compilation (clightgen) is used to translate the C source code into a DSL in Coq (CLight). By using VST, we step through the translated instructions and verify that the program satisfies the specifications. Additionally we formally define the scalar multiplication over elliptic curves and show that, under the same preconditions as used in the specification, `RFC` computes the desired results.

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 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

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

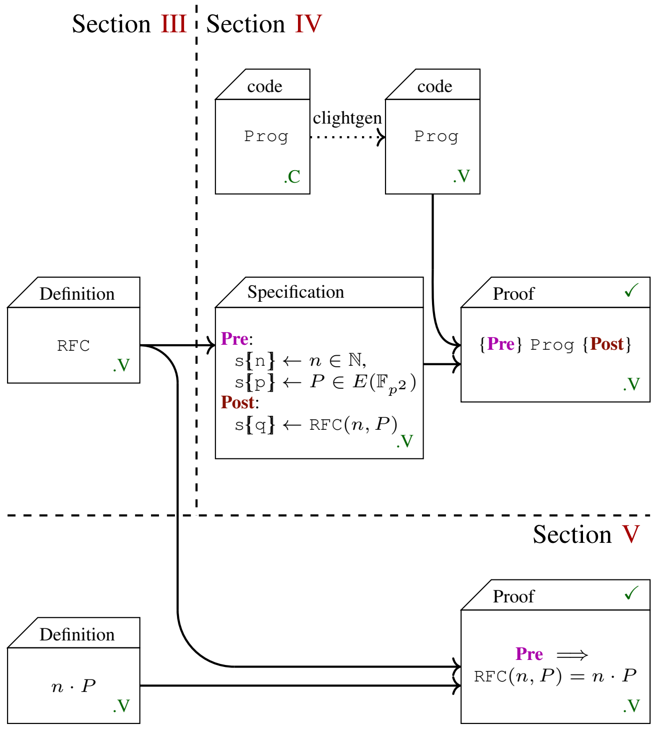


Fig. 1. Structure of the proof.

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 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 $\mathbb{F}_{p^2}$ [29].

Points in $M_{a,b}(\mathbb{K})$ can be equipped with a structure of a commutative 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 [30].

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 the operation:

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

A pseudocode description of the Montgomery ladder using this `xDBL&ADD` routine is given in Algorithm 1. The main loop iterates over the bits of the scalar n. The k^{th} iteration conditionally swaps the arguments P and Q of `xDBL&ADD` depending on the value of the k^{th} bit of n. 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}).

Algorithm 1 Montgomery ladder for scalar mult.

```
Input: x-coordinate $x_P$ of a point $P$, scalar $n$ of bitlength
  upperbound by some integer $m$
Output: x-coordinate $x_Q$ of $Q = n \cdot P$
  $Q = (X_Q : Z_Q) \leftarrow (1 : 0)$
  $R = (X_R : Z_R) \leftarrow (x_P : 1)$
  for $k := m$ down to 1 do
    $(Q, R) \leftarrow$ CSWAP($(Q, R)$, $k^{\text{th}}$ bit of $n$)
    $(Q, R) \leftarrow$ xDBL&ADD($x_P, Q, R$)
    $(Q, R) \leftarrow$ CSWAP($(Q, R)$, $k^{\text{th}}$ 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 256-bit integer n (see Section III). 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 positions 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 [6] standardizes 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}).

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 `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.

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];
```

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 in `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 all these 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; the sequence of arithmetic operations in lines 22 through 39 implement the `xDBL&ADD` routine.

E. Coq, separation logic, and VST

Coq [7] 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 [31] and it is also the system underlying the CompCert formally verified C compiler [32]. Unlike systems like F* [18], 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 [33].

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. Reasoning in separation logic assumes that certain memory regions are non-overlapping. We discuss this limitation further in Section IV-A.

The Verified Software Toolchain (VST) [34] 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 this purpose we use 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. With the help of VST we then use the strongest-postcondition approach to prove the correctness of the triple. This 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 [6].

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) :=
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 II-A) 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 strictly follows Algorithm 1 and prove those descriptions equivalent.

RFC 7748 describes the calculations done in X25519 as follows: *"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."* [6]

Operations used in the Montgomery ladder of `RFC` are performed on integers (see Appendix B-B). 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 which given n and a list l returns its little-endian decoding with radix 2^n.

Similarly, we define the projection from integers to little-endian lists.

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.

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 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 [6] (`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 VST (IV-A). We provide an example of equivalence of operations over different number representations (IV-B).

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 a pure function in Coq formalizing the RFC. 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 (λx ↦ 0 ≤ x < 2^8) p;
       Forall (λx ↦ 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 (λx ↦ 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 as follows:

> `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 represents 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`.
> `Ews{ c121665 }←(lg16)– mVI64 c_121665`
> In the global memory share `Ews`, the address `c121665` points to a list of 16 64-bit integer values corresponding to $a/4 = 121665$.
> `PROP: Forall (λx ↦ 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 as follows:

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

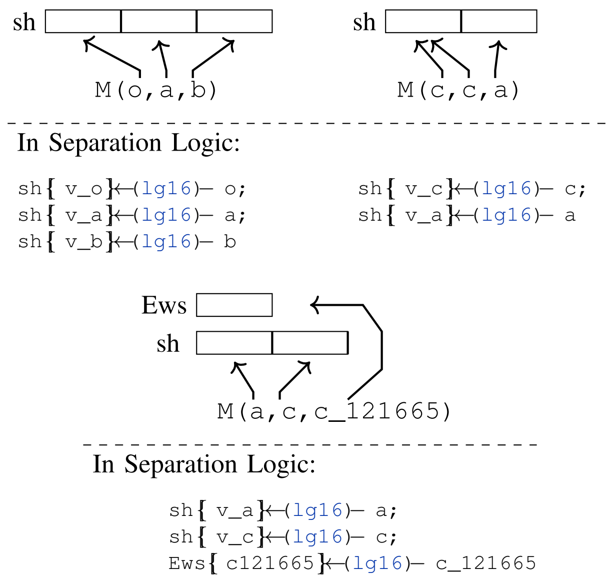
`crypto_scalarmult` computes the same result as `RFC` in Coq provided that inputs are within their respective bounds: arrays of 32 bytes.

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 the 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.



```
sh{ v_o }↤(lg16)— o;        sh{ v_c }↤(lg16)— c;
sh{ v_a }↤(lg16)— a;        sh{ v_a }↤(lg16)— a
sh{ v_b }↤(lg16)— b
```

```
sh{ v_a }↤(lg16)— a;
sh{ v_c }↤(lg16)— c;
Ews{ c121665 }↤(lg16)— c_121665
```

Fig. 2. Aliasing and Separation Logic

- 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.

B. Number representation and C implementation

As described in Section II-C, numbers in `gf` (array of 16 `long long`) are represented in base 2^{16} and we use a direct mapping to represent that array as a list of 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 III and define a notation where n is 16, to fix a 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 $\mathbb{F}_{2^{255}-19}$ 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 in $\mathbb{Z}_{2^{255}-19}$.

This is 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 the absence of overflows.

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 $2^{16} + 38$.

This is seen in Coq as follows:

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

By using each function `Low.M`; `Low.A`; `Low.Sq`; `Low.Zub`; `Unpack25519`; `clamp`; `Pack25519`; `Inv25519`; `car25519`; `montgomery_rec`, we defined `Crypto_Scalarmult` in Coq and with VST proved that 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 (λx ⇒ 0 ≤ x ∧ x < 2 ^ 8) n →
  Forall (λx ⇒ 0 ≤ x ∧ x < 2 ^ 8) p →
  Crypto_Scalarmult n p = RFC n p.
```

Using this equality, we can direct replace `Crypto_Scalarmult` in our specification by `RFC`, proving that TweetNaCl's X25519 implementation respects 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 (λx ⇒ 0 ≤ x ∧ x < 2 ^ 8) n →
  Forall (λx ⇒ 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 projective coordinates (V-A3) and prove the correctness of the ladder (V-A4). We discuss the twist of Curve25519 (V-B1) and explain how we deal with it in the proofs (V-B2).

A. Formalization of elliptic curves

Figure 3 presents the structure of the proof of the ladder's correctness. The white tiles are definitions, the orange ones are hypotheses and the green tiles represent major lemmas and theorems.

We consider the field $\mathbb{K}$ and formalize the Montgomery curves ($M_{a,b}(\mathbb{K})$). Then, by using the equivalent Weierstraß form ($E_{a',b'}(\mathbb{K})$) from the library of Bartzia and Strub, we prove that $M_{a,b}(\mathbb{K})$ forms a commutative group. Under the hypotheses that $a^2 - 4$ is not a square in $\mathbb{K}$, we prove the correctness of the ladder (Theorem 5.9).

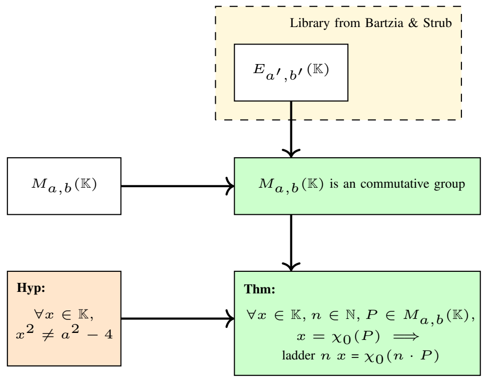

Library from Bartzia & Strub
$E_{a',b'}(\mathbb{K})$
$M_{a,b}(\mathbb{K})$
$M_{a,b}(\mathbb{K})$ is an commutative group
Hyp:
$\forall x \in \mathbb{K}, x^2 \neq a^2 - 4$
Thm:
$\forall x \in \mathbb{K}, n \in \mathbb{N}, P \in M_{a,b}(\mathbb{K}), x = \chi_0(P) \implies$ ladder n $x = \chi_0(n \cdot P)$

Fig. 3. Overview of the proof of Montgomery ladder's correctness.

We now turn our attention to the details of the proof of the ladder's correctness.

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 $\mathcal{O}$ called point at infinity:

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

1) Short Weierstraß curves: For the remainder of this text, we assume that the characteristic of $\mathbb{K}$ is neither 2 nor 3. Then, 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 $\mathcal{O}$, "at infinity". Such a curve does not have any singularities.

In this setting, Bartzia and Strub defined the parametric type `ec` which represents the points on a specific curve. It is parameterized by a `K : ecuFieldType`—the type of fields whose characteristic is neither 2 nor 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 * A³ + 27 * B² ≠ 0}.
Definition oncurve (p : point) :=
  if p is (| x, y |)
    then y² == x³ + A * x + B
    else true.
Inductive ec : Type := EC p of oncurve p.
```

Points on an elliptic curve form an abelian group when equipped with the following structure.

- The negation of a point $P = (x, y)$ is defined by reflection about the x-axis, *i.e.*, $-P = (x, -y)$.
- 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 := s² - 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 projective coordinates and other forms than the Weierstraß form. We consider the Montgomery form [30].

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 $M_{a,b}$ along with an additional formal point $\mathcal{O}$ "at infinity".

Similar to the definition of `ec`, we define the parametric type `mc` which represents the points on a specific Montgomery curve. It is parameterized by a `K : ecuFieldType`—the type of fields whose characteristic is neither 2 nor 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; _: cA² ≠ 4}.
Definition oncurve (p : point K) :=
if p is (| x, y |)
  then cB * y² == x³ + cA * x² + 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 := s² * cB - cA - x₁ - x₂ in
          (| xₛ, - s * (xₛ - x₁) - y₁ |)
    end.
```

And again we prove that the result is on the curve:

```
Lemma addO (p q : mc) : oncurve (add p q).

Definition addmc (p₁ p₂ : mc) : mc :=
  MC p₁ p₂ (addO p₁ p₂)
```

We define a bijection between a Montgomery curve and its short Weierstraß form (Lemma 5.4) and prove that it respects the addition as defined on the respective curves. In this way we get all the group laws for Montgomery curves from the Weierstraß ones.

After having verified the group properties, it follows that the bijection is a group isomorphism.

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 a Weierstraß 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 a group isomorphism between elliptic curves.

3) Projective coordinates: In a projective plane, points are represented by triples $(X : Y : Z)$ excluding $(0 : 0 : 0)$. Scalar multiples of triples are identified with each other, *i.e.*, for all $\lambda \neq 0$, the triples $(X : Y : Z)$ and $(\lambda X : \lambda Y : \lambda Z)$ represent the same point in the projective plane. For $Z \neq 0$, the point $(X : Y : Z)$ corresponds to the point $(X/Z, Y/Z)$ in the affine plane. Likewise, the point (X, Y) in the affine plane corresponds to $(X : Y : 1)$ in 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

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

Setting $Z = 0$ in this equation, we derive $X = 0$. Hence, $(0 : 1 : 0)$ is the unique point on the curve at infinity.

By restricting the parameter a of $M_{a,b}(\mathbb{K})$ such that $a^2 - 4$ is not a square in $\mathbb{K}$ (Hypothesis 5.5), we ensure that $(0, 0)$ is the only point with a y-coordinate of 0.

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

```
Hypothesis mcu_no_square : forall x : K, x² ≠ a² - 4.
```

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

Definition 5.6: Let $\chi : M_{a,b}(\mathbb{K}) \mapsto \mathbb{K} \cup \{\infty\}$ and $\chi_0 : M_{a,b}(\mathbb{K}) \mapsto \mathbb{K}$ such that

$$\chi((x,y)) = x, \qquad \chi(\mathcal{O}) = \infty, \qquad \text{and}$$
$$\chi_0((x,y)) = x, \qquad \chi_0(\mathcal{O}) = 0.$$

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

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

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 efficiently compute differential additions and point doublings using projective coordinates.

4) Scalar multiplication algorithms: By taking Algorithm 1 and replacing `xDBL&ADD` by a combination of the formulas 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 $\chi_0(n \cdot P)$

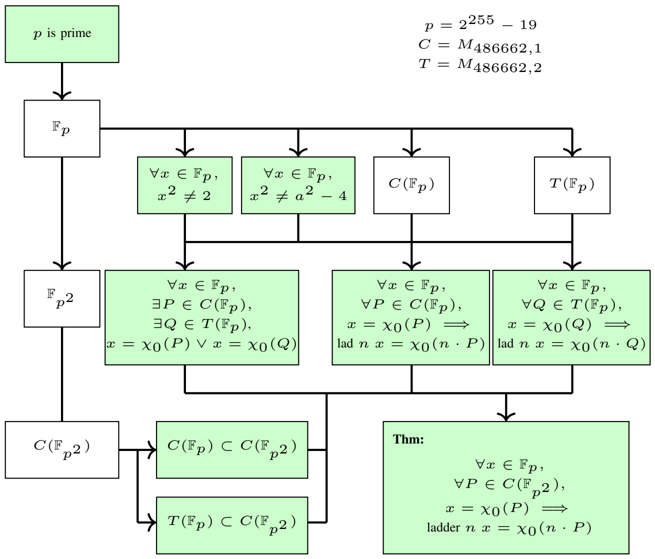
```
Theorem opt_montgomery_ok (n m: nat) (x : K) :
  n < 2^m →
  forall (p : mc M), p#x0 = x →
  opt_montgomery n m x = (p *+ n)#x0.
```

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`.

B. Curves, twists and extension fields

Figure 4 gives a high-level view of the proofs presented here. The white tiles are definitions while green tiles are important lemmas and theorems.

A brief overview of the complete proof is described below. We first set $a = 486662$, $b = 1$, $b' = 2$, $p = 2^{255} - 19$,



p is prime
$p = 2^{255} - 19$
$C = M_{486662,1}$
$T = M_{486662,2}$
$\mathbb{F}_p$
$\forall x \in \mathbb{F}_p,\ x^2 \neq 2$
$\forall x \in \mathbb{F}_p,\ x^2 \neq a^2 - 4$
$C(\mathbb{F}_p)$
$T(\mathbb{F}_p)$
$\mathbb{F}_{p^2}$
$\forall x \in \mathbb{F}_p,\ \exists P \in C(\mathbb{F}_p),\ \exists Q \in T(\mathbb{F}_p),\ x = \chi_0(P) \lor x = \chi_0(Q)$
$\forall x \in \mathbb{F}_p,\ \forall P \in C(\mathbb{F}_p),\ x = \chi_0(P) \implies$ lad n $x = \chi_0(n \cdot P)$
$\forall x \in \mathbb{F}_p,\ \forall Q \in T(\mathbb{F}_p),\ x = \chi_0(Q) \implies$ lad n $x = \chi_0(n \cdot Q)$
$C(\mathbb{F}_{p^2})$
$C(\mathbb{F}_p) \subset C(\mathbb{F}_{p^2})$
$T(\mathbb{F}_p) \subset C(\mathbb{F}_{p^2})$
Thm:
$\forall x \in \mathbb{F}_p,\ \forall P \in C(\mathbb{F}_{p^2}),\ x = \chi_0(P) \implies$ ladder n $x = \chi_0(n \cdot P)$

Fig. 4. Proof dependencies for the correctness of X25519.

with the equations $C = M_{a,b}$, and $T = M_{a,b'}$. We prove the primality of p and define the field $\mathbb{F}_p$. Subsequently, we show that neither 2 nor $a^2 - 2$ is a square in $\mathbb{F}_p$. We consider $\mathbb{F}_{p^2}$ and define $C(\mathbb{F}_p)$, $T(\mathbb{F}_p)$, and $C(\mathbb{F}_{p^2})$. We prove that for all $x \in \mathbb{F}_p$ there exists a point with x-coordinate x either on $C(\mathbb{F}_p)$ or on the quadratic twist $T(\mathbb{F}_p)$. We prove that all points in $M(\mathbb{F}_p)$ and $T(\mathbb{F}_p)$ can be projected in $M(\mathbb{F}_{p^2})$ and derive that computations done in $M(\mathbb{F}_p)$ and $T(\mathbb{F}_p)$ yield the same results if projected to $M(\mathbb{F}_{p^2})$. Using Theorem 5.9 we prove that the ladder is correct for $M(\mathbb{F}_p)$ and $T(\mathbb{F}_p)$; with the previous results, this results in the correctness of the ladder for $M(\mathbb{F}_{p^2})$, in other words the correctness of X25519.

Now that we have a red line for the proof, we turn our attention to the details. Indeed Theorem 5.9 cannot be applied directly to prove that X25519 is doing the computations over $M(\mathbb{F}_{p^2})$. This would infer that $\mathbb{K} = \mathbb{F}_{p^2}$ and we would need to satisfy Hypothesis 5.5:

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

which is not possible as there always exists $x \in \mathbb{F}_{p^2}$ such that $x^2 = a^2 - 4$. Consequently, 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.
```

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 $\mathbb{F}_p$. This allows us to study $M_{486662,1}(\mathbb{F}_p)$ and $M_{486662,2}(\mathbb{F}_p)$, one of its quadratic twists.

Definition 5.11: We instantiate `opt_montgomery` in two specific ways:

– $Curve25519\_Fp(n, x)$ for $M_{486662,1}(\mathbb{F}_p)$.
– $Twist25519\_Fp(n, x)$ for $M_{486662,2}(\mathbb{F}_p)$.

With Theorem 5.9 we derive the following two lemmas:

Lemma 5.12: Let $x \in \mathbb{F}_p$, $n \in \mathbb{N}$, $P \in \mathbb{F}_p \times \mathbb{F}_p$ such that $P \in M_{486662,1}(\mathbb{F}_p)$ and $\chi_0(P) = x$, then

$$Curve25519\_Fp(n, x) = \chi_0(n \cdot P).$$

Lemma 5.13: Let $x \in \mathbb{F}_p$, $n \in \mathbb{N}$, $P \in \mathbb{F}_p \times \mathbb{F}_p$ such that $P \in M_{486662,2}(\mathbb{F}_p)$ and $\chi_0(P) = x$, then

$$Twist25519\_Fp(n, x) = \chi_0(n \cdot P).$$

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.

Because 2 is not a square in $\mathbb{F}_p$, we can partition $\mathbb{F}_p$ as follows:

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

$$y^2 = x \quad \lor \quad 2y^2 = x.$$

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).
```

2) Curve25519 over $\mathbb{F}_{p^2}$: The quadratic extension $\mathbb{F}_{p^2}$ is defined as $\mathbb{F}_p[\sqrt{2}]$ by [3]. 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.
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 confusion.

We define $M_{486662,1}(\mathbb{F}_{p^2})$. With the rewrite rule above, it is straightforward to prove that any point in $M_{486662,1}(\mathbb{F}_p)$ is also in $M_{486662,1}(\mathbb{F}_{p^2})$. Similarly, any point in $M_{486662,2}(\mathbb{F}_p)$ also corresponds to a point in $M_{486662,1}(\mathbb{F}_{p^2})$. 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 any point $P \in \mathbb{F}_p \times \mathbb{F}_p$ on $M_{486662,1}(\mathbb{F}_p)$ (respectively on the quadratic twist $M_{486662,2}(\mathbb{F}_p)$), we have

$$\begin{aligned} P \in M_{486662,1}(\mathbb{F}_p) &\implies \varphi_c(n\cdot P) = n\cdot\varphi_c(P), \quad \text{and}\\ P \in M_{486662,2}(\mathbb{F}_p) &\implies \varphi_t(n\cdot P) = n\cdot\varphi_t(P).\end{aligned}$$

Notice that

$$\begin{aligned} \forall P \in M_{486662,1}(\mathbb{F}_p), &\quad \psi(\chi_0(\varphi_c(P))) = \chi_0(P), \quad \text{and}\\ \forall P \in M_{486662,2}(\mathbb{F}_p), &\quad \psi(\chi_0(\varphi_t(P))) = \chi_0(P).\end{aligned}$$

In summary, for all $n \in \mathbb{N}$, $n < 2^{255}$, for any 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 $\chi_0(n\cdot P)$. We have 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 $\mathbb{F}_p$. Moreover, we have proved that for any point on the curve or the twist, we can compute the scalar multiplication by n and obtain 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 $\chi_0(P) = x$, $Curve25519\_Fp(n,x)$ computes $\chi_0(n\cdot P)$.

We then prove the equivalence of operations between $\mathbb{F}_{2^{255}-19}$ 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 RFC 7748 and in the TweetNaCl implementation, correctly computes scalar 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:
$$\forall x, f(x) = g(x) \implies f = g.$$
- **Verifiable Software Toolchain**. This framework developed at Princeton allows a user to prove that a Clight code matches a 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, the whole code base of these compilers becomes part of the TCB.
- **`clightgen`**. The tool translating from C to Clight, the first step of the CompCert compilation. This compilation step is not covered by the proofs of CompCert and VST requires Clight input. For example, VST does not support the direct verification of `o[i] = a[i] + b[i]`, which `clightgen` translates to
```
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 [28], [7].

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.

Wu and 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 these 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.

Lessons learned. Most efforts in the area of high-assurance crypto are carried out by teams who at the same time work on tools and proofs and often even co-develop the implementations with the proofs. In this effort we set out to verify pre-existing software, written in a not particularly verification-friendly language using a set of tools (VST and Coq) whose development we are not actively involved in.

TweetNaCl comes with a claim of verifiability, but it became clear rather quickly that this claim is only based on the overall simplicity of the library and not supported by code written carefully such that it can efficiently be verified with existing tools. The difference between our verified version of TweetNaCl and the original TweetNaCl in Appendix A gives an idea of some minimal changes we had to apply to work with VST; many more small changes would have made the proof much easier and more elegant. As one example, in `pack25519` the subtraction of p and the carry propagation are done in a single `for` loop; had they been split into two loops, the final result would have been the same but with a much smaller verification effort.

There were many positive lessons to be learned from this verification effort; most importantly that it is possible to prove "legacy" cryptographic software written in C correct without having to co-develop proofs and tools. However, we also learned that it is still necessary to understand to some extent how these tools (in particular VST) work under the hood. VST is a collection of lemmas and proof tactics; the idea is to expose the user only to the tactics and hide the details of the underlying lemmas. At least in the VST versions we worked with, this approach did not quite work and at various stages in the proofs we had to look into the underlying lemmas. This was due to the provided tactics not terminating, for example in the last few steps of `pack25519`'s VST proof. Some struggle with VST also taught us another very pleasant lesson, namely that the VST development team is very responsive and helpful. Various of our issues were sorted out with their help and we hope that some of the experience we brought in also helped improve VST.

Extending our work. The high-level definition (Section V) can easily be ported to any other Montgomery curve 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 IV), the extension of the verification effort to Ed25519 would make direct use of the low-level arithmetic. As the ladder-steps formula is different, this would require a high level verification similar to Theorem 5.9; also, a full correctness verification of Ed25519 signatures would require verifying correctness of SHA-512.

The verification of, *e.g.*, X448 [35], [6] 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 [6] 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 [11] by Bartzia and Strub to support Montgomery curves. Using this extension we proved that the X25519 specification from the RFC matches the mathematical definitions as given in [3, Sec. 2]. Therefore in addition to proving the mathematical correctness of TweetNaCl, we also increase the trust of other works such as [19], [23], which rely on RFC 7748.

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] 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, 2, 10, 12

[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] A. Langley, M. Hamburg, and S. Turner, "RFC 7748 – elliptic curves for security," https://tools.ietf.org/html/rfc7748. 1, 2, 3, 5, 6, 12

[7] "The Coq Proof Assistant – Frequently Asked Questions," https://coq.inria.fr/faq. 1, 4, 11

[8] 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

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

[10] 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

[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, 7, 12

[12] S. Cauligi, C. Disselkoen, K. von Gleissenthall, D. M. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new spectre era," in *Programming Language Design and Implementation (PLDI)*. ACM, 2020, pp. 913–926. 1

[13] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. ACM, 2016, pp. 53–70, https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_almeida.pdf. 2

[14] T. R. Lesly-Ann Daniel, Sébastien Bardin, "BINSEC/REL: Efficient relational symbolicexecution for constant-time at binary-level," in *2020 IEEE Symposium on Security and Privacy*, 2020, pp. 1021–1038, https://people.csail.mit.edu/jgross/personal-website/papers/2019-fiat-crypto-ieee-sp.pdf. 2

[15] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "Sok: Computer-aided cryptography," Cryptology ePrint Archive, Report 2019/1393, 2019, https://eprint.iacr.org/2019/1393. 2

[16] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. Wintersteiger, and S. Zanella-Beguelin, "Evercrypt: A fast, verified, cross-platform cryptographic provider," Cryptology ePrint Archive, Report 2019/757, 2019, https://eprint.iacr.org/2019/757. 2

[17] 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. 2

[18] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy, "Verified low-level programming embedded in F*," in *Proceedings of the ACM on Programming Languages*, ser. ICFP, no. 1. ACM, 2017, p. 17, http://arxiv.org/abs/1703.00053. 2, 4

[19] 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. 2, 12

[20] 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/cpdt/. 2

[21] 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. 2

[22] 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. 2

[23] 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. 2, 12

[24] ——, "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. 2

[25] 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. 2

[26] 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. 2

[27] K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher, and A. W. Appel, "Verified correctness and security of mbedtls hmac-drbg," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, p. 2007–2020, https://doi.org/10.1145/3133956.3133974. 2

[28] 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. 2, 11

[29] 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. 3

[30] 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. 3, 8

[31] 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

[32] 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

[33] 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

[34] 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. 5

[35] M. Hamburg, "Ed448-goldilocks, a new elliptic curve," Cryptology ePrint Archive, Report 2015/625, 2015, https://eprint.iacr.org/2015/625. 12

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);
89      }
90  -   FOR(i,16) {
91  -     x[i+16]=a[i];
92  -     x[i+32]=c[i];
93  -     x[i+48]=b[i];
94  -     x[i+64]=d[i];
95  -   }
96  -   inv25519(x+32,x+32);
97  -   M(x+16,x+16,x+32);
98  -   pack25519(q,x+16);
99  +   inv25519(c,c);
100 +   M(a,a,c);
101 +   pack25519(q,a);
102     return 0;
103   }
```

In the following, 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 VI.
- 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. Thus, `x` is limited to only 16 `i64`, *i.e.*, `gf`.

B. Coq definitions

A. *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) :=
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 :=
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.
```

B. *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    (* x_2               *)
    u    (* x_3               *)
    0    (* z_2               *)
    1    (* z_3               *)
    0    (* dummy              *)
    0    (* dummy              *)
    u    (* x_1               *)
    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 require the use of *Coq 8.8.2* and *Opam 2.0* to manage the dependencies. We are aware that there exist more recent versions of Coq; VST; CompCert etc. However, to avoid dealing with backward compatibility issues, we decided to freeze our dependencies.

Associated files. The repository 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 provides all the required Coq dependencies: ssreflect (1.7), VST (2.0), CompCert (3.2), the elliptic curves library by Bartzia & Strub, and 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 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.