An Introduction to Formal Verification for Elliptic Curve Cryptography via Coq

Robert Chen, Jeff Li, Rachana Madhukara, and Yiming Zheng

Abstract: In our project, we will be connecting the ideas between using formal verification with Coq to create fast and correct implementations. While this area is well studied, it unfortunately is extremely niche and so resources are scattered. In this project, we hope to bring all these ideas together and explain what formal verification is: from definitions, to examples, and then practical use.

Contents

1. Formal Verification

Formal verification is the process of mathematically proving the correctness of a program. Doing so requires (1) modeling the program in some formal language, and (2) using a specified set of rules to prove the program's correctness.

In this section, we provide some historical background and motivation for formal verification, elucidate common misconceptions between verification and validation as well as simulation techniques vs. formal verification, and describe the major techniques for formal verification.

1.1. Motivation

The idea of formal methods of verification has existed since the 17th century, when Leibniz's *characteristica universalis* imagined a universal language governed by logical relations such that all statements could be verified for truth value and thus all disputes could be resolved via logical calculation.

Over the years, mathematicians have used formal axiomatic systems to build mathematical proofs, by expressing statements in some formal language and creating the proof by following a set of given rules. But such proofs were often far too large to compute by hand, and so interest in formalizing proofs was low. However, with the advancement of computers, such complex proofs are now possible, and formalization techniques are gaining popularity. For example, in 1997, the use of computers allowed for a simpler proof of the four color theorem, and in 2005 another proof was provided using Coq.

Yet, formal verification techniques in software have yet to be widely adopted. Given that classic engineering disciplines have well-established standards for designing and deploying products, developing such standards for software engineers seems useful and relevant [12].

In fact, the lack of such standards has led to historically disastrous outcomes, such as the following cases:

- **Therac-25**: Radiation therapy machine malfunction causing at least six accidents from radiation overdose.
- **Ariane 5**: Explosion costing $800 million, caused by a variable overflow.
- **Pentium bug**: Bug in Intel processor resulting in recall of faulty processors and loss of $500 million.

Such incidents could have been avoided with the use of formal verification. Moreover, we focus here on the the cryptography use case, where formal verification seems especially essential to ensure that program behavior does not compromise a user's security.

1.2. Verification vs. Validation

Verification and *validation*, although often used interchangeably, are distinct procedures used together to ensure that a product adheres to its specifications and fulfills its purpose. The PMBOK standard [11], adopted by the IEEE, defines them as follows:

- **Verification**: "The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process."
- **Validation**: "The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers."

Here, we are concerned with verification. However, in practice, it is important to use both validation and verification techniques in tandem. Some challenges of applying formal verification techniques include coming up with a correct specification, and ensuring that the model adheres to how the program actually behaves. Representing a program in some formal language is an abstraction process, and it can be difficult to ensure that we do not miss details that appear in real-world use cases.

1.3. Simulation vs. Verification

Software engineers often use simulation to check the correctness of software. This is done by running tests on a subset of the search space and checking that behavior conforms to what is expected. While this can work well especially with comprehensive test suites, it does not provide a guarantee and can miss crucial parts of the search space. Thus, formal verification has the advantage that it proves that the desired behavior holds everywhere in the search space.

1.4. Formal Verification Techniques

The two main formal verification techniques are theorem proving and model checking.

Theorem proving, or deductive verification, involves utilizing (1) a formal language to express formulas, (2) a set of axioms, or presupposed formulas, and (3) a set of rules used to derive new formulas from existing ones. This is the technique used by Coq.

Model Checking involves representing the program as a mathematical model (e.g., a finite state machine), and exhaustively exploring all states and transitions to verify that a desired property holds.

2. What is Coq?

Coq is a free and open-source proof assistant which allows for the development of programs as well as simultaneous verification. Coq isn't an automatic theorem prover; rather, a user must guide Coq through a proof through a sequence of tactics. These tactics include:

- `simpl`: simplifies an expression
- `reflexivity`: proves an equality statement by checking both sides are identical
- `induction`: proves a goal by induction

For convenience, Coq is also equipped with a library of common theorems and definitions. Coq is one of the most popular proof assistants. It has been used to:

- certify software such as CompCert, a compiler for C
- verify the security of JavaCard
- facilitate specifications of programming languages and the x86 and LLVM instruction sets
- prove theorems that require computationally intensive proofs, like the 4 color theorem

2.1. Coq Syntax for Beginner

Like all programming languages, every variable in Coq has a *type*. Types are defined *inductively*. For example, we can define the natural numbers as follows:

```
Inductive nat : Type :=
    | O
    | S (n : nat).
```

Here, `O` represents 0, and `S` takes a `nat` argument `n` and defines the successor of `n`. Thus, `S O` represents 1, `S (S O)` represents 2, `S (S (S O))` represents 3, and so on. Non-recursive functions can be defined as follows:

```
Definition pred (n : nat) : nat :=
  match n with
  | O => O
  | S n' => n'
  end.
```

Here, the function `pred` computes the predecessor of `n` by matching it against `O` or `S n'`. Recursive functions are defined using the `Fixpoint` syntax.

```
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.
```

The name `Fixpoint` comes from the idea that recursive functions can be thought of as fixed points of an expression. For instance, the factorial function `factorial` can be thought of as the fixed point `f` to the expression: `lambda f (lambda n (n == 0 ? 1 : n * f(n-1)))`.

2.2. *Basic Proofs in Coq*

As Coq is not an automated theorem prover, the user must guide Coq with tactics to prove a theorem. We walk though some examples given in [18]. First, consider the following proof

```
Theorem plus_O_n : forall n : nat, O + n = n.
(* The infix notation 'O + n' is the same as 'plus O n' *)
Proof.
  intros n. reflexivity. Qed.
```

The tactic `intros` moves the `forall` quantifier on `n` into the context. The tactic `reflexivity` performs some automatic simplifications (specifically simplifying `O + n` to `n` by directly applying the definition of the `plus` function) and verifies that both sides of the equality are identical. `Qed` ends the proof.

Below, we describe a few more tactics.

- **Proof by rewriting**: rewriting is a tactic that's useful in tackling proofs that involve statements of implication. For example:

  ```
  Theorem plus_id_example : forall n m:nat,
    n = m ->
    n + n = m + m.
  Proof.
    (* move both quantifiers into the context: *)
    intros n m.
    (* move the hypothesis into the context: *)
    intros H.
    (* rewrite the goal using the hypothesis: *)
    rewrite -> H.
    reflexivity. Qed.
  ```

  Here, we note that for this proof, we only focus on situations where `n` and `m` are equal. As before, `intros n m` moves `n` and `m` into the context. `intros H` moves the hypothesis `n = m` into the context and names it `H`. `rewrite -> H` rewrites the left hand side of `n + n = m + m` using the hypothesis `H`, effectively replacing `n` with `m`, reducing the statement to `m + m = m + m`, which can be easily proven with `reflexivity`, as in the next line.

- **Proof by cases**: consider the following theorem:

  ```
  Theorem plus_1_neq_0: forall n : nat,
    (n + 1) =? 0 = false.
  ```

  Trying to prove this theorem with just simplification and reflexivity will fail, because addition was defined to match the left argument with either `O` or `S n'`. In this case, however, the left argument is `n`, which is arbitrary, and so cannot be matched with `O` or `S n'` to simplify. The natural way to approach this proof is therefore to consider two cases: one where `n = O` and one where `n = S n'`. Then, we prove that the statement `(n + 1) =? 0 = false` holds in both cases. The following proof does this.

  ```
  Proof.
    intros n. destruct n as [| n'] eqn:E.
    - reflexivity.
    - reflexivity. Qed.
  ```

  The `destruct` tactic breaks the goal into two subgoals specified by `[| n']`. The first subgoal corresponds to `n = O`; the second is where `n = S n'`. In both cases, after applying the relevant case assumption, we can simply prove the goal by simplifcation and reflexivity.

Above, we showed `O` is the additive inverse on the left. Proving it is also an additive inverse on the right is a little more complicated:

```
Theorem add_0_r : forall n : nat, plus n O = n.
```

The + operation tries to match variables from left to right and `n` is unknown. For this, we can resort to induction:

```
Proof.
  intros n. induction n as [| n' IHn'].
  - (* n = 0 *) reflexivity.
  - (* n = S n' * ) simpl. rewrite -> IHn'. reflexivity. Qed.
```

Similar to the previous example, `intros n.` moves the `forall` quantifier on `n` into the context. Then we use an `induction` tactic, specifying two subgoals (separated by the vertical bar). The first subgoal is simply replacing `n` with `O`, and thus does not require further specification (hence the left side of the vertical bar is left blank). We solve this subgoal by using `reflexivity`, as shown in the subsequent line (note the `(* n = 0 *)` on this line is merely a comment, specifying which subgoal that line is trying to resolve). The second subgoal is the inductive step. This subgoal is specified by `n' IHn'`, as we are replacing `n` with `n'` (it is implicit that this is the inductive step, and we have the relationship `n = S n'`) and `IHn'`, which is a just a label for this inductive hypothesis. Then, we can resolve the subgoal by (after using `simpl.` for some simplification) using the `rewrite` tactic to rewrite `n` as described above, then applying the inductive hypothesis, i.e. the theorem for `n'`. Then a final call to `reflexivity` finishes off the proof.

3. Elliptic Curve Cryptography Background

Next we give some background on the mathematics in which elliptic curve cryptography is rooted. For the sake of simplicity, we ignore some mathematical rigor so that the intuition is clear.

Definition 3.1. A *planar algebraic curve* $\mathscr{C}$ is the set of points (x,y) satisfying some polynomial equation $f(x,y) = 0$. The *degree* of $\mathscr{C}$ is the degree of the polynomial f.

For the sake of simplicity of this survey and for the sake of discussing elliptic curves in the context of cryptography, we define elliptic curves in the following way.

Definition 3.2. An *elliptic curve* is a plane curve over a finite field which consists of the points defined by the equation

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

along with a point at infinity. This type of equaton is called a *Weierstrass equation*. Additionally, we require that this plane curve is non-singular, which is equivalent to the statement that the discriminant of the curve

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

is nonzero.

Remark 3.3. Note that the points for an elliptic curve are chosen from a fixed finite field whose characteristic is not equal to 2 or 3, since in those cases the curve equation will not be so simple to write down. Fields of characteristic 2 or 3 are just those that have order 2^n or 3^n.

Remark 3.4. The set of points which define an elliptic curve also form an abelian group under the group operation of elliptic curves, with the point at infinity acting as the identity element.

Recall that elliptic curve cryptography relies on the difficulty of reversing point addition on elliptic curves.

Definition 3.5. Point operations on elliptic curves:

1. The point at infinity $\mathscr{O}$ behaves as the identity element; in other words, adding it to any other point results in that other point.

   (a) $\mathscr{O} + \mathscr{O} = \mathscr{O}$

   (b) $\mathscr{O} + P = P$

2. Assume we want to add the two distinct points P, denoted as (x_p, y_p), and Q, denoted as (x_q, y_q). Then assuming our elliptic curve E is defined as $y^2 = x^3 + ax + b$, the sum $P + Q = R$ is defined as:

$$\lambda = \frac{y_q - y_p}{x_q - x_p}$$
$$x_r = \lambda^2 - x_p - x_q$$
$$y_r = \lambda(x_p - x_r) - y_p.$$

Conveniently, there is also a geometric representation of point addition on elliptic curves. To add two points P and Q, first find the line that goes through those points. Then after determining where that line intersects the elliptic curve at a third point, reflect that third point across the x-axis. The result of these operations is the point R, which is the result of adding the points P and Q together.

Additionally note that in order to add a point P to itself, we opt for the line tangent to the curve at the point P.

Definition 3.6. To use elliptic curve cryptography (ECC), all parties must first agree on the all the elements that define the elliptic curve, also known as the *domain parameters*. These are denoted as (p,a,b,G,n,h).

1. The size of the finite field the elliptic curve is defined over is denoted by the variable p. The size is usually prime.
2. The elliptic curve is defined using the constants a and b.
3. We define a cyclic subgroup of the elliptic curve group and the generator of this group is denoted as G. Note that this point is also called the *base point*.
4. The order of the subgroup is n. In other words, the smallest positive number n such that $nG = \mathcal{O}$, where $\mathcal{O}$ is the point at infinity.
5. We have that $h = |E(\mathbb{F}_p)|/n$ and is called the *cofactor*.

First we discuss Elliptic-curve Diffie-Hellman (ECDH). We start with an example of ECDH in practice.

Example. Suppose that Alice wants to establish a shared key with Bob. Then each party first generates a key pair suitable for elliptic curve cryptography. This is done by selecting a random integer d in the interval $[1,n-1]$ which will be the private key and a public key Q, which is the result of adding G, the base point, to itself d times. Denote Alice's key pair as (d_A,Q_A) and denote Bob's key pair as (d_B,Q_B). Then Alice computes $(x_k,y_k) = d_A \cdot Q_B$ and Bob computes $(x_k,y_k) = d_B \cdot Q_A$. Each computation leads to the same value, the shared secret. Note that no party can determine the private key unless they can solve the elliptic curve discrete logarithm problem. Similarly, no external party can compute Alice and Bob's shared secret unless they can solve the elliptic curve Diffie-Hellman problem.

4. Coq for Elliptic Curve Cryptography

Now that we have talked about some of the basics of Coq and also elliptic curve cryptography, we combine these topics. Applying formal verification, specifically using Coq, is quite natural for cryptography. This is because of how ubiquitous cryptography is now. Essentially, every internet-connected application is faced with two decisions: either to accept surveillance/modification of their network communications or to set up cryptographic countermeasures. As expected, many choose the latter.

However, they are then faced with the question: can they trust the cryptographic code? Many cryptographic algorithms have mathematically been shown to be robust. However, the same cannot be said for the implementations of these cryptographic algorithms. As reported by [15], we can check the website ianix to see that over 250 libraries claim that they provide ed25519 digital signature functionality. However, the issue is that none of these libraries have received an external audit. So how do we know which one to trust? Unfortunately, there have also been examples where highly optimized implementations written by widely respected implementers have had bugs, so we cannot just trust those implementations. It is clear that optimized implementations are necessary for fast speeds needed for web applications, for example. Therefore, we would like to eliminate the tradeoff between performance and correctness of cryptographic implementations.

And of course, the key here is to use computer-checkable proofs such as Coq. With the proofs that Coq provides, the goal would be to not have to perform manual audits of the implementation. In other words, if the proof checks out, we consider the code to be trusted. A lot of this process has been pioneered by the MIT Programming Languages and Verification Group which is led by Professor Adam Chlipala. In fact, they have developed Fiat Crypto, the largest Coq library in the world. The way this library works is that it contains numerous Coq proofs necessary for elliptic curve cryptography. The library also contains computer-checkable programs that build C code from the Coq proofs. Then others can use this C code as building blocks for whatever algorithms they are trying to implement.

Now we discuss a Coq implementation from Fiat Crypto. In order to do so, we briefly discuss projective coordinates. Recall our definition of a Weierstrass curve, which we defined using *affine coordinates*. Looking back at our definition of point addition and point doubling, we see that each requires one division step. Therefore, repeatedly adding a point to itself, as required in elliptic curve cryptography, will require hundreds of division operators.

However, scalar division is usually an expensive operation and therefore we would like to limit the number of division operations we perform.

Thus the key idea is to use *projective coordinates*. In essence, using projective coordinates allows us to defer when we actually perform the division operation to the end. This is done by repeatedly multiplying the denominator at each iteration. Then at the end, we convert projective coordinates back to affine coordinates to perform the single division operation.

So the question now is, how do we convert from affine coordinates to projective coordinates (and the other way as well)? Recall that in affine form, an elliptic curve point has two coordinates, (x,y). However in projective space each point now has three coordinates, (X,Y,Z), with the restriction that the third coordinate is never 0. Then we have the following mapping from affine coordinates to projective coordinates:

$$(x,y) \to (xz, yz, z),$$

where z is typically 1 and always nonzero. Similarly we have the following backwards mapping:

$$(X,Y,Z) \to \left(\frac{X}{Z}, \frac{Y}{Z}\right)$$

Now it is clear why we require Z to be nonzero.

Now having these definitions, we can see how this is reflected in the Fiat Crypto library (under `fiat-crypto/src/Curves/Weierstrass/Projective.v`):

```
Definition point : Type := { P : F*F*F | let '(X,Y,Z) := P in Y^2*Z = X^3 +
a*X*Z^2 + b*Z^3 /\ (Z = 0 -> Y <> 0) }.

Program Definition to_affine (P:point) : Wpoint :=
  match proj1_sig P return F*F+_ with
  | (X, Y, Z) =>
    if dec (Z = 0) then inr tt
    else inl (X/Z, Y/Z)
  end.
Next Obligation. Proof. t. fsatz. Qed.

Program Definition of_affine (P:Wpoint) : point :=
  match W.coordinates P return F*F*F with
  | inl (x, y) => (x, y, 1)
  | inr _ => (0, 1, 0)
  end.
Next Obligation. Proof. t; fsatz. Qed.
```

As we can see in the code, first we define a projective point on an elliptic curve. Then we have a definition that tells us how to get from a projective coordinate to an affine coordinate, just as we defined in math above. Although a bit harder to see, the second definition tells us how to get from an affine coordinate to a projective coordinate. From here, we can write proofs, such as ones to add points:

```
Lemma to_affine_add P Q except :
  W.eq
    (to_affine (add P Q except))
    (WeierstrassCurve.W.add (to_affine P) (to_affine Q)).
Proof using Type.
  destruct P as [p ?]; destruct p as [p Z1]; destruct p as [X1 Y1].
  destruct Q as [q ?]; destruct q as [q Z2]; destruct q as [X2 Y2].
  pose proof (not_exceptional_y_of_not_exceptional _ _ except).
  cbv [not_exceptional_y opp to_affine add] in *; t;
    clear except not_exceptional_y;
    specialize_by_assumption.
  all: try abstract fsatz.

  (* zero + P = P   -- cases for x and y *)
  { assert (X1 = 0) by (setoid_subst_rel Feq; Nsatz.nsatz_power 3%nat);
```

```
    t; fsatz. }
    { assert (X1 = 0) by (setoid_subst_rel Feq; Nsatz.nsatz_power 3%nat);
    t; fsatz. }

    (* P  + zero = P   -- cases for x and y *)
    { assert (X2 = 0) by (setoid_subst_rel Feq; Nsatz.nsatz_power 3%nat);
    t; fsatz. }
    { assert (X2 = 0) by (setoid_subst_rel Feq; Nsatz.nsatz_power 3%nat);
    t; fsatz. }
Qed.
```

This is only a small, extremely easy snippet of the expansive Fiat Crypto library. The library now contains proofs for most cryptographic primitives, has implementations of all finite field arithmetic, and has most elliptic curve types already implemented thanks to the MIT Programming Languages and Verification Group.

5. A Need for Point Compression

Here we discuss a potential new proof to include in the Fiat Crypto library. In particular, we introduce the idea of point compression, discuss the ideas behind it, including possible use cases, and detail why it is important.

The idea behind point compression is quite simple: we want to speed up ECC by storing less bits of points needed. Before we delve into discussing different point compression methods, first we discuss a bit of theory that has been used to speed up ECC.

5.1. Edwards Curves

Often Edwards curves are used in ECC since addition, doubling, and tripling can be done faster on Edwards curves than on curves given by a Weierstrass equation. This is because the addition law on Edwards curves does not have exceptions whereas addition on a Weierstrass curve has to be split into cases (for example, addition involving the identity).

Definition 5.1 (Edwards curve)**.** Let a be a non-square element of a field k whose characteristic is not 2. Then the equation

$$x^2 + y^2 = 1 + ax^2y^2$$

defines an elliptic curve with distinguished point $(0,1)$.

In specific, note that every Edwards curve can be written as a Weierstrass curve by writing down a bijection. But as that proof is beyond the scope of this paper, it is omitted here.

Definition 5.2. Let E_d be an Edwards curve given by

$$E_d : x^2 + y^2 = 1 + dx^2y^2.$$

Let $P_0 = (x_0, y_0) \in E_d$, then $-P_0 = (-x_0, y_0)$. Now let $P_1 + P_2 = P_3$ with $P_i = (x_i, y_i) \in E_d$ for $i = 1,2,3$. Then *Edwards curve point addition* is given by

$$(x_1,y_1) + (x_2,y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - x_1x_2}{1 - dx_1x_2y_1y_2} \right) = (x_3,y_3).$$

Remark 5.3. Here note that $-(x_1,y_1) = (-x_1,y_1)$ and $(0,1)$ is the identity element, which is different from Weierstrass curves. Additionally, note that the formula for point addition for Edwards curves is *complete* in that there are no exceptions to the formula. We just need that $dx_1x_2y_1y_2 \neq \pm 1$, which is satisfied when d is not a square in the field k (given in the definition).

Since the addition formula for Edwards curves is complete, it can be implemented very efficiently as a program with no branching. Additionally, complete formulas protect against *side channel* attacks. For example, if we were using separate formulas for addition versus doubling, an adversary may be able to externally distinguish these cases by observing the CPU. Therefore, using complete formulas allows us to subvert side channel attacks since exactly the same sequence of instruction is executed each time.

Remark 5.4. It should be noted here that unlike Weierstrass equations, not every elliptic curve can be written in Edwards form. In particular, every Edwards curve has a rational point of order 4, but not every elliptic curve has such a point.

Moreover, we introduce the *twists* of elliptic curves.

Definition 5.5. A *twisted Edwards curve* $E_{a,d}$ over a field k with characteristic not equal to 2 is a plane curve defined by the equation

$$E_{a,d} = ax^2 + y^2 = 1 + dx^2y^2,$$

where $a, d \in k$ are non zero and distinct. Note the special case $a = 1$ when the curve is untwisted since it reduces to a normal Edwards curve.

The operations on twisted Edwards curves are far more efficient than on most other forms of elliptic curves. This is what makes twisted Edwards curves so suitable for cryptographic applications. For example, the widely used curve Curve25519 is a twisted Edwards curve.

5.2. *Naive Point Compression*

Point compression is a technique that is deployed for a variety of reasons: to save bandwidth/memory, to protect against some attack, for example by eliminating cofactors, or a myriad of other reasons. We will discuss some of these benefits in future sections, but for now we first discuss some naive point compression algorithms. In fact, we start with the most basic example of point compresion being used.

5.2.1. General Purpose Elliptic Curve Point Compression

Recalling the Weierstrass form for an elliptic curve, we see that given a point $P = (x, y)$, we can solve for the y-coordinate in terms of the x-coordinate:

$$y = \pm\sqrt{x^3 + ax + b}.$$

However, there is a slight issue: for any x of a given point P, there are *two* possible y coordinates (heuristically we can see that this makes sense since elliptic curves have symmetry over the x-axis). Therefore, the purpose of point compression is to then encode the two possibilities for y in some sort of compressed representation so that only the x coordinate needs to be transmitted. One possible way to implement this is to transmit an extra bit with the x coordinate that contains the parity of the y coordinate. This would then allow us to pinpoint exactly which y coordinate we want.

5.3. *Benefits of Point Compression*

As alluded to in the previous section, point compression can also protect against certain attacks such as the small subgroup attack. First we discuss the attack and then detail how point compression could be used to subvert this attack using the work done in [16].

5.3.1. Small Subgroup Attack

Next we discuss an attack on ECC security to show why having a cofactor could be a pitfall.

Recall that the Diffie-Hellman (DH) key exchange is based on the perceived hardness of the discrete logarithm problem and specifically, it relies on the difficulty of computing discrete logarithms in a chosen group. The proof that DH is secure relies on the fact that the group we are working over is of prime order. However, the difference here is that most real world implementations, for example ECDH, do not use groups of prime order. In particular, while some elliptic curves do have prime order, one of the most popular choices, Curve25519, does not have prime order. In other words, while mathematical analyses of certain protocols such as DH rely on the fact that we operate over prime order groups, in reality the implementations do not, which potentially opens them up to attacks. In particular, these attacks exploit the additional structure that is present in non-prime order groups. One such example of an attack is the small subgroup attacks. The small subgroup attack on ECDH works in the following way:

1. The attacker, Eve, sends Bob a point Q of small order, pretending that it is her public key (this is in replacement of sending a legitimate curve point eP).

2. Then Bob computes bQ, where b is Bob's private key. However, since Q has small order, there are not many possibilities for the value of bQ.

3. Thus this attack will reveal n, modulo the order of Q. Note that Bob hashing bQ, for example using AES-GCM, does not change anything.

Let us discuss Eve's strategy in more detail. Note that curves typically have order $h\ell$, where ℓ is the order of the base point P and h is the cofactor. The prime ℓ is usually pretty large and h is very small. Then when Eve sends Bob a point Q, there are only a limited number of possibilities for the order of Q: either the order of Q is a divisor of h or is a divisor of h times ℓ. Then assuming that the curve is cyclic (which it is in most cases), Eve's best strategy is to choose Q such that its order is h since the attack will then reveal n mod h. Then Eve can find the value of n by a short exhaustive search. In other words, Eve is able to determine Bob's private key since she was able to force all cryptographic operations to occur within a small subgroup where an exhaustive search is feasible.

Note that one can protect against this type of attack by choosing curves where the cofactor is 1. Additionally, as detailed in [16], there have been ways to defend against this attack, although they are quite "bulky." The usual defense has been to multiply certain points by the cofactor h and end the protocol if the result at any point is the identity. However, this process has its own difficulties in that it is not clear which points to even scale by h.

5.3.2. Eliminating Cofactors Through Point Compression

Note that pitfalls associated with having a cofactor, such a small subgroup attacks, could be avoided by considering an order q subgroup of the elliptic curve. However, there exists many issues with this, such as the fact that checking for membership in the subgroup could be slow and/or complex. This is because we would need to implement an extra scalar multiplication or have to employ more complicated tactics, such as checking the roots of division polynomials or inverting isogenies. However, in [16], Hamburg suggests an alternative way to build a group of prime order q starting with an Edwards or twisted Edwards curve of order $4q$. Therefore, we are clearly eliminating the cofactor.

In Hamburg's paper, he details two ways of elininating the cofactor of 4. The first method only works for twisted Edwards curves while the second method allows for curve flexibility. Here we only focus on the first method. First we understand the algorithm and then start the process of implementing it in Coq. We start with some standard definitions from group theory:

Definition 5.6 (From [4])**.** Let H be a subgroup of a group G. Then if $xHx^{-1} = H$ for every element $x \in G$, then H is said to be a *normal subgroup* of G.

Definition 5.7. For a subgroup H of a group G and an element x of G, define xH to be the set $\{xh : h \in H\}$ and Hx to be the set $\{hx : h \in H\}$. A subset of G of the form xH for some $x \in G$ is said to be a left coset of H and a subset of the form Hx is said to be a right coset of H.

Definition 5.8 (From [6])**.** For a group G and a normal subgroup N of G, the *quotient group* of N in G, written G/N, is the set of cosets of N in G.

Definition 5.9 (From [16])**.** An element g of a group G is a *k-torsion* element if $k \cdot g = 0_G$, where 0_G denotes the identity element of the group G. Additionally, we note that the k-torsion elements of an abelian group form a subgroup denoted as $G[k]$. The k-torsion subgroup of an elliptic curve over a finite field has order dividing k^2. For example, the 2-torsion subgroup has size 1, 2, or 4.

Having these definitions, we can now describe the first proposal put forth in [16]. Calling our twisted Edwards curve as E, we consider the group $E/E[4]$. In other words, two points on E are considered equal if their difference has small order which divides 4. However with considering this group there comes three changes that need to be implemented:

1. We now need a new function that will check the equality of group elements.
2. We need an encoding function which will encode equal points to equal sequences of bits.
3. We need a decoding function that only accepts inputs that are the outputs of the encoding function.

In other words, we can use any points on E, but points which differ by a point of order 4 are considered equal and therefore will be encoded to binary in the same way. Note that this is analogous to the way projective coordinates work: although two values in memory might be considered the same point and encode to the same binary string, the actual X, Y, Z coordinates might be different. This is also the way this method protects against small subgroup attacks: while points of small order might appear internally, they will be considered equivalent to the identity element.

In the paper [16], algorithms are given for the difficult part of this procedure, encoding and decoding. The encoding algorithms are referred to as compression since the output is actually a singular field element of the underlying field rather than the two elements we expect. With these algorithms, the goal is that protocols will now we able gain the simplicity, security, and speed benefits of twisted Edwards curves without falling trap to the difficulties that a cofactor brings. Therefore, we tried to implement these algorithms in Coq, as a bonus extension to this report. (The encoding and decoding algorithms are explicitly given in section A of the appendix in [16].) However due to many factors, this proved to be difficult so we have some partial progress.

In order to start the implementation process of these encoding and decoding algorithms, we first started by looking at how only Edvard Curves algorithms were already implemented in Fiat Crypto. For example, we specifically looked at the file fiat-crypto/src/Curves/Edwards/Pre.v, where Edwards curves are defined and also point addition on Edwards curves is defined:

```
Local Notation onCurve x y := (a*x^2 + y^2 = 1 + d*x^2*y^2) (only parsing).
Lemma onCurve_zero : onCurve 0 1.
Proof using a_nonzero eq_dec field.
fsatz. Qed.

Section Addition.
    Context (x1 y1:F) (P1onCurve: onCurve x1 y1).
    Context (x2 y2:F) (P2onCurve: onCurve x2 y2).
    Lemma denominator_nonzero : (d*x1*x2*y1*y2)^2 <> 1.
    Proof using Type*.
      destruct a_square as [sqrt_a], (dec(sqrt_a*x2+y2 = 0)),
        (dec(sqrt_a*x2-y2 = 0));
        try match goal with [H: ?f (sqrt_a * x2) y2 <> 0 |- _ ]
            => pose proof (d_nonsquare ((f (sqrt_a * x1) (d * x1 * x2
                * y1 * y2 * y1))
                /(f (sqrt_a * x2) y2     *   x1 * y1        )))
            end; Field.fsatz.
    Qed.

    Lemma denominator_nonzero_x : 1 + d*x1*x2*y1*y2 <> 0.
    Proof using Type*. pose proof denominator_nonzero. Field.fsatz. Qed.
    Lemma denominator_nonzero_y : 1 - d*x1*x2*y1*y2 <> 0.
    Proof using Type*. pose proof denominator_nonzero. Field.fsatz. Qed.
    Lemma onCurve_add : onCurve ((x1*y2  +  y1*x2)/(1 + d*x1*x2*y1*y2))
        ((y1*y2- a*x1*x2)/(1 - d*x1*x2*y1*y2)).
    Proof using Type*. pose proof denominator_nonzero. Field.fsatz. Qed.
End Addition.
```

We can see by looking at this code snippet that Edwards curves are defined and point addition is also defined, with proof. Looking carefully, we see that a lot of the lemmas are resolved with 'fsatz.' Fsatz is a tactic defined within Fiat Crypto and solves simple arithmetic operations (see `fiat-crypto/src/Algebra/Field.v` for a definition). Looking at this example, we were then able to define twisted Edwards curves and set up the correct imports by looking at this file.

However we soon hit our first big difficulty. Looking at the encoding algorithm given in [16], we see that an inverse square root is necessary. However, only a very naive implementation of square roots over fields exists in Fiat Crypto and definitely no implementation of inverse square roots exists within Fiat Crypto. Therefore, this is the part where we got stuck, so we have been working with Andres Erbsen of the Programming Languages and Verification Group to try to get this implementation. Unfortunately, both the encoding and decoding algorithms rely on this inverse square root, so we are still working on that. In particular, we are strengthening our Coq knowledge so that we are able to tackle these algorithms in the near future.

6. Acknowledgements

We would like to thank the Programming Languages and Verification Group for giving us access to their Fiat Crypto library so that we could complete this project. In specific, we would like to thank Andres Erbsen for his guidance and also for specifically suggesting that we look into point compression.

References

1. ECC keys. https://cryptobook.nakov.com/asymmetric-key-ciphers/elliptic-curve-cryptography-ecc. Accessed: 2022-04-26.
2. Ecdsa compressed public key point back to uncompressed public key point. https://crypto.stackexchange.com/questions/8914/ecdsa-compressed-public-key-point-back-to-uncompressed-public-key-point. Accessed: 2022-04-26.
3. Elliptic curve point addition in projective coordinates. https://www.nayuki.io/page/elliptic-curve-point-addition-in-projective-coordinates. Accessed: 2022-04-26.

4. Normal subgroup. https://mathworld.wolfram.com/NormalSubgroup.html. Accessed: 2022-04-26.
5. Pohlig hellman and small subgroup attacks. https://crypto.stackexchange.com/questions/81851/pohlig-hellman-and-small-subgroup-attacks. Accessed: 2022-04-26.
6. Quotient group. https://mathworld.wolfram.com/QuotientGroup.html. Accessed: 2022-04-26.
7. Safecurves: choosing safe curves for elliptic-curve cryptography. http://safecurves.cr.yp.to/twist.html. Accessed: 2022-04-26.
8. What is an elliptic curve cofactor? https://crypto.stackexchange.com/questions/56344/what-is-an-elliptic-curve-cofactor. Accessed: 2022-04-26.
9. What is the math behind elliptic curve cryptography? https://hackernoon.com/what-is-the-math-behind-elliptic-curve-cryptography-f61b25253da3. Accessed: 2022-04-26.
10. What is the relationship between p (prime), n (order) and h (cofactor) of an elliptic curve? https://crypto.stackexchange.com/questions/51350/what-is-the-relationship-between-p-prime-n-order-and-h-cofactor-of-an-ell/51352#51352. Accessed: 2022-04-26.
11. Ieee draft guide: Adoption of the project management institute (pmi) standard: A guide to the project management body of knowledge (pmbok guide)-2008 (4th edition). *IEEE P1490/D1, May 2011*, pages 1–505, 2011.
12. A. Chlipala. Formal reasoning about programs. *url: http://adam. chlipala. net/frap*, 2017.
13. C. Cremers and D. Jackson. Prime, order please! revisiting small subgroup and invalid curve attacks on protocols using diffie-hellman. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 78–7815. IEEE, 2019.
14. P. N. Eagle, S. D. Galbraith, and J. Ong. Point compression for koblitz elliptic curves. *Cryptology ePrint Archive*, 2009.
15. A. Erbsen. *Crafting certified elliptic curve cryptography implementations in Coq*. PhD thesis, Massachusetts Institute of Technology, 2017.
16. M. Hamburg. Decaf: Eliminating cofactors through point compression. In *Annual Cryptology Conference*, pages 705–723. Springer, 2015.
17. D. Koshelev. New point compression method for elliptic fq2-curves of j-invariant 0. *Finite Fields and Their Applications*, 69:101774, 2021.
18. B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. Logical foundations. *Software Foundations series*, 1, 2018.
19. A. Sutherland. 18.783 lecture #11: Primality proving, 2022. [Online; accessed 25-April-2022].
20. A. Sutherland. 18.783 lecture #2: Elliptic curves as abelian groups, 2022. [Online; accessed 25-April-2022].
21. A. Sutherland. 18.783 lecture #7: Point counting, 2022. [Online; accessed 25-April-2022].
22. A. Sutherland. 18.783 lecture #8: Schoof's algorithm, 2022. [Online; accessed 25-April-2022].
23. A. Sutherland. 18.783 lecture #9: The discrete logarithm problem, 2022. [Online; accessed 25-April-2022].
24. Wikipedia contributors. Elliptic curve — Wikipedia, the free encyclopedia, 2022. [Online; accessed 28-April-2022].
25. Wikipedia contributors. Elliptic-curve cryptography — Wikipedia, the free encyclopedia, 2022. [Online; accessed 28-April-2022].
26. Wikipedia contributors. Elliptic-curve diffie–hellman — Wikipedia, the free encyclopedia, 2022. [Online; accessed 28-April-2022].
27. Wikipedia contributors. Elliptic curve point multiplication — Wikipedia, the free encyclopedia, 2022. [Online; accessed 28-April-2022].