Formalized Elliptic Curve Cryptography

Joe Hurd, Mike Gordon and Anthony Fox
University of Cambridge

Abstract

Formalizing a mathematical theory is a necessary first step to proving the correctness of programs that refer to that theory in their specification. This paper demonstrates how the mathematical theory of elliptic curves and their application to cryptography can be formalized in higher order logic. This formal development is mechanized using the HOL4 theorem prover, resulting in a collection of higher order logic functions that correctly implement the primitive operations of elliptic curve cryptography.

1 Introduction

Elliptic curve cryptography was first proposed in 1985, and by 2005 had become established enough for the National Security Agency (NSA) to include it as part of the Suite B set of cryptographic algorithms [7], intended to secure a wide range of US government data. The principal technical advantage of elliptic curve cryptography over standard public key cryptography (based on multiplication modulo large primes) is that the keys can be shorter for the same security [1], saving on bandwidth and allowing more efficient cryptographic operations.

One disadvantage of elliptic curve cryptography instead of standard public key cryptography is that elliptic curve operations are more complicated to implement than modular multiplication. Consequently, it is not easy to verify that a particular implementation is correct. The problem is exacerbated both by the use of sophisticated data representations to speed up elliptic curve operations, and low level code designed to achieve maximum performance. This is the motivation for the work described in this paper:

- formalizing elliptic curve cryptography in higher order logic;
- mechanizing the theory using the HOL4 theorem prover [4].

The end product of this work is a machine readable higher order logic theory of elliptic curve cryptography, which can be used as a specification for implementations. If an implementation can be formalized in higher order logic (e.g., using Fox's HOL4 theory of ARM machine code) then formal verification can take place by interactive proof within the HOL4 theorem prover. Since formal verification is the intended use of the HOL4 theory of elliptic curve cryptography, it is of paramount importance that it is a faithful translation of the mathematical theory. Particularly close attention is paid to matching the formal versions of the mathematical definitions as closely as possible to the versions in a standard textbook [1]. Due to space limitations this paper can only contain an abridged description of the formalized theory; a much more complete description can be found in a report available on the web[1].

The remainder of the paper is structured as follows: Sections 2–4 present the formalized definition of elliptic curves, rational points, and elliptic curve arithmetic, in which the main objective is for the resulting formalized theories to be as faithful as possible to the standard mathematical definitions; Section 5 demonstrates how the formalized theory of elliptic curve arithmetic can be executed by proof in the HOL4 theorem prover; Section 6 uses a simple formalization of ElGamal encryption to illustrate how the HOL4 theory of elliptic curves can be used in the verification of a cryptographic implementation; finally, Section 7 summarizes the work completed to date, and looks at promising areas of future research.

1.1 Research context

The work described in this paper is part of a larger project, in collaboration with the University of Utah, whose goal is to provide effective methods for creating implementations of cryptographic algorithms on ARM processors. The Cambridge team is concentrating on two topics:

1. developing high level formal verification infrastructure for elliptic curve cryptography;
2. providing a very high fidelity model of the ARM instruction set (derived from a formally verified model of an ARM processor) and building a verification platform for assembly-level ARM software on top of this.

The Utah team is providing a correct-by-construction compiler from high level mathematical models of cryptographic algorithms to low level ARM implementations. This compiler will be used at Cambridge to create formally verified implementations of elliptic curve cryptography running on the ARM processor.

This paper describes the work done and future plans for the high level verification of elliptic curve cryptography (1 above). There is a report[2] on the high fidelity formalization of the ARM instruction set (2 above), and more details can be found in previous publications [2, 3][3].

Professor Konrad Slind's group (in Utah) are developing a compiler from first order tail-recursive equations to machine code, all implemented within the HOL4 theorem prover. The intention is to provide a verified path from higher order logic definitions to execution on the Cambridge ARM processor model.

They have developed a custom assembly language, based on the ARM instruction set, that has been designed specifically as a target for compiling cryptographic algorithms. This domain specific target has abstracted away from

[1]`http://www.cl.cam.ac.uk/~jeh1004/research/papers/elliptic.html`

[2]`http://www.cl.cam.ac.uk/~mjcg/FoxReport.pdf`

[3]The source code for the formalized ARM model is available with the HOL4 distribution at `http://hol.sf.net`

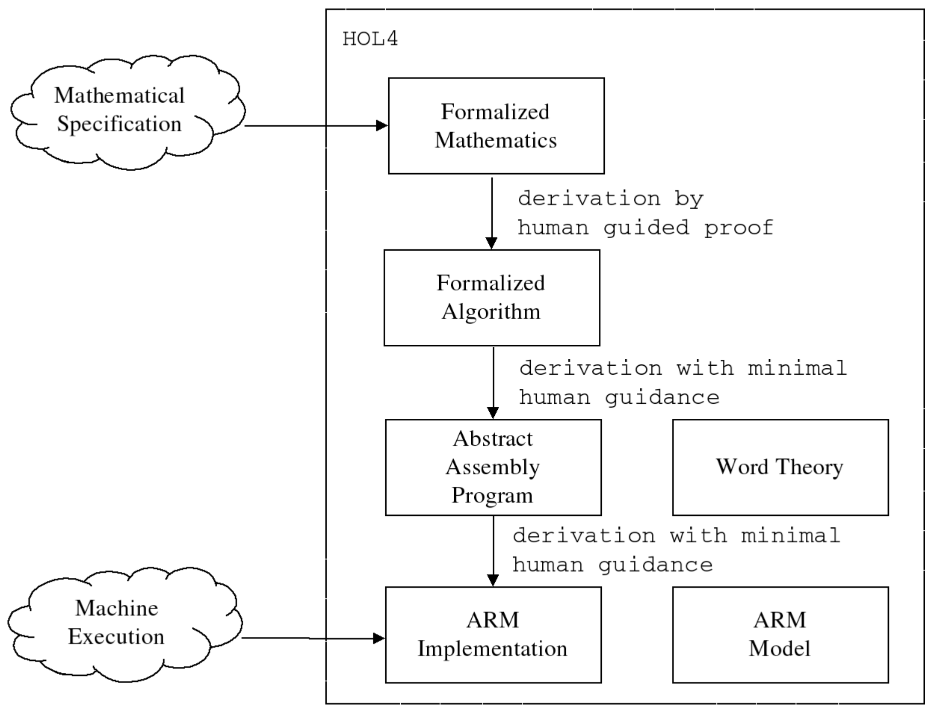
HOL4
Mathematical Specification
Formalized Mathematics
derivation by human guided proof
Formalized Algorithm
derivation with minimal human guidance
Abstract Assembly Program
Word Theory
derivation with minimal human guidance
Machine Execution
ARM Implementation
ARM Model

Figure 1: Formally verified ARM implementations.

ARM features that are not needed to support the algorithms being compiled. These simplifications have enabled a correct-by-construction compiler to be produced, and a number of guaranteed-correct implementations of cryptographic algorithms have already been successfully generated.

Cambridge and Utah plan to develop a formal translator from the Utah domain specific assembler to actual ARM instructions, thereby providing a route for compiling high level specifications to software that can be run directly on ARM processors. This verified path is illustrated in Figure 1.

Juliano Iyoda and Mike Gordon (Cambridge) have been working with the Utah team on developing a compiler from tail-recursive functions to hardware (Altera FPGAs at Cambridge and Xilinx FPGAs at Utah) – bypassing the need for assembly code [5]. The goal is to explore a hybrid approach, i.e., some functions are compiled into hardware and others are implemented in software (run on a processor). The verified ARM model supports co-processor instructions, which provide a standard way of linking hardware and software implementations.

2 Elliptic Curves

The definitions of elliptic curves, rational points and elliptic curve arithmetic presented here all come from the source textbook for the formalization: *Elliptic Curves in Cryptography* by Blake, Seroussi and Smart [1]. The purpose of this paper is to demonstrate that the formalized definitions in the theorem prover faithfully preserve the meaning of the mathematical definitions in the textbook, and so to aid direct comparison the critical definitions are copied verbatim from the textbook.

Firstly, here is the textbook definition of an elliptic curve:

> Let K be a field [and] $\overline{K}$ its algebraic closure [...] An elliptic curve over K will be defined as the set of solutions in the projective plane $\mathbb{P}^2(\overline{K})$ of a homogenous *Weierstrass equation* of the form
>
> $$E : Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3$$
>
> with $a_1, a_2, a_3, a_4, a_6 \in K$.

Note that since every term in the elliptic curve equation has degree 3, one solution (X, Y, Z) of the equation gives an entire line $\alpha(X, Y, Z) = (\alpha X, \alpha Y, \alpha Z)$ of solutions. However, this is only part of the definition, because not every equation of this form is a valid elliptic curve.

> Such a curve should be non-singular [...] Given a curve defined [as above], it is useful to define the following constants for use in later formulae:
>
> $$\begin{array}{l} b_2 = a_1^2 + 4a_2, \quad b_4 = a_1a_3 + 2a_4, \quad b_6 = a_3^2 + 4a_6, \\ b_8 = a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2, \quad [\ldots] \end{array}$$
>
> The *discriminant* of the curve is defined as
>
> $$\Delta = -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6.$$
>
> [...] A curve is then non-singular if and only if $\Delta \neq 0$.

Although this mathematical definition of an elliptic curve may seem complicated, it is straightforward to formalize it using a standard two stage process. In the first stage the syntax of elliptic curves is formalized as a new higher order logic type, using the HOL4 datatype package. The command

```
Hol_datatype
  ‘curve =
   <| field : 'a field; a1 : 'a; a2 : 'a;
      a3 : 'a; a4 : 'a; a6 : 'a |>‘
```

creates a new (polymorphic) record type α curve together with record accessor constants field, a1, ..., a6. The polymorphism is present to allow the theory of elliptic curves to be applied to any formalized elliptic curves, regardless of the higher order logic type of the underlying field elements. In addition, the standard properties that one would expect a record type to satisfy, such as

```
|- ∀f a1 a2 a3 a4 a6.
     <| field = f; a1 = a1; a2 = a2;
        a3 = a3; a4 = a4; a6 = a6 |>.field = f ,
```

are automatically proved as higher order logic theorems. In common with virtually all HOL4 proof tools, the datatype package does not introduce any new axioms. Instead, it reduces the construction of the record type and the proof of all its properties to primitive inferences of higher order logic.[4] Thus, no unsoundness can creep in to a theory from defining new types and constants.

At this point it is often useful to define some syntax in terms of the record accessors in the type definition. In this case the elliptic curve constants b_2, b_4, b_6, b_8 are needed, and are defined in HOL4 like so:

[4]The primitive inferences of higher order logic contain two primitive definition principles: one for new types, which is used to create the record type; and one for new constants, which used to create the record accessors.

```
|- curve_b2 e =                   |- curve_b4 e =
   let f = e.field in                let f = e.field in
   let $& = field_num f in           let $& = field_num f in
   let $+ = field_add f in           let $+ = field_add f in
   let $* = field_mult f in          let $* = field_mult f in
   let $** = field_exp f in          let a1 = e.a1 in
   let a1 = e.a1 in                  let a3 = e.a3 in
   let a2 = e.a2 in                  let a4 = e.a4 in
   a1 ** 2 + & 4 * a2 ;              a1 * a3 + & 2 * a4 ;

|- curve_b8 e =                   |- curve_b6 e =
   let f = e.field in                let f = e.field in
   ...                               ...
   let a3 = e.a3 in                  let a6 = e.a6 in
   let a4 = e.a4 in                  a3 ** 2 + & 4 * a6 ;
   let a6 = e.a6 in
   a1 ** 2 * a6 + & 4 * a2 * a6 -
   a1 * a3 * a4 + a2 * a3 ** 2 - a4 ** 2 .
```

The most noticeable aspect of these definitions is the use of lets to improve the readability of the field operations. Using this shorthand, it is easy to see that the formalized constants are a direct translation of the mathematical definitions. In the definition of the b_6 and b_8 constants the lets for the field operations have been elided, and this abbreviated form will be used from now on to improve the presentation.[5] Next to be formalized is the discriminant of an elliptic curve:

```
|- discriminant e =
   let f = e.field in
   ...
   let b8 = curve_b8 e in
   & 9 * b2 * b4 * b6 - b2 * b2 * b8 -
   & 8 * b4 ** 3 - & 27 * b6 ** 2 .
```

And the final piece of syntax is the definition of non-singularity:

```
|- non_singular e = ~(discriminant e = field_zero e.field)
```

The second stage of formalizing the mathematical definition of elliptic curves is to define a new class Curve consisting of all elements of type α curve that satisfy the elliptic curve axioms.[6] This is a straightforward constant definition, and results in the higher order logic theorem

```
|- Curve =
    e |
     e.field ∈ Field         ∧ e.a1 ∈ e.field.carrier ∧
     e.a2 ∈ e.field.carrier ∧ e.a3 ∈ e.field.carrier ∧
     e.a4 ∈ e.field.carrier ∧ e.a6 ∈ e.field.carrier ∧
     non_singular e  .
```

This completes the formalization of the definition of elliptic curves.

[5]These lets could be eliminated by the use of locales (as used in the Isabelle theorem prover), but locales are not currently implemented in HOL4.

[6]The word class is used here as an aid to the reader; to the theorem prover classes are just higher order logic sets.

3 Rational Points

The set $E(\hat{K})$ of $\hat{K}$-rational points on the elliptic curve E are considered next:

> Let $\hat{K}$ be a field satisfying $K \subseteq \hat{K} \subseteq \overline{K}$. A point (X, Y, Z) on the curve is $\hat{K}$-rational if $(X, Y, Z) = \alpha(\hat{X}, \hat{Y}, \hat{Z})$ for some $\alpha \in \overline{K}$, $(\hat{X}, \hat{Y}, \hat{Z}) \in \hat{K}^3 - \{(0, 0, 0)\}$, i.e., up to projective equivalence, the coordinates of the points are in $\hat{K}$.

Note that if $K \subseteq \hat{K}$ then the coefficients of the elliptic curve equation can be considered to be from $\hat{K}$. Thus the formalized definition of rational points assumes that $\hat{K} = K$:

```
|- curve_points e =
   let f = e.field in
   ...
   let a6 = e.a6 in
    project f [x; y; z] |
      [x; y; z] ∈ nonorigin f ∧
      (y ** 2 * z + a1 * x * y * z + a3 * y * z ** 2 =
       x ** 3 + a2 * x ** 2 * z + a4 * x * z ** 2 + a6 * z ** 3)   .
```

This is a case where the formalized definition significantly deviates from the mathematical definition, in which the rational points over the field $\hat{K}$ are a subset of the rational points over the algebraic closure $\overline{K}$. However, if the rational points were formalized as a subset, then this would result in the field elements of $\hat{K}$ having the same higher order logic type as the field elements of $\overline{K}$. For many fields, especially the finite fields which are of principal interest in cryptography, it would be difficult and unnatural to formalize them with this constraint. Therefore, only the field $\hat{K}$ is mentioned in the definition of rational points: the reference to $\overline{K}$ is completely dropped; and K is not needed if the coefficients are considered to come from $\hat{K}$.

The above definition of rational points uses projective space, but it is usually more convenient to use affine coordinates:

> The curve has exactly one rational point with coordinate Z equal to zero, namely $(0, 1, 0)$. This is the *point at infinity*, which will be denoted by $\mathcal{O}$.
>
> For convenience, we will most often use the *affine* version of the Weierstrass equation, given by
>
> $$E : Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6$$
>
> where $a_i \in K$. The $\hat{K}$-rational points in the affine case are the solutions to E in $\hat{K}^2$, and the point at infinity $\mathcal{O}$. [...] We will switch freely between the projective and affine presentations of the curve, denoting the equation in both cases by E. For $Z \neq 0$, a projective point (X, Y, Z) satisfying [the projective version of E] corresponds to the affine point $(X/Z, Y/Z)$ satisfying [the affine version of E].

For example, taking the underlying field to be $\mathbb{R}$, the curves in Figure 2 depict the solutions in $\mathbb{R}^2$ of different elliptic curve equations in affine coordinates.

The first step to formalizing the affine version of elliptic curves is to define the point at infinity $\mathcal{O}$:

```
|- curve_zero e =
   project e.field
     [field_zero e.field; field_one e.field; field_zero e.field] .
```

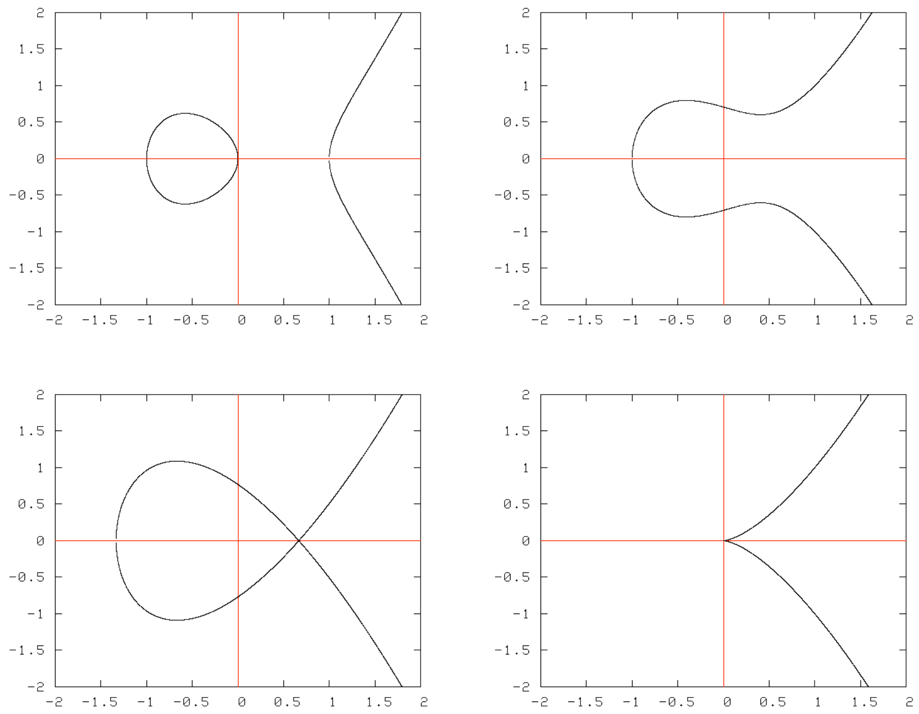

Figure 2: Example elliptic curves, clockwise from top left: $y^2 = x^3 - x$; $y^2 = x^3 - \frac{1}{2}x + \frac{1}{2}$; $y^2 = x^3$; and $y^2 = x^3 - \frac{4}{3}x + \frac{16}{27}$.

From the formalized definition of rational points on the projective version of elliptic curves, it is possible to recover the affine version as a theorem:

```
|- ∀e ∈ Curve. curve_zero e ∈ curve_points e ;

|- ∀e ∈ Curve. ∀x y ∈ (e.field.carrier).
     affine e.field [x; y] ∈ curve_points e =
     let f = e.field in
     ...
     let a6 = e.a6 in
     y ** 2 + a1 * x * y + a3 * y =
     x ** 3 + a2 * x ** 2 + a4 * x + a6 .
```

The mathematical definition of rational points in affine coordinates states explicitly that the every rational points is either $\mathcal{O}$ or is a solution of the elliptic curve equation, and implicitly assumes the 'obvious fact' that the point at infinity $\mathcal{O}$ is cannot be expressed in affine coordinates. Both these facts are proved as theorems in the formalization:

```
|- ∀e ∈ Curve. ∀p ∈ curve_points e.
     (p = curve_zero e) ∨
     ∃x y ∈ (e.field.carrier). p = affine e.field [x; y] ;

|- ∀e ∈ Curve. ∀x y.
     ~(curve_zero e = affine e.field [x; y]) .
```

4 Elliptic Curve Arithmetic

This section describes a formalization of elliptic curve arithmetic, again focusing on the comparison with the mathematical definitions as given in [1]. This uses the formalization of the affine version of the elliptic curve equation, because the textbook presents elliptic curve arithmetic in affine coordinates. Thus before tackling the definitions of elliptic curve arithmetic, a 'case theorem' is proved that supports the definition of functions on elliptic curve points using affine coordinates:

```
|- ∀e ∈ Curve. ∀z f.
     (curve_case e z f (curve_zero e) = z) ∧
     ∀x y. curve_case e z f (affine e.field [x; y]) = f x y .
```

Although this looks like a theorem, it is actually a definition of the constant curve_case by new specification.[7] The best way to see how curve_case is used is by example, and the operations of elliptic curve arithmetic will provide several.

The textbook defines all the operations of elliptic curve arithmetic in one passage, reproduced here in full:

> Let E denote an elliptic curve given by
>
> $$E : Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6$$
>
> and let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ denote points on the curve. Then
>
> $$-P_1 = (x_1, -y_1 - a_1x_1 - a_3) \ .$$

[7] Constants are defined by new specification by proving a 'witness theorem' of the form $\vdash \exists x.\, \phi(x)$, after which a new constant c is created with defining property $\vdash \phi(\mathsf{c})$.

The first and simplest operation to be formalized is negation, which can be expressed using the new curve_case constant:

```
|- curve_neg e =
   let f = e.field in
   ...
   let a3 = e.a3 in
   curve_case e (curve_zero e)
     (λx1 y1.
        let x = x1 in
        let y = ~y1 - a1 * x1 - a3 in
        affine f [x; y]) .
```

How does this work, say to evaluate curve_neg E P? Expanding the definition of curve_neg above (and the lets) will result in a term of the form

$$\mathsf{curve\_case}\ E\ \mathcal{O}\ (\lambda\, x_1, y_1.\ \ldots)\ P\ .$$

Now, using the definition of curve_case, if the argument P is the point at infinity $\mathcal{O}$, then the result will be the second argument of curve_case, which in this case is $\mathcal{O}$. If P is not the point at infinity then it must be a point on the curve that can be expressed as affine K x_1 y_1 (where K is the underlying field of E), and in this case the function in the third argument is called with arguments x_1 and y_1. The end result is the two theorems

$$\vdash\ \mathsf{curve\_neg}\ E\ \mathcal{O} = \mathcal{O}\ ,$$
$$\vdash\ \mathsf{curve\_neg}\ E\ (\mathsf{affine}\ K\ x_1\ y_1) = \mathsf{affine}\ K\ x_1\ (-y_1 - a_1x_1 - a_3)\ .$$

It is not much harder to formalize point addition in the same way, although a careful reading is required to be sure of correctly catching and handling the four special cases $P = \mathcal{O}$, $Q = \mathcal{O}$, $P = Q$ and $P = -Q$.

5 Verified Execution

The definitions of elliptic curve arithmetic were formalized in terms of affine coordinates, faithful to the presentation in the source textbook. However, there is an additional benefit to this choice: the definitions of the elliptic curve operations are sufficiently close to functional programs that they can be executed directly in the HOL4 theorem prover.

A simple test of the elliptic curve operations is provided by formalizing a simple exercise for the reader in [6].

The exercise uses the example curve $Y^2+Y = X^3-X$ over the field GF(751); the HOL4 primality prover and simplifier together can prove that the field satisfies the field laws and the elliptic curve is non-singular:

```
|- GF 751 IN Field ;
|- ec = curve (GF 751) 0 0 1 750 0 ;
|- ec IN Curve .
```

Note the use of 750 in the formalized version instead of -1 in the mathematics: a typical example of representation choices during formalization resulting in a loss of succintness.

The exercise next defines two points which the HOL4 simplifer can prove lie on the curve:

```
|- affine (GF 751) [361; 383] IN curve_points ec ;
|- affine (GF 751) [241; 605] IN curve_points ec .
```

The exercise requires the reader to perform some elliptic curve arithmetic, and check that the results lie on the curve. Again, this is no problem for the HOL4 simplifier:

```
|- curve_neg ec (affine (GF 751) [361; 383]) =
   affine (GF 751) [361; 367] ;
|- affine (GF 751) [361; 367] IN curve_points ec ;

|- curve_add ec (affine (GF 751) [361; 383])
                (affine (GF 751) [241; 605]) =
   affine (GF 751) [680; 469] ;
|- affine (GF 751) [680; 469] IN curve_points ec ;

|- curve_double ec (affine (GF 751) [361; 383]) =
   affine (GF 751) [710; 395] ;
|- affine (GF 751) [710; 395] IN curve_points ec .
```

Together, the verified execution of these six theorems took 72 seconds and 961,068 primitive inference rules to complete: a performance that reflects the highly abstract nature of the definitions involved.

6 Group Based Cryptography

6.1 Elliptic Curve Groups

Many useful cryptographic operations are based on the discrete logarithm problem, which in turn is based on an arbitrary group. The security of the cryptographic operations thus depends on the precise group used, and (thus far) elliptic curve groups have proved highly resistant to attack.

Given an elliptic curve E with underlying field K, then

$$(E(K),\ \mathcal{O},\ -,\ +)$$

is an Abelian group, where $-$ is negation of elliptic curve points and $+$ is addition. It is straightforward to formalize the definition of the elliptic curve group:

```
|- curve_group e =
   <| carrier := curve_points e; id := curve_zero e;
      inv := curve_neg e; mult := curve_add e |> .
```

6.2 ElGamal Encryption

ElGamal encryption demonstrates how the discrete logarithm problem based on a group G can be used as a public key encryption algorithm. The presentation of the algorithm given here is the standard one, and follows [8]. Bob generates an instance $g^x = h$ of the discrete logarithm problem to create a new public and private key. Bob publishes the public key (g, h) while keeping the private key x secret. The following algorithm allows Alice to send a message $m \in G$ to Bob that cannot be read by a third party (this security property is called *confidentiality*).

1. Alice obtains a copy of Bob's public key (g, h).
2. Alice generates a randomly chosen natural number $k \in \{1, \ldots, \sharp G - 1\}$ and computes $a = g^k$ and $b = h^k m$.
3. Alice sends the encrypted message (a, b) to Bob.
4. Bob receives the encrypted message (a, b). To recover the message m he computes
$$ba^{-x} = h^k m g^{-kx} = g^{xk-xk} m = m \ .$$

The first step in formalizing ElGamal encryption is to define the packet that Alice sends to Bob:

```
|- elgamal G g h m k =
   (group_exp G g k, G.mult (group_exp G h k) m) .
```

This follows the algorithm precisely.

The following theorem demonstrates the correctness of ElGamal encryption, i.e., that Bob can decrypt the ElGamal encryption packet to reveal Alice's message (assuming he knows his private key x):

```
|- ∀G ∈ Group. ∀g h m ∈ G.carrier. ∀k x.
     (h = group_exp G g x) ⟹
     (let (a,b) = elgamal G g h m k in
      G.mult (G.inv (group_exp G a x)) b = m)
```

The formalized version diverges slightly from the standard algorithm by having Bob compute $a^{-x}b$ instead of ba^{-x}, but results in a stronger correctness theorem since the underlying group G does not have to be Abelian.

Suppose an implementation of ElGamal encryption over an elliptic curve group has been formalized, and verified to correctly implement the operations of elliptic curve arithmetic. The above correctness theorem of ElGamal encryption is sufficient to guarantee that executing the implementation of encryption followed by the implementation of decryption will always return the original message.

7 Summary

This paper has presented a formalization of elliptic curve theory in higher order logic, mechanized using the HOL4 theorem prover. The principal goal of the project is for the formalization to 'get close to the mathematics', and this paper has demonstrated that in the case of elliptic curve theory it is possible to get very close indeed. In contrast to the more common approach of coding some mathematical operations as programs and then justifying their correctness, the approach described in this paper is to formalize the mathematics directly and then implement proof tools to execute the definitions.

The most important contribution of this project is a 'gold standard' set of elliptic curve operations mechanized in the HOL4 theorem prover. The stage is now set for a verified path from these mathematical definitions of the elliptic curve operations right down to an implementation in ARM machine code. In addition, the tools are now available to support verified execution of the gold standard definitions, which can provide test vectors for prototypes before a full proof of correctness is attempted.

7.1 Future Work

At the moment the formalized definitions of elliptic curve operations are highly abstract, and need sophisticated proof tools to execute them even inefficiently. The compiler being developed for the next step of the verified path to ARM machine code requires its input to be in the form of first order tail-recursive equations, and so the elliptic curve operations must be ported to this language and proved equivalent to the mathematical definitions. A concrete representation for elliptic curve points will need to be chosen: an inevitable trade-off between the complexity of the equivalence proof and the efficiency of the final implementation in ARM machine code.

Now that there is a mechanized 'gold standard' for elliptic curve operations, other implementations of elliptic curve cryptography can be formally verified. For example, one idea is to start with a μCryptol program implementing a cryptographic operation based on an elliptic curve group, and make a shallow embedding of the program in higher order logic. A mechanized proof of the group law for elliptic curves reduces the functional correctness of the embedded μCryptol program to a proof that it correctly implements the elliptic curve operations. The end result is a μCryptol program formally verified to be functionally correct because it implements operations that happen to satisfy a group law, and moreover the group is an elliptic curve group. The 'moreover' part where the 'gold standard' formalization plays a critical role in guaranteeing the security of the μCryptol program.

References

[1] Ian Blake, Gadiel Seroussi, and Nigel Smart. *Elliptic Curves in Cryptography*, volume 265 of *London Mathematical Society Lecture Note Series.* Cambridge University Press, 1999.

[2] Anthony Fox. Formal specification and verification of ARM6. In David Basin and Burkhart Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2003*, volume 2758 of *Lecture Notes in Computer Science*, pages 25–40, Rome, Italy, September 2003. Springer.

[3] Anthony Fox. An algebraic framework for verifying the correctness of hardware with input and output: A formalization in HOL. In J. L. Fiadeiro et al., editor, *CALCO 2005*, volume 3629 of *Lecture Notes in Computer Science*, pages 157–174. Springer, 2005.

[4] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL (A theorem-proving environment for higher order logic).* Cambridge University Press, 1993.

[5] Mike Gordon, Juliano Iyoda, Scott Owens, and Konrad Slind. A proof-producing hardware compiler for a subset of higher order logic. In Joe Hurd, Edward Smith, and Ashish Darbari, editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, number PRG-RR-05-02 in Oxford University Computing Laboratory Research Reports, pages 59–75, August 2005.

[6] Neal Koblitz. *A Course in Number Theory and Cryptography.* Number 114 in Graduate Texts in Mathematics. Springer, 1987.

[7] NSA. Fact sheet NSA Suite B cryptography. Published on the web at `http://www.nsa.gov/ia/industry/crypto_suite_b.cfm`, 2005.

[8] Bruce Schneier. *Applied Cryptography.* Wiley, second edition, 1996.