Formally Verified Elliptic Curve Cryptography For ARM Processors

Joe Hurd

Computing Laboratory
University of Oxford

Department of Computing
Imperial College
Monday 29 January 2007

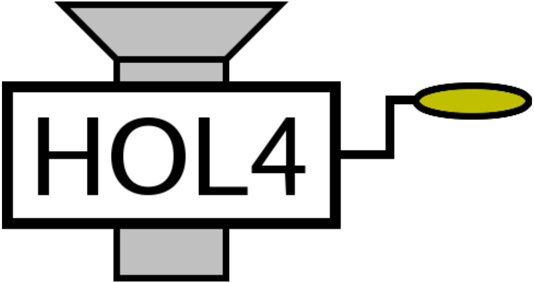
Talk Plan

Verified ARM Implementations

- **Motivation:** How to ensure that low level cryptographic software is both correct and secure?
  - Critical application, so need to go beyond bug finding to assurance of correctness.
- **Project goal:** Create formally verified ARM implementations of elliptic curve cryptographic algorithms.
  - Joint project between Cambridge University and the University of Utah, managed by Mike Gordon.

Illustrating the Verification Flow

- Elliptic curve ElGamal encryption
- Key size = 320 bits



- Verified ARM machine code

The Verification Flow

- A formal specification of elliptic curve operations derived from mathematics (Hurd, Cambridge). This talk!
- A verifying compiler from higher order logic functions to a low level assembly language (Slind & Li, Utah).
- A verifying back-end targeting ARM assembly programs (Tuerk, Cambridge).
- An assertion language for ARM assembly programs (Myreen, Cambridge).
- A very high fidelity model of the ARM instruction set derived from a processor model (Fox, Cambridge).

The whole verification takes place in the HOL4 theorem prover.

The HOL4 Theorem Prover

- Developed by Mike Gordon's Hardware Verification Group in Cambridge, first release was HOL88.
- Latest release called HOL4, developed jointly by Cambridge, Utah and ANU.
- Implements classical Higher Order Logic (a.k.a. simple type theory).
- Sprung from the Edinburgh LCF project, so has a small logical kernel to ensure soundness.

Assumptions and Guarantees

- Assumptions that must be checked by humans:
  - **Specification:** The formalized theory of elliptic curve cryptography is faithful to standard mathematics. This talk!
  - **Model:** The formalized ARM machine code is faithful to the real world execution environment.
- Guarantee provided by formal methods:
  - The resultant block of ARM machine code faithfully implements an elliptic curve cryptographic algorithm.
  - Functional correctness + a security guarantee.
- Of course, there is also an implicit assumption that the HOL4 theorem prover is working correctly.

Elliptic Curve Cryptography

- First proposed in 1985 by Koblitz and Miller.
- Part of the 2005 NSA Suite B set of cryptographic algorithms.
- Certicom the most prominent vendor, but there are many implementations.
- Advantages over standard public key cryptography:
  - Known theoretical attacks much less effective,
  - so requires much shorter keys for the same security,
  - leading to reduced bandwidth and greater efficiency.
- However, there are also disadvantages:
  - Patent uncertainty surrounding many implementation techniques.
  - The algorithms are more complex, so it's harder to implement them correctly.

Elliptic Curve Cryptography: More Secure?

- This table shows equal security key sizes:

standard	elliptic curve
1024 bits	173 bits
4096 bits	313 bits

- But... there has been less theoretical effort made to attack elliptic curve cryptosystems.

Cryptography Based On Groups

- The Discrete Logarithm Problem over a group G tests the difficulty of inverting the power operation:
  - Given $x, y \in G$, find a k such that $x^k = y$.
- The difficulty of this problem depends on the group G.
- For some groups, such as integer addition modulo n, the problem is easy.
- For some groups, such as multiplication modulo a large prime p (a.k.a. standard public key cryptography), the problem is difficult.
- Warning: the number field sieve can solve this in sub-exponential time.

Elliptic Curve Cryptography: A Comparison

Standard Public Key Cryptography

- Needed: a large prime p and a number g.
- Group Operation: multiplication mod p.
- Power operation: $k \mapsto g^k \bmod p$.

Elliptic Curve Cryptography

- Needed: an elliptic curve E and a point p.
- Group Operation: adding points on E.
- Power operation: $k \mapsto p + \cdots + p$ (k times).

ElGamal Encryption (1)

The ElGamal encryption algorithm can use any instance $g^x = h$ of the Discrete Logarithm Problem.

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 uses his private key x to compute

$$ba^{-x} = h^k m g^{-kx} = g^{xk-xk} m = m \ .$$

ElGamal Encryption (2)

Formalize the ElGamal encryption packet that Alice sends to Bob:

Constant Definition

```
elgamal_encrypt G g h m k =
(group_exp G g k, G.mult (group_exp G h k) m)
```

And the ElGamal decryption operation that Bob performs:

Constant Definition

```
elgamal_decrypt G x (a,b) =
G.mult (G.inv (group_exp G a x)) b
```

Note: Encryption follows the textbook algorithm precisely, but decryption computes $a^{-x}b$ instead of ba^{-x}.

ElGamal Encryption (3)

Formally verify that ElGamal encryption followed by decryption reveals the original message, assuming that:

- Alice and Bob use the same group; and
- the private key that Bob uses correctly pairs with the public key that Alice uses.

Theorem

```
⊢ ∀G ∈ Group. ∀g h m ∈ G.carrier. ∀k x.
      (h = group_exp G g x) ⟹
      (elgamal_decrypt G x
         (elgamal_encrypt G g h m k) = m)
```

Note: The tweak that we made to the ElGamal decryption operation results in a stronger theorem, since the group G no longer has to be Abelian.

Formalized Elliptic Curves

- Formalized theory of elliptic curves mechanized in the HOL4 theorem prover.
- Currently about 7500 lines of ML, comprising:
  - 1000 lines of custom proof tools;
  - 6000 lines of definitions and theorems; and
  - 500 lines of example operations.
- Complete up to the theorem that elliptic curve arithmetic forms an Abelian group.
- Formalizing this highly abstract theorem will add evidence that the specification is correct. . .
- . . . but is anyway required for functional correctness of elliptic curve cryptographic operations.

Assurance of the Specification

How can evidence be gathered to check whether the formal specification of elliptic curve cryptography is correct?

1. Comparing the formalized version to a standard mathematics textbook.
2. Deducing properties known to be true of elliptic curves.
3. Deriving checkable calculations for example curves.

Will illustrate all three methods.

Source Material

- The primary way to demonstrate that the specification of elliptic curve cryptography is correct is by comparing it to standard mathematics.
- The definitions of elliptic curves, rational points and elliptic curve arithmetic that we present come from the source textbook for the formalization (*Elliptic Curves in Cryptography*, by Ian Blake, Gadiel Seroussi and Nigel Smart.)
- A guiding design goal of the formalization is that it should be easy for an evaluator to see that the formalized definitions are a faithful translation of the textbook definitions.

Elliptic Curves

- An elliptic curve over the reals is the set of points (x,y) satisfying an equation of the form

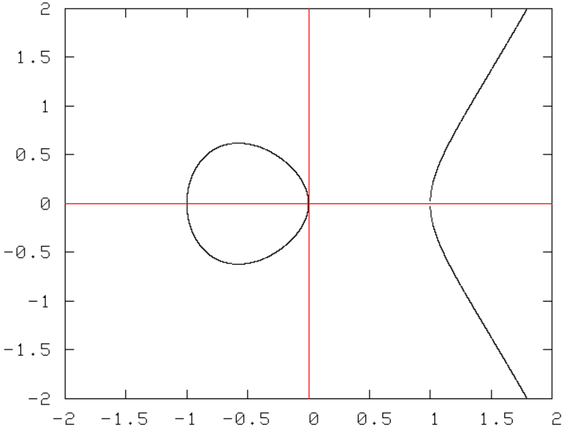
$$E : y^2 = x^3 + ax + b \ .$$

- Despite the name, they don't look like ellipses!
- It's possible to 'add' two points on an elliptic curve to get a third point on the curve.
- Elliptic curves are used in number theory; Wiles proved Fermat's Last Theorem by showing that the elliptic curve

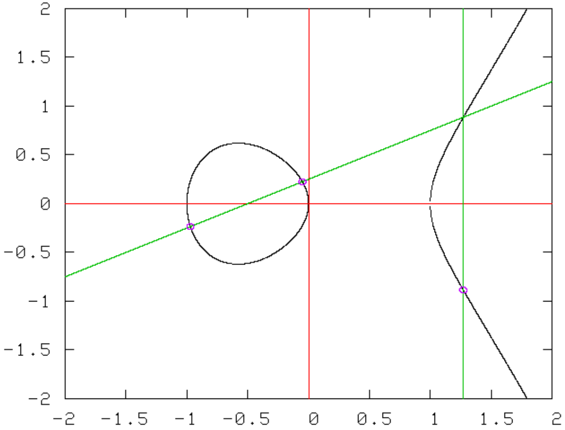
$$y^2 = x(x - a^n)(x + b^n)$$

generated by a counter-example $a^n + b^n = c^n$ cannot exist.

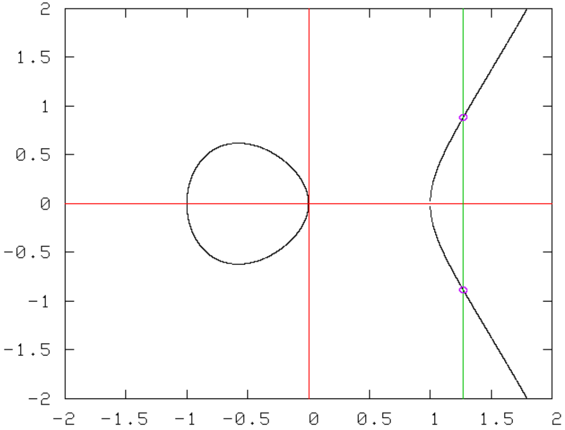
The Elliptic Curve $y^2 = x^3 - x$



The Elliptic Curve $y^2 = x^3 - x$: Addition



The Elliptic Curve $y^2 = x^3 - x$: Negation



Negation of Elliptic Curve Points (1)

Blake, Seroussi and Smart define negation of elliptic curve points using affine coordinates:

> *"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 $P_1 = (x_1, y_1)$ [denote a point] on the curve. Then*
>
> $$-P_1 = (x_1, -y_1 - a_1x_1 - a_3) \ .\text{"}$$

Negation of Elliptic Curve Points (2)

Negation is formalized by cases on the input point, which smoothly handles the special case of $\mathcal{O}$:

Constant Definition

```
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])
```

$$“-P_1 = (x_1, -y_1 - a_1 x_1 - a_3)”$$

Negation of Elliptic Curve Points (3)

The curve_case function makes it possible to define functions on elliptic curve points by separately treating the 'point at infinity' $\mathcal{O}$ and the other points (x, y):

Theorem

```
⊢ ∀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
```

Negation of Elliptic Curve Points (4)

Negation maps points on the curve to points on the curve.

Theorem

$\vdash$ $\forall$e $\in$ `Curve.` $\forall$p $\in$ `curve_points e.`
    `curve_neg e p` $\in$ `curve_points e`

Verified Elliptic Curve Calculations

- It is often desirable to derive calculations that provably follow from the definitions.
  - Can be used to sanity check the formalization,
  - or provide a ‘golden’ test vector.
- A custom proof tool performs these calculations.
  - The tool mainly consists of unfolding definitions in the correct order.
  - The numerous side conditions are proved with predicate subtype style reasoning.

Verified Calculations: Elliptic Curves Points

Use an example elliptic curve from a textbook exercise (Koblitz, 1987).

Example

```
ec = curve (GF 751) 0 0 1 750 0
```

Prove that the equation defines an elliptic curve and that two points given in the exercise lie on the curve.

Example

```
⊢ ec ∈ Curve
⊢ affine (GF 751) [361; 383] ∈ curve_points ec
⊢ affine (GF 751) [241; 605] ∈ curve_points ec
```

Verified Calculations: Elliptic Curve Arithmetic

Perform some elliptic curve arithmetic calculations and test that the results are points on the curve.

Example

```
⊢ curve_neg ec (affine (GF 751) [361; 383]) =
    affine (GF 751) [361; 367]

⊢ affine (GF 751) [361; 367] ∈ 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] ∈ curve_points ec

⊢ curve_double ec (affine (GF 751) [361; 383]) =
    affine (GF 751) [710; 395]

⊢ affine (GF 751) [710; 395] ∈ curve_points ec
```

Doing this revealed a typo in the formalization of point doubling!

The Elliptic Curve Group

The (current) high water mark of the HOL4 formalization of elliptic curves is the ability to define the elliptic curve group.

Constant Definition

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

To prove that this is an Abelian group ‘merely’ requires showing that it satisfies all the group axioms plus commutativity.

I nominate the associativity law as a challenge problem for formalized mathematics.

HOL Source Code

The first step of compilation is to define an equivalent function in a subset of HOL:

- The only supported types are tuples of words (Fox).
- A fixed set of supported word operations.
- Functions must be first order and tail recursive.

Constant Definition

```
add_mod_751 (x : word32, y : word32) =
let z = x + y in
if z < 751 then z else z - 751
```

Testing In C

Tuerk has created a prototype that emits a set of functions in the HOL subset as a C library, for testing purposes.

Code

```
word32 add_mod_751 (word32 x, word32 y) {
   word32 z;
   z = x + y;
   word32 t;
   if (z < 751) {
      t = z;
   } else {
      t = z - 751;
   }
   return t;
}
```

Hoare Triples for Real Machine Code

- Real processors have exceptions, finite memory, and status flags.
- It's still possible to specify machine code programs using Hoare triples.
- But specifying all the things that *don't* change makes them difficult to read and prove.
- Myreen uses the $*$ operator of separation logic to create Hoare triples that obey the frame rule:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

Formally Verified ARM Implementation

Using Slind & Li's compiler with Tuerk's back-end targeting Myreen's Hoare triples for Fox' ARM machine code:

Theorem

```
⊢ ∀rv1 rv0.
    ARM_PROG
      (R 0w rv0 * R 1w rv1 * ~S)
      (MAP assemble
         [ADD AL F 0w 0w (Dp_shift_immediate (LSL 1w) 0w);
          MOV AL F 1w (Dp_immediate 0w 239w);
          ORR AL F 1w 1w (Dp_immediate 12w 2w);
          CMP AL 0w (Dp_shift_immediate (LSL 1w) 0w); B LT 3w;
          MOV AL F 1w (Dp_immediate 0w 239w);
          ORR AL F 1w 1w (Dp_immediate 12w 2w);
          SUB AL F 0w 0w (Dp_shift_immediate (LSL 1w) 0w);
          B AL 16777215w])
      (R 0w (add_mod_751 (rv0,rv1)) * ~R 1w * ~S)
```

Formally Verified Netlist Implementation

- Iyoda has a verifying hardware compiler that accepts the same HOL subset as Slind & Li's compiler.
- It generates a formally verified netlist ready to be synthesized:

Theorem

```
⊢ InfRise clk  ⟹
  (∃v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10.
     DTYPE T (clk,load,v3) ∧ COMB $~ (v3,v2) ∧
     COMB (UNCURRY $∧) (v2 <> load,v1) ∧ COMB $~ (v1,done) ∧
     COMB (UNCURRY $+) (inp1 <> inp2,v8) ∧ CONSTANT 751w v7 ∧
     COMB (UNCURRY $<) (v8 <> v7,v6) ∧
     COMB (UNCURRY $+) (inp1 <> inp2,v5) ∧
     COMB (UNCURRY $+) (inp1 <> inp2,v10) ∧ CONSTANT 751w v9 ∧
     COMB (UNCURRY $-) (v10 <> v9,v4) ∧
     COMB (λ(sw,in1,in2). (if sw then in1 else in2))
       (v6 <> v5 <> v4,v0) ∧ ∃v. DTYPE v (clk,v0,out)) ==>
  DEV add_mod_751
    (load at clk,(inp1 <> inp2) at clk,done at clk,out at clk)
```

Results So Far

- So far only initial results—both verifying compilers need extending to handle full elliptic curve cryptography examples.
- The ARM compiler can compile simple 32 bit field operations.
- The hardware compiler can compile field operations with any word length, but with 320 bit numbers the synthesis tool runs out of FPGA gates.

Summary

- This talk has given an overview of an ongoing project to generate formally verified ARM machine code.
- There's much work still to be done completing and scaling up all levels of the project, and more cryptographic algorithms to be included (ECDSA).
- The hardware compiler provides another verified implementation platform, and it would be interesting to extend the C output to generate reference implementations in other languages (μCryptol).