Formally Verified Elliptic Curve Cryptography

Joe Hurd

Computing Laboratory
University of Oxford

Cambridge University
Tuesday 13 March 2007

Talk Plan

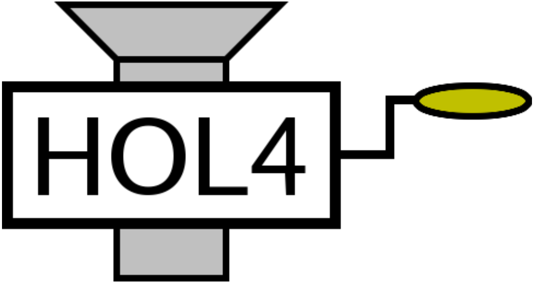
1. Introduction
2. Formalized Elliptic Curves
3. (Towards) Verified Implementations
4. Summary

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.

Illustrating the Verification Flow

- Elliptic curve ElGamal encryption
- Key size = 320 bits



- Verified ARM machine code

The Verification Flow

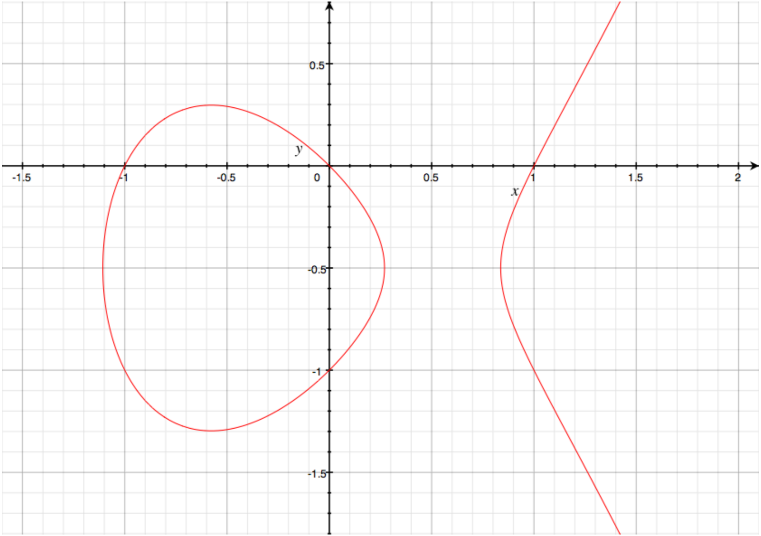
- A formal specification of elliptic curve operations derived from mathematics (Hurd, Cambridge). This talk!
- A verifying compiler from higher order logic functions to a low level assembly language (Slind & Li, Utah).
- A verifying back-end targeting ARM assembly programs (Tuerk, Cambridge).
- An assertion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

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.

$Y^2 + Y = X^3 - X$



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.

The elliptic curve specification can be checked using 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.

Negation of Elliptic Curve Points

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*
>
> $$-P_1 = (x_1, -y_1 - a_1x_1 - a_3)\ ."$$

Checking the Spec 1: Comparison with the Textbook

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])
```

$$``-P_1 = (x_1, -y_1 - a_1 x_1 - a_3)"$$

Checking the Spec 2: Deducing Known Properties

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

Theorem

$\vdash$ ∀e ∈ `Curve`. ∀p ∈ `curve_points e`.
    `curve_neg e p` ∈ `curve_points e`

Checking the Spec 3: Example Calculations

Example elliptic curve from a textbook exercise (Koblitz 1987).

Example

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

⊢ ec ∈ Curve

⊢ affine (GF 751) [361; 383] ∈ curve_points ec

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

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

The Elliptic Curve Group

Still need to complete the proof that elliptic curve arithmetic forms an Abelian group:

Constant Definition

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

Complete apart from the challenge problem of associativity.

There's a light at the end of the tunnel: Recently Thèry has formalized a proof of associativity in the Coq theorem prover.

Executable Higher Order Logic

The first step of the verification flow is an elliptic curve cryptography library in the following executable subset of higher order logic:

- The only supported types are tuples of (Fox) `word32s`.
- A fixed set of supported word operations.
- Functions must be first order and tail recursive.

Elliptic Curve Cryptography Example 0

To test the machinery, we have defined a tiny elliptic curve cryptography library implementing ElGamal encryption using the example curve

$$Y^2 + Y = X^3 - X$$

over the field GF(751).

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 a status report of the effort to generate formally verified elliptic curve cryptography in ARM machine code.
- There's much work still to be done to complete the effort, 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 (e.g., Cryptol).