Formally Verified ARM Code

Joe Hurd

Computing Laboratory
University of Oxford

High Confidence Software and Systems
Thursday 10 May 2007

Joint work with Anthony Fox (Cambridge),
Mike Gordon (Cambridge) and Konrad Slind (Utah)

Talk Plan

Verified ARM Implementations

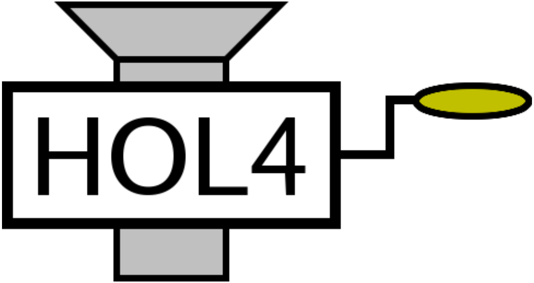
- **Motivation:** How to ensure that low level cryptographic software is both correct and secure?
  - Critical application, so need to go beyond bug finding to assurance of correctness.
- **Project goal:** Create formally verified ARM implementations of elliptic curve cryptographic algorithms.
  - This talk will recap project material presented at HCSS last year, followed by work done this year.

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.

Illustrating the Verification Flow

- Elliptic curve ElGamal encryption
- Key size = 320 bits



- Verified ARM machine code

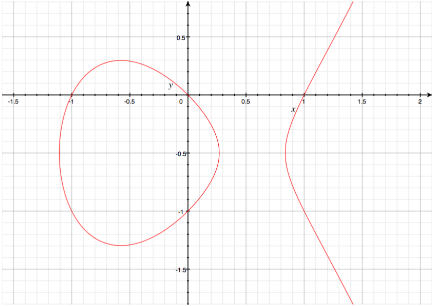
Assumptions and Guarantees

- Assumptions that must be checked by humans:
  - **Specification:** The formalized theory of elliptic curve cryptography is faithful to standard mathematics.
  - **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 proof assistant is working correctly.

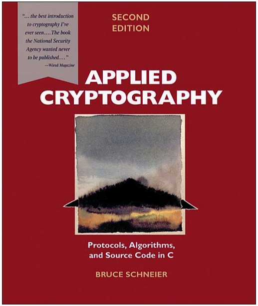
The HOL4 Proof Assistant

- 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.
- Models written in a functional language.
- Reasoning in Higher Order Logic.

Specification of Elliptic Curve Cryptography



+



Assurance of the Specification

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

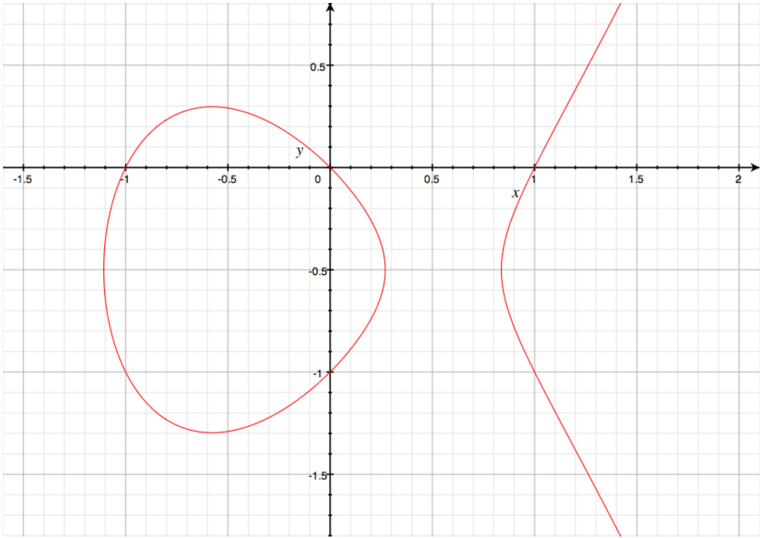
1. Comparing the formalized version to a standard mathematics textbook.
2. Deducing properties known to be true of elliptic curves.
3. Deriving checkable calculations for example curves.

The elliptic curve cryptography specification can be checked using all three methods.

Formalized Elliptic Curve Cryptography

- Formalized theory of elliptic curve crypography mechanized in the HOL4 proof assistant.
- The definitions of elliptic curves, rational points and elliptic curve arithmetic come from the textbook *Elliptic Curves in Cryptography*, by Ian Blake, Gadiel Seroussi and Nigel Smart.
- Designed to be easy for an evaluator to see that the formalized definitions are a faithful translation of the textbook definitions.

Example Elliptic Curve: $Y^2 + Y = X^3 - X$



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*
>
> $$E : Y^2 + a_1 XY + a_3 Y = X^3 + a_2 X^2 + a_4 X + a_6$$
>
> *and let $P_1 = (x_1, y_1)$ [denote a point] on the curve. Then*
>
> $$-P_1 = (x_1, -y_1 - a_1 x_1 - a_3) \ ."$$

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])
```

$$\text{“}-P_1 = (x_1, -y_1 - a_1 x_1 - a_3)\text{”}$$

Checking the Spec 2: Deducing Known Properties

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

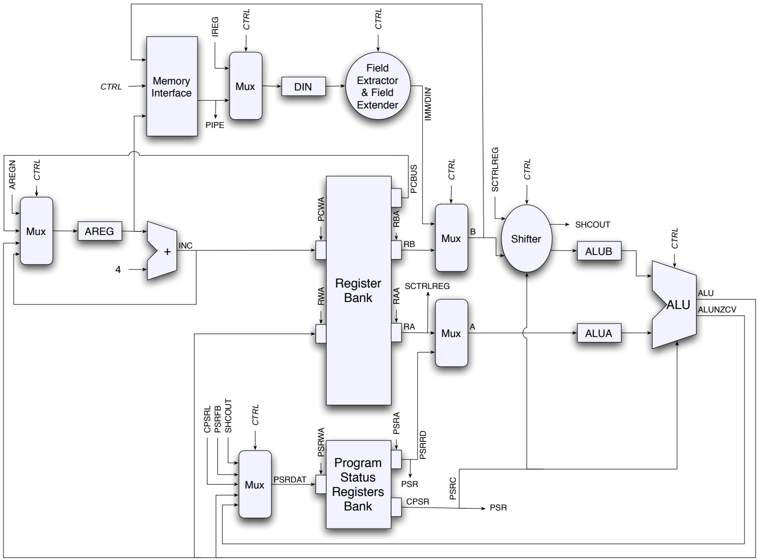
Checking the Spec 3: Example Calculations

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
```

ARM Datapath



ARM6 Verification Project

- ARMv3 Instruction Set Architecture (ISA) modelled in functional subset of higher order logic.
- ARM6 microarchitecture also modelled in higher order logic.
- Models proved equivalent using the HOL4 proof assistant.
  - Took a year, but would be much quicker now.
  - Infrastructure developed (e.g., for reasoning about words).
- CPU and memory separately modelled.
  - Simple memory model currently used for software execution.
  - More realistic models possible (future research).

Formalized ARM Instruction Set

- Started from formal model of ARMv3 ISA verified against a model of the ARM6 microarchitecture.
- Upgraded ISA model to ARMv4 (ARMv5, Thumb planned).
  - Can formally reason about a wider range of ARM programs.
  - Caution: Upgrades are not verified against a processor model.
  - An ML processor simulator can be automatically extracted from the ISA model; executes 10,000 instructions per second.
- Central problem: How to reason about real ARM programs?
  - Exceptions, finite memory, and status flags.
  - Must specify the processor state changed by an instruction.
  - Worse: Must specify the state *not changed* by an instruction.

Specifications for ARM Code

- Myreen uses the $*$ operator of separation logic to create Hoare triples for ARM code that obey the frame rule:

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

  - This avoids having to specify all the processor state that the code C *doesn't* change.

- Specifications of the ARM move and store instructions:

$$\begin{array}{cc} \{\mathrm{R}\ a\ x * \mathrm{R}\ b\ \_\} & \{\mathrm{R}\ a\ x * \mathrm{R}\ b\ (\mathrm{addr}\ y) * \mathrm{M}\ y\ \_\} \\ \mathrm{MOV}\ b,\ a & \mathrm{STR}\ a,\ [b] \\ \{\mathrm{R}\ a\ x * \mathrm{R}\ b\ x\} & \{\mathrm{R}\ a\ x * \mathrm{R}\ b\ (\mathrm{addr}\ y) * \mathrm{M}\ y\ x\} \end{array}$$

  - Instruction specifications are derived from the processor model.

Example: Deriving Specifications

Show that the decrement-and-store instruction

$$\begin{gathered}\{\mathrm{R}\ a\ x * \mathrm{R}\ b\ (\mathrm{addr}\ y) * \mathrm{M}\ (y-1)\ \_\} \\ \mathrm{STR}\ a,\ [b, \# -4]! \\ \{\mathrm{R}\ a\ x * \mathrm{R}\ b\ (\mathrm{addr}\ (y-1)) * \mathrm{M}\ (y-1)\ x\}\end{gathered}$$

can be used as a stack push, where

$$\begin{aligned}\text{stack}\ y\ [x_0, \ldots, x_{m-1}]\ n \ \equiv\ & \mathrm{R}\ 13\ (\mathit{addr}\ y) * \\ & \underbrace{\mathrm{M}\ (y+m-1)\ x_{m-1} * \cdots * \mathrm{M}\ y\ x_0}_{[x_{m-1}, \ldots, x_0]} * \underbrace{\mathrm{M}\ (y-1)\ \_ * \cdots * \mathrm{M}\ (y-n)\ \_}_{\text{empty slots}}\end{aligned}$$

Example: Deriving Specifications

Show that the decrement-and-store instruction

$$\begin{array}{c}\{\mathrm{R}\ a\ x * \mathrm{R}\ b\ (\mathrm{addr}\ y) * \mathrm{M}\ (y-1)\ \_ * P\} \\ \mathrm{STR}\ a,\ [b, \# -4]! \\ \{\mathrm{R}\ a\ x * \mathrm{R}\ b\ (\mathrm{addr}\ (y-1)) * \mathrm{M}\ (y-1)\ x * P\}\end{array}$$

can be used as a stack push, where

$$\begin{array}{l}\text{stack}\ y\ [x_0, \ldots, x_{m-1}]\ n \ \equiv\ \mathrm{R}\ 13\ (addr\ y) * \\ \quad \underbrace{\mathrm{M}\ (y+m-1)\ x_{m-1} * \cdots * \mathrm{M}\ y\ x_0}_{[x_{m-1}, \ldots, x_0]} * \underbrace{\mathrm{M}\ (y-1)\ \_ * \cdots * \mathrm{M}\ (y-n)\ \_}_{\text{empty slots}}\end{array}$$

Example: Deriving Specifications

Show that the decrement-and-store instruction

$$\{\mathrm{R}\ a\ x * \mathrm{stack}\ y\ xs\ (n+1)\}$$
$$\mathrm{STR}\ a,\ [13, \# - 4]!$$
$$\{\mathrm{R}\ a\ x * \mathrm{stack}\ (y-1)\ (x :: xs)\ n\}$$

can be used as a stack push, where

$$\mathrm{stack}\ y\ [x_0, \ldots, x_{m-1}]\ n \ \equiv\ \mathrm{R}\ 13\ (addr\ y) *$$
$$\underbrace{\mathrm{M}\ (y+m-1)\ x_{m-1} * \cdots * \mathrm{M}\ y\ x_0}_{[x_{m-1}, \ldots, x_0]} * \underbrace{\mathrm{M}\ (y-1)\ \_ * \cdots * \mathrm{M}\ (y-n)\ \_}_{\text{empty slots}}$$

The Verification Flow

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

The whole verification takes place in the HOL4 proof assistant.

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) `word32`s.
- 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;
}
```

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 the project to generate formally verified elliptic curve cryptography in ARM machine code.
- There's much work still to be done to generate, say, a formally verified ARM machine code implementation of 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).