Elliptic Curve Cryptography

A case study in formalization using a higher order logic theorem prover

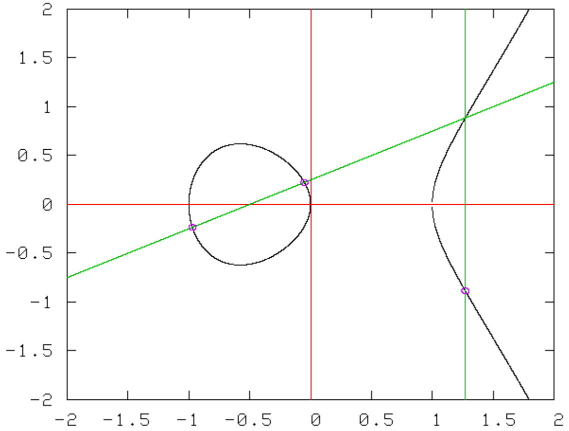
Joe Hurd

Computing Laboratory
Oxford University

8 August 2005
Galois Connections, Inc

Talk Plan

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



The Etymology of “Elliptic Curve”

- Elliptic curves over the reals are pairs (x,y) satisfying an equation of the form

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

- Studying the arc lengths of an ellipse leads to *elliptic integrals*

$$\int \frac{dx}{\sqrt{4x^3 - g_2 x - g_3}} \ .$$

- The inverse of an elliptic integral is a doubly periodic function called an *elliptic function*.
- Every doubly periodic function $\wp$ satisfies

$$\wp'^2 = 4\wp^3 - g_2 \wp - g_3$$

which is an instance of the elliptic curve equation.

Cryptography Based On Groups

- Many cryptographic primitives make use of the Discrete Logarithm Problem (DLP) over a group G.
  - DLP: given $x, y \in G$, find a k such that $x^k = y$.
- The difficulty of DLP depends on the group G.
- For some groups, such as integer addition modulo n, the problem is easy.
- For some groups, such as the multiplicative group $GF(p)^*$ of the finite field $GF(p)$, the problem is difficult.
  - Warning: the number field sieve can solve this in sub-exponential time.

Cryptography Based On Elliptic Curve Groups

- We can base DLP on an elliptic curve group $E(GF(q))$ over a finite field $GF(q)$.
  - There are no known sub-exponential algorithms to solve this problem.
- This table shows equal security key sizes in DLP:

$GF(p)^*$	$E(GF(q))$
1024 bits	173 bits
4096 bits	313 bits

- Elliptic curve groups require shorter keys than multiplicative groups.
  - Elliptic curve cryptography benefits security applications with constraints on bandwidth or computation power.

Elliptic Curve Cryptography Project

- A joint project between Oxford, Cambridge and Utah.
- Overall goal: to formally verify ARM assembly code programs implementing elliptic curve cryptography.
    - Need a model of ARM assembly code in HOL4. $\surd$
    - Need a formalization of elliptic curve cryptography. $\Longleftarrow$
- Aim to prove a HOL4 theorem that ARM code correctly implements elliptic curve arithmetic.
- Proving the correctness of ARM code implementing elliptic curve cryptography will rely on the theorem that elliptic curve arithmetic forms a group.
    - An unusual example of a practical verification requiring the formalization of highly abstract mathematics.

Formalization Status

The formalization of elliptic curves in HOL4 is *a work in progress.*

- Currently about 4000 lines of ML, comprising:
  - 3500 lines of definitions and theorems; and
  - 500 lines of extensions to the system proof tools.
- Complete up to the theorem that elliptic curve arithmetic forms an Abelian group.
  - Specifically, stuck on the lemma that doubling a point on the curve results in a point on the curve.
- Will progress by improving either the proof strategy (to reduce the size of the formulas) or the tactics (to deal with larger formulas).

Source Material

- 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 design goal in this work is for the formalized definitions to match the textbook definitions as closely as possible, and so the critical definitions are simply copied verbatim from the textbook.

Elliptic Curve Points

The Blake, Seroussi and Smart definition of elliptic curve points:

> *"An elliptic curve over a [field] 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 $a_1, a_2, a_3, a_4, a_6 \in K$."*

Elliptic Curve Points in Projective Coordinates

The HOL4 definition of elliptic curve points:

Constant Definition

```
curve_points e =
(let f = e.field in
 ...
 let a6 = e.a6 in
 { project f [x; y; z] |
   [x; y; z] IN 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) }
```

Elliptic Curve Points in Affine Coordinates (1)

In common with most texts, Blake, Seroussi and Smart use affine coordinates as well as projective 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*."*

Elliptic Curve Points in Affine Coordinates (2)

To avoid switching between two different representations, we treat affine coordinates as an abbreviation for projective coordinates.

Constant Definition

```
affine f v = project f (v ++ [field_one f])
```

Equality in affine coordinates is much simpler than equality in projective coordinates.

Theorem

```
|- !f :: Field. !v1 v2.
     (affine f v1 = affine f v2) = (v1 = v2)
```

Elliptic Curve Points in Affine Coordinates (3)

First define the point at infinity $\mathcal{O}$:

Constant Definition

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

Elliptic Curve Points in Affine Coordinates (4)

From the definition of elliptic curve points in projective coordinates, it is possible to recover the equation for affine coordinates:

Theorem

```
|- !e :: Curve. curve_zero e IN curve_points e

|- !e :: Curve. !x y :: (e.field.carrier).
     affine e.field [x; y] IN 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
```

Elliptic Curve Points in Affine Coordinates (5)

We also prove that all elliptic curve points are either $\mathcal{O}$ or can be expressed in affine coordinates, and that $\mathcal{O}$ cannot be expressed in affine coordinates:

Theorem

```
|- !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])
```

Elliptic Curve Points in Affine Coordinates (6)

Finally we prove a ‘case theorem’ allowing us to define functions on elliptic curve points using affine coordinates.

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
```

At this point there is no further need to reason about the projective version of elliptic curves.

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

Negation of Elliptic Curve Points (2)

Negation is formalized using the case theorem, 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)
  (\x1 y1.
     let x = x1 in
     let y = ~y1 - a1 * x1 - a3 in
     affine f [x; y])
```

Negation of Elliptic Curve Points (3)

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

Theorem

```
|- !e :: Curve. !p :: curve_points e.
     curve_neg e p IN curve_points e
```

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 |>
```

ElGamal Encryption (1)

The ElGamal encryption algorithm uses an 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)

We first formalize the ElGamal encryption packet that Alice sends to Bob.

Constant Definition

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

This follows the algorithm precisely.

ElGamal Encryption (3)

We next prove the theorem that Bob can decrypt the ElGamal encryption packet to reveal the message (assuming he knows his private key).

Theorem

```
|- !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)
```

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

Verified Elliptic Curve Calculations

- It is often desirable to derive calculations that provably follow from the definitions.
  - Can be used to test the model,
  - or provide ‘golden reference’ results.
- The inference rule is fairly basic, and just consists of unfolding definitions in the correct order.
  - The numerous side conditions are proved with predicate subtype style reasoning.
- For this version we simply assume that $751 \in \text{Prime}$ (and thus $\text{GF}(751) \in \text{Field}$), but in future we’ll need a way to prove that our field sizes really are prime.

Rational Points on Elliptic Curves

We begin by defining an elliptic curve equation from an exercise in Koblitz (1987).

Constant Definition

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

We next prove that it defines an elliptic curve and that two points given in the exercise do indeed lie on the curve.

Theorem

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

Efficiently Calculating Field Inverses

- Elliptic curve arithmetic requires computing field inverses.
- Field inverse in GF(p) is defined as $\lambda k.\, k^{p-2}$.
- To make this more efficient, we verify a version of exponentiation using repeated squaring.

Constant Definition

```
modexp a n m =
if n = 0 then 1
else if n MOD 2 = 0 then modexp ((a * a) MOD m) (n DIV 2) m
else (a * modexp ((a * a) MOD m) (n DIV 2) m) MOD m
```

Theorem

```
|- !a n m. 1 < m ==> (modexp a n m = (a ** n) MOD m)
```

Elliptic Curve Arithmetic

We verify 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] 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
```

Doing this revealed a bug in the definition of point doubling!

Summary

- This talk has given a brief overview of the elliptic curve formalization in higher order logic, showing how to bridge the gap between mathematics and executable programs.
- The formalization is designed to interface with the verifying compilation to ARM assembly code, but could be ‘retargeted’ to export Cryptol programs or verified test vectors.