Compositional Verification of Elliptic Curve Cryptography

John Launchbury
Layered Assurance Workshop, 2012

The Cryptol team at Galois, past and present:
Lennart Augustsson, Sally Browning, Ledah Casburn, Iavor Diatchki, Trevor Elliot, Levent Erkok, Sigbjorn Finne, Adam Foltzer, Andy Gill, Fergus Henderson, Joe Hendrix, Joe Hurd, John Launchbury, Jeff Lewis, Lee Pike, John Matthews, Thomas Nordin, Mark Shields, Joel Stanley, Frank Seaton Taylor, Jim Teisher, Aaron Tomb, Mark Tullsen, Philip Weaver, Adam Wick, Edward Yang
In 2007, Harry Reimann discovered a bug in `BN_nist_mod_384`, a function used for field division in OpenSSL’s implementation of the NIST P-384 elliptic curve

- Edge case that occurred on less than 1 in $2^{29}$ inputs; no known exploit at the time
- In 2012, an adaptive attack allowed full key recovery by triggering the bug

Our goal: create an efficient verified implementation of ECDSA over NIST P-384 curve in Java
What is an Elliptic Curve?

\[ y^2 = x^3 + ax + b \]
$y^2 = x^3 - x + 1$
Discrete setting: $\mathbb{Z}_{19}$

$$y^2 = x^3 - x + 1$$
Addition

$$y^2 = x^3 - x + 1$$
Doubling

\[ y^2 = x^3 - x + 1 \]
Algorithm for addition

\[ P + Q = (R_x, R_y) \]

where \( s = (Q_y - P_y) / (Q_x - P_x) \)

\[ R_x = s^2 - P_x - Q_x \]

\[ R_y = s(P_x - R_x) - P_y \]
Algorithm for addition

\[ P + Q = (R_x, R_y) \]

where
\[ s = \frac{(Q_y - P_y)}{(Q_x - P_x)} \]

\[ R_x = s^2 - P_x - Q_x \]
\[ R_y = s(P_x - R_x) - P_y \]
For large discrete elliptic curves, scalar multiplication is a one-way function:

\[ Q = k \cdot P \]

(Easy to compute \( k \cdot P \); hard to find \( k \) from \( Q \) and \( P \))

- This operation is used to implement Suite B algorithms ECDSA (digital signatures) and ECDH (key agreement)
NIST P384 Curve

- Prime field $P_{384}$

\[ P_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1 \]

- Curve Equation: $y^2 = x^3 - 3x + b$, where

\[ b = \text{b3312fa7 e23ee7e4 988e056b e3f82d19 181d9c6e fe814112} \]
\[ \quad \text{0314088f 5013875a c656398d 8a2ed19d 2a85c8ed d3ec2aef} \]
Implementing ECC

Cryptographic Protocols

- **ECDSA**
  - Digital Signatures
- **ECDH**
  - Key Agreement

One Way Functions

- \( R = s \cdot P \) (Scalar Multiplication)
- \( R = s \cdot P + t \cdot Q \) (Twin Multiplication)

Point Operations

- \( R = P + Q \) (Addition)
- \( R = P - Q \) (Subtraction)
- \( R = 2 \cdot P \) (Doubling)

Field Operations

- Multiplication
- Squaring
- Division
- Addition
- Subtraction
- Doubling
Implementing ECC

Cryptographic Protocols
- **ECDSA**
  - Digital Signatures
- **ECDH**
  - Key Agreement

One Way Functions
- $R = s \cdot P$
- $R = s \cdot P + t \cdot Q$

Point Operations
- $R = P + Q$
- $R = P - Q$
- $R = 2 \cdot P$

Field Operations
- Multiplication
- Squaring
- Division

Optimize modular reduction for specific field prime
Implementing ECC

Use projective coordinates to avoid field division and minimize multiplications
Implementing ECC

- Cryptographic Protocols
  - **ECDSA**
    - Digital Signatures
  - **ECDH**
    - Key Agreement

- One Way Functions
  - Scalar Multiplication: \( R = s \cdot P \)
  - Twin Multiplication: \( R = s \cdot P + t \cdot Q \)

- Point Operations
  - Addition: \( R = P + Q \)
  - Subtraction: \( R = P - Q \)
  - Doubling: \( R = 2 \cdot P \)

- Field Operations
  - Multiplication
  - Squaring
  - Division
  - Doubling

Use sign digit encoding to reduce the average number of points additions.
Implementing ECC

- Cryptographic Protocols
  - ECDSA: Digital Signatures
  - ECDH: Key Agreement

- One Way Functions
  - Use twin multiplication when needed

- Point Operations
  - Addition: \( R = P + Q \)
  - Subtraction: \( R = P - Q \)
  - Doubling: \( R = 2P \)

- Field Operations
  - Multiplication
  - Squaring
  - Division
  - Doubling
ECC Benchmarks
Sign & Verify

BC (64bit) - 70ms
Galois (32bit) - 30ms
OpenSSL (32bit) - 20ms
Galois (64bit) - 10ms
OpenSSL (64bit) - 5ms

64 bit implementation is realistic and competitive
Using Cryptol
One specification – many uses

w₀ = u - I₁ mod p + u - I₁ w₁ mod p
s = f * (w₀ + p w₂) mod q
Basic Verification Strategy

1. Use forward symbolic simulation to unroll implementations, and generate terms that precisely describe results.

2. Show equivalence of two terms through rewriting, and off-the-shelf theorem provers, including abc or Yices.

ABC   Yices   Rewriting
1. Use forward symbolic simulation to unroll implementations, and generate terms that precisely describe results.

2. Show equivalence of two terms through rewriting, and off-the-shelf theorem provers, including abc or Yices.
1. Use forward symbolic simulation to unroll implementations, and generate terms that precisely describe results.

2. Show equivalence of two terms through rewriting, and off-the-shelf theorem provers, including abc or Yices.
1. Use forward symbolic simulation to unroll implementations, and generate terms that precisely describe results

2. Show equivalence of two terms through rewriting, and off-the-shelf theorem provers, including abc or Yices
# Suite B Problem Sizes

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Lines of Code</th>
<th>Logic Size</th>
<th>Decomposition Steps Required</th>
<th>Verification Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-128 BouncyCastle (Java) AESFastEngine</td>
<td>817</td>
<td>1MB</td>
<td>None needed, Fully automatic</td>
<td>40 min</td>
</tr>
<tr>
<td>SHA-384 libgcrypt (C)</td>
<td>423</td>
<td>3.2MB</td>
<td>12 steps, Easy composition, all solved via SAT</td>
<td>160 min</td>
</tr>
<tr>
<td>ECDSA (P-384) Galois (Java)</td>
<td>2348</td>
<td>More than 5GB</td>
<td>48 steps, Richer compositional approach required</td>
<td>10 min</td>
</tr>
</tbody>
</table>
Elliptic Curve Crypto (ECC)

Cryptographic Protocols
- **ECDSA**
  - Digital Signatures
- **ECDH**
  - Key Agreement

One Way Functions
- **Scalar Multiplication**
  - $R = s \cdot P$
- **Twin Multiplication**
  - $R = s \cdot P + t \cdot Q$

Point Operations
- **Addition**
  - $R = P + Q$
- **Subtraction**
  - $R = P - Q$
- **Doubling**
  - $R = 2 \cdot P$

Field Operations
- **Multiplication**
- **Squaring**
- **Division**
- **Doubling**
Elliptic Curve Crypto (ECC)

Cryptographic Protocols
- **ECDSA**: Digital Signatures
- **ECDH**: Key Agreement

One Way Functions
- Scalar Multiplication: $R = s \cdot P$
- Twin Multiplication: $R = s \cdot P + t \cdot Q$

Point Operations
- Addition: $R = P + Q$
- Subtraction: $R = P - Q$
- Doubling: $R = 2 \cdot P$

Field Operations
- Multiplication
- Addition
- Subtraction
- Doubling

Solvable using SAT-based equivalence checking
Elliptic Curve Crypto (ECC)

Cryptographic Protocols
- **ECDSA**
  - Digital Signatures
- **ECDH**
  - Key Agreement

One Way Functions

Scalar Multiplication
- $R = s \cdot P$
- $R = s \cdot P + t \cdot Q$

Digital Signatures
- **ECDSA**

Key Agreement
- **ECDH**

Point Operations
- Addition $R = P + Q$
- Subtraction $R = P - Q$
- Doubling $R = 2 \cdot P$

Field Operations
- Multiplication
- Squaring
- Division
- Doubling

Symbolic simulation can construct models up to point layer, but verification was infeasible
Key Idea

- Establish specification-implementation correspondences
- Use correspondences to produce simpler models of enclosing methods

Example:

```c
void ec_double(JacobianPoint r) {
    ...  
    field_add(t4, r.x, t4);  
    field_mul(t5, t4, t5);  
    field_mul3(t4, t5);  
    ...  
}
```

- Don’t produce code model for field_add. Instead, replace value at t4 with specification of ref_field_add

Had to do this process >40 times to verify ECDSA
Results

- Found three bugs (in our optimized code)
  - Sign & verify failed to clear all intermediate results
  - Boundary condition due to use of less-than where less-than-or-equal was needed
  - Modular reduction failed to propagate one overflow
- After fixing these, verification completes in 10 minutes
Modular division bug

NISTCurve.java (line 964):

```java
    d = (z[ 0] & LONG_MASK) + of;
    z[ 0] = (int) d; d >>= 32;
    d = (z[ 1] & LONG_MASK) - of;
    z[ 1] = (int) d; d >>= 32;
    d += (z[ 2] & LONG_MASK);
```
NISTCurve.java (line 964):

```java
    d = (z[ 0] & LONG_MASK) + of;
    z[ 0] = (int) d; d >>= 32;
    d += (z[ 1] & LONG_MASK) - of;
    z[ 1] = (int) d; d >>= 32;
    d += (z[ 2] & LONG_MASK);
```
Modular division bug

Bug only occurs when this addition overflows
Previous code guaranteed that (0 < of < 5)

NISTCurve.java (line 964):

```
d = (z[ 0] & LONG_MASK) + of;
z[ 0] = (int) d; d >>= 32;
d += (z[ 1] & LONG_MASK) - of;
z[ 1] = (int) d; d >>= 32;
d += (z[ 2] & LONG_MASK);
```
Modular division bug

NISTCurve.java (line 964):

```java
    d = (z[ 0] & LONG_MASK) + of;
    z[ 0] = (int) d; d >>= 32;
    d += (z[ 1] & LONG_MASK) - of;
    z[ 1] = (int) d; d >>= 32;
    d += (z[ 2] & LONG_MASK);
```

abc found bug in 20 seconds. Testing found bug after 2 hours (8 billion field reductions)

Note: testing would have taken ~1M years if the bug had been in this line
Summary

- We’ve successfully verified *efficient* implementations of the main cryptographic algorithms used in Suite B
- The level of effort required for verification depends on the algorithm
- Verification of complex algorithms benefits from tools that offer a variety of verification techniques, and requires compositional reasoning