|galois|

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

An Improbable Bug

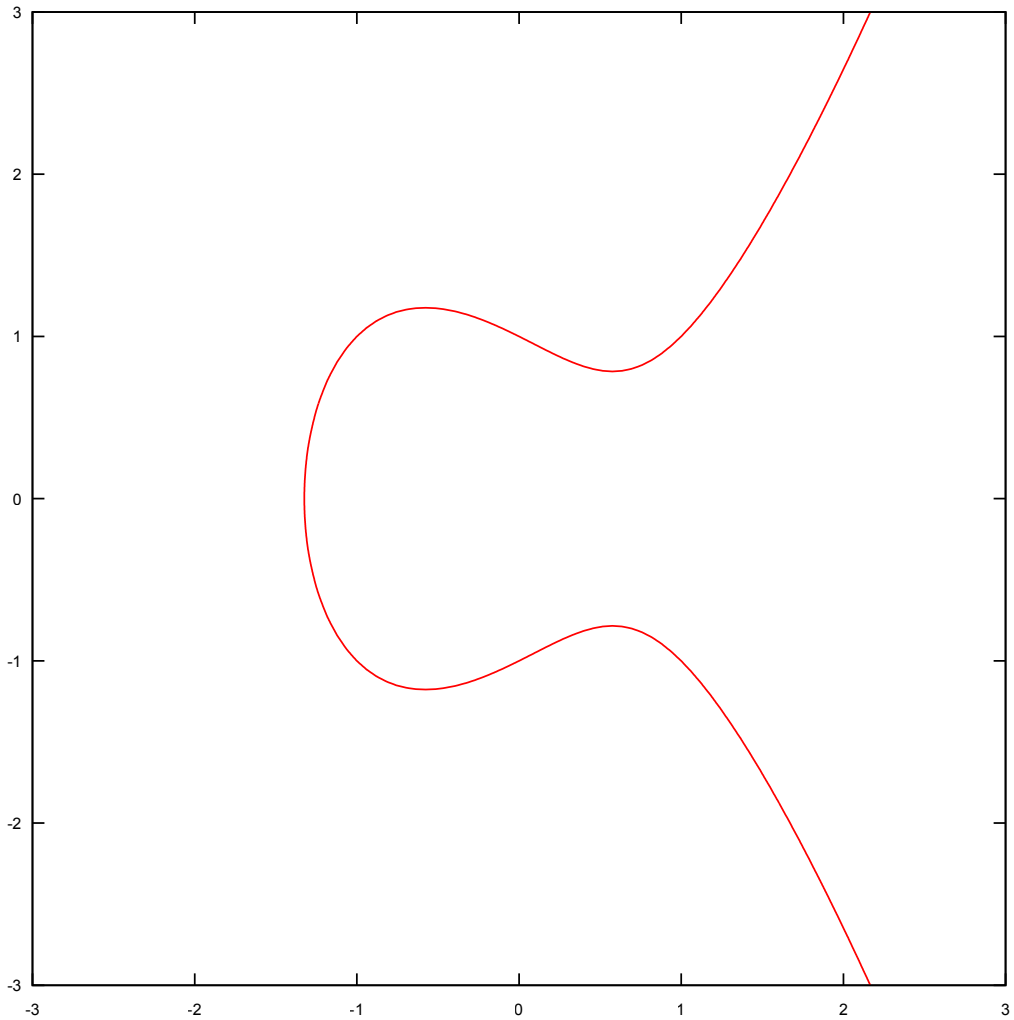
In 2007, Harry Reimann discovered a bug in `BN_nist_mod_384`, a function used for field division in OpenSSL's implementation of the NIST P-384 elliptic curve

- 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

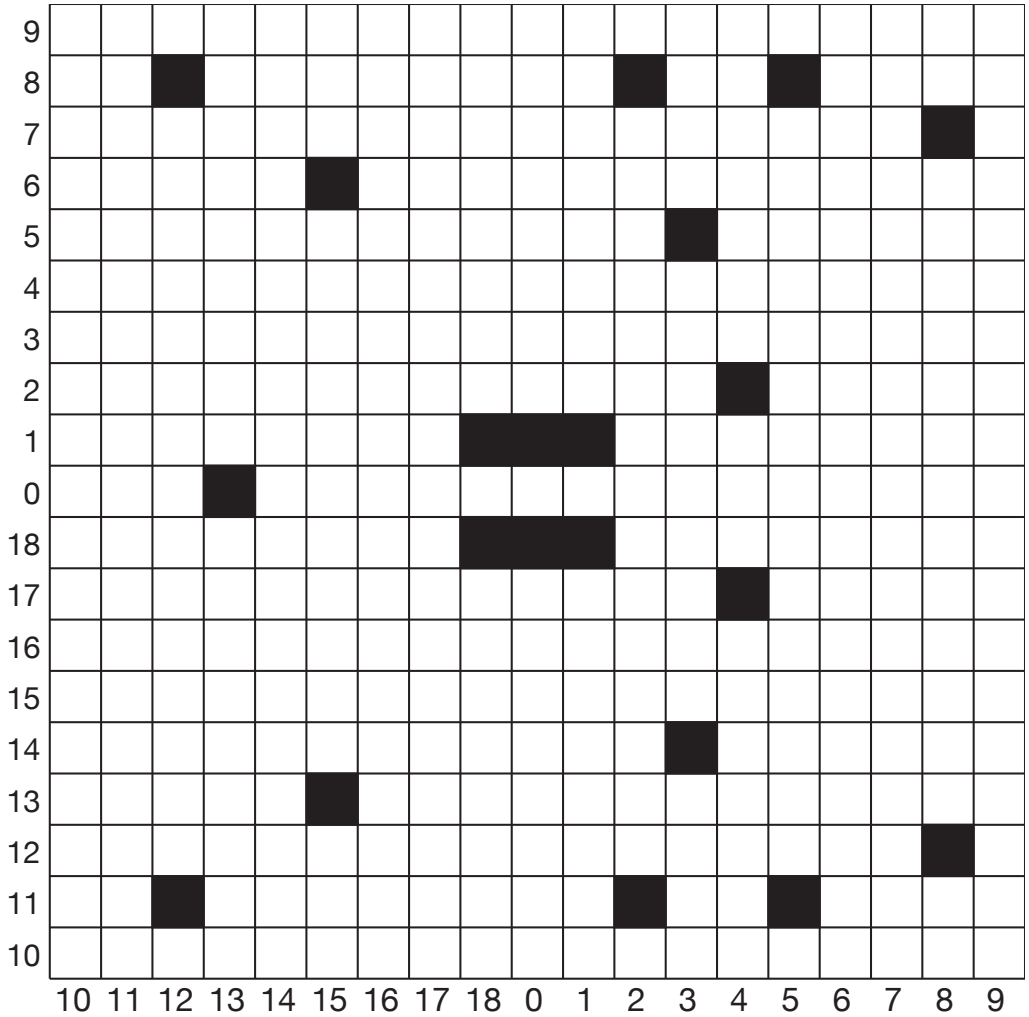
What is an Elliptic Curve?

$$y^2 = x^3 + ax + b$$



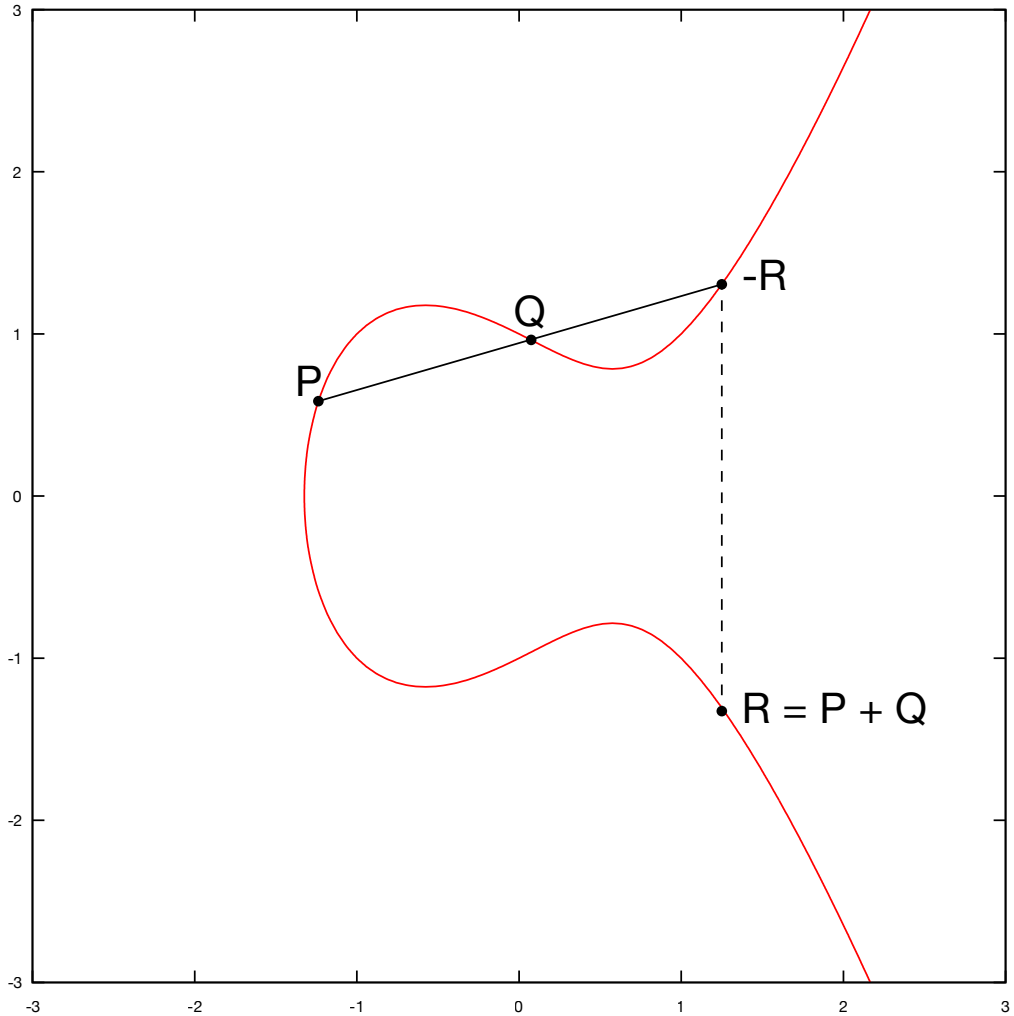
$$y^2 = x^3 - x + 1$$

Discrete setting: Z_{19}



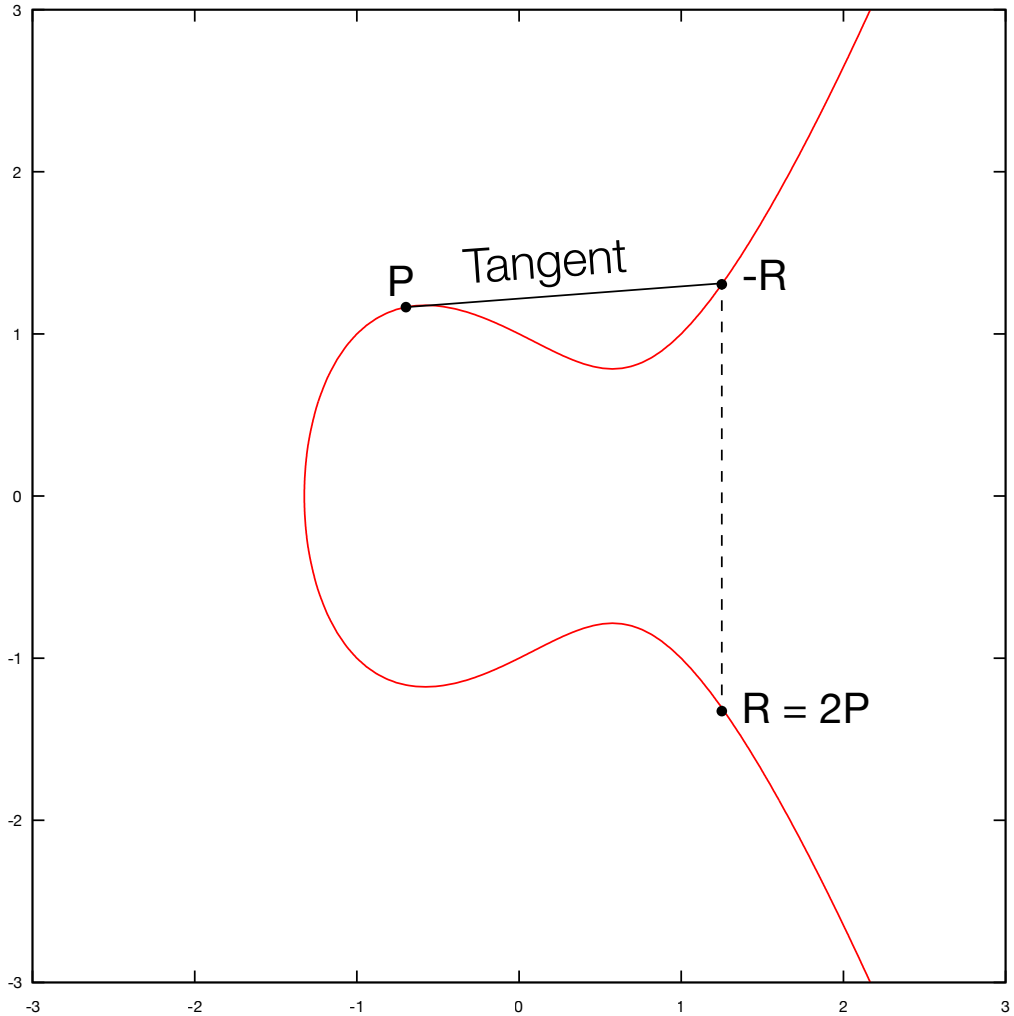
$$y^2 = x^3 - x + 1$$

Addition



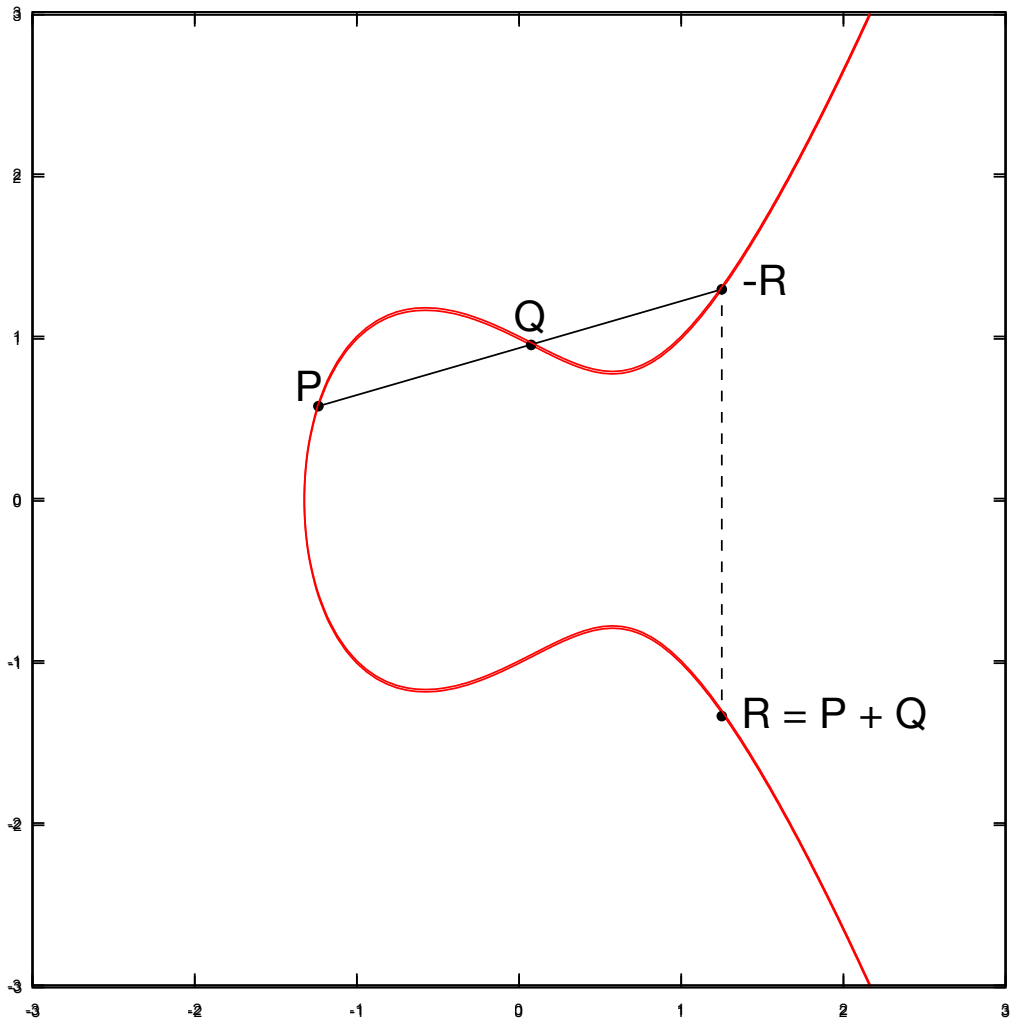
$$y^2 = x^3 - x + 1$$

Doubling



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

Algorithm for addition



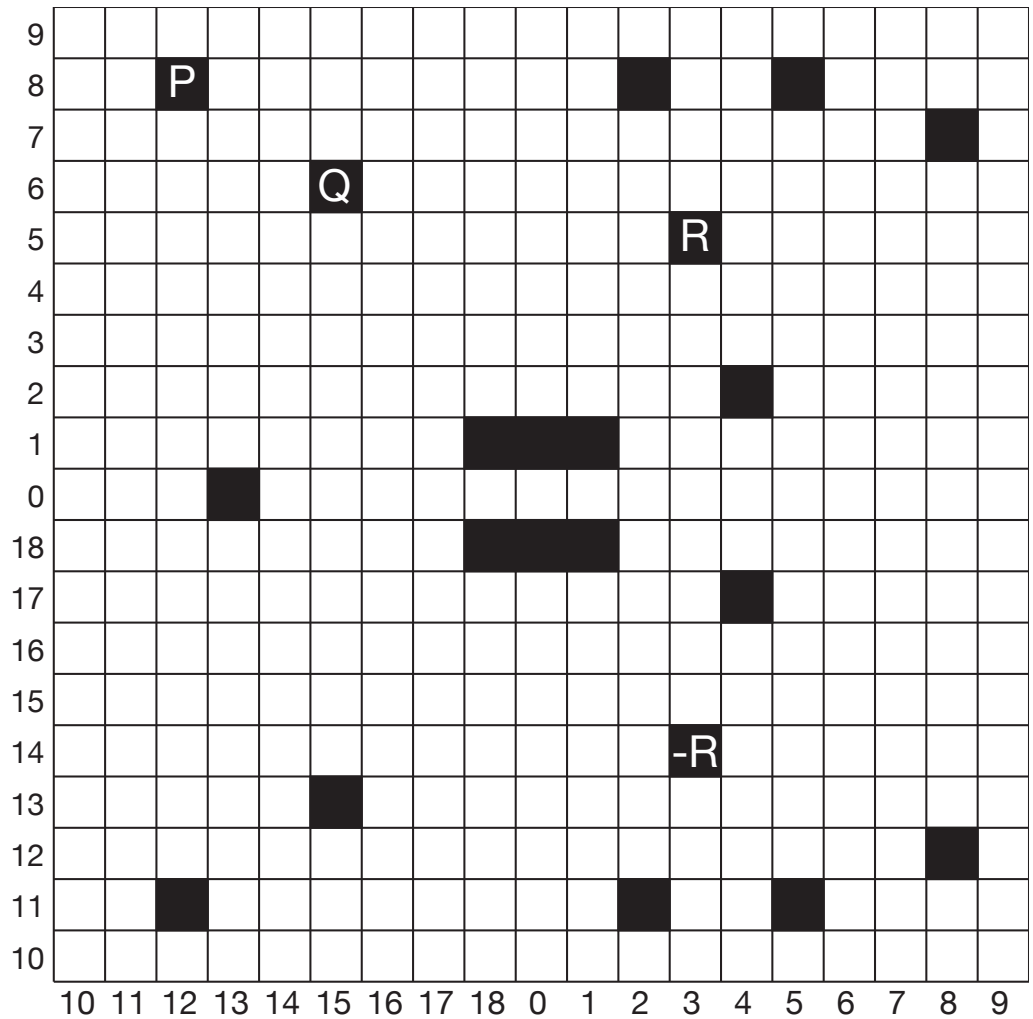
$$P + Q = (R_x, R_y)$$

$$\text{where } s = (Q_y - P_y)/(Q_x - P_x)$$

$$R_x = s^2 - P_x - Q_x$$

$$R_y = s(P_x - R_x) - P_y$$

Algorithm for addition



$$P + Q = (R_x, R_y)$$

$$\text{where } s = (Q_y - P_y)/(Q_x - P_x)$$

$$R_x = s^2 - P_x - Q_x$$

$$R_y = s(P_x - R_x) - P_y$$

Use in Cryptography

- 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

Security level comparable to AES-192

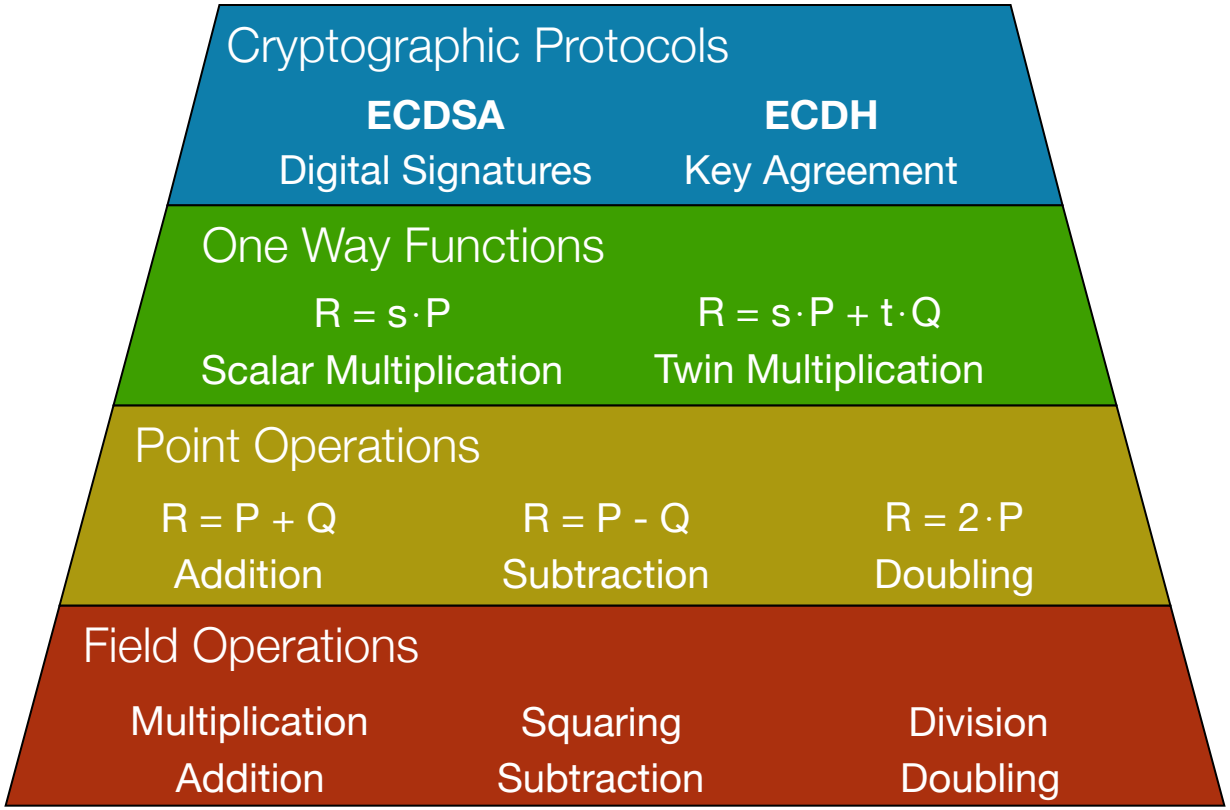
- Prime field P_{384}

$$P_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$$

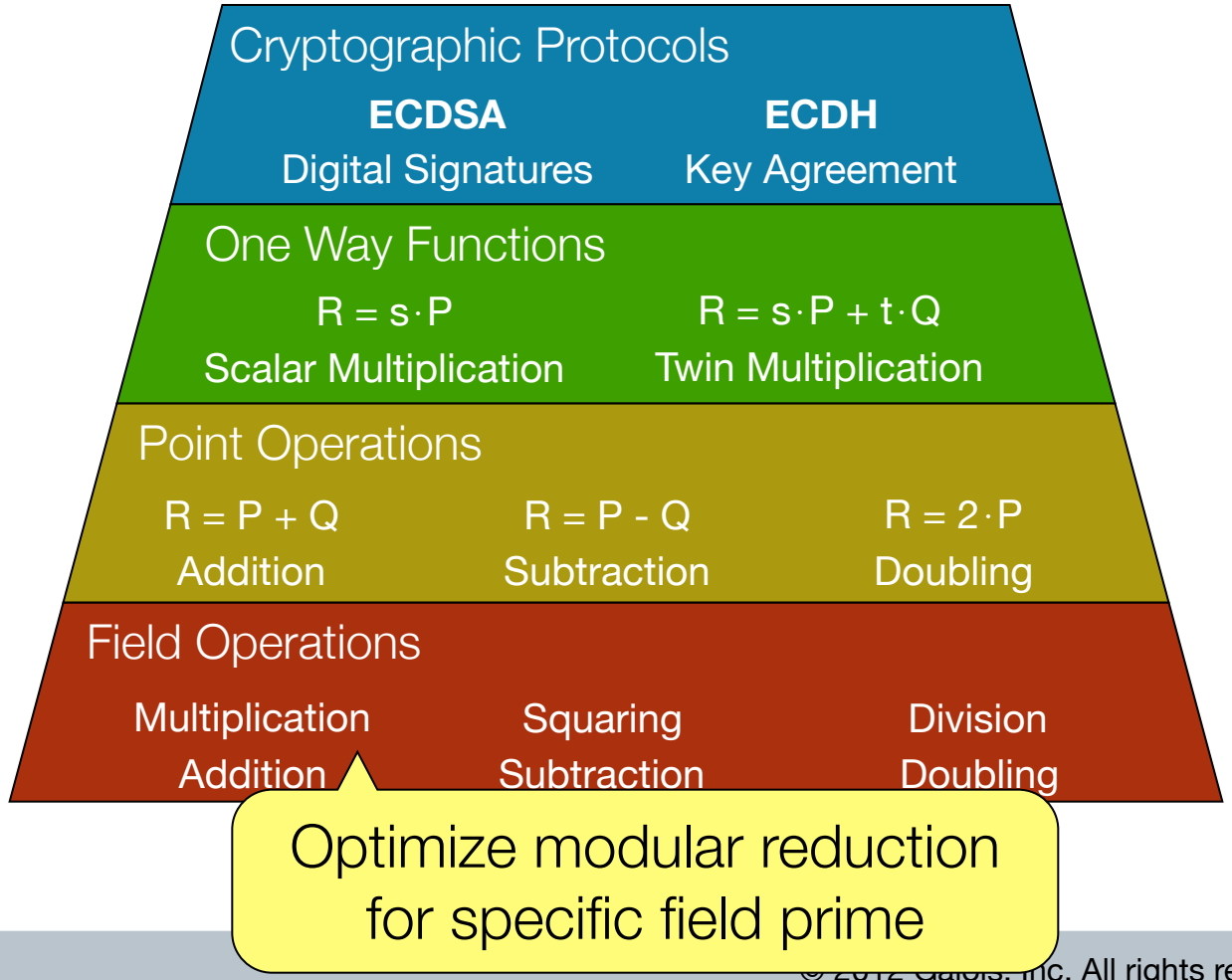
- Curve Equation: $y^2 = x^3 - 3x + b$, where

```
b = b3312fa7 e23ee7e4 988e056b e3f82d19 181d9c6e fe814112
    0314088f 5013875a c656398d 8a2ed19d 2a85c8ed d3ec2aef
```

Implementing ECC

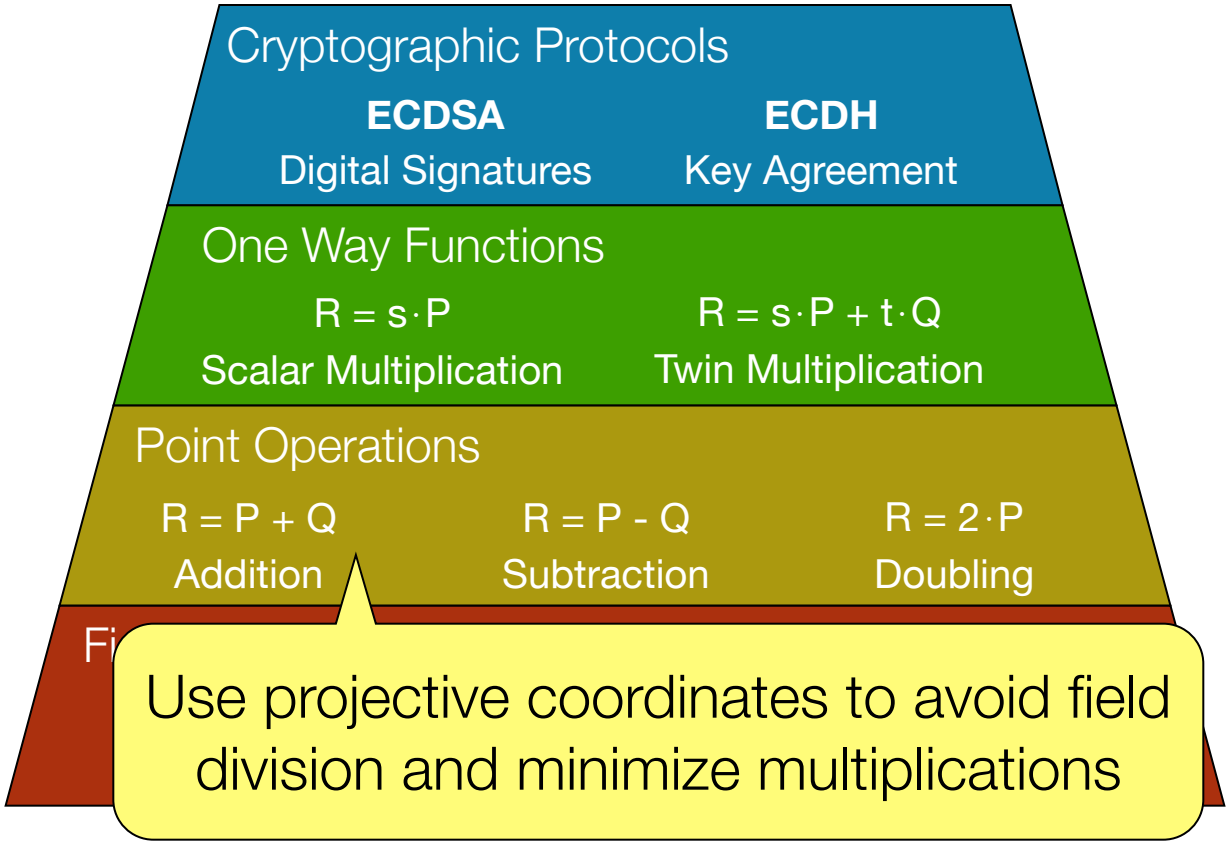


Implementing ECC

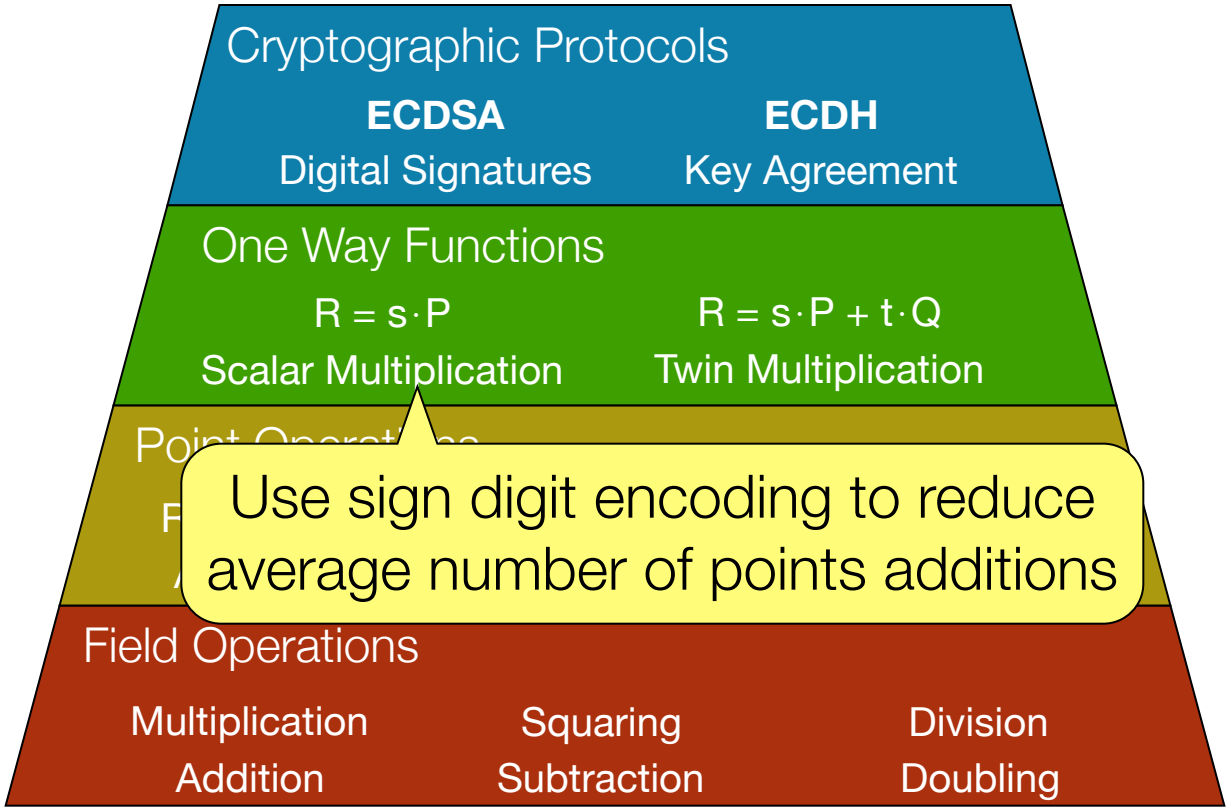


Optimize modular reduction for specific field prime

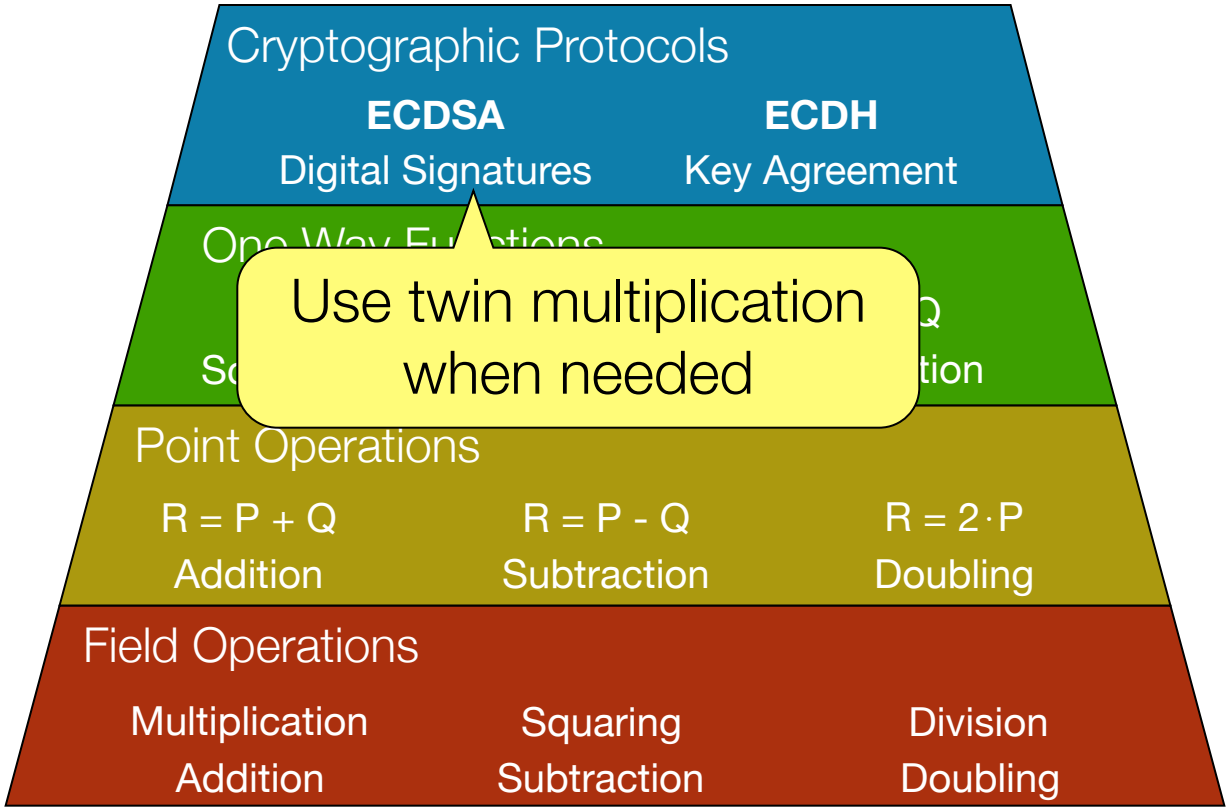
Implementing ECC



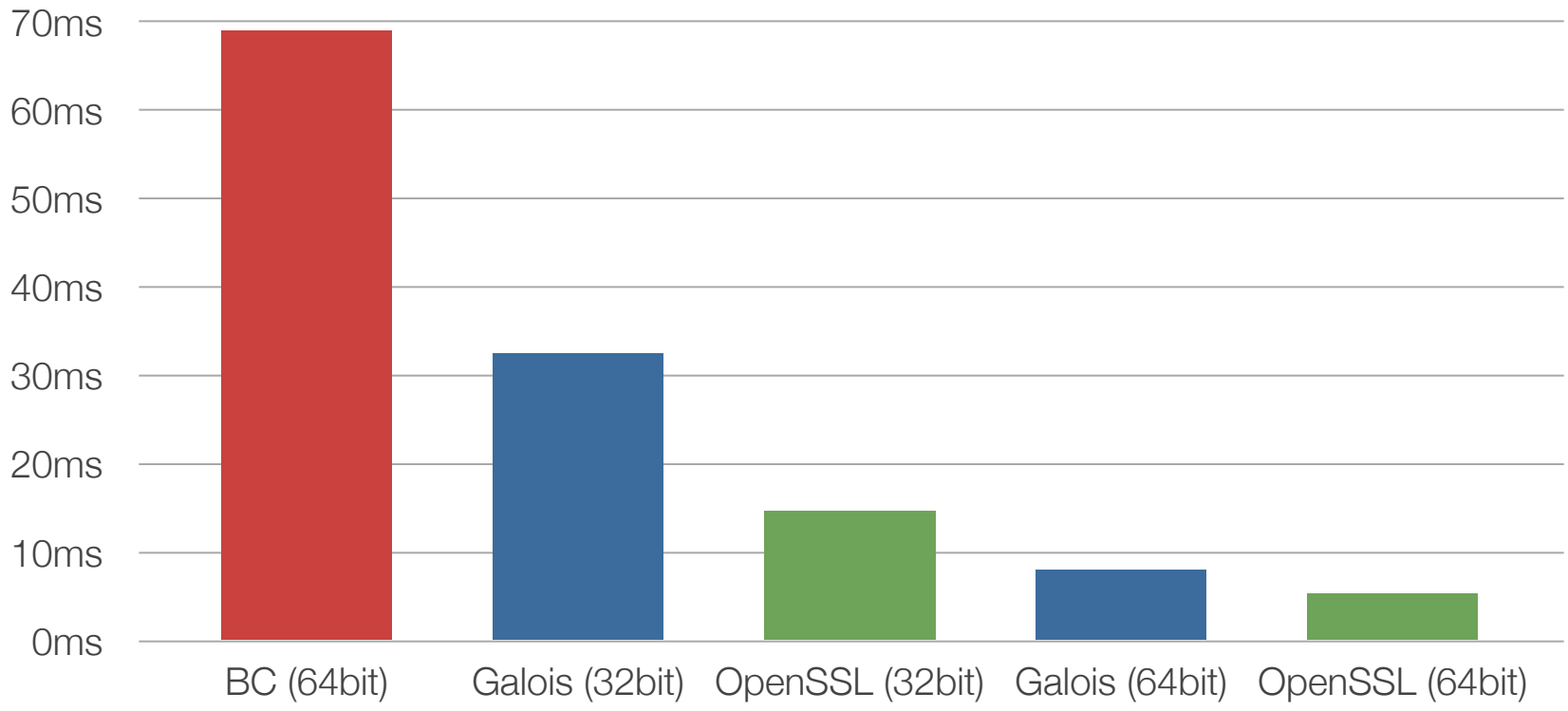
Implementing ECC



Implementing ECC

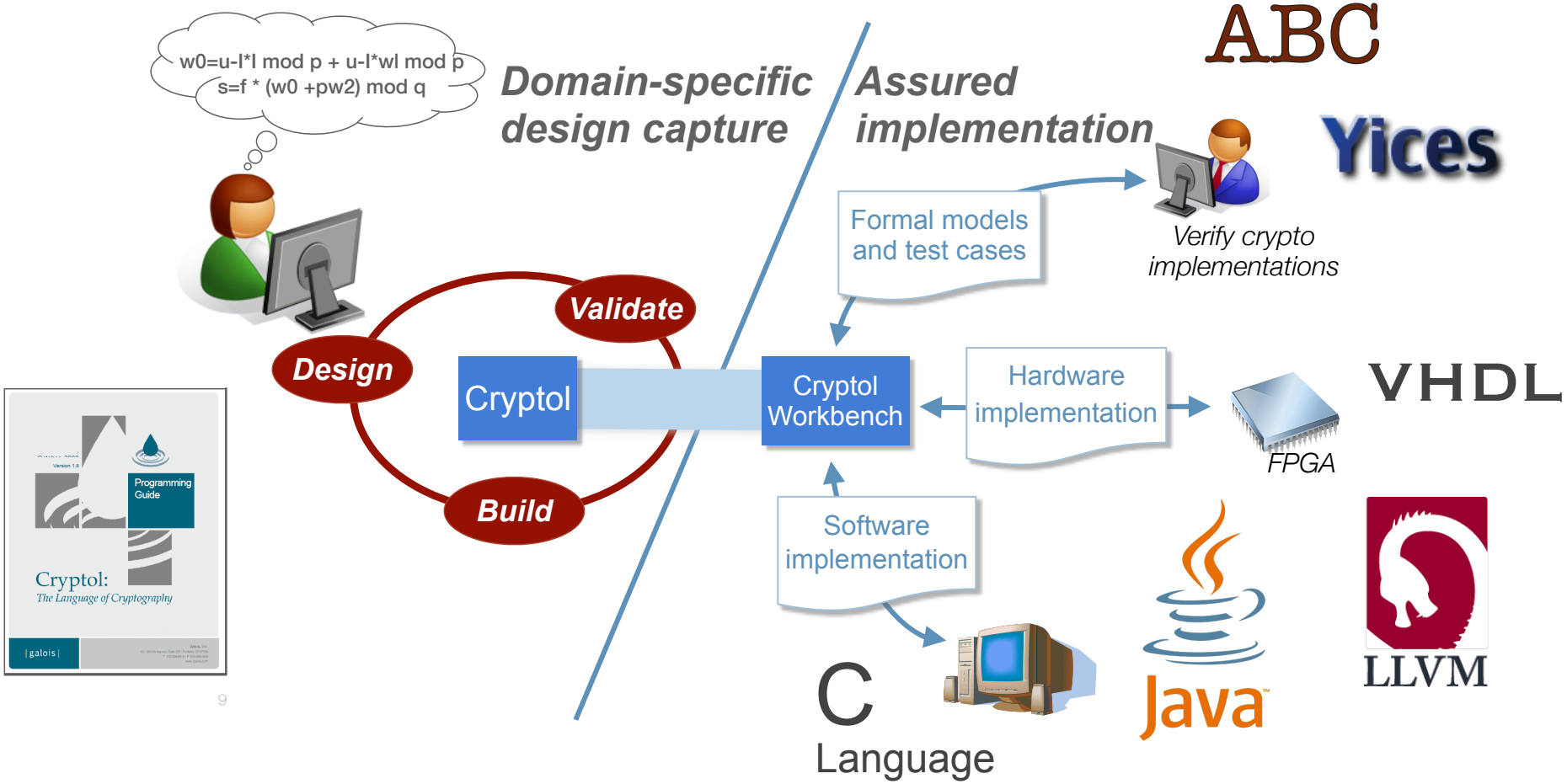


ECC Benchmarks
Sign & Verify



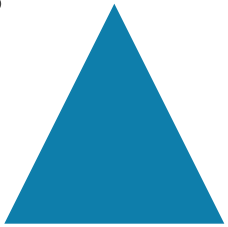
64 bit implementation is realistic and competitive

Using Cryptol
One specification – many uses

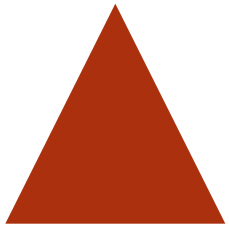


Basic Verification Strategy

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



Specification



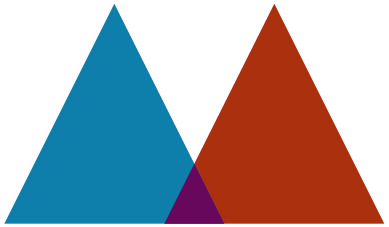
Implementation

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

ABC Yices Rewriting

Basic Verification Strategy

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



Specification Implementation

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

ABC Yices Rewriting

Basic Verification Strategy

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

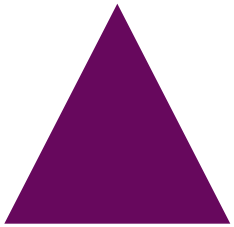
Specification Implementation

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

ABC Yices Rewriting

Basic Verification Strategy

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



Specification Implementation

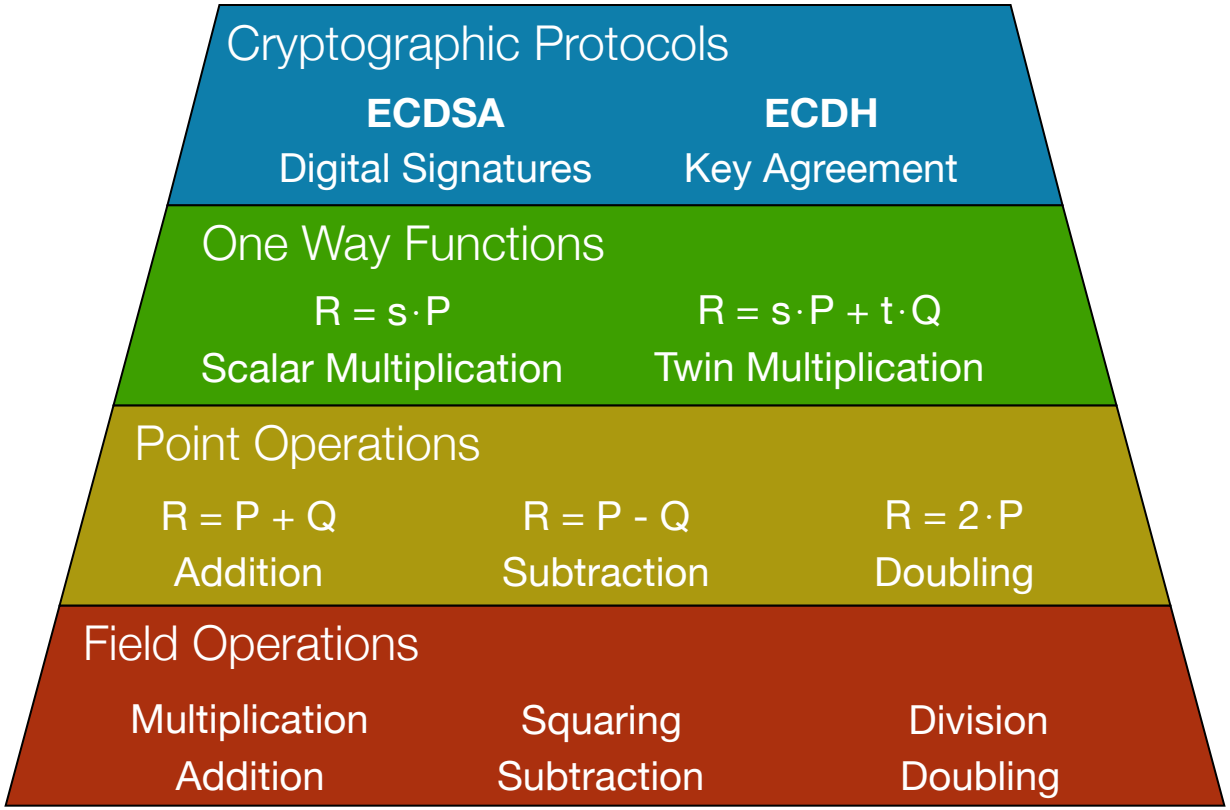
2. Show equivalence of two terms through rewriting, and off-the-shelf theorem provers, including abc or Yices

ABC Yices Rewriting

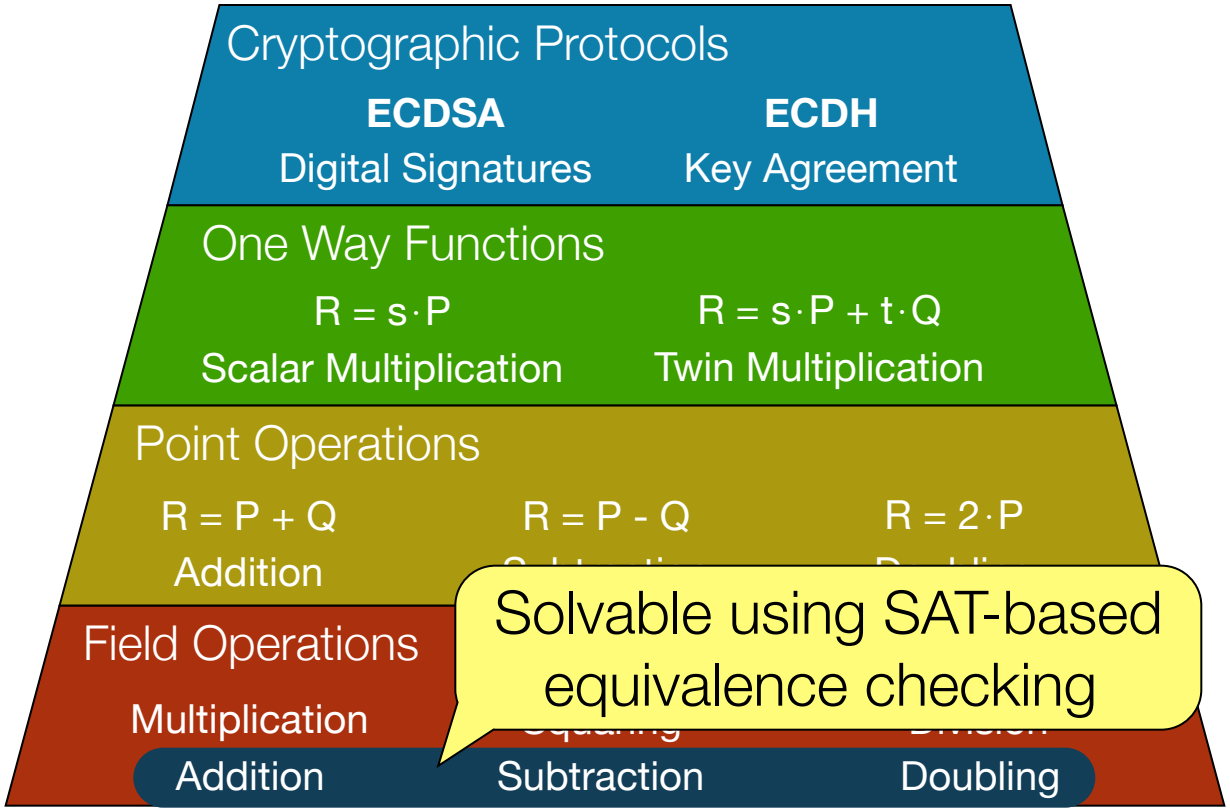
Suite B Problem Sizes

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

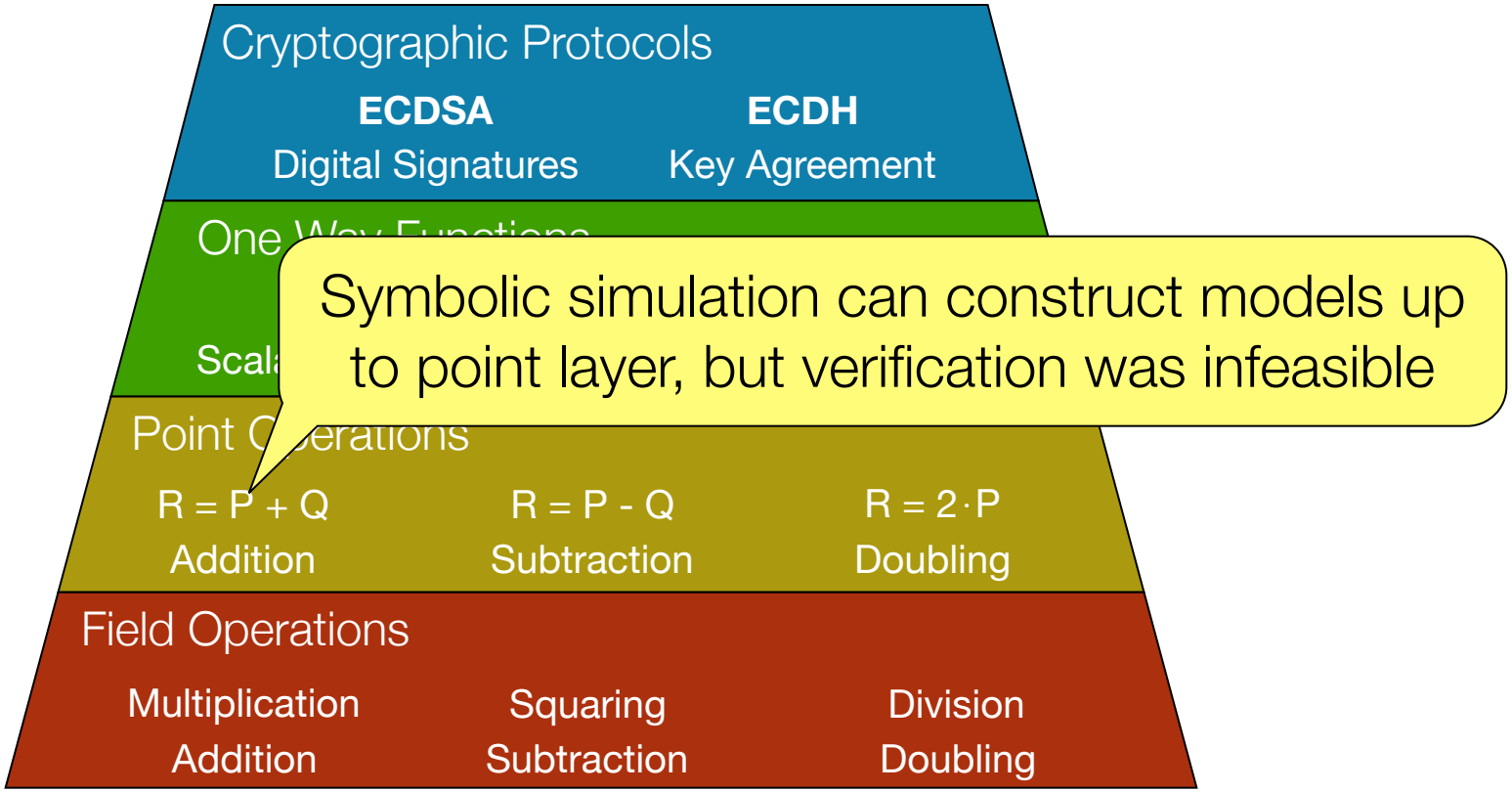
Elliptic Curve Crypto (ECC)



Elliptic Curve Crypto (ECC)



Elliptic Curve Crypto (ECC)



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

Had to do this process >40 times to verify ECDSA

- Example:

```
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`

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):

```
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):

```
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

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

```
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);
```

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