|galois|

Verification of Elliptic Curve Cryptography

Joe Hendrix, Galois, Inc

HCSS | May 2012

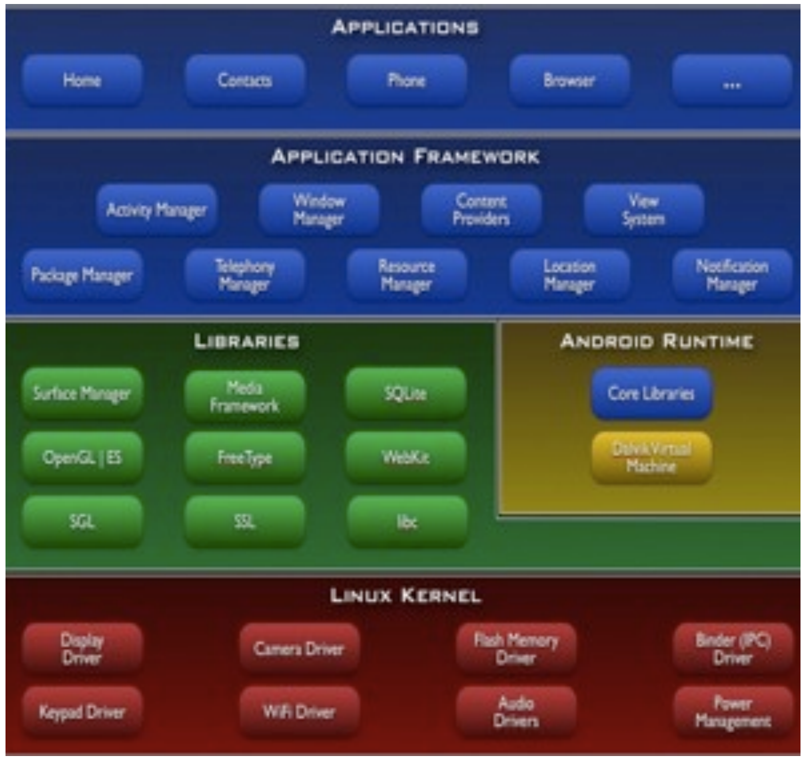
The Cryptol team, past and present:

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

Cryptographic algorithms are a small, but essential part of critical systems.

Defects in cryptographic implementation can compromise security of the entire system.

Testing is insufficient to find all bugs.



Cryptography: A Foundation of Trust

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.

- A similar bug occurred in the implementation of the NIST P-256 elliptic curve.
- Edge case that occurred on less than 1 in 2^{29} inputs; no known exploit at the time.
- Found day before release of OpenSSL0.9.8g; fix committed 6 months later.

Exploiting ECDH in OpenSSL

In 2012, Brumley, Barbosa, Page, and Vercauteren published a paper showing an adaptive attack that allowed full key recovery by triggering the bug.

- Ephemeral keys provide a mitigation.
- Several Linux distributions were still unpatched.
- The authors call for formal verification:

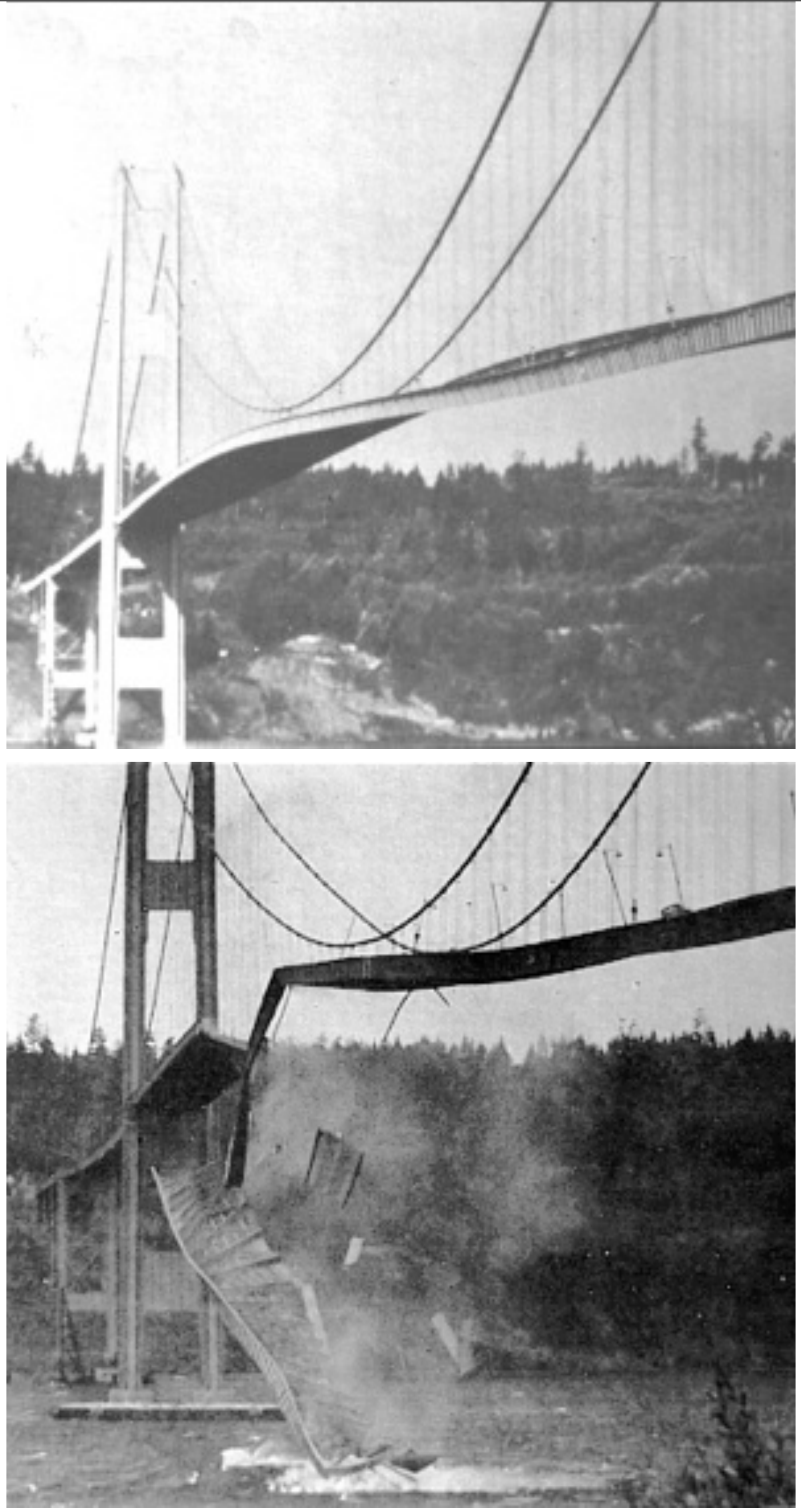
  We suggest that the effort required to adopt a development strategy capable of supporting formal verification is both warranted, and an increasingly important area for future work.

Statistics from the testing laboratories show that 48 percent of the cryptographic modules and 27 percent of the cryptographic algorithms brought in for voluntary testing had security flaws that were corrected during testing.

Without this program, the federal government would have had only a 50-50 chance of buying correctly implemented cryptography.

NIST Computer Security Division, 2008 Annual report

Software is a digital artifact — potential for much greater confidence in the correctness of our software than in the correctness of our bridges.



Bugs Are Prevalent

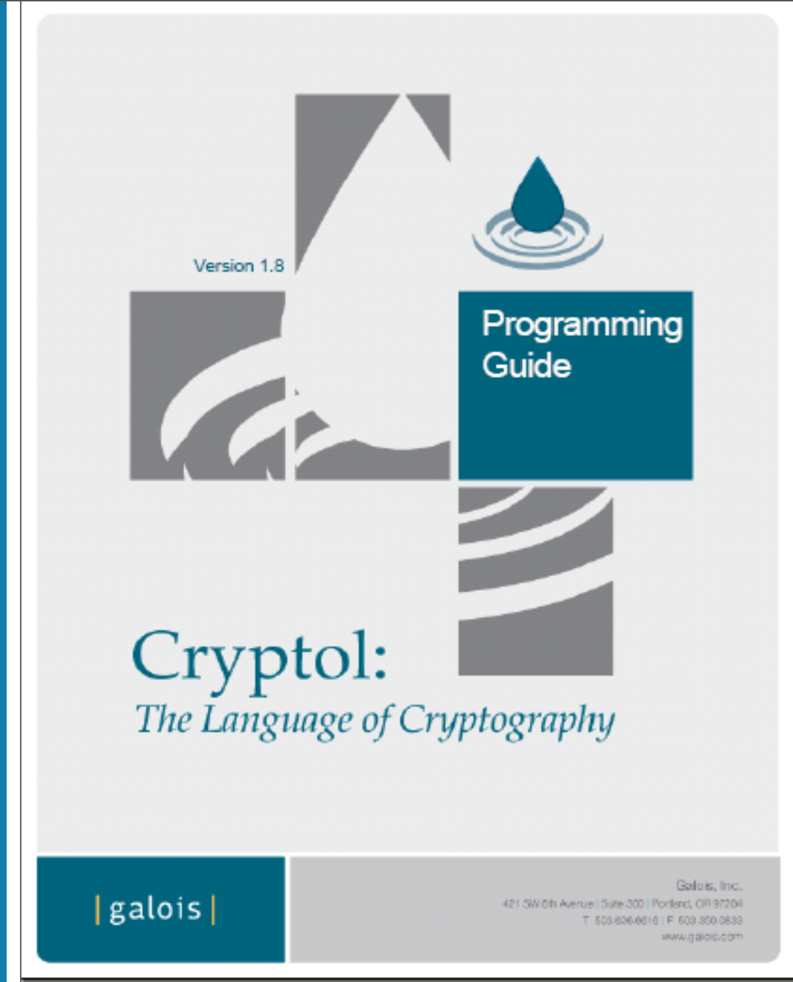
|galois|

Our Contribution

Galois has developed tools for showing that **different** cryptographic implementations compute the **same** values for all possible keys and inputs.

Uses formal verification techniques including symbolic simulation, rewriting, and third-party SAT and SMT-solvers.

From a user's perspective, our tool performs exhaustive test coverage.



ABC

Yices

Cryptol:
The Language of Cryptography

- Declarative specification language
  - Language tailored to the crypto domain.
  - Designed with feedback from NSA.

Cryptol Examples

- Add two 384-bits numbers

```
add : ([384],[384]) -> [384];
add(x,y) = x + y;
```

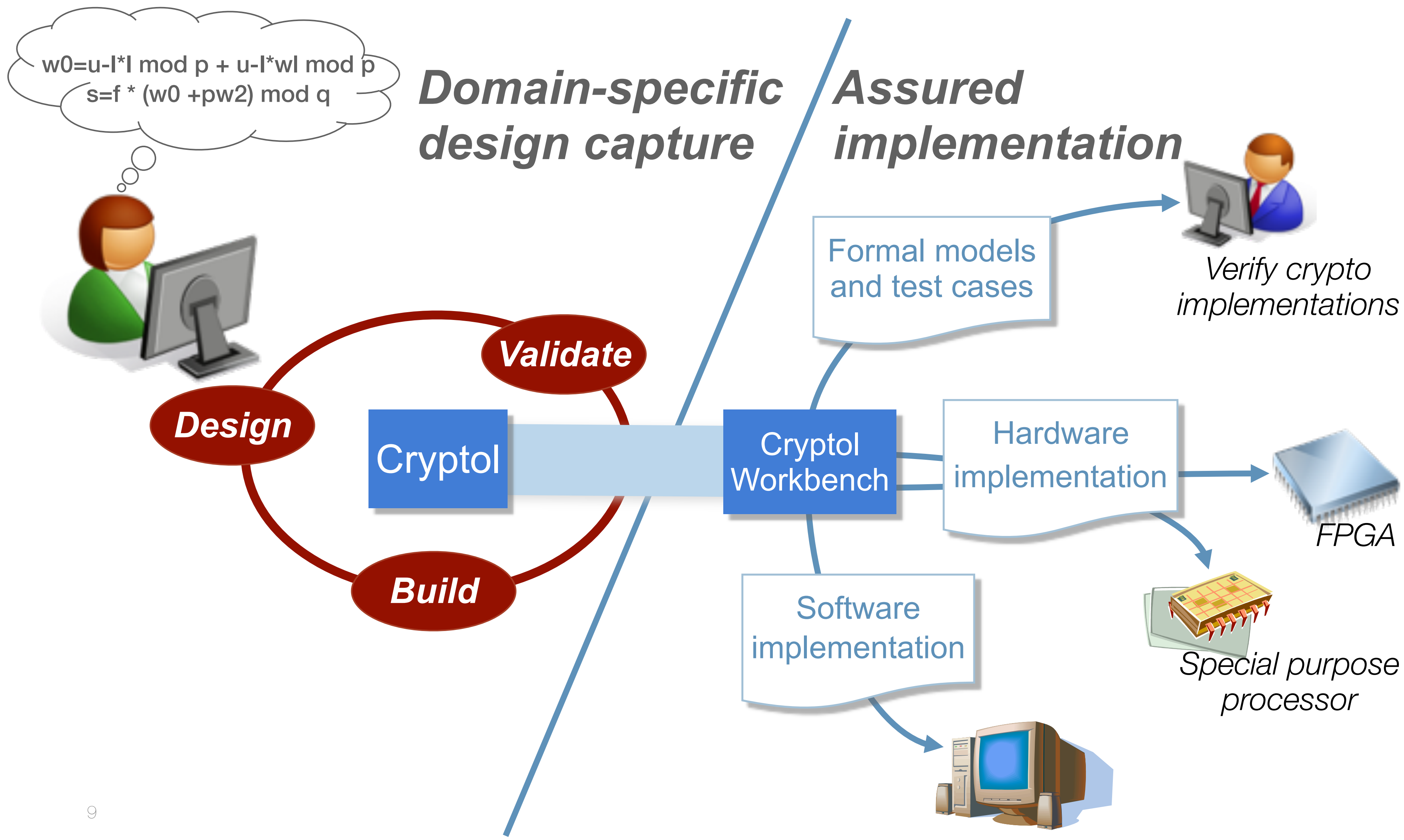
- Bit manipulation

```
ext : {n} (fin n) => [n] -> [n+1];
ext(x) = x # zero;

trim : {n} (fin n) => [n+1] -> [n];
trim(x) = reverse (tail (reverse x));
```

- Addition modulo

```
add_mod : {n} (fin n) => ([n],[n],[n]) -> [n];
add_mod(x,y,p) = trim((ext x + ext y) % ext p);
```

One specification - Many uses



VHDL

C Language

LLVM

Java

Cryptol

Verification Ecosystem

Verification Strategy

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

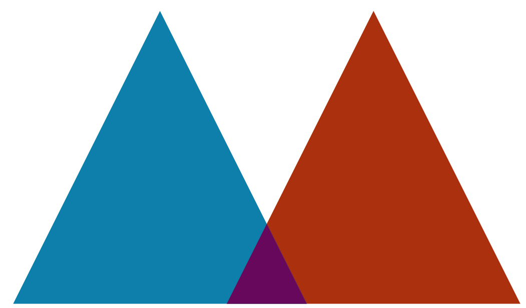
Implementation A   Implementation B

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

ABC   Yices   Rewriting

Verification Strategy

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



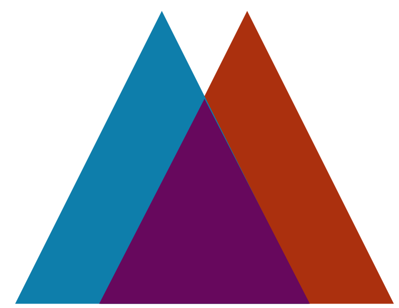
Implementation A   Implementation B

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

ABC   Yices   Rewriting

Verification Strategy

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



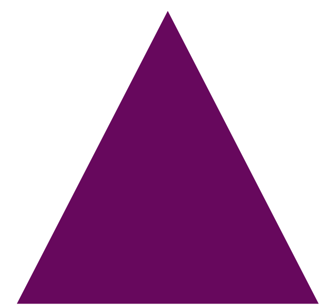
Implementation A Implementation B

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

ABC Yices Rewriting

Verification Strategy

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



Implementation A   Implementation B

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

ABC   Yices   Rewriting

Suite B Verification Efforts

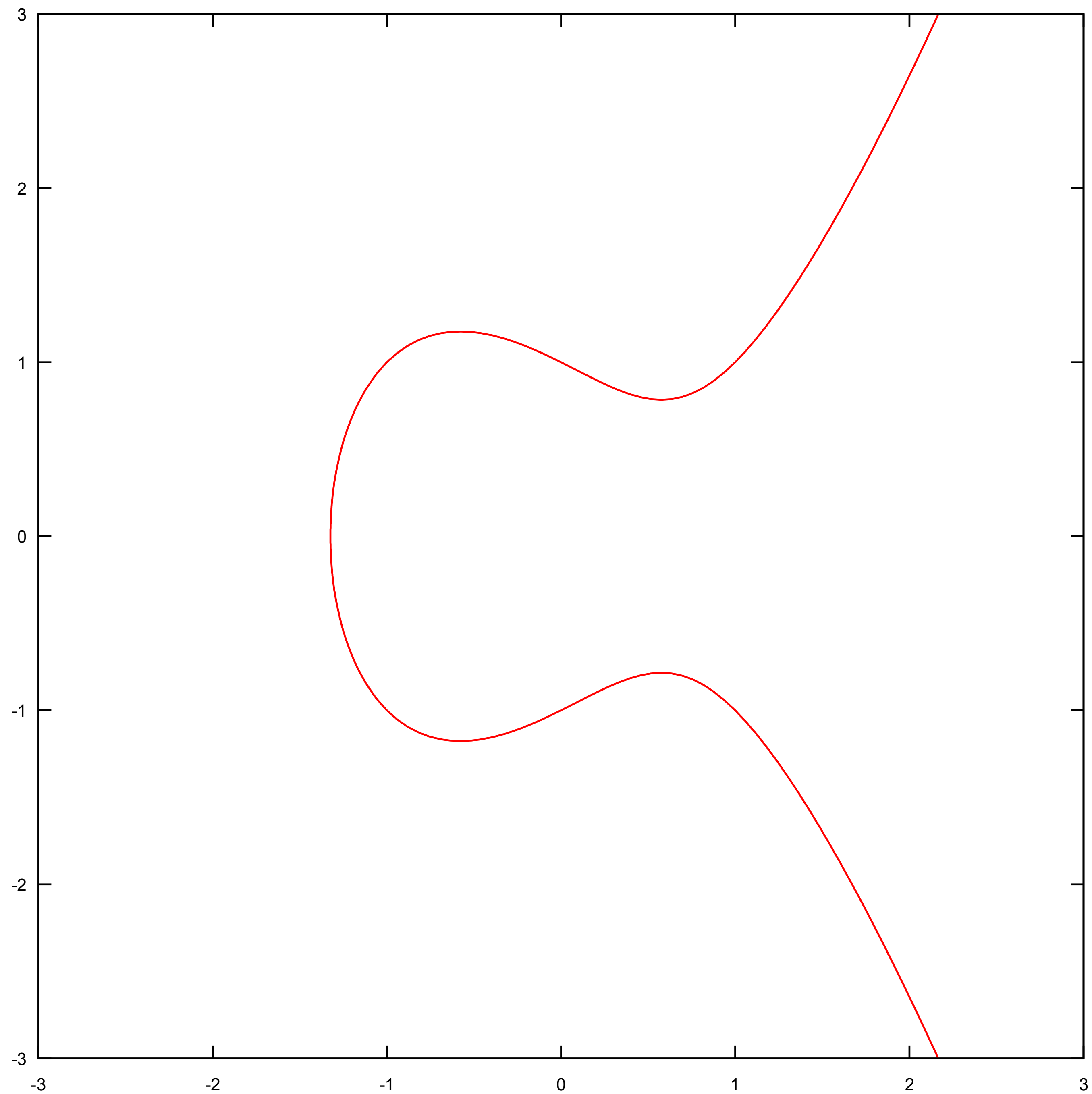
	Role	Implementation	Lines of Code
AES-128	Symmetric Key Cipher	BouncyCastle (Java)	817
SHA-384	Secure Hash Function	libgcrypt (C)	423
ECDSA (P-384)	Digital Signature Scheme	galois (Java)	2348

Suite B Problem Sizes

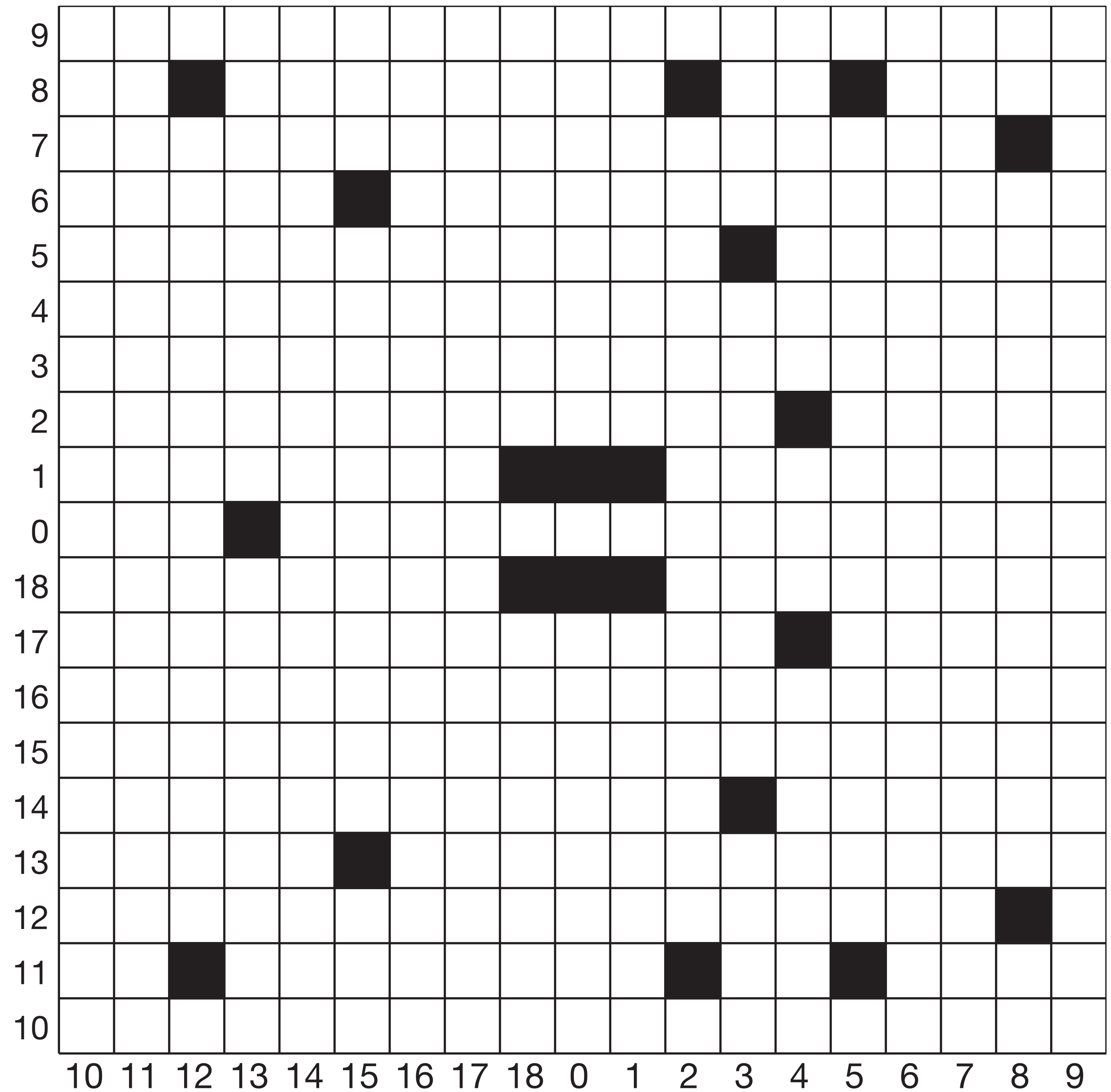
	Lines of Code	AIG Size	Decomposition Steps Required	Verification Time
AES-128 BouncyCastle AESFastEngine	817	1MB	None needed Fully automatic	40 min
SHA-384 libgcrypt	423	3.2MB	12 steps All solved via SAT	160 min
ECDSA (P-384) (galois)	2348	More than 5GB	48 steps Multiple tactics required	10 min

What is an Elliptic Curve?

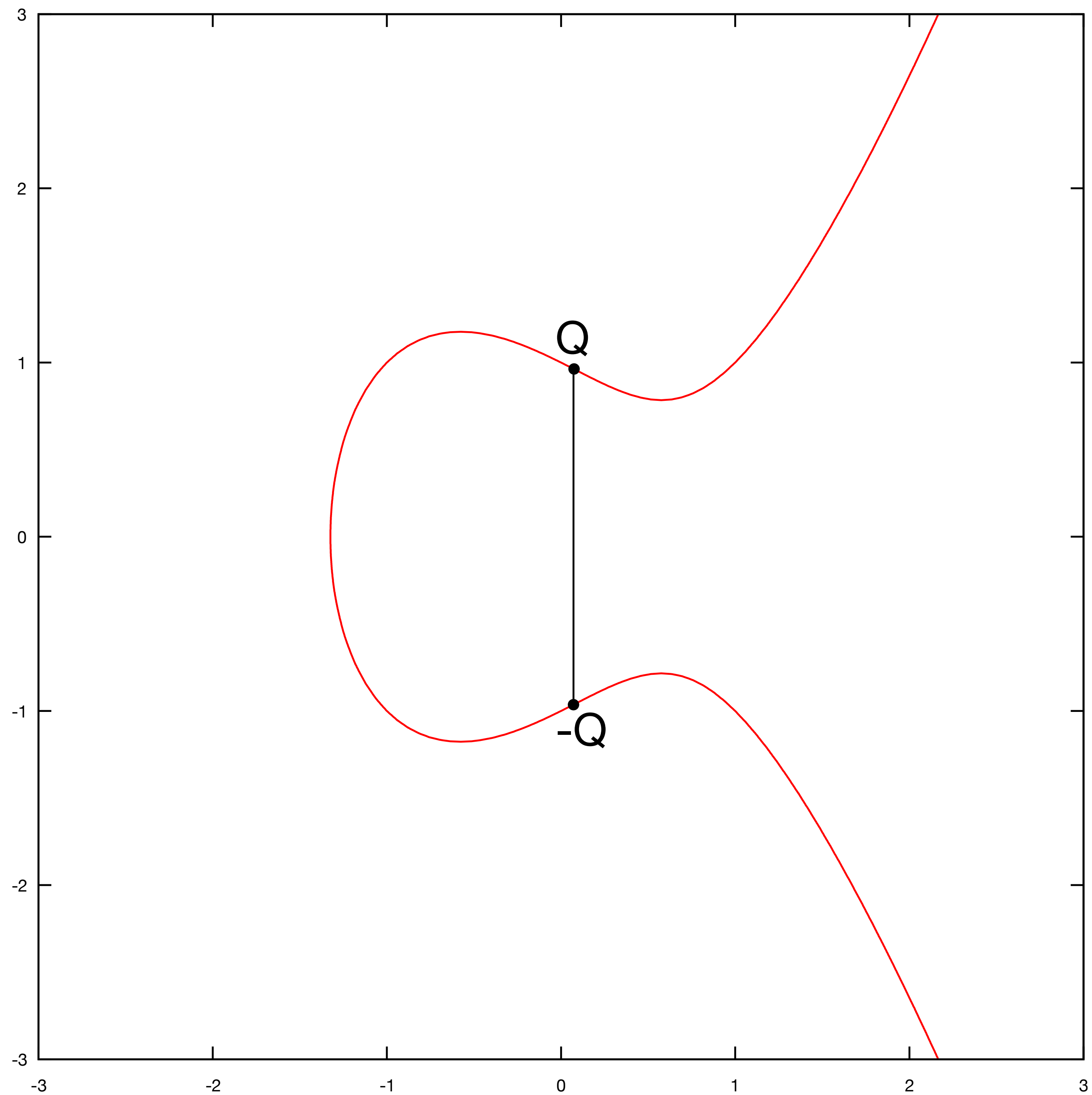
$$y^2 = x^3 + ax + b$$



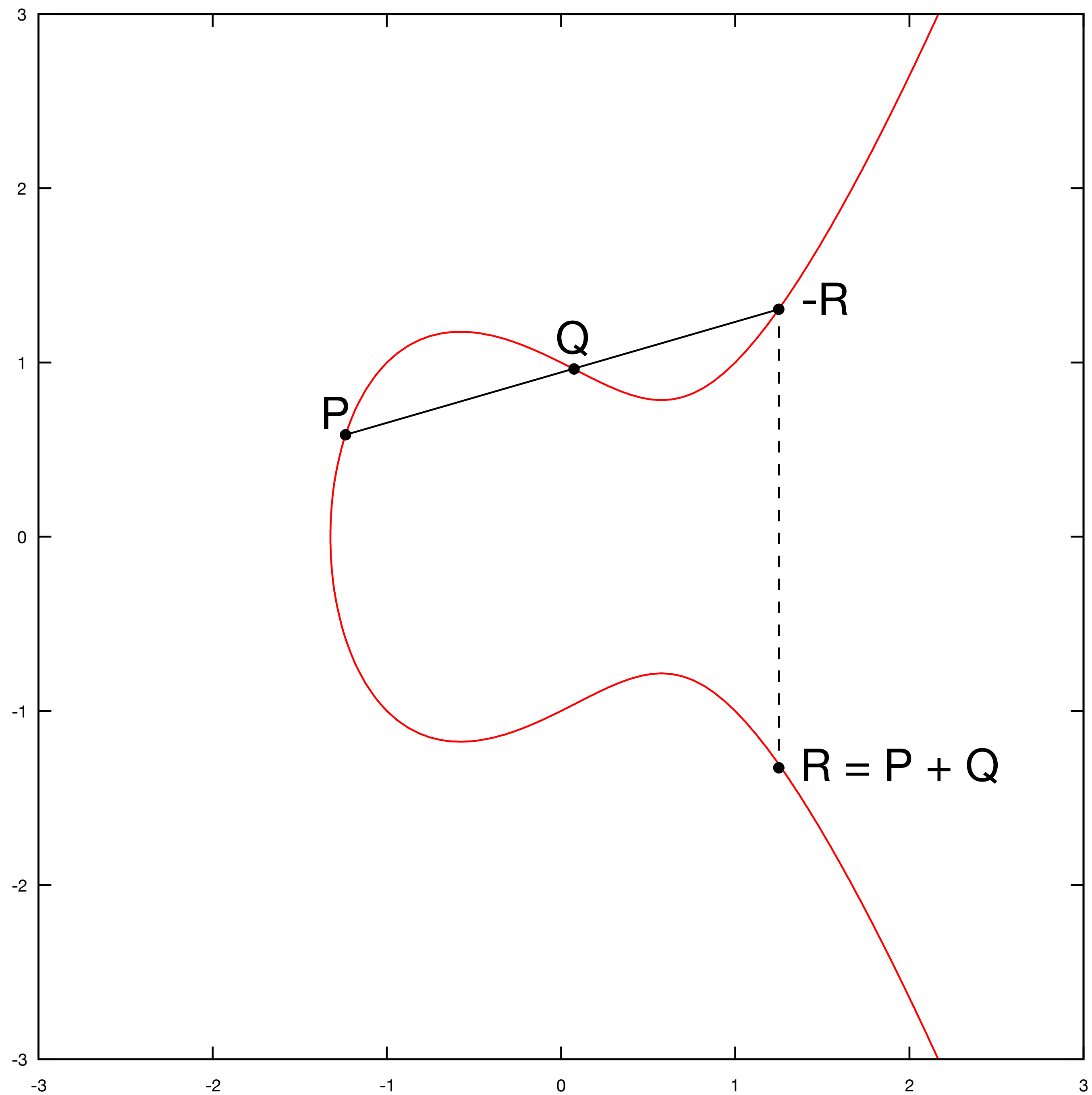
$$y^2 = x^3 - x + 1$$


9 8 7 6 5 4 3 2 1 0 18 17 16 15 14 13 12 11 10
10 11 12 13 14 15 16 17 18 0 1 2 3 4 5 6 7 8 9

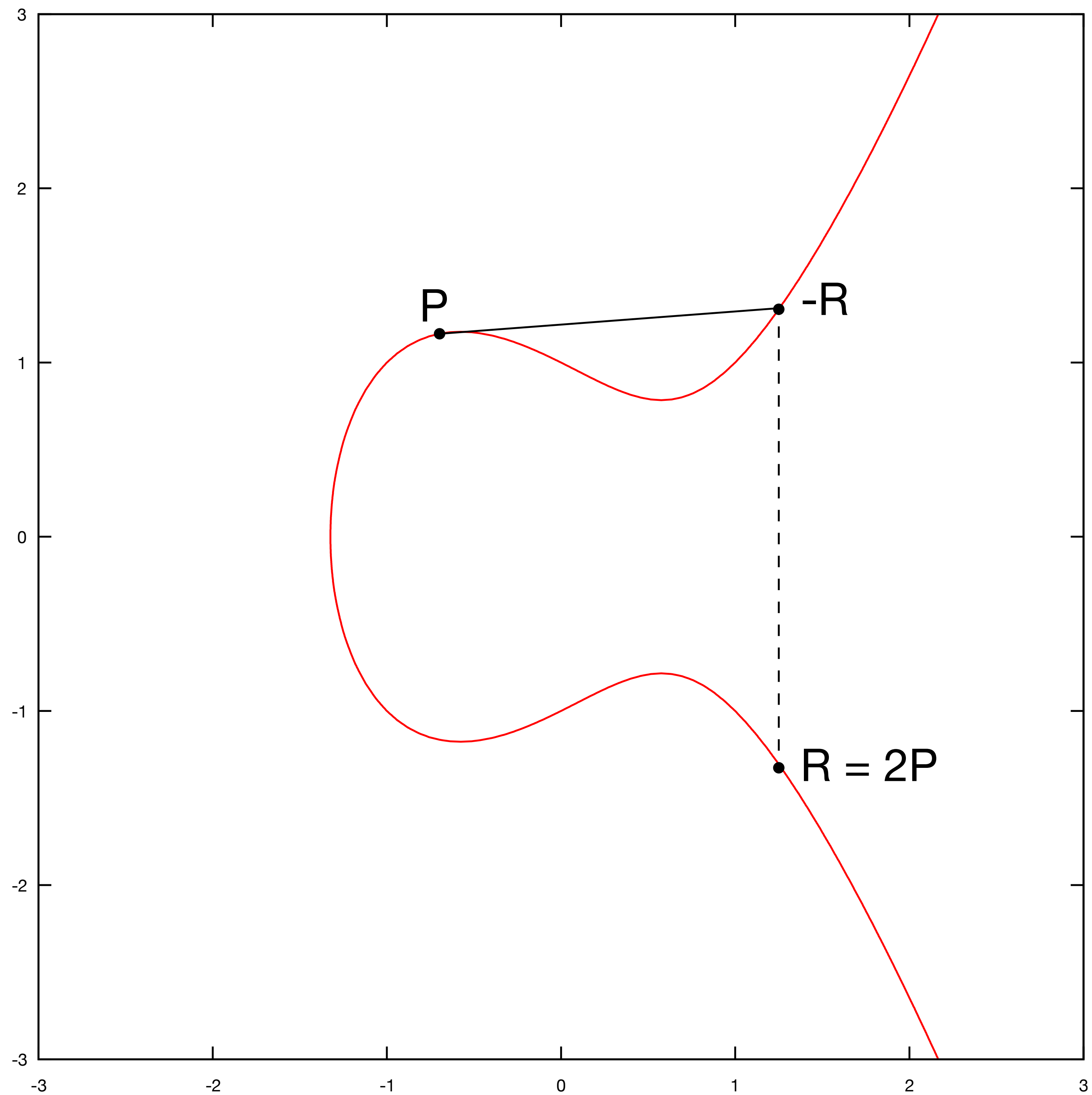
$$y^2 = x^3 - x + 1$$



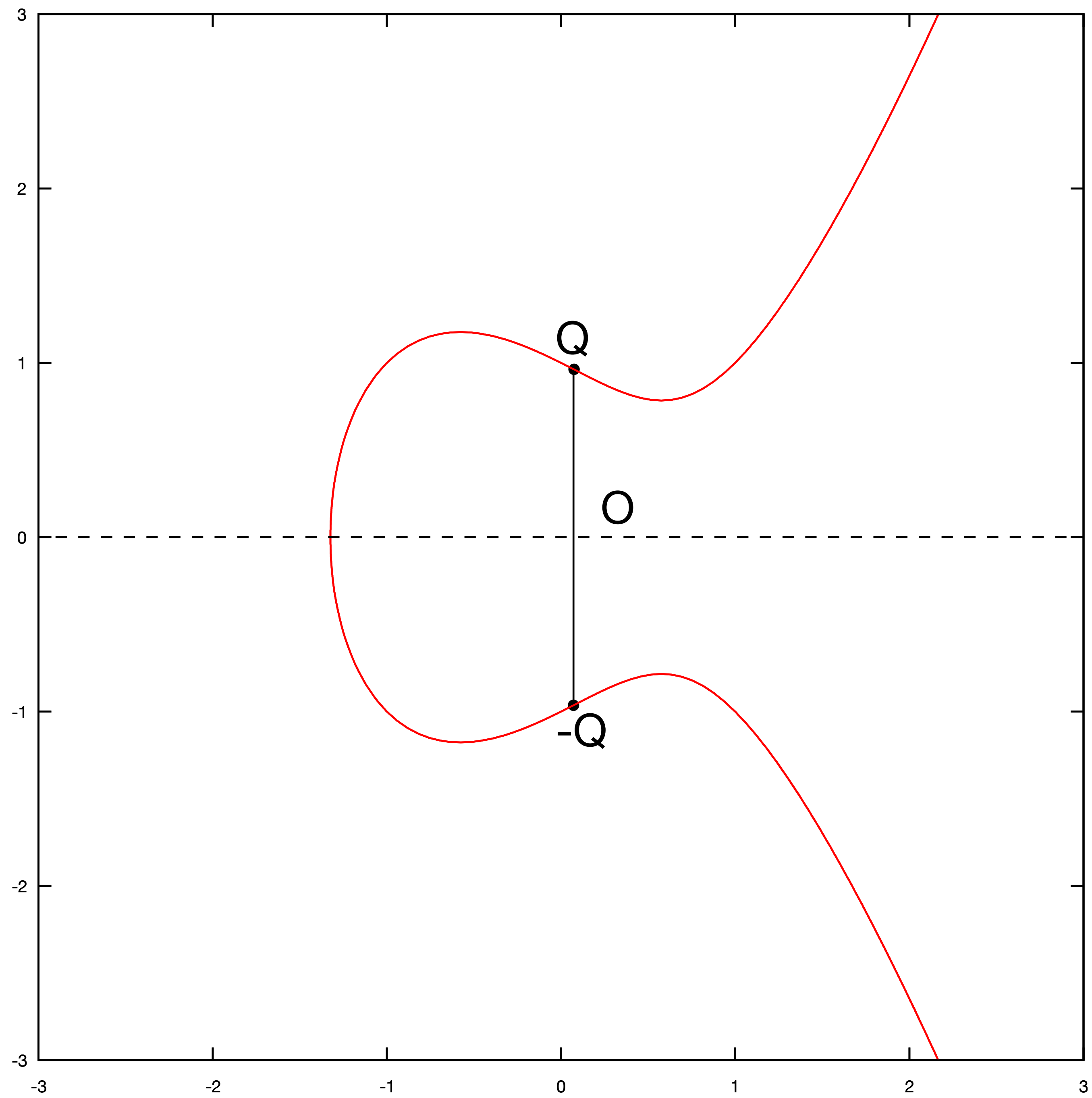
$$y^2 = x^3 - x + 1$$



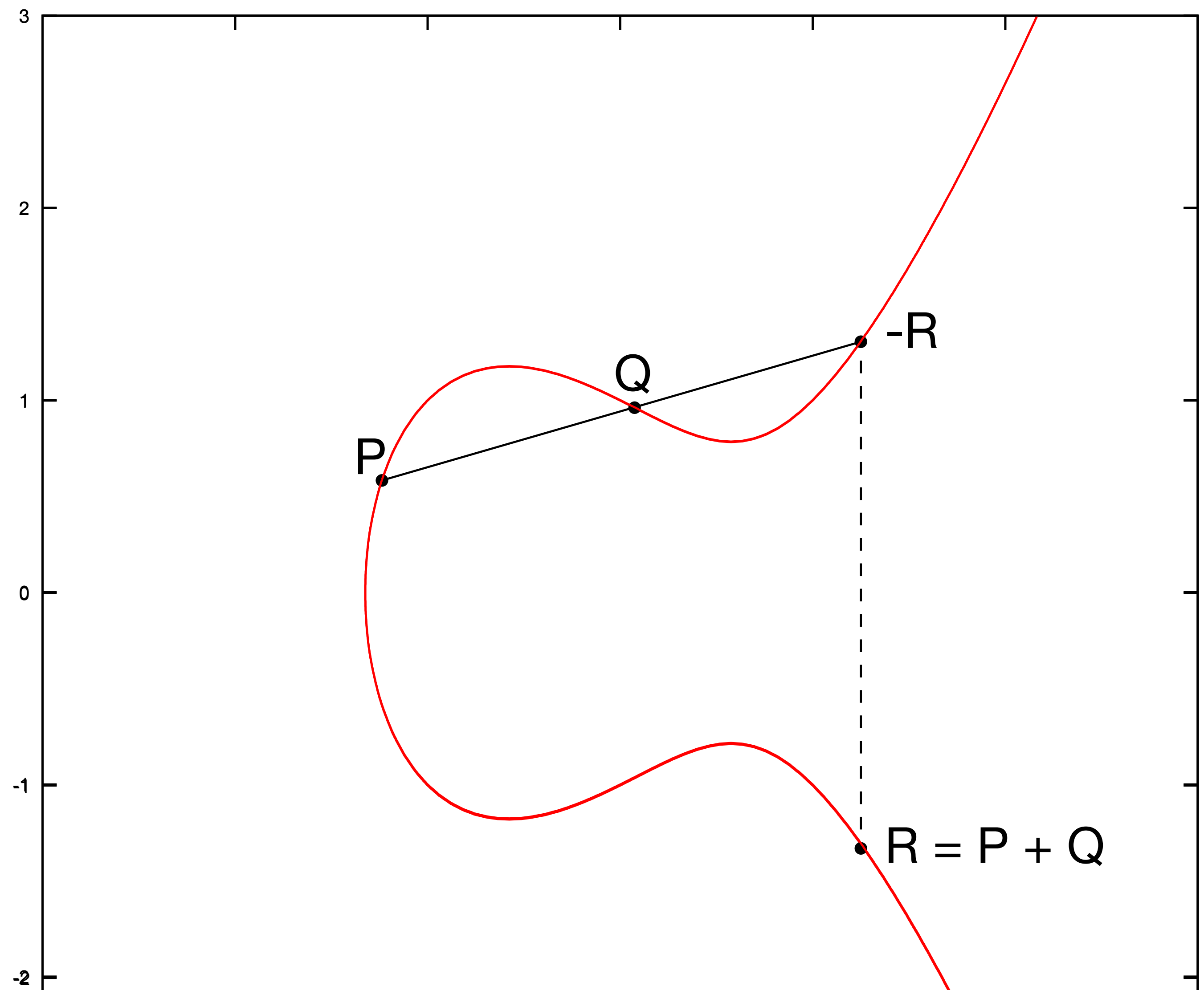
$$y^2 = x^3 - x + 1$$



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



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

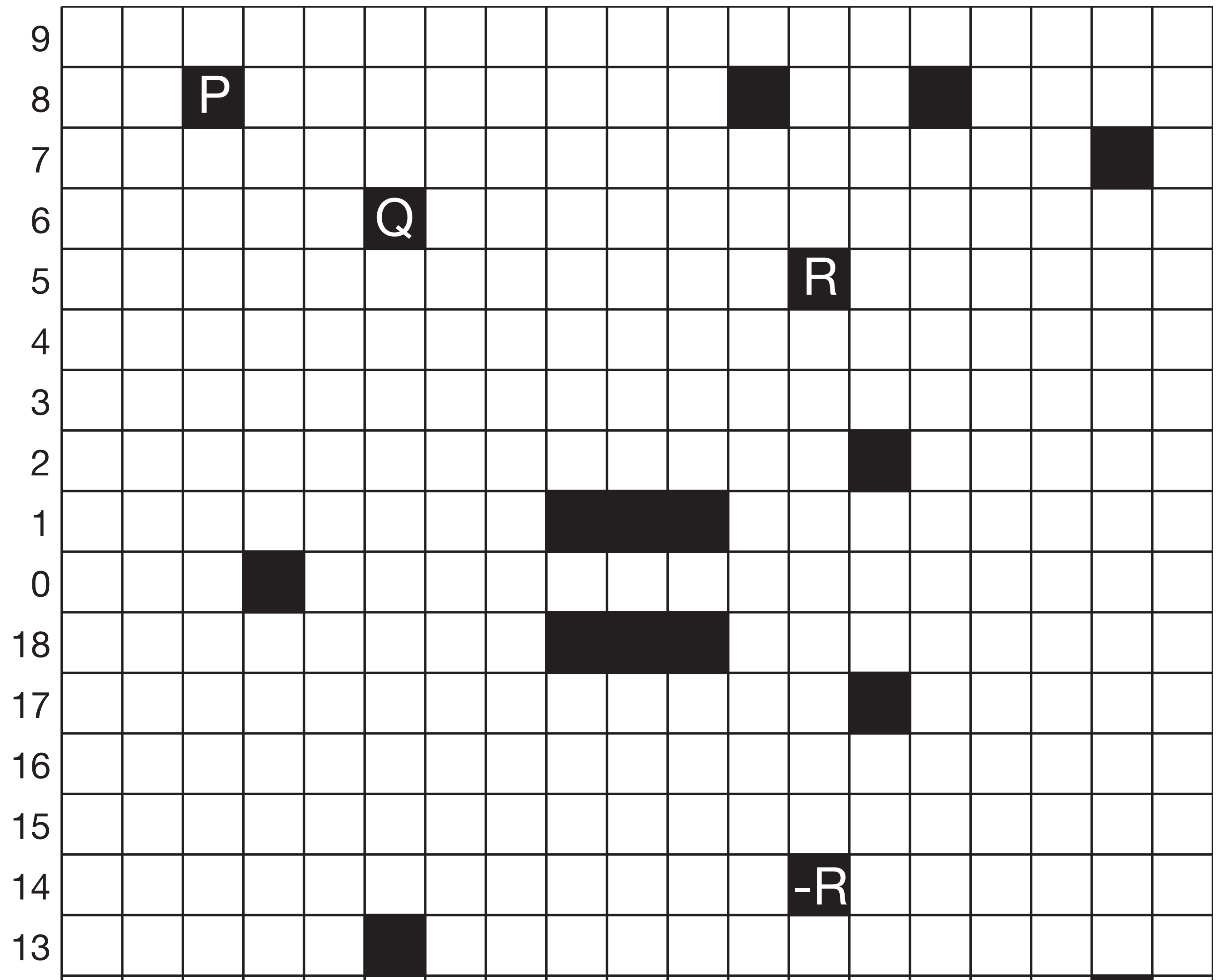


$$P + Q = (R_x, R_y)$$

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

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

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



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

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

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

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

Friday, May 11, 12

Use in Cryptography

- For large elliptic curves, scalar multiplication is a one-way function:

$$Q = k \cdot P$$

(Easy to compute $k \cdot P$; hard to compute Q/P)

- This operation is used to implement ECDSA and ECDH. (digital signatures and key agreement)

NIST P384 Curve

ECC is a family of algorithms, with many options.

- We implemented the NIST P-384 curve.
- Part of NSA Suite B.

Symmetric Key Size (bits)	Elliptic Curve Key Size (bits)	RSA Key Size (bits)
128	256	3072
192	384	7680
256	521	15360

NIST Recommended Key Sizes

NIST P384 Curve

- Prime field P_{384}

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

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

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

Project Goals

- Create an efficient verified implementation of ECDSA over NIST P-384 curve in Java.
  - Use known optimizations such as twin multiplication, projective coordinates, optimized field arithmetic.
  - Specification can use the same algorithms as the implementation. It doesn't have to start from first principals.
  - Implementation uses many low-level tricks for improving efficiency.

Implementing ECC

Cryptographic Protocols

ECDSA	**ECDH**
Digital Signatures	Key Agreement

One Way Functions

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

Point Operations

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

Field Operations

Multiplication	Squaring	Division
Addition	Subtraction	Doubling

Implementing ECC

Cryptographic Protocols

ECDSA — Digital Signatures

ECDH — Key Agreement

One Way Functions

$R = s \cdot P$ — 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

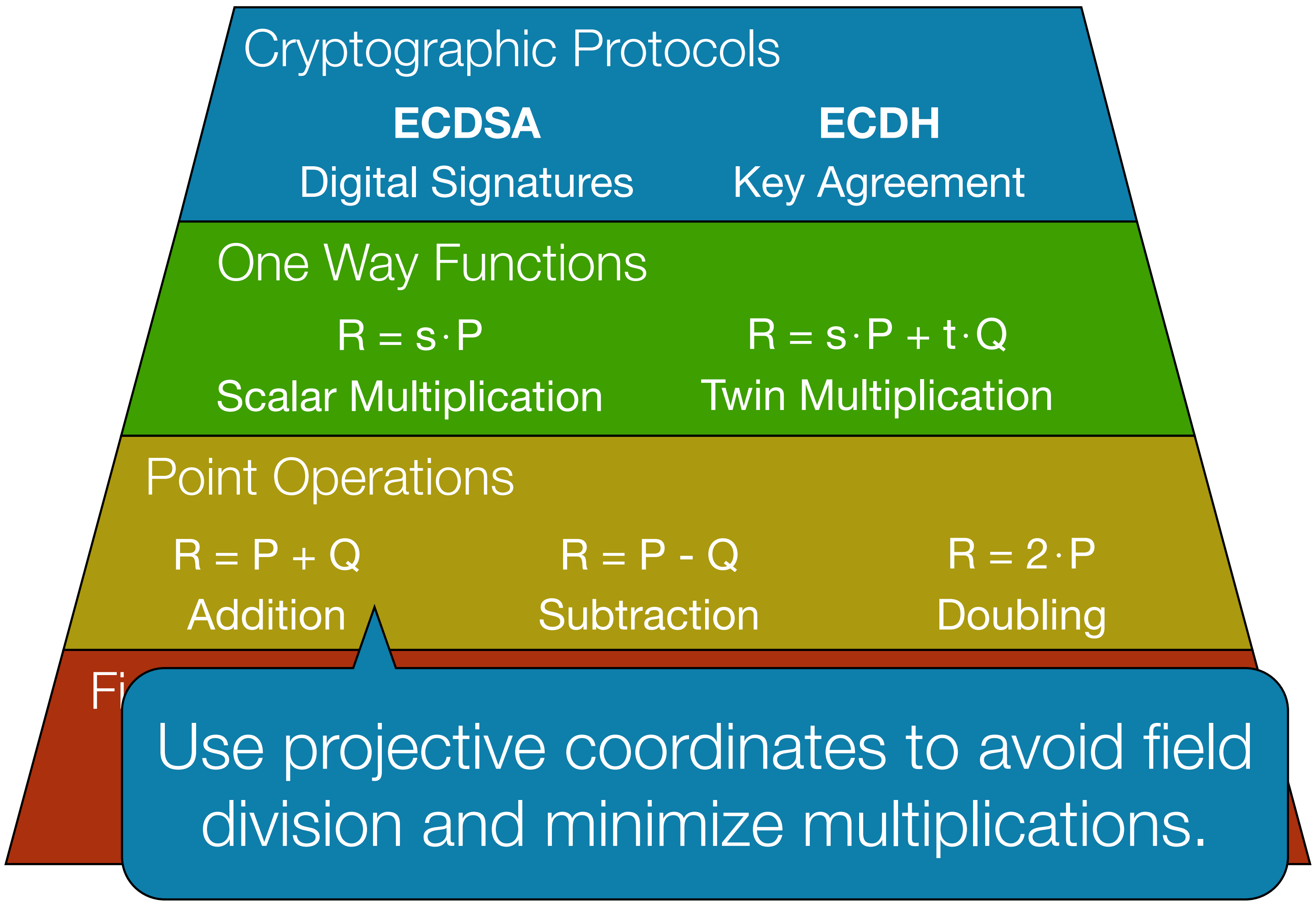
Multiplication, Addition

Squaring, Subtraction

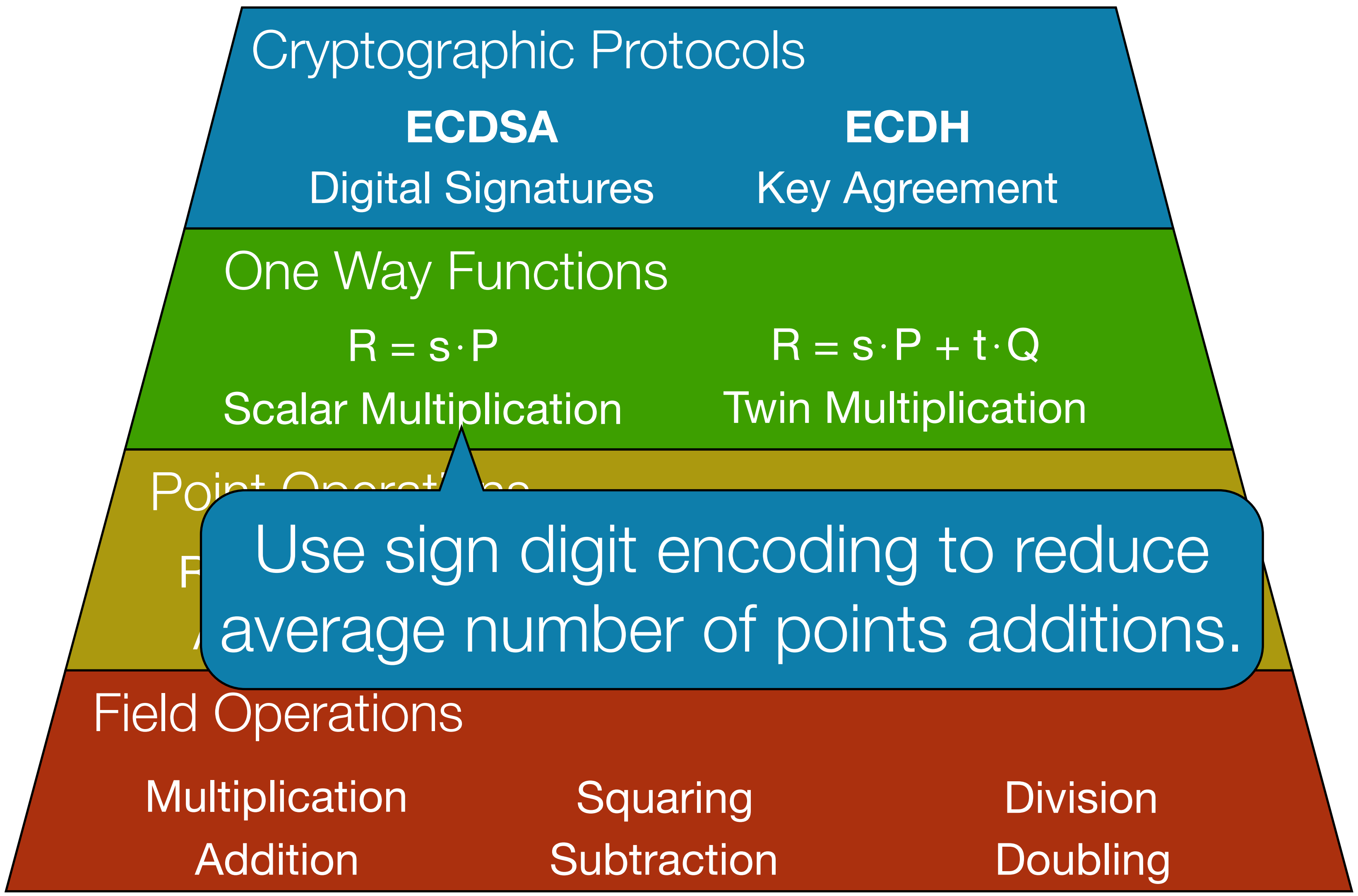
Division, Doubling

Optimize modular reduction for specific field prime.

Implementing ECC



Implementing ECC



Implementing ECC

Cryptographic Protocols

ECDSA Digital Signatures

ECDH Key Agreement

unctions

$R = s \cdot P + t \cdot Q$ Twin Multiplication

ication

Use twin multiplication when needed.

Point Operations

$R = P + Q$ Addition

$R = P - Q$ Subtraction

$R = 2 \cdot P$ Doubling

Field Operations

Multiplication, Addition

Squaring, Subtraction

Division, Doubling

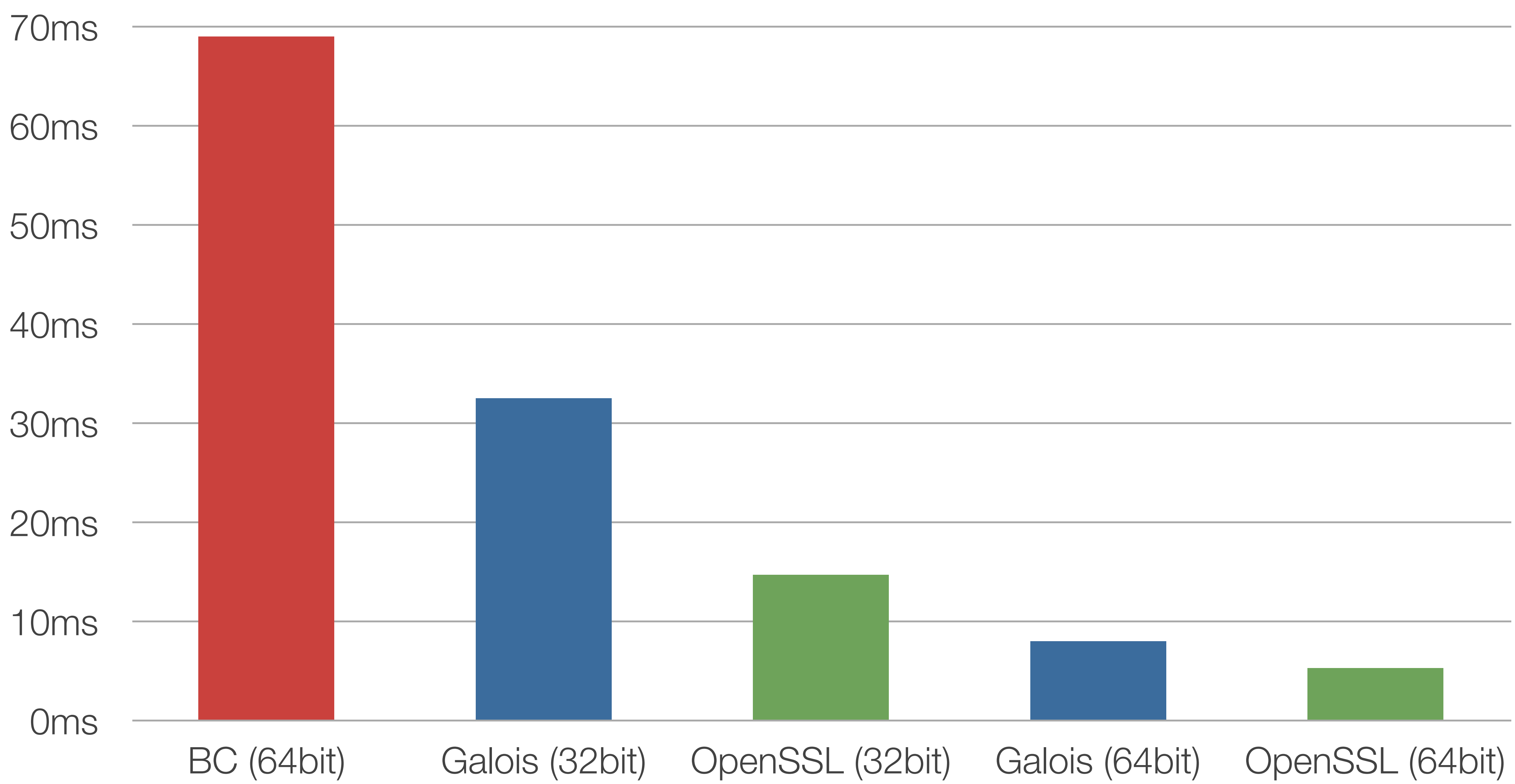
Field addition in Java


```
/** Assigns z = x + y (mod field_prime). */
public void field_add(int[] z, int[] x, int[] y) {
  if (add(z, x, y) != 0 || leq(field_prime, z)) decFieldPrime(z);
}

int[] field_prime = { -1, 0, 0, -1, -2, -1, -1, -1, -1, -1, -1, -1 };

static final long LONG_MASK = 0xFFFFFFFFL;

/** Assigns z = x + y and returns carry. */
protected int add(int[] z, int[] x, int[] y) {
  long c = 0;
  for (int i = 0; i != z.length; ++i) {
    c += (x[i] & LONG_MASK) + (y[i] & LONG_MASK);
    z[i] = (int) c; c = c >> 32;
  }
  return (int) c;
}

static boolean leq(int[] x, int[] y) { ... }
protected int decFieldPrime(int[] x) { ... }
```

Field addition in Cryptol


```
p384_field_add : ([384],[384]) -> [384];
p384_field_add(x,y) = mod_add(x,y,384_prime);

p384_prime : [384];
p384_prime = 2 ** 384 - 2 ** 128 - 2 ** 96 + 2 ** 32 - 1;

mod_add : {n} (fin n) => ([n],[n],[n]) -> [n];
mod_add(x,y,p) = if sum >= ext(p) then
                   trim(sum) - p
                 else
                   trim(sum)
  where sum = ext(x) + ext(y);

ext : {n} (fin n) => [n] -> [n+1];
ext(x) = x # zero;

trim : {n} (fin n) => [n+1] -> [n];
trim(x) = reverse (tail (reverse x));
```

ECC Benchmarks
Sign & Verify



SAWScript:

Language for Compositional Verification

Elliptic Curve Crypto (ECC)

Cryptographic Protocols

ECDSA	**ECDH**
Digital Signatures	Key Agreement

One Way Functions

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

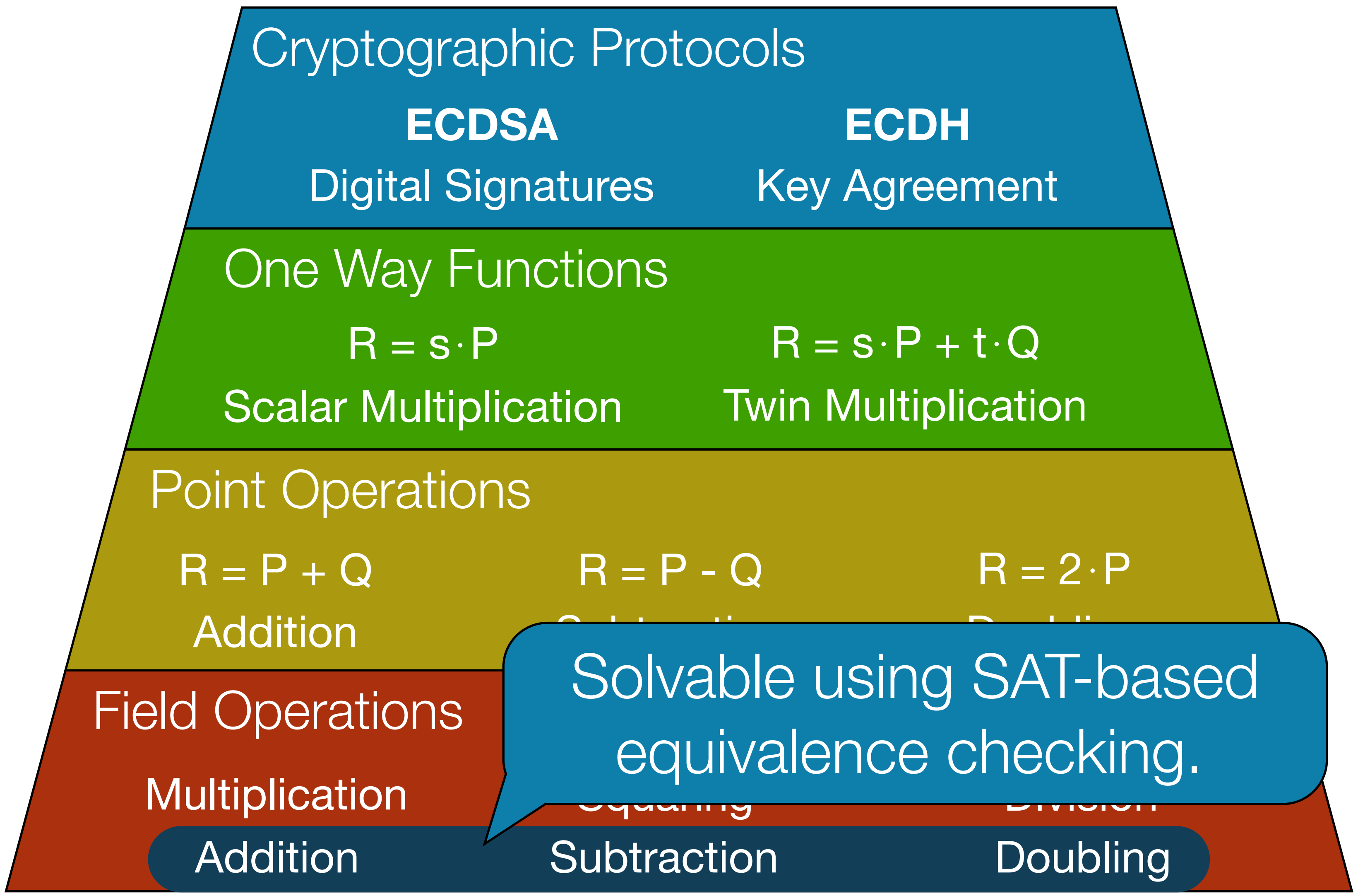
Point Operations

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

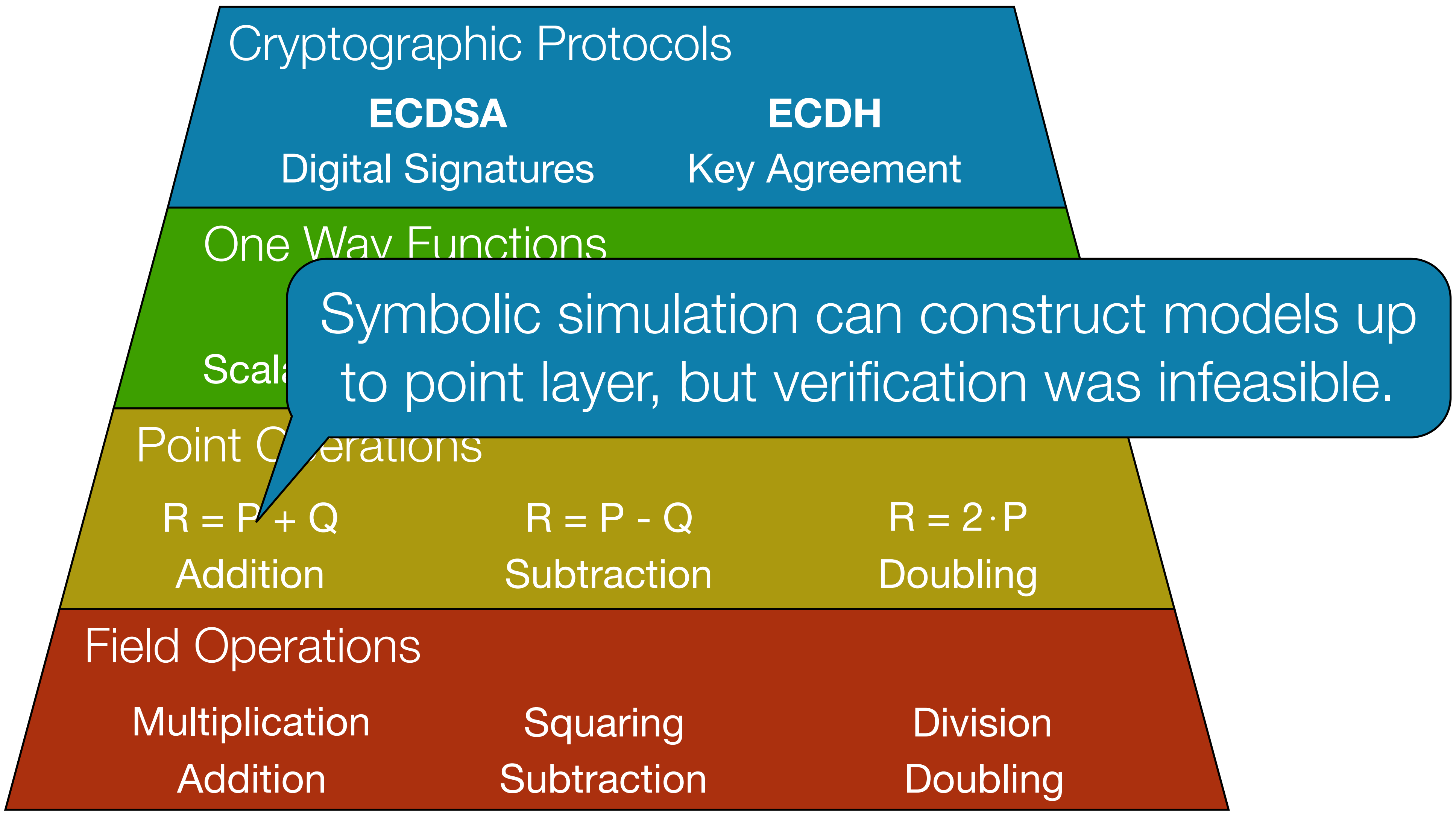
Field Operations

Multiplication	Squaring	Division
Addition	Subtraction	Doubling

Elliptic Curve Crypto (ECC)



Elliptic Curve Crypto (ECC)



SAWScript Capabilities

- Allows behavior of Java methods, including side effects, to be precisely defined using Cryptol functions.
- Method specifications are used in two ways:
  - As statements to be proven.
  - As lemmas to help verify later methods.
- SAWScript has a simple tactic language for user control over verification steps.

Method Specification Requirements

- Cryptol types for Java variables, including lengths for arrays.
- Assumptions on inputs.
- Which references can alias other references.
- Expected results when method terminates.
- Optionally, postconditions at intermediate breakpoints within method.
- Tactics for verifying method.

field_add Specification

Import specification from Cryptol

```
extern SBV ref_field_add("sbv/p384_field_add.sbv")
  : ([384],[384]) -> [384];

let field_prime = <| 2^384 - 2^128 - 2^96 + 2^32 - 1 |> : [384];

method com.galois.ecc.P384ECC64.field_add
{
  var z, x, y :: int[12];
  mayAlias { z, x, y };

  var this.field_prime :: int[12];
  assert valueOf(this.field_prime) := split(field_prime) : [12][32];

  let jx = join(valueOf(x));
  let jy = join(valueOf(y));
  ensure valueOf(z) := split(ref_field_add(jx, jy)) : [12][32];

  verify { rewrite; yices; };
};
```

field_add Specification


```
extern SBV ref_field_add("sbv/p384_field_add.sbv")
  : ([384],[384]) -> [384];

let field_prime = <| 2^384 - 2^128 - 2^96 + 2^32 - 1 |> : [384];

method com.galois.ecc.P384ECC64.field_add
{
  var z, x, y :: int[12];
  mayAlias { z, x, y };

  var this.field_prime :: int[12];
  assert valueOf(this.field_prime) := split(field_prime) : [12][32];

  let jx = join(valueOf(x));
  let jy = join(valueOf(y));
  ensure valueOf(z) := split(ref_field_add(jx, jy)) : [12][32];

  verify { rewrite; yices; };
};
```

Constants support arbitrary bitwidths.

`field_add` Specification

```
extern SBV ref_field_add("sbv/p384_field_add.sbv")
  : ([384],[384]) -> [384];

let field_prime = <| 2^384 - 2^128 - 2^96 + 2^32 - 1 |> : [384];

method com.galois.ecc.P384ECC64.field_add
{
  var z, x, y :: int[12];
  mayAlias { z, x, y };

  var this.field_prime :: int
  assert valueOf(this.field_prime) := split(field_prime) : [12][32];

  let jx = join(valueOf(x));
  let jy = join(valueOf(y));
  ensure valueOf(z) := split(ref_field_add(jx, jy)) : [12][32];

  verify { rewrite; yices; };
};
```

Declare arguments.

field_add Specification

```
extern SBV ref_field_add("sbv/p384_field_add.sbv")
  : ([384],[384]) -> [384];

let field_prime = <| 2^384 - 2^128 - 2^96 + 2^32 - 1 |> : [384];

method com.galois.ecc.P384ECC64.field_add
{
  var z, x, y :: int[12];
  mayAlias { z, x, y };

  var this.field_prime :: int[12];
  assert valueOf(this.field_prime) := split(field_prime) : [12][32];

  let jx = join(valueOf(x));
  let jy = join(valueOf(y));
  ensure valueOf(z) := split(ref_fi

  verify { rewrite; yices; };
};
```

Declare initialized field value.

field_add Specification


```
extern SBV ref_field_add("sbv/p384_field_add.sbv")
  : ([384],[384]) -> [384];

let field_prime = <| 2^384 - 2^128 - 2^96 + 2^32 - 1 |> : [384];

method com.galois.ecc.P384ECC64.field_add
{
  var z, x, y :: int[12];
  mayAlias { z, x, y };

  var this.field_prime :: int[12];
  assert valueOf(this.field_prime) := split(field_prime) : [12][32];

  let jx = join(valueOf(x));
  let jy = join(valueOf(y));
  ensure valueOf(z) := split(ref_field_add(jx, jy)) : [12][32];

  verify { rewrite; yices; };
};
```

Define post-condition.

field_add Specification

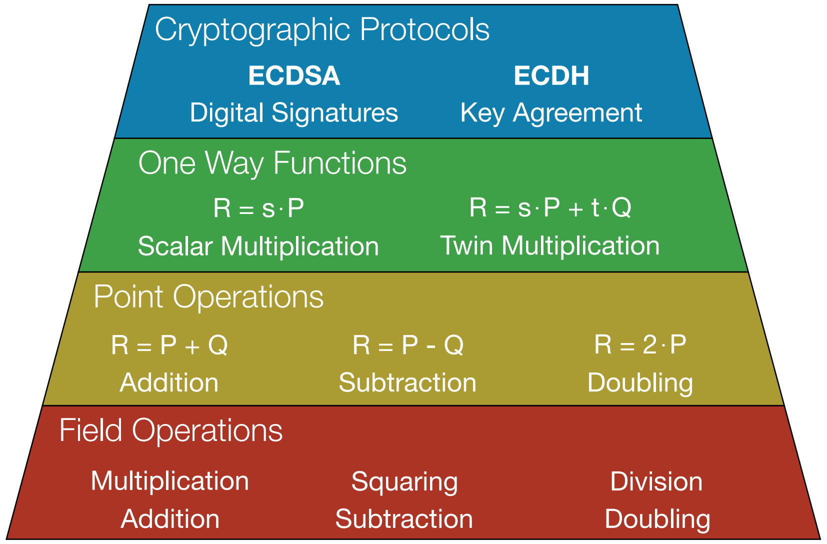

```
extern SBV ref_field_add("sbv/p384_field_add.sbv")
  : ([384],[384]) -> [384];

let field_prime = <| 2^384 - 2^128 - 2^96 + 2^32 - 1 |> : [384];

method com.galois.ecc.P384ECC64.field_add
{
  var z, x, y :: int[12];
  mayAlias { z, x, y };

  var this.field_prime :: int[12];
  assert valueOf(this.field_prime) := split(field_prime) : [12][32];

  let jx = join(valueOf(x));
  let jy = join(valueOf(y));
  ensure valueOf(z) := split(ref_field_add(jx, jy)) : [12][32];

  verify { rewrite; yices; };
};
```

Specify tactics.

Compositional Verification

- Once a specification is defined, it can be used to simplify later methods.

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



- Rather than execute code for field_add, simulator simply replaces value at `t4` with an application of Cryptol `ref_field_add`.

ECC Verification Results

Verification Results

- We were able to successfully verify the Java implementation against a Cryptol specification using SAWScript.
  - Specification can use the same algorithms as the implementation. It doesn't have to start from first principals.
  - Specification can be independently validates using theorem proving where desired.

Verification Statistics

- 48 Method Specifications Total
  - 2 protocol specifications (verify & sign)
  - 8 scalar multiplication specifications.
  - 3 point specifications (add, subtract, double).
  - 20 field specifications.
  - 15 bitvector specifications.
- Total verification time is under 10 minutes.

Found Three Bugs

- 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.

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

```
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.ja

      d = (z
      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).

Verification Features Used

- Rewriter used 30 times (18 in conjunction with another solver).
- Yices used 23 times.
- abc used in 13 times.
- Yices was often faster, but used uninterpreted functions; counterexamples could be spurious.
- Specification with inductive assertions only used once (modular division).

Proof Engineering

- Modified implementation to make verification easier.
  - In large loops, such as scalar multiplication, we moved loop body into a separate function, and verified the body independently.
  - Other minor syntactic changes to make rewriting easier.
  - Code performance was not affected significantly.
- Also modified Cryptol large word multiplication specification to ease verification.
  - Introduces risk of bugs in specification; risk could be reduced by proving properties about specification.

Summary

- We’ve successfully verified 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.

Thanks!