# Software Analysis Workbench (SAW)

## A Tool suite for Compositional Cryptographic Verification

Galois, Inc.
Presented by Joe Hendrix

# Team

Sally Browning

John Matthews

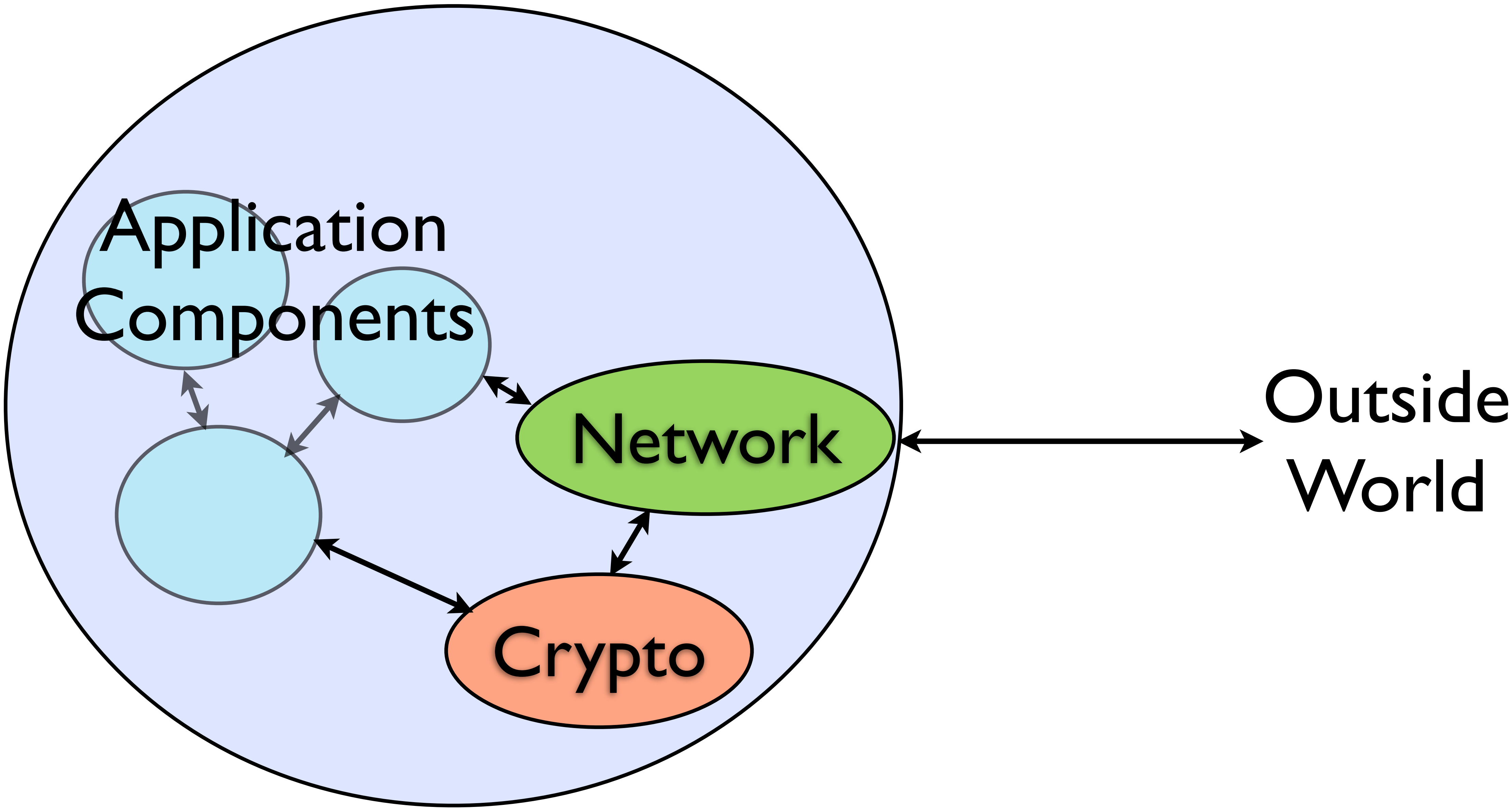
Joe Hendrix

Joel Stanley

Joe Hurd

Aaron Tomb

# Cryptographic Verification

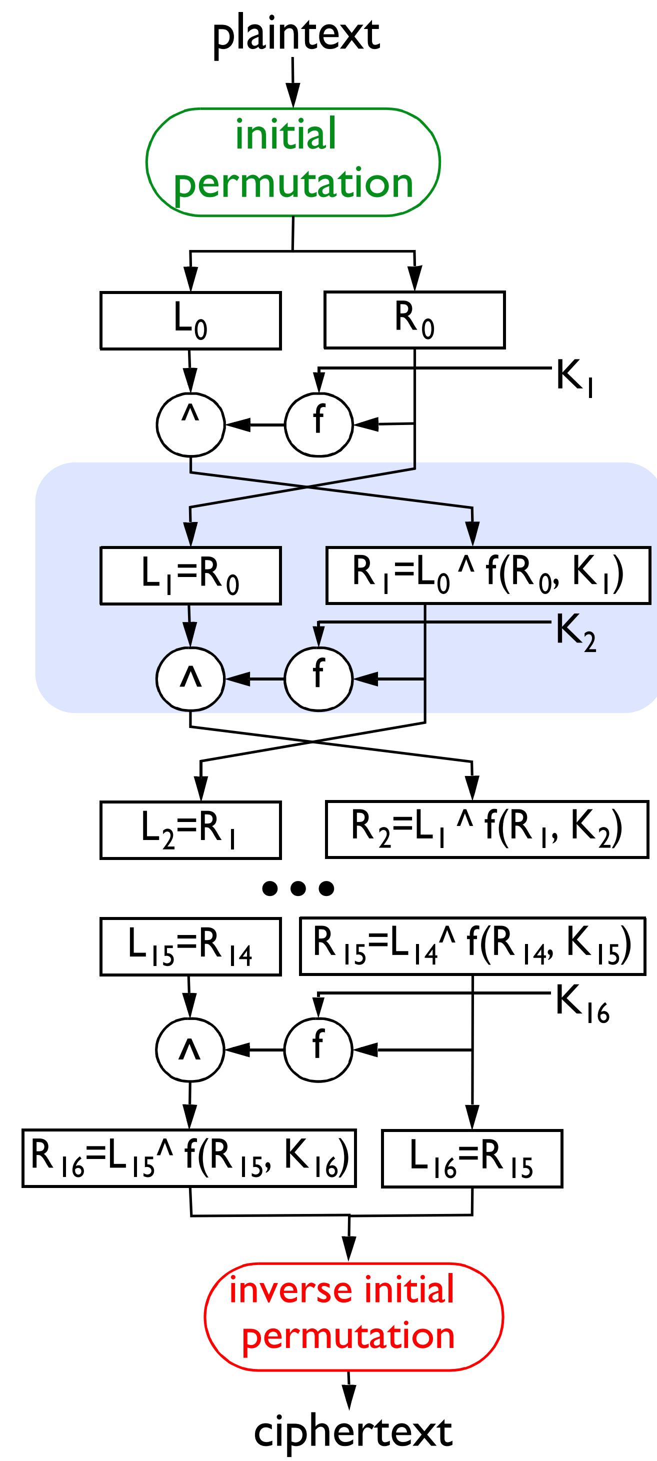



Cryptographic algorithms are a small, but critical component of networked systems.

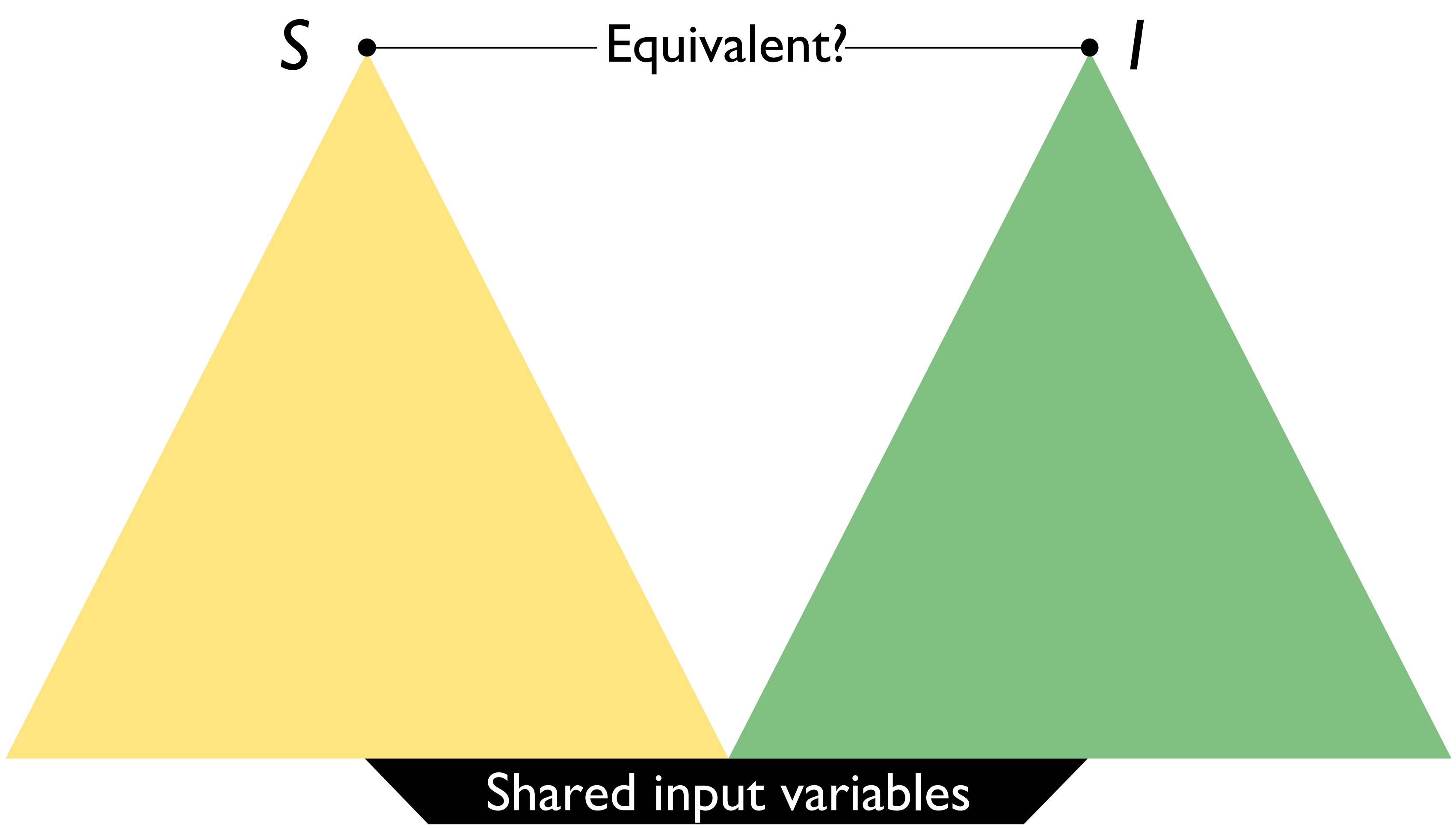
# Structure of a Block Cipher

```
des : ([64],[56]) -> [64];
des (pt, key) = permute (FP, last)
  where {
    pt' = permute(IP, pt);
    iv = [| round(lr, key, rnd)
         || rnd <- [0 .. 15]
         || lr <- [(split pt')] # iv
         |];
    last = join (swap (iv @ 15));
    swap [a b] = [b a];
  };
```

```
round : ([2][32], [56], [4]) -> [2][32];
round([l r], key, rnd) = [r (l^f(r, kx))]
  where {
    kx = expand(key, rnd);
    f(r,k) =
      permute(PP, SBox(k^permute(EP, r)));
  };
```

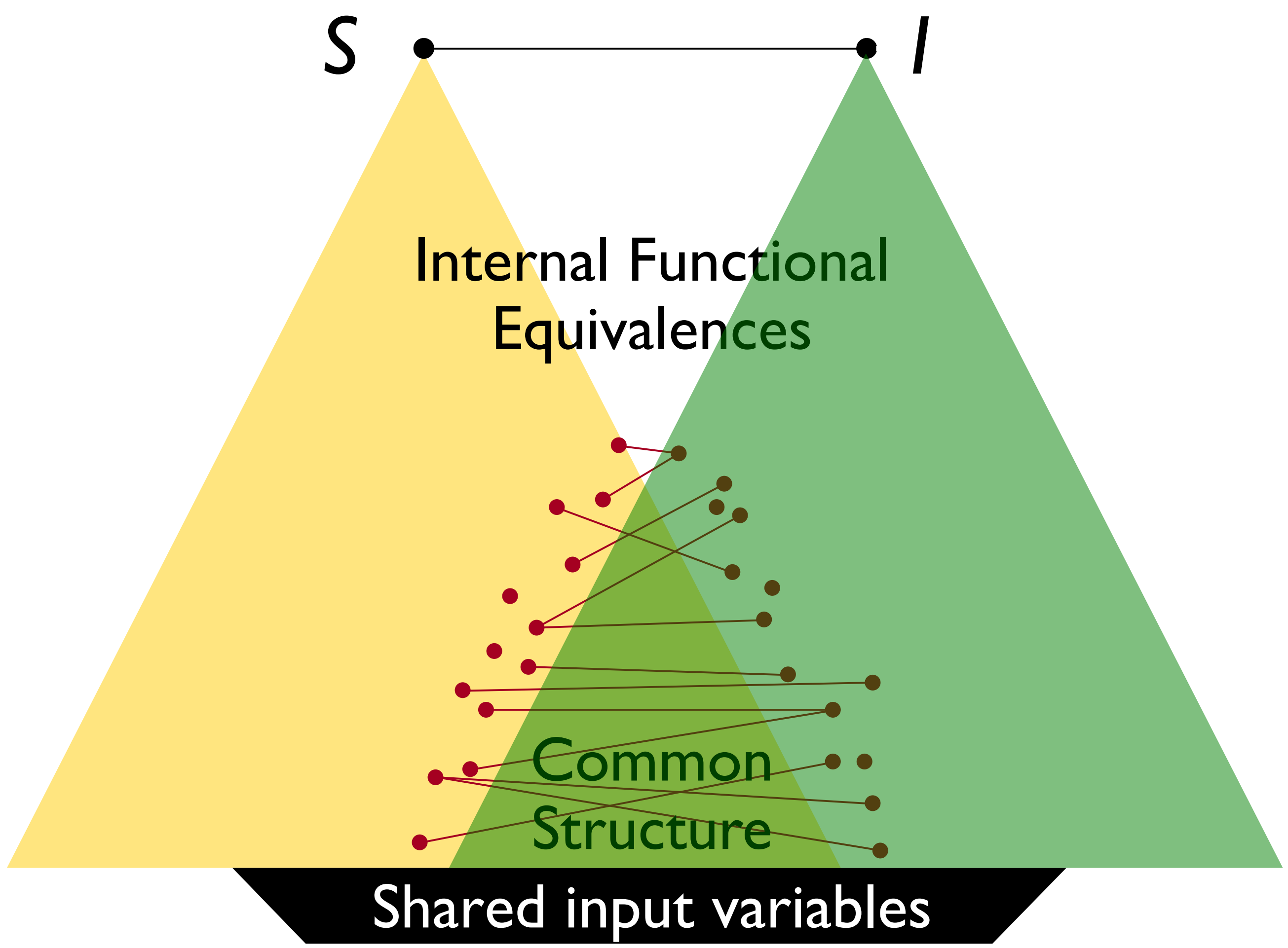


# Equivalence Checking



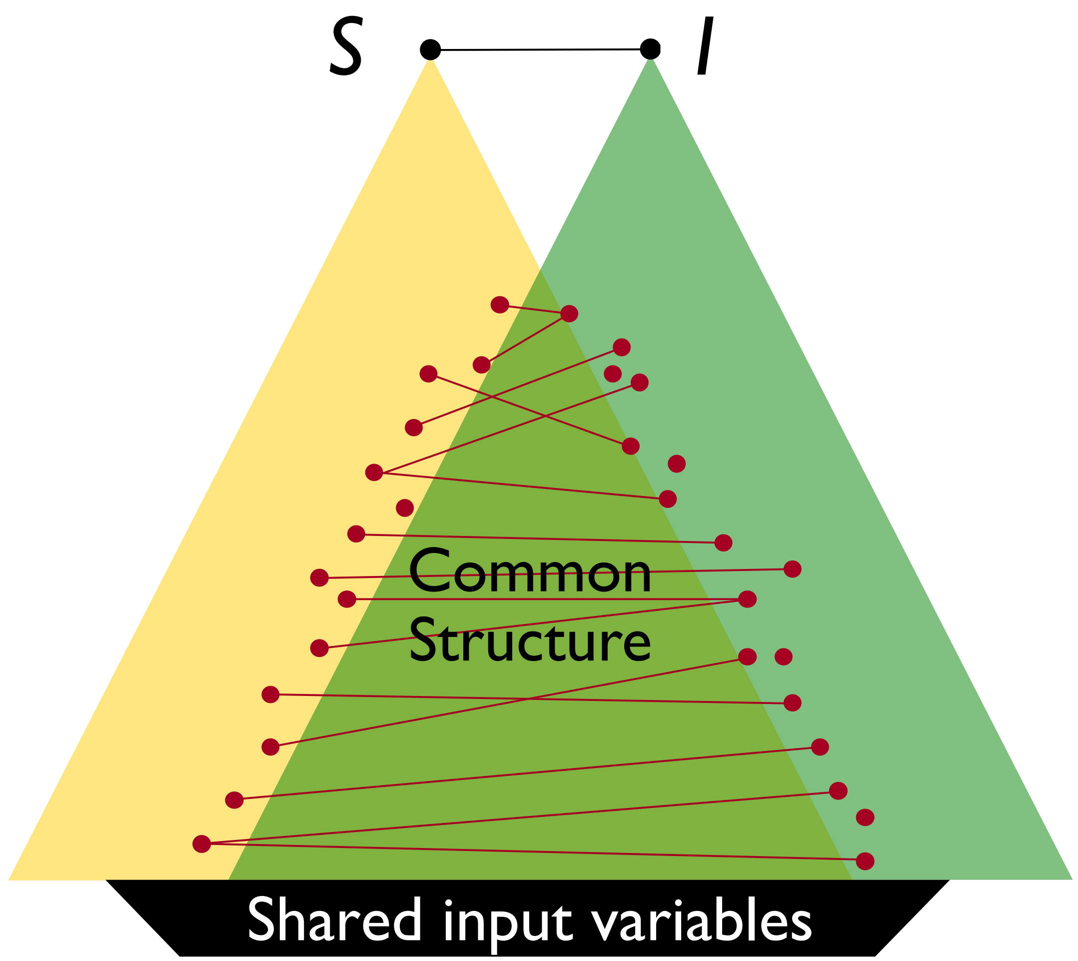

From A. Biere, "SAT in Formal Hardware Verification"

# Equivalence Checking



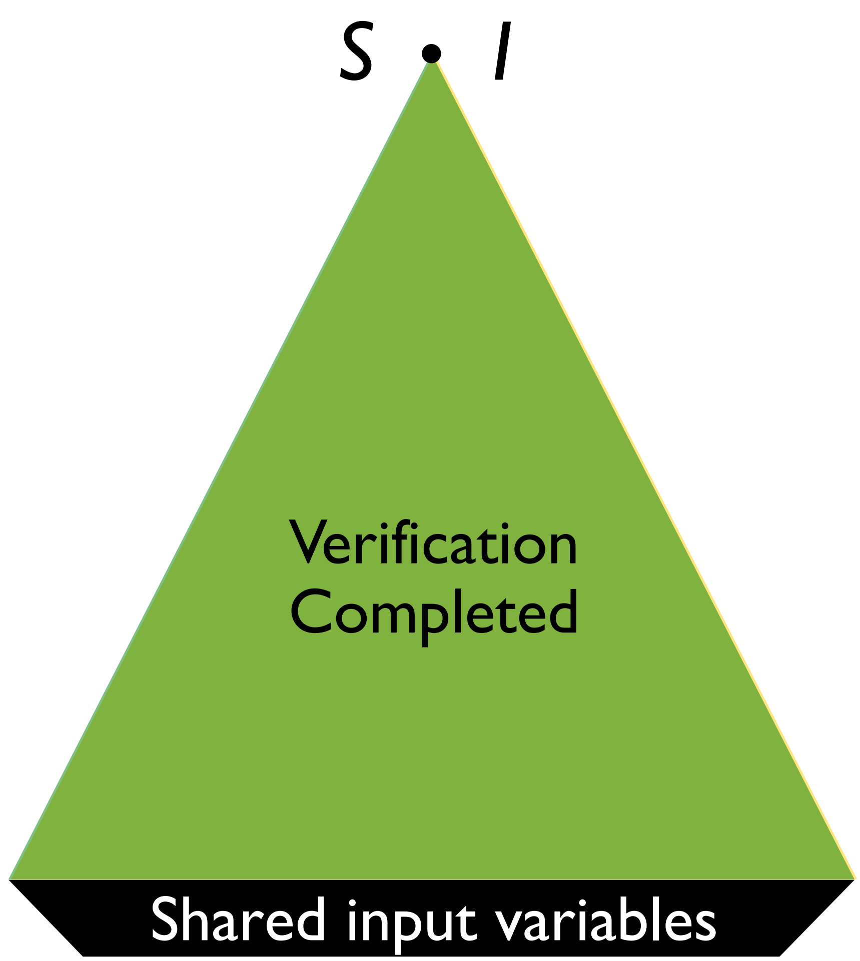

From A. Biere, "SAT in Formal Hardware Verification"

# Equivalence Checking




From A. Biere, "SAT in Formal Hardware Verification"

# Equivalence Checking




From A. Biere, "SAT in Formal Hardware Verification"

# Public Key Cryptography

RSA & Diffie Hillman

Elliptic Curve Cryptography (ECC)

Homomorphic Encryption?

Public key cryptography contain hard algorithms for automated verification, such as

- Large word multiplication
- Field Division
- Modular Exponentiation

# Public Key Cryptography

RSA & Diffie Hillman

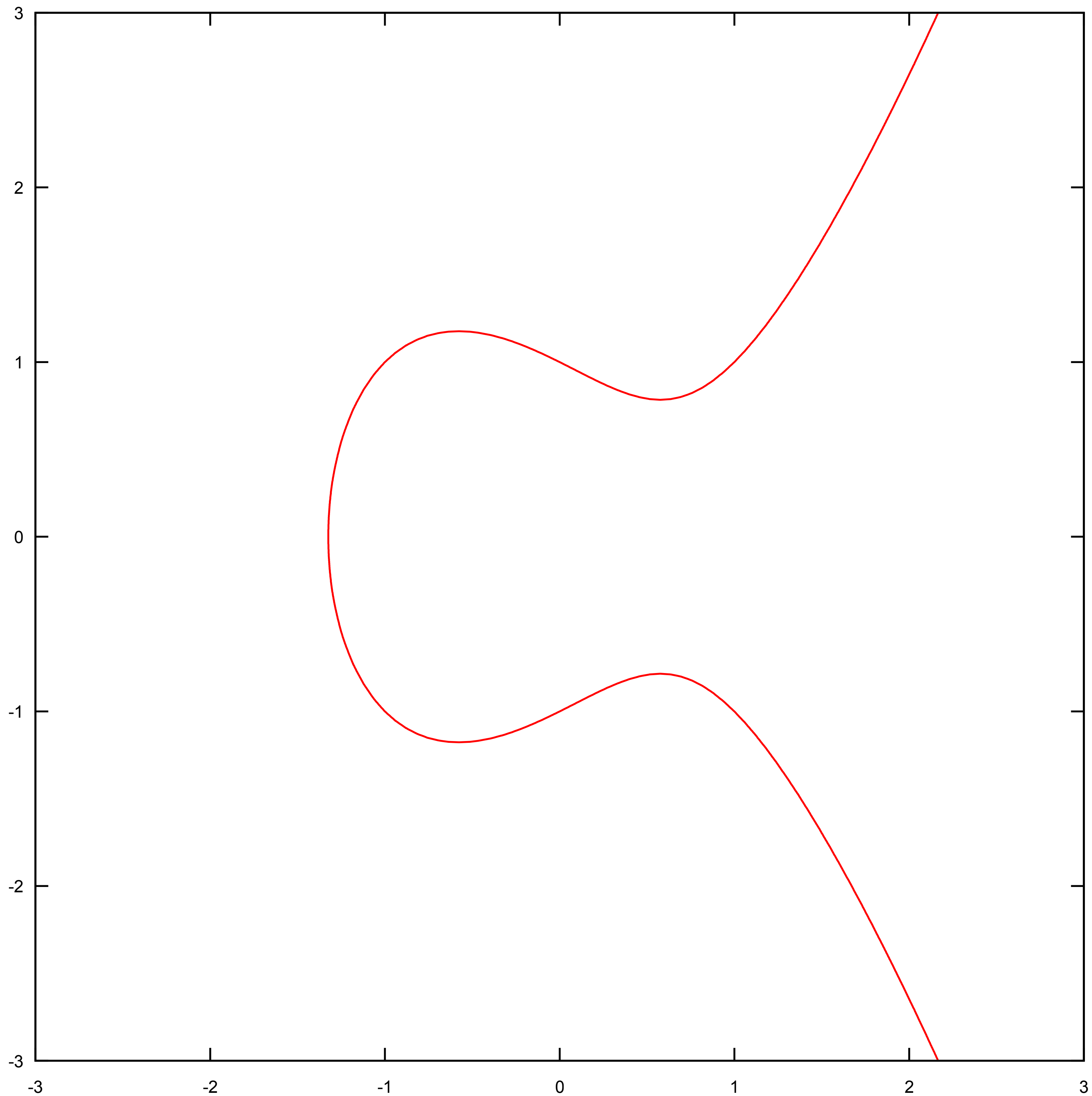
Elliptic Curve Cryptography (ECC)

Focus of this talk

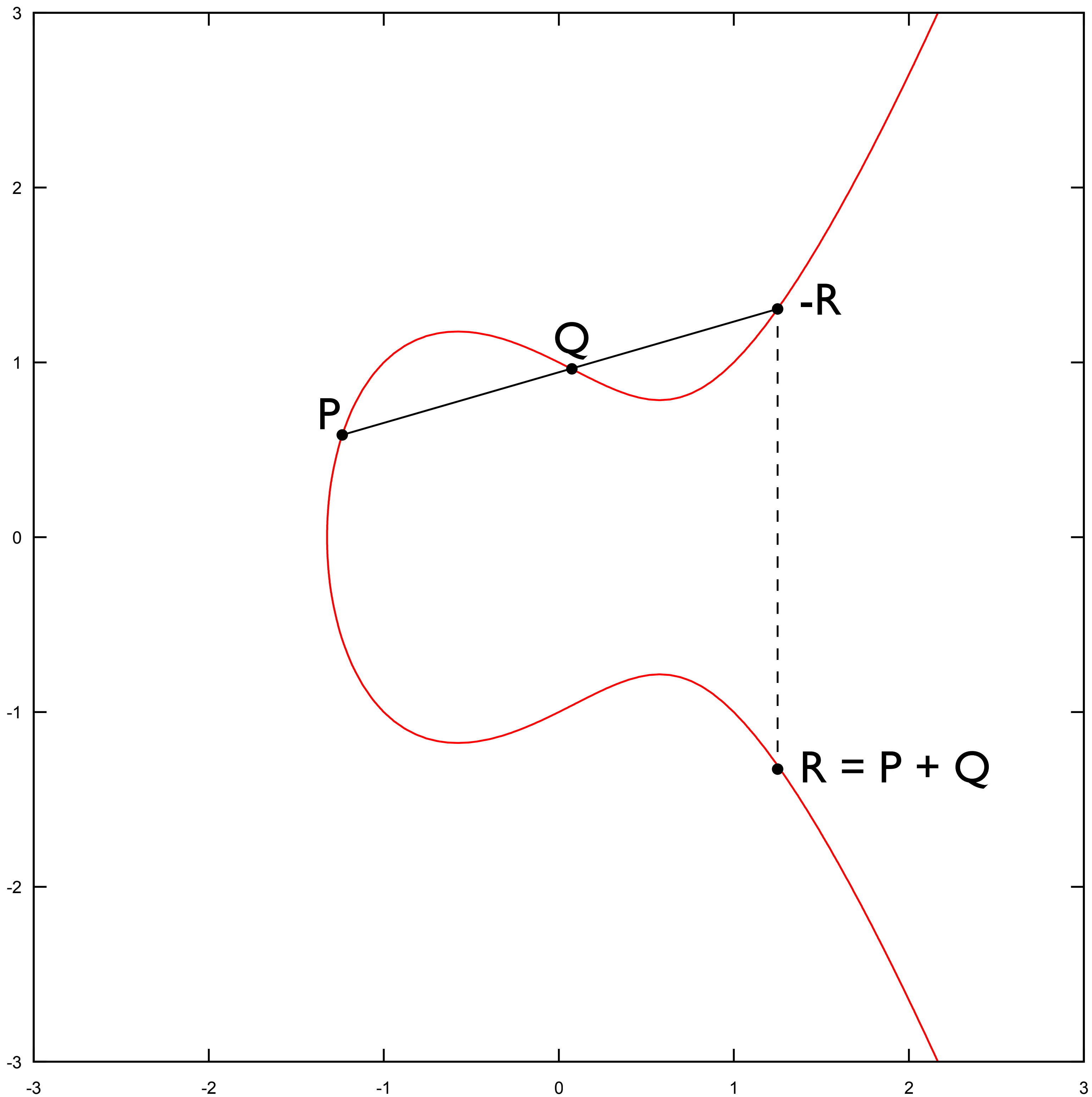
Homomorphic Encryption?

Public key cryptography contain hard algorithms for automated verification, such as

- Large word multiplication
- Field Division
- Modular Exponentiation




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




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

# Elliptic Curve Crypto (ECC)

**Cryptographic Protocols**

| ECDSA | ECDH |
|---|---|
| Digital Signatures | Key Agreement |

**Module Operations**

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

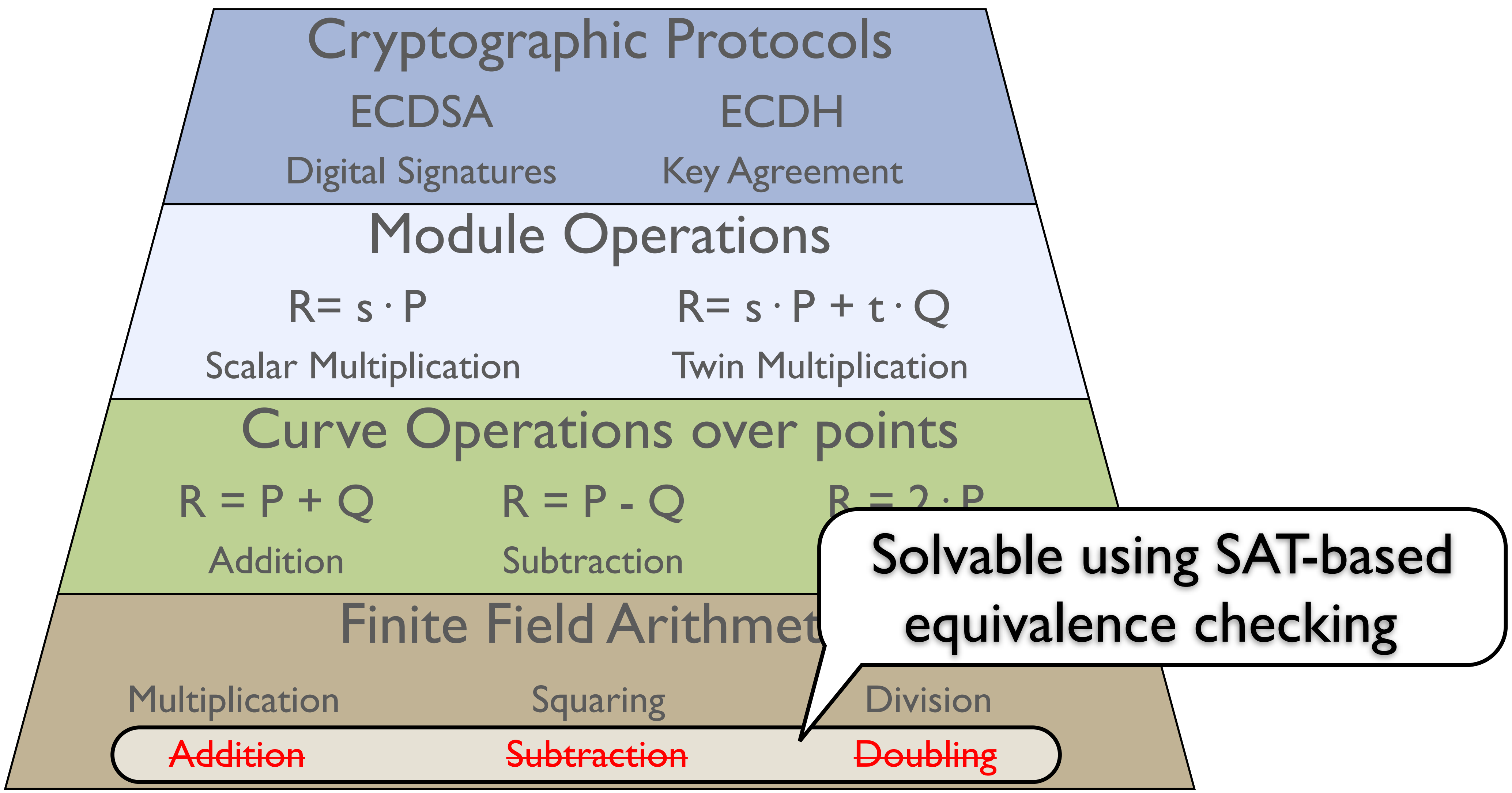
**Curve Operations over points**

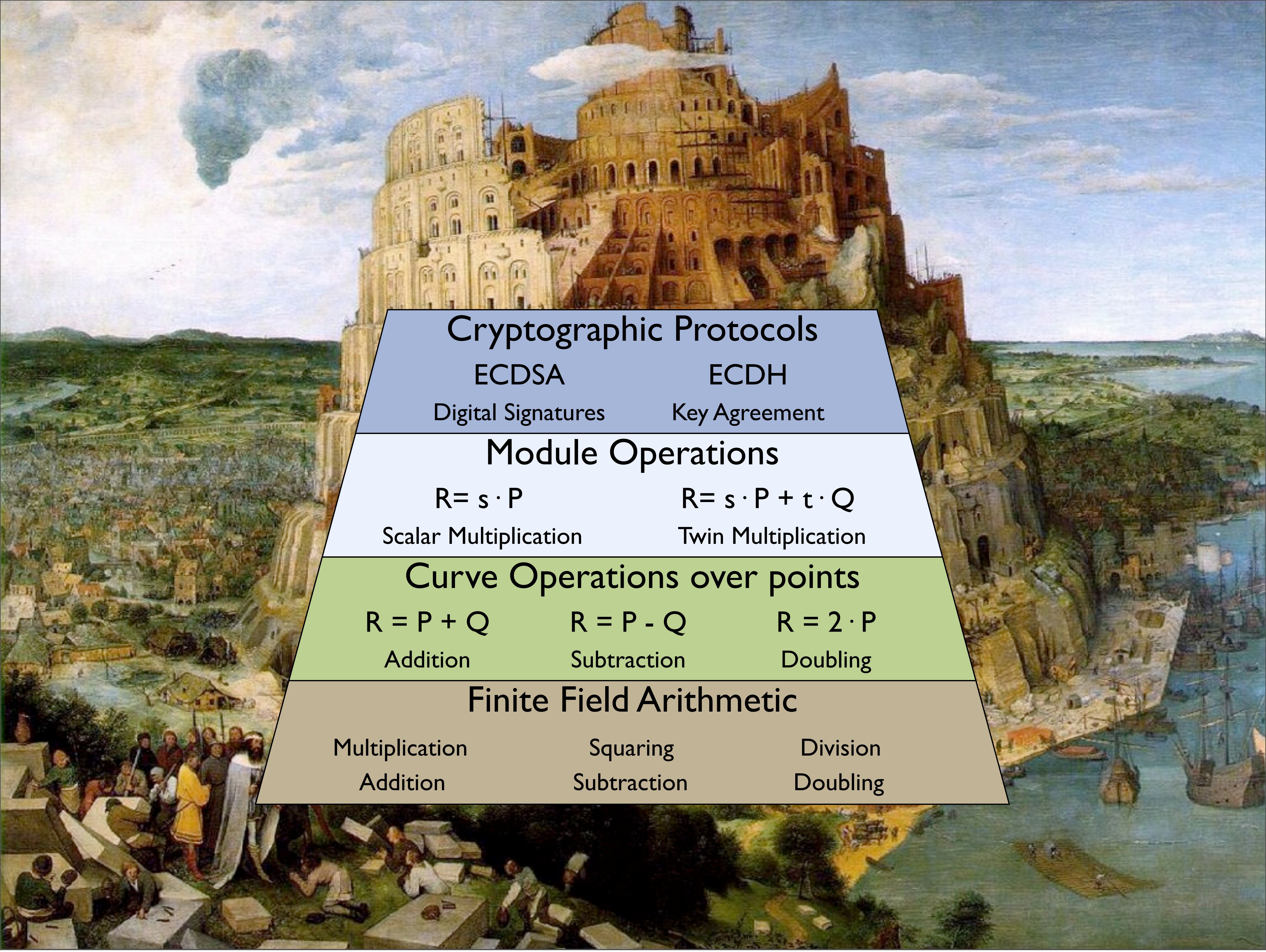
| $R = P + Q$ | $R = P - Q$ | $R = 2 \cdot P$ |
|---|---|---|
| Addition | Subtraction | Doubling |

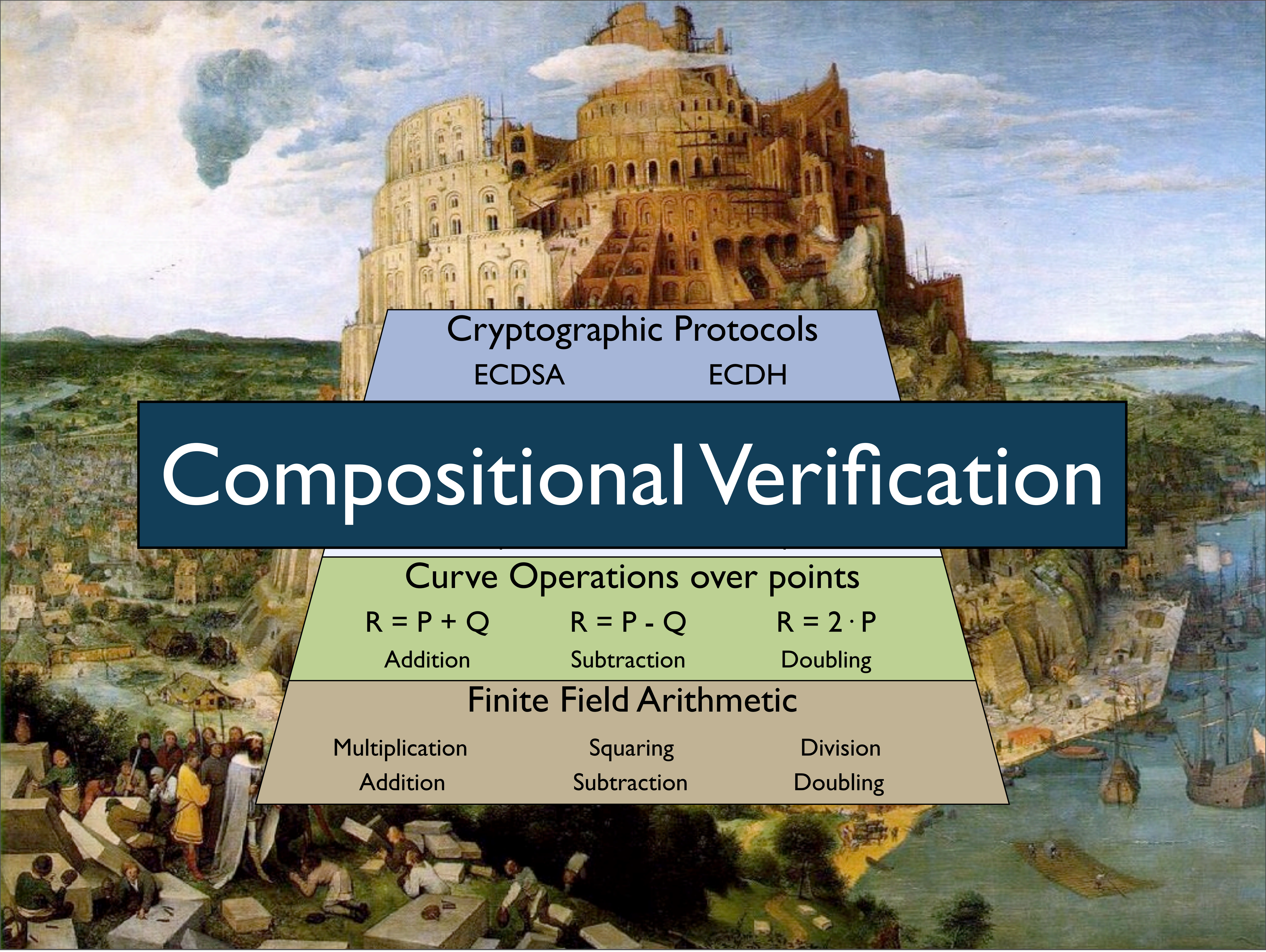
**Finite Field Arithmetic**

| | | |
|---|---|---|
| Multiplication | Squaring | Division |
| Addition | Subtraction | Doubling |

# Elliptic Curve Crypto (ECC)




Cryptographic Protocols
ECDSA
Digital Signatures
ECDH
Key Agreement
Module Operations
R= s · P
Scalar Multiplication
R= s · P + t · Q
Twin Multiplication
Curve Operations over points
R = P + Q
Addition
R = P - Q
Subtraction
R = 2 · P
Doubling
Finite Field Arithmetic
Multiplication
Squaring
Division
Addition
Subtraction
Doubling


Cryptographic Protocols
ECDSA
ECDH
Compositional Verification
Curve Operations over points
$R = P + Q$
Addition
$R = P - Q$
Subtraction
$R = 2 \cdot P$
Doubling
Finite Field Arithmetic
Multiplication
Addition
Squaring
Subtraction
Division
Doubling

# NIST P384 Curve

ECC is a family of algorithms, with many choices...

- NIST P384 is a standardized curve that is 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

# Implementing P384

| Cryptol Specification | Java Implementation |
| --- | --- |
| Clarity | Performance |
| Declarative | Imperative |
| 384bit Integers | Arrays of 32bit Integers |
| Higher-order Functions | Object Oriented |

# Field addition in Cryptol

```
/* Returns x + y (mod p384_prime). */
p384_add(x,y) = prime_field_add(x, y, p384_prime);

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

prime_field_add : {n} (fin n) => ([n],[n],[n]) -> [n];
prime_field_add(x,y,p) = mod(uext(x) + uext(y), p)
  where {
    /* Unsigned word extension. */
    uext : {n} (fin n) => [n] -> [n+1];
    uext(x) = x # zero;
    /* Modular reduction on input. */
    mod : {n} (fin n) => ([n+1],[n]) -> [n];
    mod(x,p) = take(width(p), x % uext(p));
  };
```

# Field addition in Cryptol

```
/* Returns x + y (mod p384_prime). */
p384_add(x,y) = prime_field_add(x, y, p384_prime);

p384_pr
p384_pr

prime_field_add : {n} (fin n) => ([n],[n],[n]) -> [n];
prime_field_add(x,y,p) = mod(uext(x) + uext(y), p)
  where {
    /* Unsigned word extension. */
    uext : {n} (fin n) => [n] -> [n+1];
    uext(x) = x # zero;
    /* Modular reduction on input. */
    mod : {n} (fin n) => ([n+1],[n]) -> [n];
    mod(x,p) = take(width(p), x % uext(p));
  };
```

Definition of field addition for P384

# Field addition in Cryptol

```
/* Returns x + y (mod p384_prime). */
p384_add(x,y) = prime_field_add(x, y, p384_prime);

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

prime_fie
prime_fie
  where {
    /* Unsigned word extension. */
    uext : {n} (fin n) => [n] -> [n+1];
    uext(x) = x # zero;
    /* Modular reduction on input. */
    mod : {n} (fin n) => ([n+1],[n]) -> [n];
    mod(x,p) = take(width(p), x % uext(p));
  };
```

Definition of field prime.

# Field addition in Cryptol

```
/* Returns x + y (mod p384_prime). */
p384_add(x,y) = prime_field_add(x, y, p384_prime);

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

prime_field_add : {n} (fin n) => ([n],[n],[n]) -> [n];
prime_field_add(x,y,p) = mod(uext(x) + uext(y), p)
  where {
    /* Unsigned word extension. */
    uext : {n} (fin n) => [n] -> [n+1];
    uext(x) = x # zero;
    /* Modular reduction on input. */
    mod :
    mod(x
  };
```

Extend precision to avoid overflow

# Field addition in Cryptol

```
/* Returns x + y (mod p384_prime). */
p384_add(x,y) = prime_field_add(x, y, p384_prime);

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

prime_field_add : {n} (fin n) => ([n],[n],[n]) -> [n];
prime_field_add(x,y,p) = mod(uext(x) + uext(y), p)
  where {
    /* Unsigned word extension. */
    uext : {n} (fin n) => [n] -> [n+1];
    uext(x) = x # zero;
    /* Modular reduction on input. */
    mod : {n} (fin n) => ([n+1],[n]) -> [n];
    mod(x,p) = take(width(p), x % uext(p));
  };
```

Perform modular reduction to original precision.

# 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 Java

In-place modification of result

```
/** 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 Java

```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

/** 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 prime is an int array

# 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 r
protected int add(int[] z,
  long c = 0;
  for (int i = 0; i != z.le
    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) { ... }
```

Mask used for unsigned conversion to long

# 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) { ... }
```

Addition loop with explicit carry

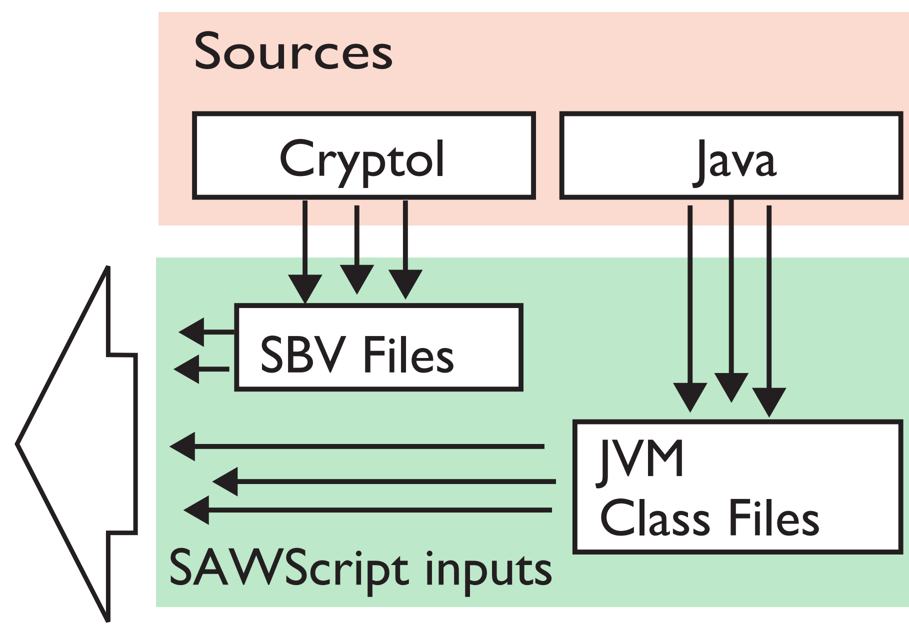
# Software Analysis Workbench

- Allows Java methods to be specified using SAWScript, a bit-precise specification language relating Java and Cryptol.
- Specifications are sequentially processed by SAWScript verification tool.

```
sawScript -j ecc.jar:classes.jar ecc.saw
```

# SAWScript Workflow

*sawScript -j ecc.jar:classes.jar ecc.saw*




SAWScript Execution

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

method ECC.field_add {

  ...specification
  verifyUsing: abc;
}

...more declarations
method ECC.ec_double {
```

If all succeed

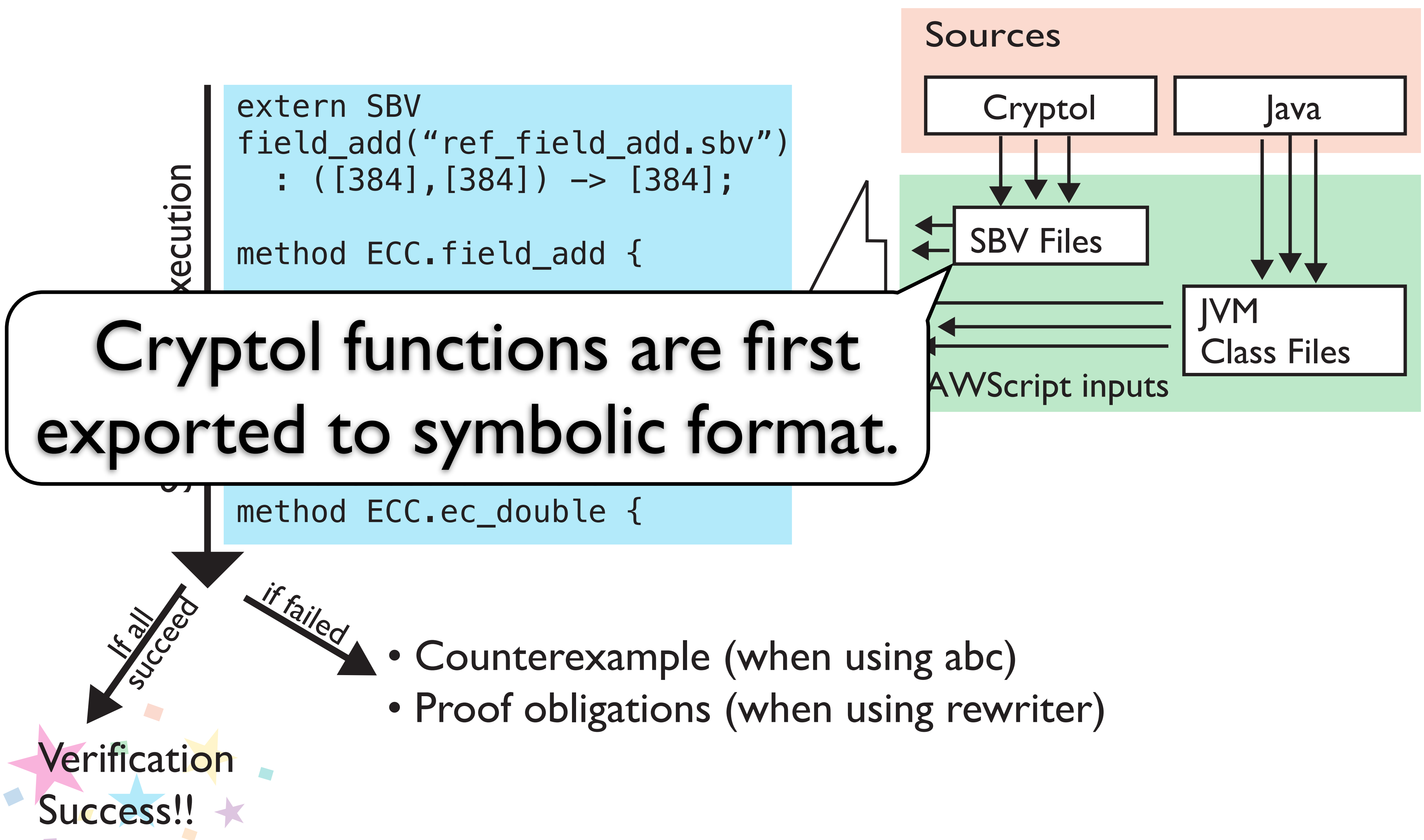
Verification Success!!

if failed

- Counterexample (when using abc)
- Proof obligations (when using rewriter)

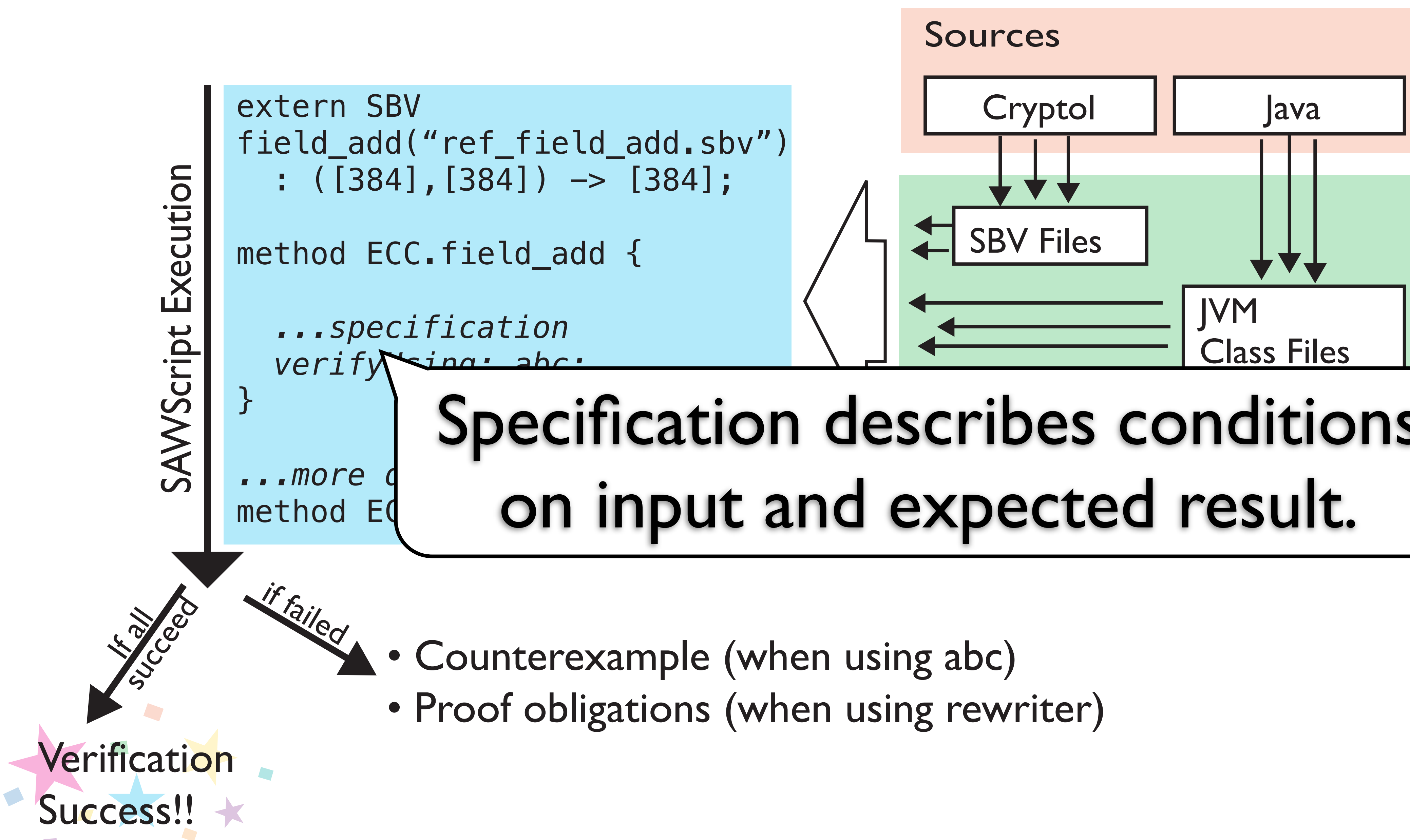
# SAWScript Workflow

*sawScript -j ecc.jar:classes.jar ecc.saw*



# SAWScript Workflow

*sawScript -j ecc.jar:classes.jar ecc.saw*

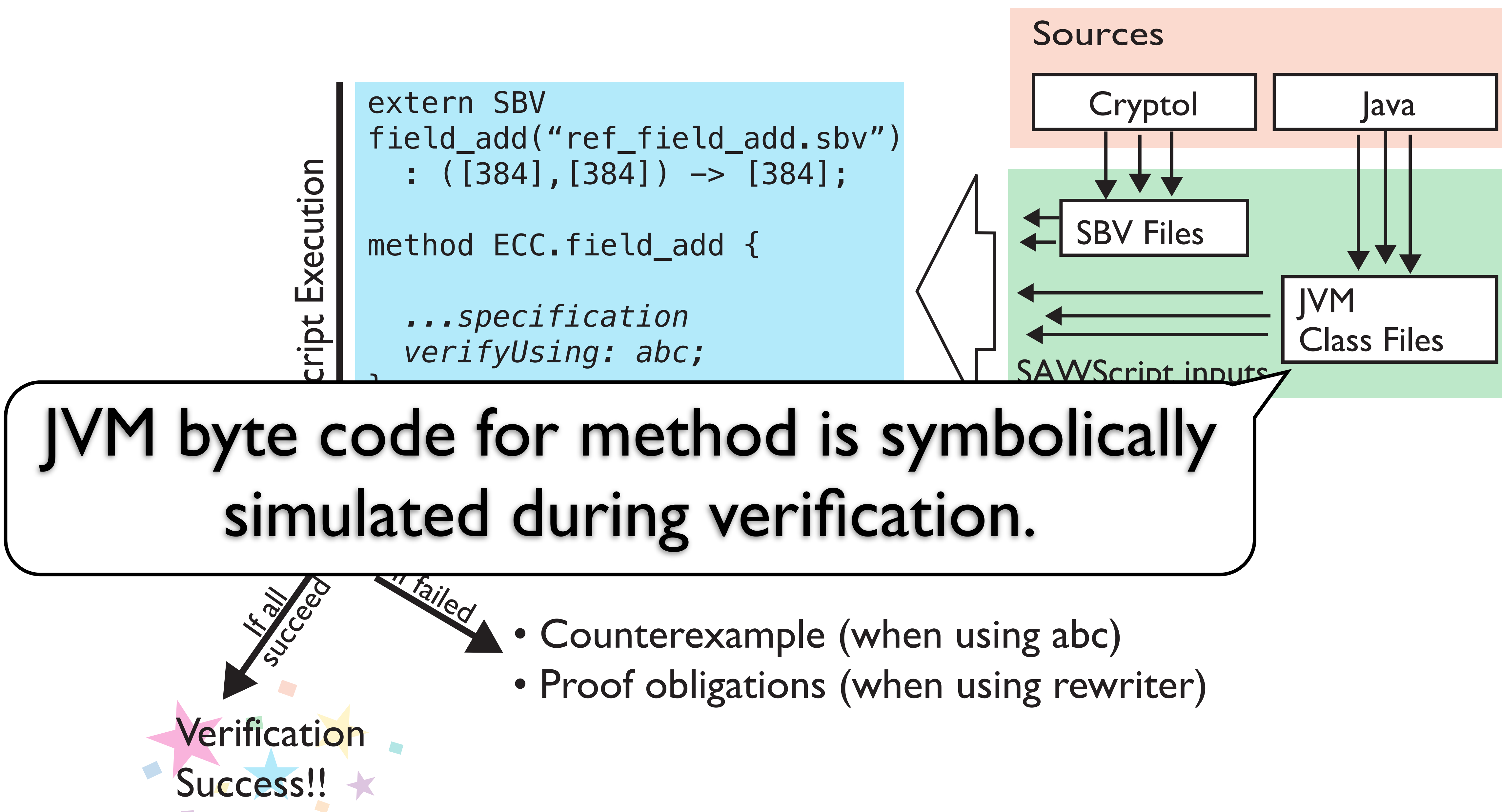



**Specification describes conditions on input and expected result.**

- Counterexample (when using abc)
- Proof obligations (when using rewriter)

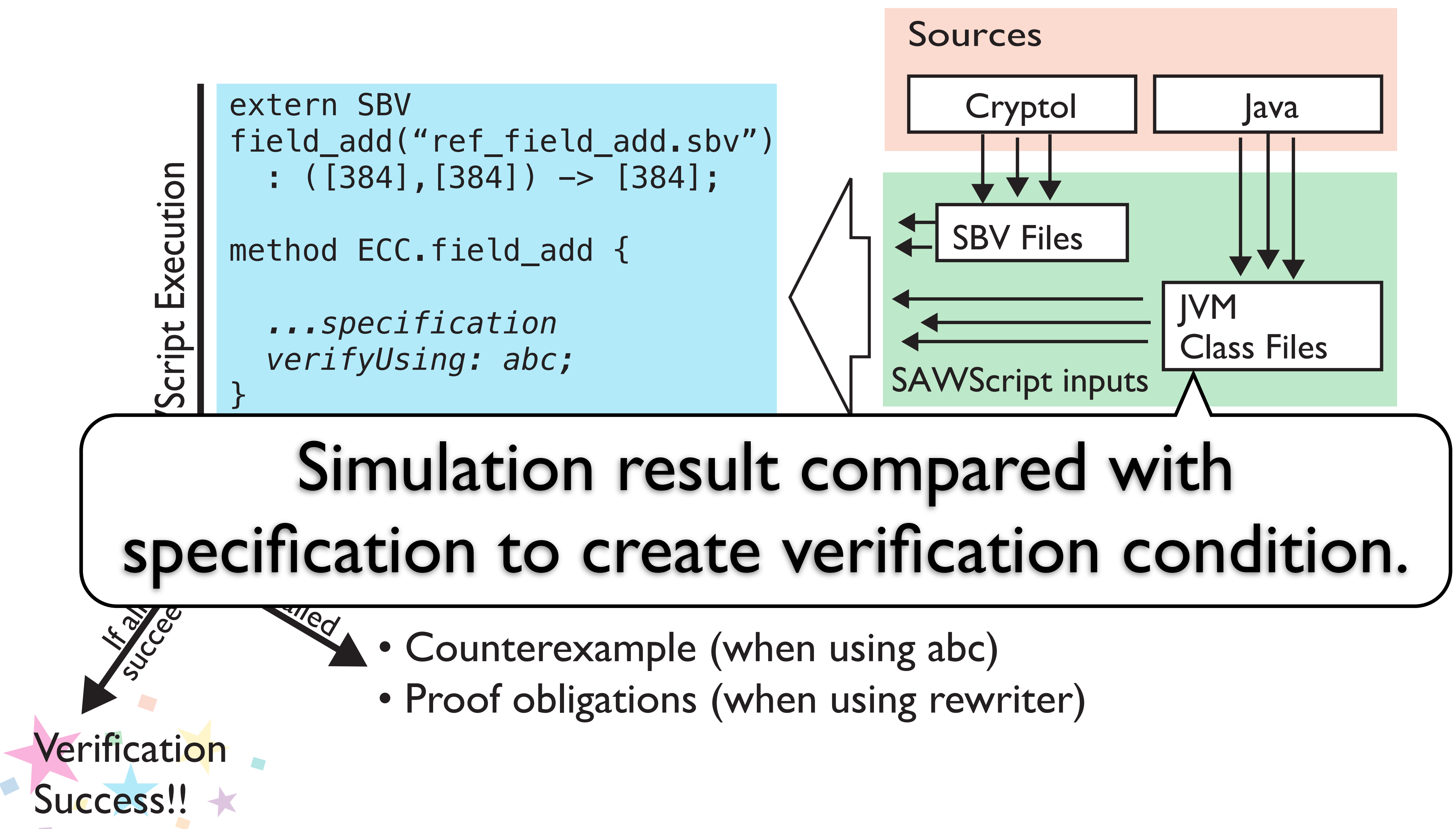
# SAWScript Workflow

```
sawScript -j ecc.jar:classes.jar ecc.saw
```



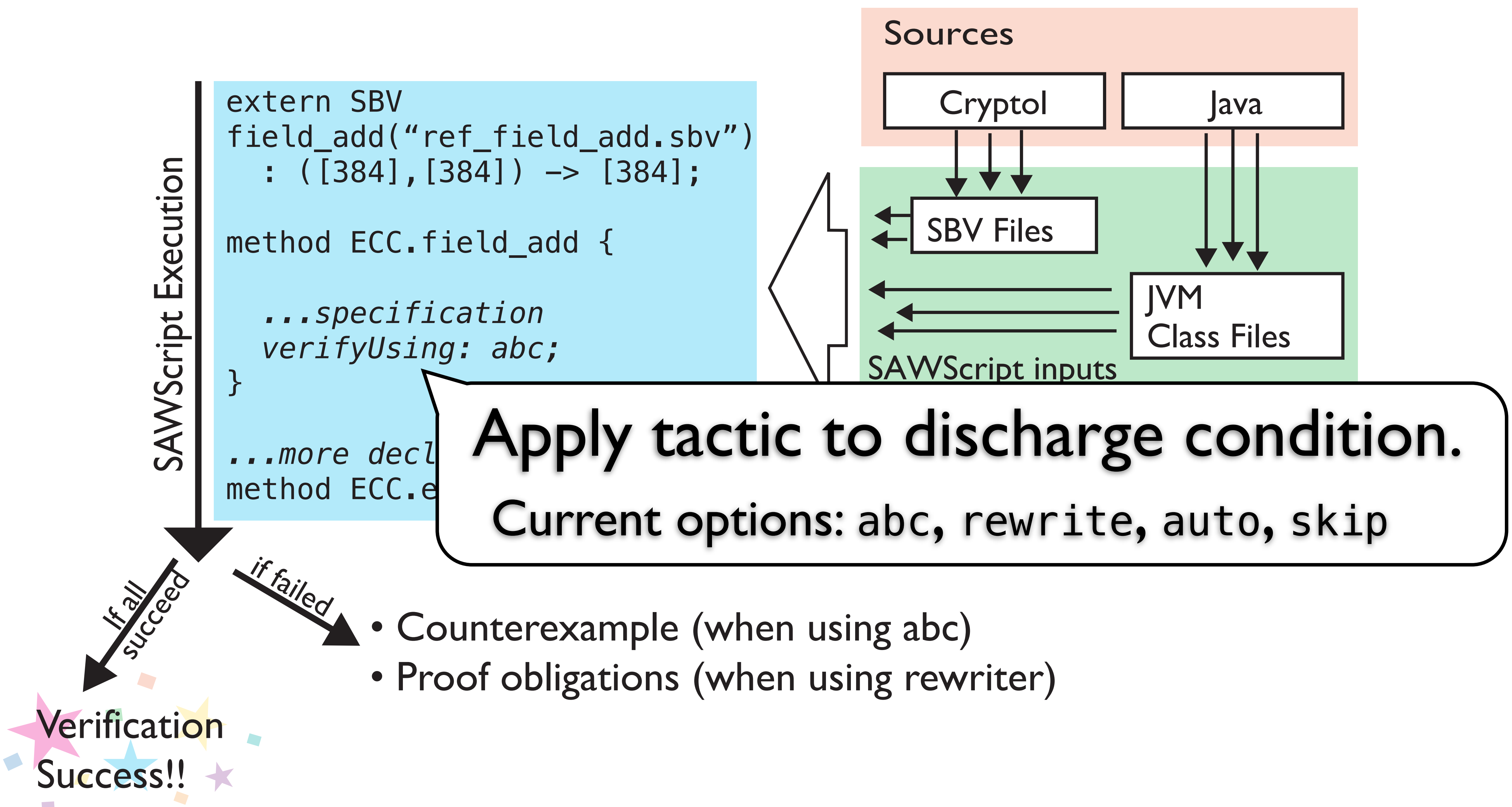
# SAWScript Workflow

```
sawScript -j ecc.jar:classes.jar ecc.saw
```



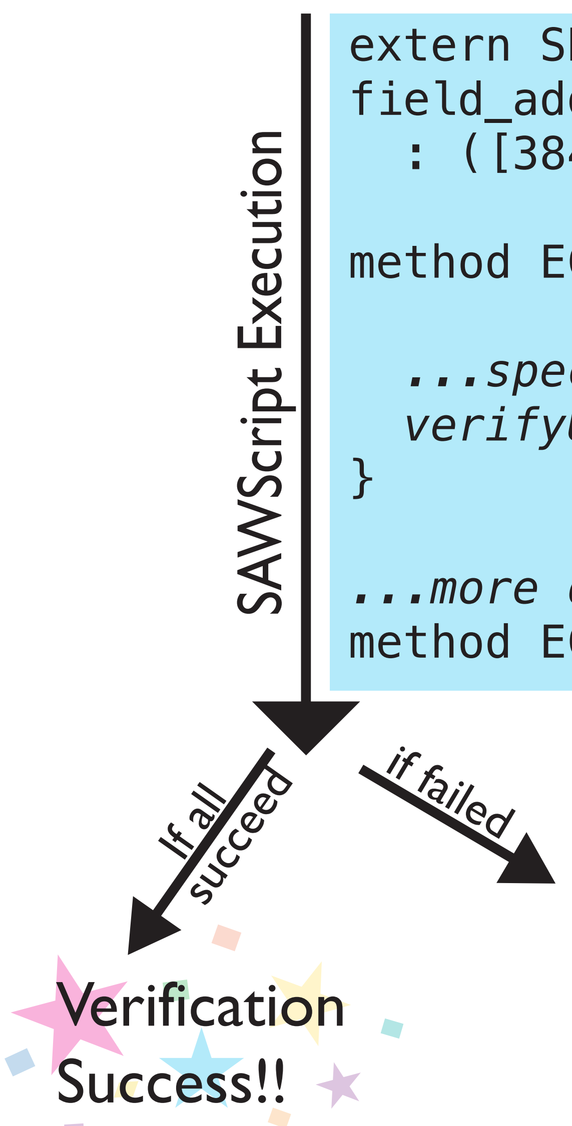
# SAWScript Workflow

`sawScript -j ecc.jar:classes.jar ecc.saw`



# SAWScript Workflow

*sawScript -j ecc.jar:classes.jar ecc.saw*

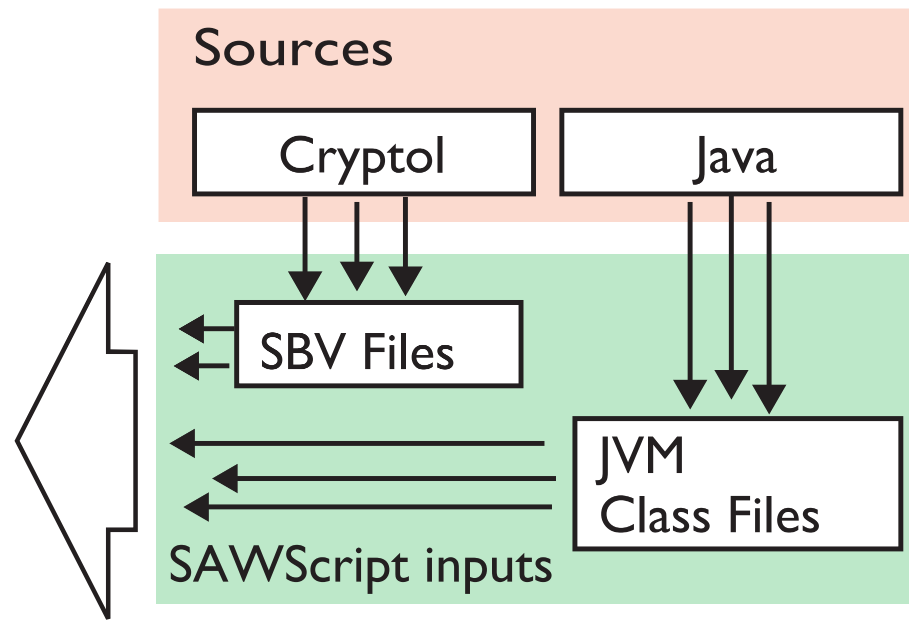



```
extern SBV
field_add("ref_field_add.sbv")
  : ([384],[384]) -> [384];

method ECC.field_add {

  ...specification
  verifyUsing: abc;
}

...more declarations
method ECC.ec_double {
```




- Cou
- Proc

`field_add` specification will be used in later methods that call it.

# `field_add` Specification

```
extern SBV p384_add("sbv/ref_p384_add.sbv") : ([384],[384]) -> [384];

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

method com.galois.ecc.P384ECC64.field_add {

  var args[0], args[1], args[2] : int[12];

  mayAlias { args[0], args[1], args[2] };

  const this.field_prime := split(field_const) : [12][32];

  ensures args[0] :=
    split(p384_add(join(fromJava(args[1])),
                   join(fromJava(args[2])))) : [12][32];

  verifyUsing: abc;
};
```

# field_add Specification

```
extern SBV p384_add("sbv/ref_p384_add.sbv") : ([384],[384]) -> [384];

let field_const

method com.galois.ecc.P384ECC64.field_add {

  var args[0], args[1], args[2] : int[12];

  mayAlias { args[0], args[1], args[2] };

  const this.field_prime := split(field_const) : [12][32];

  ensures args[0] :=
    split(p384_add(join(fromJava(args[1])),
                   join(fromJava(args[2])))) : [12][32];

  verifyUsing: abc;
};
```

Import Cryptol field addition.

# `field_add` Specification

```
extern SBV p384_add("sbv/ref_p384_add.sbv") : ([384],[384]) -> [384];

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

**Declare constant for field prime**

```
method com.galo

  var args[0], args[1], args[2] : int[12];

  mayAlias { args[0], args[1], args[2] };

  const this.field_prime := split(field_const) : [12][32];

  ensures args[0] :=
    split(p384_add(join(fromJava(args[1])),
                   join(fromJava(args[2])))) : [12][32];

  verifyUsing: abc;
};
```

# field_add Specification

```
extern SBV p384_add("sbv/ref_p384_add.sbv") : ([384],[384]) -> [384];

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

method com.galois.ecc.P384ECC64.field_add {

  var args[0], args[1], ar

  mayAlias { args[0], args[1], args[2] };

  const this.field_prime := split(field_const) : [12][32];

  ensures args[0] :=
    split(p384_add(join(fromJava(args[1])),
                   join(fromJava(args[2])))) : [12][32];

  verifyUsing: abc;
};
```

Identify Java method

# field_add Specification

```
extern SBV p384_add("sbv/ref_p384_add.sbv") : ([384],[384]) -> [384];

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

method com.galois.ecc.P384ECC64.field_add {

  var args[0], args[1], args[2] : int[12];

  mayAlias { args[0], args[1], args[

  const this.field_prime := split(field_const) : [12][32];

  ensures args[0] :=
    split(p384_add(join(fromJava(args[1])),
                   join(fromJava(args[2])))) : [12][32];

  verifyUsing: abc;
};
```

Specify bit-precise types

# field_add Specification

```
extern SBV p384_add("sbv/ref_p384_add.sbv") : ([384],[384]) -> [384];

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

method com.galois.ecc.P384ECC64.field_add {

  var args[0], args[1], args[2] : int[12];

  mayAlias { args[0], args[1], args[2] };

  const this.field_p

  ensures args[0] :=
    split(p384_add(join(fromJava(args[1])),
                   join(fromJava(args[2])))) : [12][32];

  verifyUsing: abc;
};
```

Specify potential aliasing

# `field_add` Specification

```
extern SBV p384_add("sbv/ref_p384_add.sbv") : ([384],[384]) -> [384];

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

method com.galois.ecc.P384ECC64.field_add {

  var args[0], args[1], args[2] : int[12];

  mayAlias { args[0], args[1], args[2] };

  const this.field_prime := split(field_const) : [12][32];

  ensures args[0] :=
    split(p384_add(joi
                   joi

  verifyUsing: abc;
};
```

Specify Constants

# `field_add` Specification

```
extern SBV p384_add("sbv/ref_p384_add.sbv") : ([384],[384]) -> [384];

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

method com.galois.ecc.P384ECC64.field_add {

  var args[0], args[1], args[2] : int[12];

  mayAlias { args[0], args[1], args[2] };

  const this.field_prime := split(field_const) : [12][32];

  ensures args[0] :=
    split(p384_add(join(fromJava(args[1])),
                   join(fromJava(args[2])))) : [12][32];

  verifyUsing: ab
};
```

Specify Postcondition

# `field_add` Specification

```
extern SBV p384_add("sbv/ref_p384_add.sbv") : ([384],[384]) -> [384];

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

method com.galois.ecc.P384ECC64.field_add {

  var args[0], args[1], args[2] : int[12];

  mayAlias { args[0], args[1], args[2] };

  const this.field_prime := split(field_const) : [12][32];

  ensures args[0] :=
    split(p384_add(join(fromJava(args[1])),
                   join(fromJava(args[2])))) : [12][32];

  verifyUsing: abc;
};
```

Select Verification Method

# Bit-Precise Rewriting

- Rewriting is a general-purpose tactic in many theorem provers.
- SAWScript's rewrite engine has been designed to support Cryptol's type system.
- Multiple rewrite rules are compiled into a single automaton for efficiency as in many theorem provers and rewrite engines.

# Results so far

- Verified 13 out of 18 Java methods in Java ECC implementation.
- Identified one error in modular reduction:

```
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);
```

# Results so far

- Verified 13 out of 18 Java methods in Java ECC implementation.
- Identified one error in modular reduction:

```
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);
```

# Results so far

- Verified 13 out of 18 Java methods in Java ECC implementation.
- Identified one error in modular reduction:

```
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);
```

# Results so far

- Verified 13 out of 18 Java methods in Java ECC implementation.
- Identified one error in modular reduction:

Bug only occurs if this addition overflows.

```
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);
```

# Results so far

- Verified 13 out of 18 Java methods in Java ECC implementation.
- Identified one error in modular reduction:

```
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;
```

`of` is guaranteed to be less than 4.

# Next Steps

- Add inductive assertions to SAWScript.
  - Field division uses extended gcd algorithm that is not symbolically terminating.
  - Inductive assertions needed to handle division.
- Complete remaining proofs for example ECC implementation.

# Next Steps

- General purpose rewriting as a proof tactic is powerful, but labor intensive.
- Decision procedures for common theories:
  - Integrate SMT Solving.
  - Decision procedures for fields and Abelian groups.
  - Computer algebra techniques such as Gröbner Basis seem promising.

# Summary

- Need for automated verification tools that can handle public key cryptographic implementations.
- Tool must support compositional verification, and a variety of proof tools for discharging obligations.
- Have infrastructure in place, and more work remains.