Verifying Postquantum Cryptography

Ming-Hsien Tsai

December 27, 2024

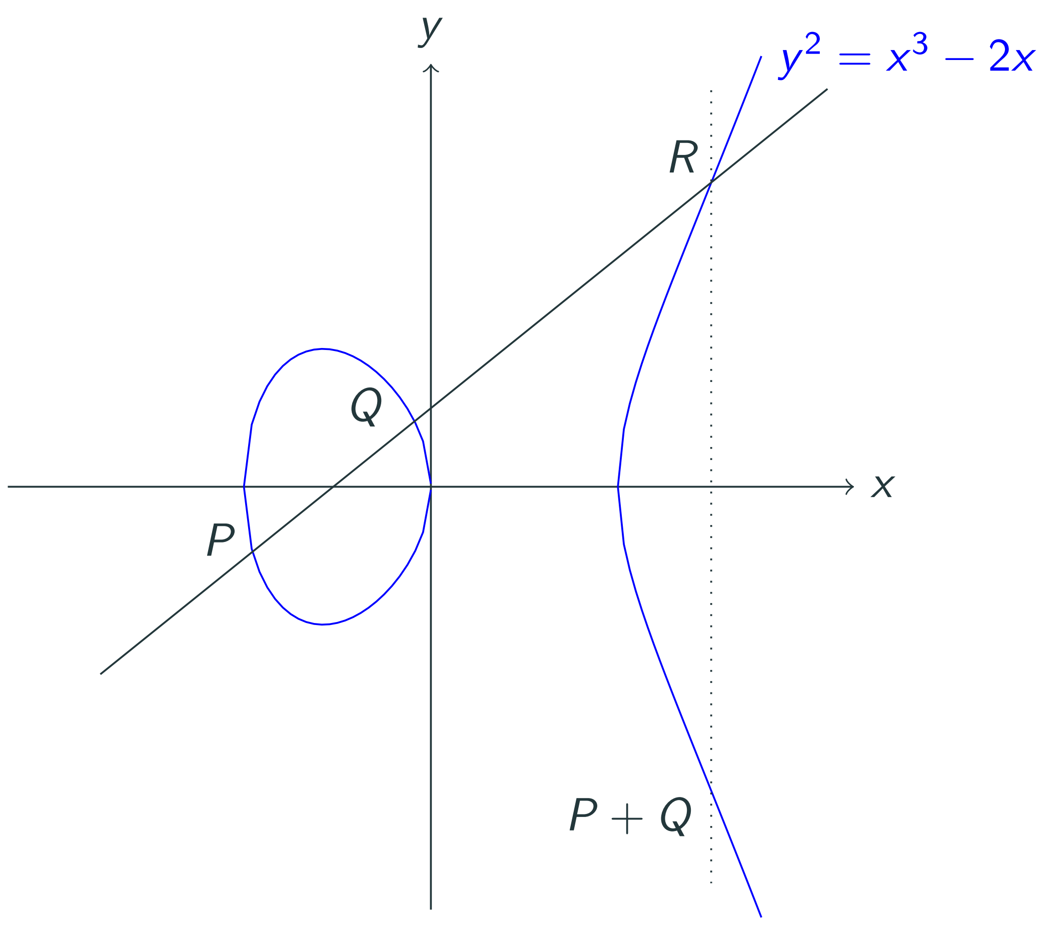
TAIWAN TECH
NATIONAL TAIWAN UNIVERSITY OF
SCIENCE AND TECHNOLOGY

Section 1

Introduction

Cryptography

- Modern cryptography relies on complex mathematical structures.
  - RSA: 2048-bit modulo computation
  - elliptic curves: complex group operations based on large finite fields
  - lattices: polynomial rings with finite coefficients of high degrees



- A field (such as $\mathbb{Q}$) has addition and multiplication and their inverse operations.
- Each point is represented by two field elements.
  - A finite (prime) field is obtained by modulo arithmetic.
  - $\mathbb{F}_q = \{0, 1, \dots, q-1\}$ with a prime q.
- In Ed25519, we have
  - the finite field $\mathbb{F}_q = 2^{255} - 19$;
  - the curve $-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$.

Computer Cryptography

- Mathematically, all operations in cryptography have simple representation.
  - RSA: m^e mod pq where p and q are 1024-bit prime numbers.
  - elliptic curves: $P + Q$ where P and Q are points on an elliptic curve.
  - lattices: $f(X) \times g(X) \bmod X^{256} + 1$ where $f(X)$ and $g(X)$ are in the ring $\mathbb{F}_{3329}[X]$.
    - A ring (such as $\mathbb{Z}$) has addition, its inverse operation, and multiplication.
- However, no computer can perform such complex operations with simple instructions.
- To employ modern cryptography, all operations must be implemented by programs on different (say, 32- or 64-bit) architectures.
- How many programmers have written multi-precision arithmetic programs?
  - the GNU multi-precision arithmetic library (`gmp`)

Real World Computer Cryptography

- Complex operations (multi-precision arithmetic and polynomial multiplication) are only small steps in computer cryptography.
- Advanced algorithms are implemented to improve performance.
  - Karatsuba multiplication, Montgomery reduction, Number theoretic transform, etc.
- In the real world, even advanced algorithms are not good enough.
- The OpenSSL project has many assembly programs for such operations.
- How many programmers are comfortable writing multi-precision arithmetic in assembly?
- And the story began in 2009...

Cryptographic Primitives

- We want to verify assembly implementations of such primitive operations in real-world cryptography.
- Specifically, we want to verify the following operations
  - field arithmetic over large finite fields
  - group operations on elliptic curves
  - polynomial multiplication in large finite rings
- We want to show programs compute corresponding mathematical functions correctly.
  - This is called functional correctness.
  - We are not verifying security properties.

Problems and Difficulties

- Non-linear computation is hard to verify.
  - SAT/SMT solvers do not work.
    - If they did, RSA would be broken already.
    - more about this later.
- Cryptographic programs are succinct.
  - Every bit counts.
- There are many cryptographic assembly programs.
  - 32 bits: x86 and armv7
  - 64 bits: x86_64 and aarch64
  - and more: avx, avx2, avx512, and neon

Problems and Difficulties

- Non-linear computation is hard to verify.
  - SAT/SMT solvers do not work.
    - If they did, RSA would be broken already.
    - more about this later.
- Cryptographic programs are succinct.
  - Every bit counts.
- There are many cryptographic assembly programs.
  - 32 bits: x86 and armv7
  - 64 bits: x86_64 and aarch64
  - and more: avx, avx2, avx512, and neon

SAT formula: $p \wedge (q \vee \neg r)$

Problems and Difficulties

- Non-linear computation is hard to verify.
  - SAT/SMT solvers do not work.
    - If they did, RSA would be broken already.
    - more about this later.
- Cryptographic programs are succinct.
  - Every bit counts.
- There are many cryptographic assembly programs.
  - 32 bits: x86 and armv7
  - 64 bits: x86_64 and aarch64
  - and more: avx, avx2, avx512, and neon

SAT formula: $p \wedge (q \vee \neg r)$

SMT formula: $3 \leq i \leq 7 \wedge a[i] = n$

Problems and Difficulties

- Non-linear computation is hard to verify.
  - SAT/SMT solvers do not work.
    - If they did, RSA would be broken already.
    - more about this later.
- Cryptographic programs are succinct.
  - Every bit counts.
- There are many cryptographic assembly programs.
  - 32 bits: x86 and armv7
  - 64 bits: x86_64 and aarch64
  - and more: avx, avx2, avx512, and neon

SAT formula: $p \wedge (q \vee \neg r)$

SMT formula: $3 \leq i \leq 7 \wedge a[i] = n$

integer theory

Problems and Difficulties

- Non-linear computation is hard to verify.
  - SAT/SMT solvers do not work.
    - If they did, RSA would be broken already.
    - more about this later.
- Cryptographic programs are succinct.
  - Every bit counts.
- There are many cryptographic assembly programs.
  - 32 bits: x86 and armv7
  - 64 bits: x86_64 and aarch64
  - and more: avx, avx2, avx512, and neon

SAT formula: $p \land (q \lor \neg r)$

SMT formula: $3 \leq i \leq 7 \land a[i] = n$

integer theory array theory

Section 2

Algebraic Abstraction

SMT QF_BV

- SMT (Satisfiability Modulo Theories) solvers support different theories.
- Quantifier-Free Bit-Vector logic in SMT can model computation at bit level.
  - SMT QF_BV solvers translate QF_BV queries to SAT queries through bit blasting.
- In 2014, we use BOOLECTOR to verify an academic assembly program for the field multiplication in $\mathbb{F}_q$ where $q = 2^{255} - 19$.
  - about 200 instructions
  - without annotation: fail to verify, with LOTS of annotation: 4 days
  - COQ is needed to prove a simple algebraic property.
- Not useful!

gfverif

- In 2015, the gfverif project uses the computer algebra system SAGE to verify algebraic properties in C program.
- Instead of crunching bits, computer algebra systems support arithmetic natively.
  - Consider proving $x \cdot y = y \cdot x$ by bits and by algebra.
- Lesson: it is better to verify non-linear computation algebraically than logically.

Montgomery Reduction

Algorithm	Code
(* $R = 2^{64}, 0 \le T < R^2$ *)	(* $T = 2^{64} T_H + T_L$ *)
(* $N \cdot N' + 1 \equiv 0 \bmod R$ *)	ASSUME $N \times N' + 1 \equiv 0 \bmod [2^{64}]$
$m \leftarrow ((T \bmod R) \cdot N') \bmod R$	$dc : m \leftarrow$ MULL T_L N'
$t \leftarrow (T + m \cdot N)/R$	$mN_H : mN_L \leftarrow$ MULL m N
	$carry : t_L \leftarrow$ ADDS T_L mN_L
	$c : t \leftarrow$ ADCS T_H mN_H $carry$
	ASSERT $t_L \equiv 0 \bmod [2^{64}]$
	ASSUME $t_L = 0$
(* $t \cdot R \equiv T \bmod N$ *)	ASSERT $(c \times 2^{64} + t) \times 2^{64} \equiv T_H \times 2^{64} + T_L \bmod [N]$

- In the code, c and *carry* are bit variables; others are 64-bit variables.
- Given a 128-bit number $T_H \cdot 2^{64} + T_L$ and two 64-bit constants $N \cdot N' + 1 \equiv 0 \bmod [2^{64}]$, it computes a 65-bit number $2^{64} \cdot (c \cdot 2^{64} + t) \equiv (T_H \cdot 2^{64} + T_L) \bmod [N]$ without division.
- BOOLECTOR fails to verify it in 7 days.

Montgomery Reduction

Algorithm	Code
(* $R = 2^{64}, 0 \leq T < R^2$ *)	(* $T = 2^{64} T_H + T_L$ *)
(* $N \cdot N' + 1 \equiv 0 \bmod R$ *)	ASSUME $N \times N' + 1 \equiv 0 \bmod [2^{64}]$
$m \leftarrow ((T \bmod R) \cdot N') \bmod R$	$dc : m \leftarrow$ MULL T_L N'
$t \leftarrow (T + m \cdot N)/R$	$mN_H : mN_L \leftarrow$ MULL m N
	$carry : t_L \leftarrow$ ADDS T_L mN_L
	$c : t \leftarrow$ ADCS T_H mN_H $carry$
	ASSERT $t_L \equiv 0 \bmod [2^{64}]$
	ASSUME $t_L = 0$
(* $t \cdot R \equiv T \bmod N$ *)	ASSERT $(c \times 2^{64} + t) \times 2^{64} \equiv T_H \times 2^{64} + T_L \bmod [N]$

- In the code, c and *carry* are bit variables; others are 64-bit variables.
- Given a 128-bit number $T_H \cdot 2^{64} + T_L$ and two 64-bit constants $N \cdot N' + 1 \equiv 0 \bmod [2^{64}]$, it computes a 65-bit number $2^{64} \cdot (c \cdot 2^{64} + t) \equiv (T_H \cdot 2^{64} + T_L) \bmod [N]$ without division.
- BOOLECTOR fails to verify it in 7 days.

Montgomery Reduction

Algorithm	Code
(* $R = 2^{64}, 0 \leq T < R^2$ *)	(* $T = 2^{64} T_H + T_L$ *)
(* $N \cdot N' + 1 \equiv 0 \bmod R$ *)	ASSUME $N \times N' + 1 \equiv 0 \bmod [2^{64}]$
$m \leftarrow ((T \bmod R) \cdot N') \bmod R$	$dc : m \leftarrow$ MULL T_L N'
$t \leftarrow (T + m \cdot N)/R$	$mN_H : mN_L \leftarrow$ MULL m N
	$carry : t_L \leftarrow$ ADDS T_L mN_L
	$c : t \leftarrow$ ADCS T_H mN_H $carry$
	ASSERT $t_L \equiv 0 \bmod [2^{64}]$
	ASSUME $t_L = 0$
(* $t \cdot R \equiv T \bmod N$ *)	ASSERT $(c \times 2^{64} + t) \times 2^{64} \equiv T_H \times 2^{64} + T_L \bmod [N]$

modulo

division

- In the code, c and $carry$ are bit variables; others are 64-bit variables.
- Given a 128-bit number $T_H \cdot 2^{64} + T_L$ and two 64-bit constants $N \cdot N' + 1 \equiv 0 \bmod [2^{64}]$, it computes a 65-bit number $2^{64} \cdot (c \cdot 2^{64} + t) \equiv (T_H \cdot 2^{64} + T_L) \bmod [N]$ without division.
- BOOLECTOR fails to verify it in 7 days.

Montgomery Reduction

Algorithm	Code
$(*\ R = 2^{64}, 0 \leq T < R^2\ *)$	$(*\ T = 2^{64} T_H + T_L\ *)$
$(*\ N \cdot N' + 1 \equiv 0 \bmod R\ *)$	ASSUME $N \times N' + 1 \equiv 0 \bmod [2^{64}]$
$m \leftarrow ((T \bmod R) \cdot N') \bmod R$	$dc : m \leftarrow$ MULL $T_L\ N'$
$t \leftarrow (T + m \cdot N)/R$	$mN_H : mN_L \leftarrow$ MULL $m\ N$
	$carry : t_L \leftarrow$ ADDS $T_L\ mN_L$
	$c : t \leftarrow$ ADCS $T_H\ mN_H\ carry$
	ASSERT $t_L \equiv 0 \bmod [2^{64}]$
	ASSUME $t_L = 0$
$(*\ t \cdot R \equiv T \bmod N\ *)$	ASSERT $(c \times 2^{64} + t) \times 2^{64} \equiv T_H \times 2^{64} + T_L \bmod [N]$

- In the code, c and *carry* are bit variables; others are 64-bit variables.
- Given a 128-bit number $T_H \cdot 2^{64} + T_L$ and two 64-bit constants $N \cdot N' + 1 \equiv 0 \bmod [2^{64}]$, it computes a 65-bit number $2^{64} \cdot (c \cdot 2^{64} + t) \equiv (T_H \cdot 2^{64} + T_L) \bmod [N]$ without division.
- BOOLECTOR fails to verify it in 7 days.

Montgomery Reduction

Algorithm	Code
(* $R = 2^{64}, 0 \leq T < R^2$ *)	(* $T = 2^{64} T_H + T_L$ *)
(* $N \cdot N' + 1 \equiv 0 \bmod R$ *)	ASSUME $N \times N' + 1 \equiv 0 \bmod [2^{64}]$
$m \leftarrow ((T \bmod R) \cdot N') \bmod R$	$dc : m \leftarrow$ MULL T_L N'
$t \leftarrow (T + m \cdot N)/R$	$mN_H : mN_L \leftarrow$ MULL m N
	$carry : t_L \leftarrow$ ADDS T_L mN_L
	$c : t \leftarrow$ ADCS T_H mN_H $carry$
	ASSERT $t_L \equiv 0 \bmod [2^{64}]$
	ASSUME $t_L = 0$
(* $t \cdot R \equiv T \bmod N$ *)	ASSERT $(c \times 2^{64} + t) \times 2^{64} \equiv T_H \times 2^{64} + T_L \bmod [N]$

- In the code, c and *carry* are bit variables; others are 64-bit variables.
- Given a 128-bit number $T_H \cdot 2^{64} + T_L$ and two 64-bit constants $N \cdot N' + 1 \equiv 0 \bmod [2^{64}]$, it computes a 65-bit number $2^{64} \cdot (c \cdot 2^{64} + t) \equiv (T_H \cdot 2^{64} + T_L) \bmod [N]$ without division.
- BOOLECTOR fails to verify it in 7 days.

Polynomial Equations

- Idea: translate programs into polynomial equations.

Code	Equations
ASSUME $N \times N' + 1 \equiv 0 \bmod [2^{64}]$	$N \times N' + 1 \equiv 0 \bmod [2^{64}]$
$dc : m \leftarrow$ MULL T_L N'	$dc \cdot 2^{64} + m = T_L \cdot N'$
$mN_H : mN_L \leftarrow$ MULL m N	$mN_H \cdot 2^{64} + mN_L = m \cdot N$
$carry : t_L \leftarrow$ ADDS T_L mN_L	$carry \cdot (carry - 1) = 0$ $carry \cdot 2^{64} + t_L = T_L + mN_L$
$c : t \leftarrow$ ADCS T_H mN_H $carry$	$c \cdot (c - 1) = 0$ $c \cdot 2^{64} + t = T_H + mN_H + carry$

ASSERT $t_L \equiv 0 \bmod [2^{64}]$

- To ensure soundness, all program traces must be solutions to all equations.
  - No overflow, no underflow, etc.
- Soundness conditions are checked by SMT QF_BV solvers.

Polynomial Equations

- Idea: translate programs into polynomial equations.

Code	Equations
ASSUME $N \times N' + 1 \equiv 0 \bmod [2^{64}]$	$N \times N' + 1 \equiv 0 \bmod [2^{64}]$
$dc : m \leftarrow$ MULL T_L N'	$dc \cdot 2^{64} + m = T_L \cdot N'$
$mN_H : mN_L \leftarrow$ MULL m N	$mN_H \cdot 2^{64} + mN_L = m \cdot N$
$carry : t_L \leftarrow$ ADDS T_L mN_L	$carry \cdot (carry - 1) = 0$ $carry \cdot 2^{64} + t_L = T_L + mN_L$
$c : t \leftarrow$ ADCS T_H mN_H $carry$	$c \cdot (c - 1) = 0$ $c \cdot 2^{64} + t = T_H + mN_H + carry$

ASSERT $t_L \equiv 0 \bmod [2^{64}]$

- To ensure soundness, all program traces must be solutions to all equations.
  - No overflow, no underflow, etc.
- Soundness conditions are checked by SMT QF_BV solvers.

Root Entailment Problem

- Idea: verify assertions by checking roots.

Equations	Root Entailment
	$\forall N, N', m, T_L, T_H, mN_L, mN_H, t_L, t, dc, carry, c.$
$N \times N' + 1 \equiv 0 \bmod [2^{64}]$	$(\quad N \times N' + 1 \equiv 0 \bmod [2^{64}] \quad \wedge$
$dc \cdot 2^{64} + m = T_L \cdot N'$	$dc \cdot 2^{64} + m - T_L \cdot N' = 0 \quad \wedge$
$mN_H \cdot 2^{64} + mN_L = m \cdot N$	$mN_H \cdot 2^{64} + mN_L - m \cdot N = 0 \quad \wedge$
$carry \cdot (carry - 1) = 0$	$carry \cdot (carry - 1) = 0 \quad \wedge$
$carry \cdot 2^{64} + t_L = T_L + mN_L$	$carry \cdot 2^{64} + t_L - (T_L + mN_L) = 0 \quad \wedge$
$c \cdot (c - 1) = 0$	$c \cdot (c - 1) = 0 \quad \wedge$
$c \cdot 2^{64} + t = T_H + mN_H + carry$	$c \cdot 2^{64} + t - (T_H + mN_H + carry) = 0 \quad)$
ASSERT $t_L \equiv 0 \bmod [2^{64}]$	$\Longrightarrow \quad t_L \equiv 0 \bmod [2^{64}]$

- The root entailment problem: given a system Σ of polynomial equations, verify whether all solutions to Σ are also solutions to the assertion.

Root Entailment Problem

$f = g \Rightarrow f - g = 0$

- Idea: verify assertions by checking roots.

Equations	Root Entailment
	$\forall N, N', m, T_L, T_H, mN_L, mN_H, t_L, t, dc, carry, c.$
$N \times N' + 1 \equiv 0 \bmod [2^{64}]$	$(\quad N \times N' + 1 \equiv 0 \bmod [2^{64}] \quad \wedge$
$dc \cdot 2^{64} + m = T_L \cdot N'$	$dc \cdot 2^{64} + m - T_L \cdot N' = 0 \quad \wedge$
$mN_H \cdot 2^{64} + mN_L = m \cdot N$	$mN_H \cdot 2^{64} + mN_L - m \cdot N = 0 \quad \wedge$
$carry \cdot (carry - 1) = 0$	$carry \cdot (carry - 1) = 0 \quad \wedge$
$carry \cdot 2^{64} + t_L = T_L + mN_L$	$carry \cdot 2^{64} + t_L - (T_L + mN_L) = 0 \quad \wedge$
$c \cdot (c - 1) = 0$	$c \cdot (c - 1) = 0 \quad \wedge$
$c \cdot 2^{64} + t = T_H + mN_H + carry$	$c \cdot 2^{64} + t - (T_H + mN_H + carry) = 0 \quad)$
ASSERT $t_L \equiv 0 \bmod [2^{64}]$	$\Longrightarrow t_L \equiv 0 \bmod [2^{64}]$

- The root entailment problem: given a system Σ of polynomial equations, verify whether all solutions to Σ are also solutions to the assertion.

Ideal Membership Problem

Root Entailment	Ideal Membership
$\forall N, N', m, T_L, T_H, mN_L, mN_H, t_L, t, dc, carry, c.$ $\big(N \times N' + 1 \equiv 0 \bmod [2^{64}] \;\wedge\; dc \cdot 2^{64} + m - T_L \cdot N' = 0 \;\wedge\; mN_H \cdot 2^{64} + mN_L - m \cdot N = 0 \;\wedge\; carry \cdot (carry - 1) = 0 \;\wedge\; carry \cdot 2^{64} + t_L - (T_L + mN_L) = 0 \;\wedge\; c \cdot (c - 1) = 0 \;\wedge\; c \cdot 2^{64} + t - (T_H + mN_H + carry) = 0 \big)$ $\Longrightarrow t_L \equiv 0 \bmod [2^{64}]$	$t_L \in \big\langle N \times N' + 1 - k \cdot 2^{64},\; dc \cdot 2^{64} + m - T_L \cdot N',\; mN_H \cdot 2^{64} + mN_L - m \cdot N,\; carry \cdot (carry - 1),\; carry \cdot 2^{64} + t_L - (T_L + mN_L),\; c \cdot (c - 1),\; c \cdot 2^{64} + t - (T_H + mN_H + carry),\; 2^{64} \big\rangle$

- $f \in \langle g_0, g_1, \ldots, g_n \rangle$ if $f = h_0 \cdot g_0 + h_1 \cdot g_1 + \cdots h_n \cdot g_n$ for some $h_0, h_1, \ldots, h_n$.
  - Given $f, g_0, g_1, \ldots, g_n$, the ideal membership problem checks if $f \in \langle g_0, g_1, \ldots, g_n \rangle$.
- The ideal membership problem is solved by computing Gröbner bases.

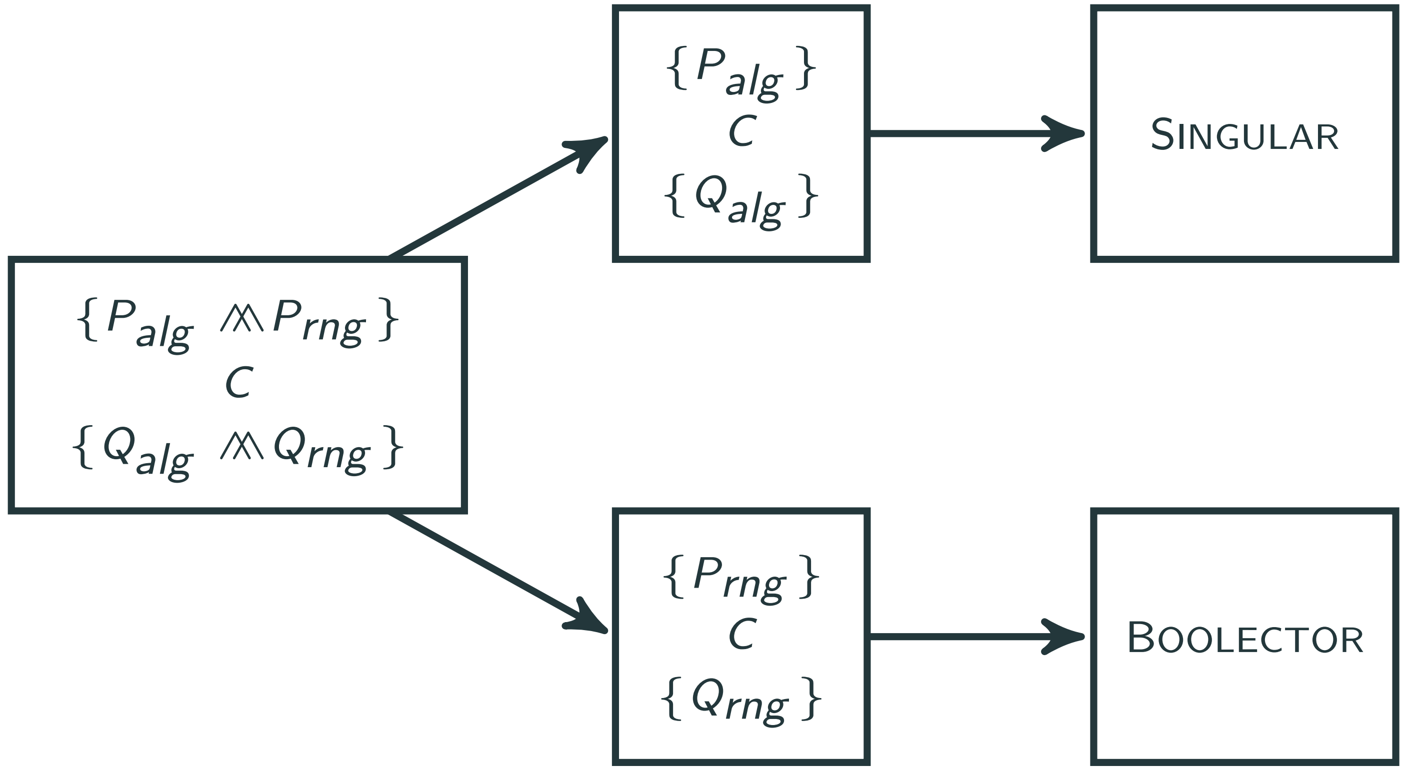
Ideal Membership Problem

Root Entailment	Ideal Membership
$\forall N, N', m, T_L, T_H, mN_L, mN_H, t_L, t, dc, carry, c.$ $(\quad N \times N' + 1 \equiv 0 \bmod [2^{64}] \quad \wedge$ $dc \cdot 2^{64} + m - T_L \cdot N' = 0 \quad \wedge$ $mN_H \cdot 2^{64} + mN_L - m \cdot N = 0 \quad \wedge$ $carry \cdot (carry - 1) = 0 \quad \wedge$ $carry \cdot 2^{64} + t_L - (T_L + mN_L) = 0 \quad \wedge$ $c \cdot (c - 1) = 0 \quad \wedge$ $c \cdot 2^{64} + t - (T_H + mN_H + carry) = 0 \quad)$ $\Longrightarrow \; t_L \equiv 0 \bmod [2^{64}]$	$t_L \in \left\langle \begin{array}{c} N \times N' + 1 - k \cdot 2^{64} \\ dc \cdot 2^{64} + m - T_L \cdot N' \\ mN_H \cdot 2^{64} + mN_L - m \cdot N \\ carry \cdot (carry - 1) \\ carry \cdot 2^{64} + t_L - (T_L + mN_L) \\ c \cdot (c - 1) \\ c \cdot 2^{64} + t - (T_H + mN_H + carry) \\ 2^{64} \end{array} \right\rangle$

- $f \in \langle g_0, g_1, \ldots, g_n \rangle$ if $f = h_0 \cdot g_0 + h_1 \cdot g_1 + \cdots h_n \cdot g_n$ for some $h_0, h_1, \ldots, h_n$.
  - Given $f, g_0, g_1, \ldots, g_n$, the ideal membership problem checks if $f \in \langle g_0, g_1, \ldots, g_n \rangle$.
- The ideal membership problem is solved by computing Gröbner bases.

CryptoLine

- CryptoLine is a formal verification tool for cryptographic assembly programs.
- It has two verification cores:
  - The algebraic core implements algebraic abstraction and employs computer algebra systems.
  - The range core employs SMT QF_BV solvers.
- CryptoLine verifies Montgomery reduction in 1 second.



Section 3

Certified Verification

Bugs in Verification?

- Verification tools are very complex programs themselves.
- A typical verification tool has the following phases:
  - A reduction phase transforms verification problems to well-established problems.
  - A proof phase employs efficient provers to solve well-established problems.
- Any mistake can lead to incorrect verification results.
- Many provers are known to have bugs.
- How much do you trust your verification tools?
  - “Model checkers are nice tools, but their results may be dubious.”

    Prof. Jean-François Monin, VERIMAG
- Besides, our competitors always complain our trusted computing base is large.

Formally Verified Algorithm

- CRYPTOLINE has several reduction phases:
  - It reduces CRYPTOLINE assertions to the ideal membership problem.
  - It reduces soundness conditions to SMT QF_BV queries.
  - It moreover reduces SMT QF_BV queries to SAT queries (bit blasting).
    - to avoid bugs in SMT QF_BV solvers
- All these reduction algorithms are specified and proven in COQ.
  - For example, consider bit_blast(ϕ) where ϕ is an SMT QF_BV query.
  - We give a formal COQ proof for the following theorem:

Theorem

For all SMT QF_BV query ϕ, ϕ is satisfiable if and only if the SAT query bit_blast(ϕ) is satisfiable.

Certified Results

- To ensure our queries are solved correctly, we ask external efficient provers to provide a certificate for each query.
  - Formally verified provers would be too inefficient.
  - SAT competition requires certificates since 2013.
- Two types of certificates are needed: one for ideal membership and the other for SAT.
- Each certificate is validated by an independent certificate checker.
  - To further improve assurance, we develop a formally verified certificate checker for ideal membership and use a formally verified certificate checker for SAT.

CoqQFBV and CoqCryptoLine

- We build two formally verified verification tools.
- CoqQFBV is a formally verified SMT QF_BV solver.
    - It is based on OCaml programs automatically extracted from Coq bit blasting algorithms.
    - It employs the formally verified SAT certificate checker GRAT.
- CoqCryptoLine is a formally verified verification tool for cryptographic assembly programs.
    - It is based on OCaml programs automatically extracted from our reduction algorithms.
    - It employs our formally verified certificate checker for the ideal membership problem.
- Model checkers can be trustful if we build them right.

Section 4

Experiments

Classical Cryptography

- We verify field arithmetic and group operations in two different curves from four different security libraries:
  - secp256k1: bitcoin
  - curve25519: boringSSL, nss, and OpenSSL.
- 47 cryptographic C programs are verified in experiments.
  - We obtain their GCC Gimple IR and translate them to CRYPTOLINE.
- Experiments are running on an Ubuntu 22.04 server with 4x 1.5 GHz AMD EPYC 7763 64-core CPUs.

Results i

L_{CL}: lines of CryptoLine instructions

T_{CCL}: time took by CoqCryptoLine

T_{CL}: time took by CryptoLine

Function	L_{CL}	T_{CCL}	T_{CL}	Function	L_{CL}	T_{CCL}	T_{CL}
bitcoin/asm/secp256k1_fe_*							
mul_inner	269	91.58	4.46	sqr_inner	226	39.22	2.64
bitcoin/field/secp256k1_fe_*							
add	35	0.09	0.02	cmov	95	3.14	0.03
mul_inner	172	76.81	3.26	mul_int	26	1.15	0.02
negate	31	0.62	0.03	sqr_inner	155	46.85	1.90
from_storage	100	0.14	0.03	normalize_weak	36	0.30	0.05
bitcoin/group/							
secp256k1_ge_neg					51	0.32	0.04
secp256k1_ge_from_storage					100	0.20	0.04
secp256k1_gej_double_var.part.14					1347	1578.91	30.37
bitcoin/scalar/secp256k1_scalar_*							
add	152	2.95	0.12	mul_512	478	55.60	3.61
mul	1232	310.87	11.83	reduce	147	2.01	0.11
sqr	1193	249.39	9.46	sqr_512	439	40.09	4.00
secp256k1_scalar_reduce_512					754	86.16	3.56
boringssl/fiat_curve25519/fe_*							
add	35	0.08	0.02	mul_impl	152	71.25	3.44
sub	40	0.10	0.03	sqr_impl	124	36.69	1.88
fe_mul121666					74	1.40	0.14
x25519_scalar_mult_generic					1530	1257.98	346.05
boringssl/fiat_curve25519_x86/fe_*							
add	70	0.16	0.03	mul_impl	435	109.97	3.05
sqr_impl	339	41.12	1.68	sub	80	0.23	0.06
fe_mul121666					136	2.35	0.15
x25519_scalar_mult_generic					4247	5305.38	305.46

Results i

Function	L_{CL}	T_{CCL}	T_{CL}	Function	L_{CL}	T_{CCL}	T_{CL}
bitcoin/asm/secp256k1_fe_*							
mul_inner	269	91.58	4.46	sqr_inner	226	39.22	2.64
bitcoin/field/secp256k1_fe_*							
add	35	0.09	0.02	cmov	95	3.14	0.03
mul_inner	172	76.81	3.26	mul_int	26	1.15	0.02
negate	31	0.62	0.03	sqr_inner	155	46.85	1.90
from_storage	100	0.14	0.03	normalize_weak	36	0.30	0.05
bitcoin/group/							
secp256k1_ge_neg					51	0.32	0.04
secp256k1_ge_from_storage					100	0.20	0.04
secp256k1_gej_double_var.part.14					1347	1578.91	30.37
bitcoin/scalar/secp256k1_scalar_*							
add	152	2.95	0.12	mul_512	478	55.60	3.61
mul	1232	310.87	11.83	reduce	147	2.01	0.11
sqr	1193	249.39	9.46	sqr_512	439	40.09	4.00
secp256k1_scalar_reduce_512					754	86.16	3.56
boringssl/fiat_curve25519/fe_*							
add	35	0.08	0.02	mul_impl	152	71.25	3.44
sub	40	0.10	0.03	sqr_impl	124	36.69	1.88
fe_mul121666					74	1.40	0.14
x25519_scalar_mult_generic					1530	1257.98	346.05
boringssl/fiat_curve25519_x86/fe_*							
add	70	0.16	0.03	mul_impl	435	109.97	3.05
sqr_impl	339	41.12	1.68	sub	80	0.23	0.06
fe_mul121666					136	2.35	0.15
x25519_scalar_mult_generic					4247	5305.38	305.46

Field operations

L_{CL}: lines of CryptoLine instructions

T_{CCL}: time took by CoqCryptoLine

T_{CL}: time took by CryptoLine

Results i

Function	L_{CL}	T_{CCL}	T_{CL}	Function	L_{CL}	T_{CCL}	T_{CL}
bitcoin/asm/secp256k1_fe_*							
mul_inner	269	91.58	4.46	sqr_inner	226	39.22	2.64
bitcoin/field/secp256k1_fe_*							
add	35	0.09	0.02	cmov	95	3.14	0.03
mul_inner	172	76.81	3.26	mul_int	26	1.15	0.02
negate	31	0.62	0.03	sqr_inner	155	46.85	1.90
from_storage	100	0.14	0.03	normalize_weak	36	0.30	0.05
bitcoin/group/							
secp256k1_ge_neg					51	0.32	0.04
secp256k1_ge_from_storage					100	0.20	0.04
secp256k1_gej_double_var.part.14					1347	1578.91	30.37
bitcoin/scalar/secp256k1_scalar_*							
add	152	2.95	0.12	mul_512	478	55.60	3.61
mul	1232	310.87	11.83	reduce	147	2.01	0.11
sqr	1193	249.39	9.46	sqr_512	439	40.09	4.00
secp256k1_scalar_reduce_512					754	86.16	3.56
boringssl/fiat_curve25519/fe_*							
add	35	0.08	0.02	mul_impl	152	71.25	3.44
sub	40	0.10	0.03	sqr_impl	124	36.69	1.88
fe_mul121666					74	1.46	0.14
x25519_scalar_mult_generic					1530	1257.98	346.05
boringssl/fiat_curve25519_x86/fe_*							
add	70	0.16	0.03	mul_impl	435	109.97	3.05
sqr_impl	339	41.12	1.68	sub	80	0.23	0.06
fe_mul121666					136	2.35	0.15
x25519_scalar_mult_generic					4247	5305.38	305.46

L_{CL}: lines of CryptoLine instructions

T_{CCL}: time took by CoqCryptoLine

T_{CL}: time took by CryptoLine

Group operations

Results ii

L_{CL}: lines of CryptoLine instructions

T_{CCL}: time took by CoqCryptoLine

T_{CL}: time took by CryptoLine

Function	L_{CL}	T_{CCL}	T_{CL}	Function	L_{CL}	T_{CCL}	T_{CL}
nss/Hacl_Curve25519_51/							
fadd0	20	0.11	0.03	fsub0	25	0.15	0.04
fmul0	146	165.11	32.84	fmul1	81	15.09	0.57
fsqr0	112	69.36	5.17	fsqr20	224	124.89	5.11
fmul20					276	230.15	37.69
point_add_and_double					1483	3240.20	465.32
point_double					729	1352.25	24.55
openssl/curve25519/fe51_*							
add	35	0.10	0.03	sub	50	0.09	0.03
mul	147	59.98	2.63	sq	119	34.53	1.50
fe51_mul121666					75	1.16	0.13
x25519_scalar_mult					1481	1598.86	306.86

- CryptoLine finishes all cases within 10 minutes.
  - Field arithmetic is verified in a minute. Point addition is verified in 10 minutes.
- CoqCryptoLine finishes all cases within 90 minutes.
  - Field arithmetic is verified in 5 minutes. Point addition is verified in 90 minutes.
- Some point addition programs are verified but not fully certified (missing 1 out of 3).

Post Quantum Cryptography

- Classical cryptography will be broken by large-scale quantum computers.
  - RSA and elliptic curve cryptography
- To retain security on classical computers, post quantum cryptography is developed to prevent quantum attacks.
  - Note that post quantum cryptography is running on classical computers.
- NIST called for PQC competition in 2016 and announced winners in 2022.
- Three (Kyber for KEM, Dilithium, SPHINCS+ for DSA) have been standardized, and one (FALCON for DSA) will be standardized in a few months.

Results

- Kyber is a lattice-based PQC KEM.
- It uses the polynomial ring $\mathbb{F}_q[X]/\langle X^{256}+1\rangle$ with $q = 3329$.
- Each $f \in \mathbb{F}_q[X]/\langle X^{256}+1\rangle$ is of the form $\sum_{i=0}^{255} c_i X^i$ with $c_i \in \mathbb{F}_q$ for all i.
- Let $f = \sum_{i=0}^{255} c_i X^i, g = \sum_{i=0}^{255} d_i X^i \in \mathbb{F}_q[X]/\langle X^{256}+1\rangle$. Define
  - $f \pm g = (f \pm g) \bmod q = \sum_{i=0}^{255}(c_i \pm d_i \bmod q) \cdot X^i$.
  - $f \times g = h \bmod X^{256} + 1$ where $h = (f \cdot g) \bmod q$.
- To compute $f \times g$, Kyber specification uses a discrete Fourier transform called Number Theoretic Transform (NTT).
  - $\mathbb{F}_q[X]/\langle X^{256}+1\rangle \cong \mathbb{F}_q[X]/\langle X^{128}-1729\rangle \times \mathbb{F}_q[X]/\langle X^{128}+1729\rangle$ $(1729^2 \equiv -1 \bmod 3329)$

Function	L_{CL}	T_{CCL}	T_{CL}
PQClean/kyber/NTT			
PQCLEAN_KYBER512_CLEAN_ntt	10375	2641.49	92.22
PQCLEAN_KYBER768_AVX2_ntt	8975	1047.99	92.23

Hash Block Functions

- Hash functions are widely used in cryptography.
- Typical hash functions compute by blocks.
- Such hash block functions need be very efficient.
  - OpenSSL has 6 assembly implementations for SHA-256 and 5 for SHA-3.
- We also develop techniques to verify them.
  - Our technique converts assembly and reference implementations to logic circuits and applies logic equivalence checking.

Has Any Bug Been Found?

- Microsoft Research also entered the NIST PQC competition.
- SIDH is an isogeny-based PQC.
- Its source code is available at PQCrypto-SIDH.
- CryptoLine found an error in the aarch64 implementation of the field multiplication.
- SIDH was broken in 2022.

Conclusion

- Real-world cryptographic assembly programs are formally verified in reasonable time.
  - CryptoLine: 10 minutes (uncertified) or CoqCryptoLine: 90 minutes (certified)
- An effective high-assurance formal verification tool is built.
  - verification + certification
- We are actively verifying PQC assembly implementations.
  - Both avx2 and aarch64 implementations for Dilithium NTT are verified in July 2024.
- Hopefully, we will have correct and efficient PQC libraries in a few years.

Thank you for your attention.

Question?