Lower-level Verifications for Cryptographic Software

involving Elliptic Curves and others

Bo-Yin Yang

Academia Sinica

ECC 2018, November 20, Osaka

Verification

- Verification: the study of showing how something works as designed. The discipline considers "worst cases" by design.
  - Tries to show that there are no failure possiblities; and
  - ideally identifies possible failures if we cannot verify correctness.
- The most well-established application of verification is in chip design.
- We will apply it to cryptographic software.

Verification in Practice

- Usually carried out with
  - Proof Assistants, such as COQ
  - Satisfiability Module Theory (SMT) and SAT solvers, e.g. MINISAT.
  - Specifically designed tools
- We will use SAT solvers and some home-brewed tools

Cryptography and Its Software as a Subject of Study I

- Cryptography has lots of real world applications from private communication to digital currency.
- Similar to formal verification, cryptography necessarily expects the worst scenario.
- Modern cryptography uses much sophisticated, complex mathematical structures.
- Secure cryptosystems must be designed and analyzed thoroughly.
  - There is little room for trial and error in cryptography.

Cryptography and Its Software as a Subject of Study II

- The sophisticated mathematical structures in modern cryptography often require complicated arithmetic computation over large numbers.
  - In RSA, modulo arithmetic over $n = pq$ where p, q are prime.
  - In NIST P-256, modular arithmetic over $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$.
  - In Curve25519, modular arithmetic over $2^{255} - 19$.
- Commodity computers only support up to 64-bit integers.
  - This makes the program even more complicated.

Cryptography and Its Software as a Subject of Study III

- To make cryptography practical, cryptographers must design cryptosystems for security and efficiency.
- Parameters are chosen for efficiency, not for a reader's understanding.
  - Reduction in $GF(2^{256} - 2^{224} + 2^{192} + 2^{96} - 1)$, performed through bitwise masking and shifting (NIST P-256);
  - Reduction in $GF(2^{255} - 19)$ performed by bitwise shifting and multiplication (X25519).
- To attain the best performance, primitive cryptographic algorithms are even often implemented in assembly.
  - OpenSSL and boringSSL.
- Not many cryptographers also program assembly language well.

An Ideal Research Problem for Verification

- Not all programs need to be verified.
- However, cryptographic programs are
  - critical
  - indispensable
  - complex
  - highly visible
- Moreover, practical cryptographers do appreciate verification.
  - See comments in OpenSSL
- Colleagues recognize the importance of verification when informed of this work.
  - Many computer scientists know of OpenSSL.

Challenges I

- Verifying non-linear computation is hard.
  - Cryptographic assembly programs perform such computation in hundreds of bits.
- Such programs must be proven correct for all inputs.
  - For cryptographic assembly programs, every bit and flag count.
- Assembly programs are very succinct.
  - Abstraction is unlikely to work.

Challenges II

- An algorithm has different instantiations on different mathematical structures.
- Consider, say, modular multiplication.
  - In NIST P-256, modular multiplication is over $GF(2^{256} - 2^{224} + 2^{192} + 2^{96} - 1)$ (256 bits).
  - In X25519, modular multiplication is over $GF(2^{255} - 19)$ (255 bits).
- Since numbers are different, reduction is computed differently.
  - In NIST P-256, it is implemented by bitwise masks and shifts.
  - In X25519, it is implemented by bitwise shifts and multiplication.
- Each instantiation must be verified.

Challenges III

- Algorithm instance implement differently on different architectures.
- Different architectures (x86, ARM) have different instruction sets.
- Different generations of x86_64 have slightly different instructions.
- In OpenSSL, two different implementations for modular multiplication are available.
  - In Broadwell microarchitecture, it is possible to perform two threads of addition simultaneously with `adox`.
- Vectorized instructions are also widely used.
  - OpenSSL has 3 Poly1305 implementations (sequential, avx, avx2).
- All implementations need to be verified.

Related Work

- Fiat (MIT) is a C program synthesis tool for cryptographic programs.
- Jasmin (INRIA) is a portable assembly language with formal semantics.
- HACL* (INRIA) is a verified cryptographic library in F*.
- Vale (Microsoft Research) is a framework to write correct assembly programs for different architectures.
- None of them really addresses the cryptographic assembly program verification problem.

Previous Work

- Our first idea is to verify cryptographic assembly programs by SMT/SAT solvers via bit blasting.
- In 2014, we use BOOLECTOR to verify an academic implementation of modular multiplication in X25519.
  - It took 4 days (without annotation) or 5 hours (with extensive manual annotation).
  - Moreover, we had to prove a simple mathematical property in COQ.
- Verifying a hundred of assembly instructions in 4 days is perhaps better than using proof assistants.
- Not very useful!

The gfverif Project

- In 2015, Daniel J. Bernstein and Peter Schwabe announces their gfverif project.
- Their tool verifies algebraic properties of C programs using a computer algebra system.
- Idea:
  - Translate a C program and its specification to an algebraic problem;
  - Solve the algebraic problem by a computer algebra system.
- It sounds reasonable.
  - Why do we use SMT/SAT solvers to solve algebraic problems?

An Almost Certified Automatic Verification Tool

- In 2017, we extend the idea of gfverif to assembly programs and certify algebraic results with Coq.
- Unfortunately, results from SMT/SAT solvers are yet to be certified.
  - Efficient certification implies P = coNP.
- This tool verifies the same academic implementation of modular multiplication in 1.5 minutes without annotation.
- It also verifies an academic implementation of Montgomery ladderstep (about 1300 instructions) in 5.5 days.
  - Montgomery ladderstep is used in elliptic curve point operations.
- It is probably useful.
  - suitable for production release, not for daily development
  - not industrial implementation
  - we translated from qhasm (X25519), so not many instances

More Recent Work

- We further optimize our tool.
- We verify industrial implementations in OpenSSL and boringSSL.
- We verify the OpenSSL multi-precision Montgomery modular multiplication for RSA, and its implementation for NIST P-256.
- We also verify the boringSSL Montgomery ladderstep implementation for X25519.
  - Previously, we only verify an academic implementation for X25519.
- We also decide *not* to certify the tool.
  - Main reason: lack of manpower.

CryptoLine

- The CryptoLine tool consists of three parts:
  - the modeling language for cryptographic assembly programs
  - the specification language for functional properties
  - the verification algorithm
- We also provide a tool chain to
  - extract assembly programs from execution
  - translate assembly programs into the modeling language
- The tool chain enables us to produce models for verification quickly.
  - It is essential to tool adoption.

The CryptoLine Modeling Language I

- CryptoLine covers common assembly instructions used in cryptographic programs.
  - bvAssign (assignment)
  - bvAdd, bvAddC, bvAdc, bvAdcC (addition)
  - bvSub, bvSubC, bvSub, bvSbbC(subtraction)
  - bvMul, bvMulf (multiplication)
  - bvShl, bvConcatShl (left shift)
  - bvSplit (splitting)
  - bvCmove (condition move)
  - bvAssert, bvAssume (assertion and assumption)
- Flags must be specified explicitly.
  - Missing flags induce under- or over-flow checks (bvAdd and bvSub).

The CRYPTOLINE Modeling Language II

- Special instructions are added for modeling purposes.
  - bvConcatShl (concatenate then shift), Split (split into parts), bvCmove (conditional move)
  - more about this in case study
- Instructions for verification are available.
  - bvAssert and bvAssume
- There is no branching instruction.
  - In practical cryptography, running time is a side channel.
  - Cryptographic programs need be data-independent (called *constant-time*).
  - Secret-Dependent Branches are not allowed.

The CryptoLine Specification Language I

- The CryptoLine specification language specifies a conjunction of range and algebraic properties:
  - Range properties: $E < E'$ or $E \leq E'$.
  - Algebraic properties: $E = E'$ or $E \equiv E' \mod E''$.
- We also add syntactic sugar for common expressions.
  - For instance, $[c_0 : c_1 : \cdots : c_k]$ stands for $\sum_{i=0}^{k} c_i \times 2^{64 \cdot i}$.

The CRYPTOLINE Specification Language II

- For instance, the multiplication in X25519 is specified by

$$\left\{ \begin{array}{c} a_0 < 2^{52} \wedge a_1 < 2^{52} \wedge a_2 < 2^{52} \wedge a_3 < 2^{52} \wedge a_4 < 2^{52} \wedge \\ b_0 < 2^{52} \wedge b_1 < 2^{52} \wedge b_2 < 2^{52} \wedge b_3 < 2^{52} \wedge b_4 < 2^{52} \\ \Lambda\!\!\!\Lambda \\ \top \end{array} \right\}$$

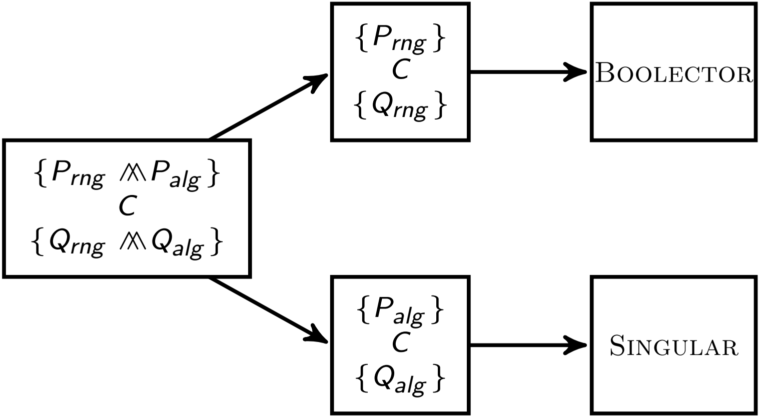
$$MUL([r_0 : r_1 : r_2 : r_3 : r_4], [a_0 : a_1 : a_2 : a_3 : a_4], [b_0 : b_1 : b_2 : b_3 : b_4])$$

$$\left\{ \begin{array}{c} r_0 < 2^{52} \wedge r_1 < 2^{52} \wedge r_2 < 2^{52} \wedge r_3 < 2^{52} \wedge r_4 < 2^{52} \\ \Lambda\!\!\!\Lambda \\ (a_0 + a_1 \cdot 2^{52} + a_2 \cdot 2^{104} + a_3 \cdot 2^{156} + a_4 \cdot 2^{208}) \times (b_0 + b_1 \cdot 2^{52} + b_2 \cdot 2^{104} + b_3 \cdot 2^{156} + b_4 \cdot 2^{208}) \equiv \\ r_0 + r_1 \cdot 2^{52} + r_2 \cdot 2^{104} + r_3 \cdot 2^{156} + r_4 \cdot 2^{208} \mod (2^{255} - 19) \end{array} \right\}$$

- Notice that 256-bit numbers are divided into 5 51-bit limbs.

Hybrid Verification Technique

- Here is the CryptoLine verification algorithm:



Verifying Range Specifications

- CRYPTOLINE translates a program and its range specification to a formula in the SMT quantifier-free bit vector theory.
- The formula is unsatisfiable iff the program fulfills its range specification.
- We use BOOLECTOR to check the satisfiability of the formula.
- BOOLECTOR+MINISAT works better for most cases.
- A handful of cases need BOOLECTOR+LINGELING.
- Both BOOLECTOR and Z3 fail for a number of realistic assembly programs.

Verifying Algebraic Specifications I

- CryptoLine first checks there is no overflow using SMT/SAT.
- It then translates a program and its algebraic specification to the ideal membership problem.
  - A set $I \subseteq \mathbb{Z}[x_0, x_1, \ldots, x_n]$ is an *ideal* if $f + g, p \cdot f \in I$ for every $f, g \in I$ and $p \in \mathbb{Z}[x_0, x_1, \ldots, x_n]$.
  - Given an ideal I and a polynomial $p \in \mathbb{Z}[x_0, x_1, \ldots, x_n]$, the ideal membership problem asks if $p \in I$.
- $p \in I$ implies the program fulfills its algebraic specification.
- We use Singular to solve the ideal membership problem.

Verifying Algebraic Specifications II

- To see how it works, consider a system of polynomial equations $f_i(\overline{x}) = 0$ derived from assembly instructions.
  - For instance, `mul %rcx` translates to $\%rdx' \times 2^{64} + \%rax' = \%rax \times \%rcx$.
- Suppose we want to prove an equality $g(\overline{x}) = 0$.
- Formally, we want to show $\forall \overline{x}. \bigwedge_i f_i(\overline{x}) = 0 \implies g(\overline{x}) = 0$.
- Then $g(\overline{x}) \in \langle f_1(\overline{x}), f_2(\overline{x}), \ldots, f_k(\overline{x}) \rangle$ implies $\forall \overline{x}. \bigwedge_i f_i(\overline{x}) = 0 \implies g(\overline{x}) = 0$.
  - $g(\overline{x}) = \sum_i h_i(\overline{x}) f_i(\overline{x}) = 0$ for any $\overline{x}$ such that $\bigwedge_i f_i(\overline{x}) = 0$.

Verification Flow

- Here are the verification steps:
  1. Compile into a standalone program.
     - `gcc ecp_nistz256_mul.c $OPENSSLDIR/libcrypto.a`
  2. Extract execution trace.
     - `itrace.py a.out ecp_nistz256_mul_mont > ecp_nistz256_mul_mont.gas`
  3. Manually add x86_64 to CryptoLine translation rules.
  4. Apply the translation rules.
     - `to_bvdsl.py ecp_nistz256_mul_mont.gas > ecp_nistz256_mul_mont.cl`
  5. Manually add pre- and post-conditions.
  6. Manually tune the CryptoLine program to match semantics.
     - More about this later.
  7. Run the tool.
     - `cv.native ecp_nistz256_mul_mont.cl`

Current Requirements

All available for stock Ubuntu server install.

- O'Caml Package Manager (`opam`)
  - With O'Caml 4.07.0
  - With `lwt`, `lwt_ppx`, `num` packages
- SINGULAR version 4
- BOOLECTOR-3.0.0 with LINGELING, MINISAT, EDICAL.

Translation Rules

- The PYTHON script `to_bvdsl.py` translates x86_64 assembly to CRYPTOLINE by rules provided by users.
- Consider the following rule:

```
mov $1v, $2v -> bvAssign $2v (bvVar $1v)
```

- It translates `mov %rbp, %rax` to `bvAssign rax (bvVar rbp)`.
- Here is another rule:

```
add $1v, $2v -> bvAddC carry $2v (bvVar $1v) (bvVar $2v)
```

- It translates `add %rax, %r9` to `bvAddC carry r9 (bvVar rax) (bvVar r9)`.
- Most assembly instructions are thus translated automatically.

Fine Tune

- Consider the fragment:

```
mov %r8, %rbp
shl $0x20, %r8
shr $0x20, %rbp
```

- What it does is to assign
  - the high 32 bits of old `%r8` to the low 32 bits of `%rbp`; and
  - the low 32 bits of old `%r8` to the high 32 bits of `%r8`.
- Manual translation is needed.
- Here is the correct translation:

```
bvSplit rbp r8 (bvVar r8) 32;
bvShl r8 (bvVar r8) 32;
```

- Only 4 manual translations are needed in `ecp_nistz256_mul_mont`.

Evaluation on a 2.8GHz Broadwell Xeon

library	program	ln	assert	range	alg	total
OpenSSL	ecp_nistz256_add	89	0.44	4.17	0.03	4.63
	ecp_nistz256_sub	88	-	18.54	~0	18.55
	ecp_nistz256_from_mont	82	-	0.41	0.02	0.45
	ecp_nistz256_mul_mont	192	-	21.49	0.03	21.53
	ecp_nistz256_mul_mont^{+}	153	-	15.43	0.03	15.47
	ecp_nistz256_mul_by_2	49	-	0.05	0.02	0.08
	ecp_nistz256_sqr_mont	148	-	16.43	0.03	16.47
	ecp_nistz256_sqr_mont^{+}	131	-	22.50	0.03	22.54
	x86_64_mont_2	228	832.60	13.41	0.03	846.05
	x86_64_mont_4	490	8279.87	523.27	0.91	8804.06
boringSSL	x25519_x86_64_mul	226	-	28.73	0.03	28.78
	x25519_x86_64_sqr	171	-	6.14	0.03	6.18
	x25519_x86_64_ladderstep	1459	-	2921.82	107.93	3029.78
mbedTLS	mbedtls_mpi_mul_mpi_2	76	0.46	0.42	0.03	0.92
	mbedtls_mpi_mul_mpi_4	249	12.85	9.27	0.02	22.16

- Time is in seconds; + is for Broadwell architectures
- In 2017, X25519 modular multiplication and Montgomery ladderstep took 90 seconds and 5.5 days respectively.
- CryptoLine is useful even for daily development!

Recent Activity

Active Research on CryptoLine

- CryptoLine now supports compositional reasoning and is multi-threaded.
- Montgomery ladderstep in boringSSL is verified in 307 seconds.
  - was 3029 seconds
- For multi-precision Montgomery modular multiplication:
  - 256-bit version is verified in 7.5 seconds (was 8804 seconds).
  - 1024-bit version is verified in 295 seconds.

New stuff

- We are extending our efforts to postquantum crypto
- We are extending verification to compiler intermediate representations

Verification of Postquantum Crypto I

Lattice-based encryption schemes

- NTT-based Ring-LWE: Kyber, NewHope
- non-NTT based Ring-LWE: NTRU, NTRU Prime
- Others: Frodo

NTT-based Ring-LWE

- Verified $n = 256$ NTT and inverse NTT $\pmod{7681}$ for Kyber.
- working ongoing on the similar NewHope

non-NTT-based Ring-LWE

NTRU and NTRU Prime should be doable, under study

Verification of Postquantum Crypto II

Other classes of PQC than Lattices with Work in Progress:

- Multivariates: should be doable, operations in $GF(2^k)$ or small $GF(p)$.
- Coding-bases: should be doable, operations in $GF(2^k)$.
- Supersingular Isogenies: experience from ECC/RSA valuable?

Not on the docket

Hash-based: not our domain

Verification of Compiler Intermediate Representations

Why not Assembly

- We can't have assembly for every architecture
- For reference implementations, clarity and correctness are more important than efficiency
- Similarly for prototypes of algorithms.

Why not C itself?

- CompCert and similar certified compilers are seldom used for production work.
- Standard compilers (`gcc` and `clang`) do strange things to your code.

clang Strangeness on OpenSSL code I

Taken from https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/crypto/ec/curve25519.

From fe51_mul121666 in curve25519.c

```
u128 h2 = f[2] * (u128)121666;
g2 = (uint64_t)h2 & MASK51;
```

Constant MASK51=0x7FFFFFFFFFFF

clang Intermediate Representation

```
h2 = mul i128 f_2 121666;
conv15 = trunc h2;
g2 = and i64 conv15 0x7FFFFFFFFFFE
```

clang Strangeness on OpenSSL code II

From function fe51_mul in curve25519.c

```
g2 = (uint64_t)h2 & MASK51;
g2 += (uint64_t)(h1 >> 51);
g3 += g2 >> 51;
g2 &= MASK51;
```

clang IR output

```
conv109 = trunc h2                                  //(uint64_t)h2
...
shr122 = lshr i128 h1 51
conv123 = trunc shr122                              //(uint64_t)(h1>>51)
g2 = and i64 conv109 0x7FFFFFFFFFFF
add124 = add i64 conv123 g2
...                                                 //g3 += g2>>51
fold = add i64 conv123 conv109
and135 = and i64 fold 0x7FFFFFFFFFFF
```

What we have done with `clang` IR I

- Identify a subset LLVMCRYPTOLINE of `clang` IR in use for crypto
- Translate LLVMCRYPTOLINE to CRYPTOLINE.
- Add assertions and assumptions as needed.
- Hand-adjust as needed.
- Verify.

What we have done with `clang` IR II

program	function	loc (IR)	modified	time (s)
ecp_nistp224.c	felem_diff_128_64	30	×	0.35
	felem_diff	30	×	0.26
	felem_mul_reduce	99	✓	18.10
	felem_mul	60	×	5.34
	felem_neg	47	✓	0.74
	felem_reduce	75	✓	1.40
	felem_scalar	15	×	0.10
	felem_square_reduce	79	✓	16.40
	felem_square	43	×	0.97
	felem_sum	22	×	0.15
	widefelem_diff	54	×	0.77
	widefelem_scalar	31	×	1.19
ecp_nistp521.c	felem_diff128	61	×	0.44
	felem_diff64	61	×	0.50
	felem_neg	43	×	0.34
	felem_scalar128	36	×	0.62
	felem_scalar64	35	×	0.21
	felem_scalar	43	×	0.24
	felem_sum64	52	×	0.19
	felem_reduce	144	✓	1.81
	felem_diff_128_64	70	×	-
	felem_mul	289	×	-
	felem_square	158	×	-

Note that the three unverified programs contain anomalies which we suspect are possible mistakes in range specification.

What we have done with `clang` IR III

program	function	loc (IR)	modified	time (s)
ecp_nistp256.c	felem_shrink	63	✓	1.33
	felem_small_mul	111	×	10.24
	felem_small_sum	26	×	0.14
	felem_sum	22	×	0.14
	smallfelem_mul	109	✓	1.79
	smallfelem_neg	22	×	0.07
	smallfelem_square	70	✓	1.80
curve25519.c	fe51_add	32	×	0.06
	fe51_mul121666	57	✓	0.18
	fe51_mul	124	✓	1.88
	fe51_sq	94	✓	0.79
	fe51_sub	37	×	0.11
	x25519_scalar_mult[1]	1235	✓	871.00

[1] Only the part of Montgomery Ladderstep is verified.

Conclusions

- For the first time, we are able to verify industrial low-level cryptographic programs practically.
  - 5 minutes for 1400 assembly instructions!
- This project combines several techniques:
  - SMT/SAT solving and computer algebra
- Formal verification and practical cryptography is a perfect match.
  - Practical cryptography needs efficient and correct programs.
  - Formal verification needs real applications.
- Lots of new opportunities in high assurance cryptographic software.

Thanks to

My Colleagues at IIS, Sinica

- Ming-Hsien Tsai
- Bow-Yaw Wang

Also must thank these smart people

- Jiaxiang Liu and Xiaomu Shi, Shenzhen University
- Andy Polyakov, OpenSSL

Thank you for your attention.
Question?