Learning about Computer Arithmetic by Formally Verifying It

John Harrison
Amazon Web Services

RAIM Meeting 2025

Wed 5th Nov 2025 (11:00–12:00)

1998-2017: Verifying floating-point arithmetic at Intel

At the IEEE floating-point meeting 2006

2018-?: Verifying crypto bignums at AWS

AWS's Automated Reasoning Group in 2019

Similarities

Similarities

- When starting, largely ignorant about the subject matter

Similarities

- When starting, largely ignorant about the subject matter
- Plenty of real problems motivate the application of verification

Similarities

- When starting, largely ignorant about the subject matter
- Plenty of real problems motivate the application of verification
- Much information is/was ‘folklore’, not available in textbooks

Similarities

- When starting, largely ignorant about the subject matter
- Plenty of real problems motivate the application of verification
- Much information is/was ‘folklore’, not available in textbooks
- At least we have precise specifications

Similarities

- When starting, largely ignorant about the subject matter
- Plenty of real problems motivate the application of verification
- Much information is/was ‘folklore’, not available in textbooks
- At least we have precise specifications . . . more or less

Similarities

- When starting, largely ignorant about the subject matter
- Plenty of real problems motivate the application of verification
- Much information is/was 'folklore', not available in textbooks
- At least we have precise specifications . . . more or less
- Technological progress opens up new sub-areas of application

Similarities

- When starting, largely ignorant about the subject matter
- Plenty of real problems motivate the application of verification
- Much information is/was 'folklore', not available in textbooks
- At least we have precise specifications . . . more or less
- Technological progress opens up new sub-areas of application
- Many mathematical similarities and analogies

Differences

Differences

- Open-sourcing
  - Intel work was internal and never made public
  - Cryptographic code an open-source library s2n-bignum

Differences

- Open-sourcing
  - Intel work was internal and never made public
  - Cryptographic code an open-source library s2n-bignum
- Popularity/competition
  - Floating-point verification was relatively new when I started
  - Everyone and their dog did cryptographic FV even in 2018

Differences

- Open-sourcing
  - Intel work was internal and never made public
  - Cryptographic code an open-source library s2n-bignum
- Popularity/competition
  - Floating-point verification was relatively new when I started
  - Everyone and their dog did cryptographic FV even in 2018
- Implementation and verification (much more satisfying)
  - Intel work was initially verifying existing code, but eventually led to new implementations
  - s2n-bignum was from the start implementation and verification together

Differences

- Open-sourcing
  - Intel work was internal and never made public
  - Cryptographic code an open-source library s2n-bignum
- Popularity/competition
  - Floating-point verification was relatively new when I started
  - Everyone and their dog did cryptographic FV even in 2018
- Implementation and verification (much more satisfying)
  - Intel work was initially verifying existing code, but eventually led to new implementations
  - s2n-bignum was from the start implementation and verification together
- We want efficiency and correctness

Differences

- Open-sourcing
  - Intel work was internal and never made public
  - Cryptographic code an open-source library s2n-bignum
- Popularity/competition
  - Floating-point verification was relatively new when I started
  - Everyone and their dog did cryptographic FV even in 2018
- Implementation and verification (much more satisfying)
  - Intel work was initially verifying existing code, but eventually led to new implementations
  - s2n-bignum was from the start implementation and verification together
- We want efficiency and correctness *and* security

Differences

- Open-sourcing
  - Intel work was internal and never made public
  - Cryptographic code an open-source library s2n-bignum
- Popularity/competition
  - Floating-point verification was relatively new when I started
  - Everyone and their dog did cryptographic FV even in 2018
- Implementation and verification (much more satisfying)
  - Intel work was initially verifying existing code, but eventually led to new implementations
  - s2n-bignum was from the start implementation and verification together
- We want efficiency and correctness *and* security
  - 'Constant-time' code

Motivations for FP verification

RESEARCH NEWS

MATHEMATICS

How Number Theory Got the Best of the Pentium Chip

Chalk one up for number theory. With lurid accounts of the flaw in Intel's Pentium processor making front-page and network news, users of the personal computer chip in fields ranging from science to banking are finding cases where its faulty logic sends their computations awry. But the problem might have gone undetected for much longer if the chip had not slipped up months ago during a long series of calculations in number theory, raising the suspicions of a dogged mathematics professor.

To other mathematicians, the discovery of the flaw by Thomas Nicely of Lynchburg College in Virginia emphasizes the value of number theory—the study of subtle properties of ordinary counting numbers—for providing quality control for new computer systems. By forcing a computer to perform simple operations repeatedly on many different numbers, number-theory calculations "push machines to their limits," says Peter Borwein of Simon Fraser University in Burnaby, British Columbia. Many computer makers have adopted these calculations as a shakedown test for systems intended for heavy-duty scientific computation, and although the practice has yet to spread to personal computers, Borwein and some other mathematicians think that might be a good idea.

Intel had actually found the flaw by other means after the chip had gone into production, but had decided that it was not likely to affect ordinary users. But the company hadn't counted on the use that Nicely had in mind. When he fired up a Pentium computer last March, Nicely was adding its number-crunching power to a project in computational number theory he had begun the year before. He was trying to improve on previous estimates of a number called Brun's sum, which is related to the distribution of prime numbers.

The sequence of prime numbers—2, 3, 5, 7, 11, 13, 17, 19, etc.—is a continuing source of fascination to mathematicians. Since the time of Euclid, they have known that there are infinitely many primes, but although primes are relatively abundant early on, they become scarce among larger numbers. For example, roughly 23% of two-digit numbers are prime (21 of 90), but the figure for ten-digit numbers is just 4%, and among hundred-digit numbers, the fraction of primes is less than half a percent. As a consequence, the gap between consecutive prime numbers tends to increase. However, every so often two odd numbers in a row turn out to be prime: 3 and 5, 41 and 43, 101 and 103, and 10,007 and 10,009, for example.

Mathematicians conjecture that such "twin primes" pop up infinitely often. But in 1919, the Norwegian mathematician Viggo Brun proved that even if there are infinitely many twin primes, the sum obtained by adding their reciprocals—the sum (1/3 + 1/5) + (1/5 + 1/7) + (1/11 + 1/13) + … —converges to a finite value, much as the sum 1/2 + 1/4 + 1/8 + 1/16 + … converges to 1. Brun's sum is known only to the first few digits, however—and even there, the accuracy is based on conjectures about the frequency with which twin primes occur. Number theorists think it's unlikely that clumps of twin primes are lurking among very large numbers, but they have been unable to prove it. One way to check up on this assumption is to compute better estimates for Brun's sum.

"In desperation, I ran this portion of the calculation on one of the 486s. … The error disappeared."
—Thomas Nicely

In 1974, two mathematicians working for the Navy, Daniel Shanks and John Wrench Jr., reported the first computationally intensive estimate of Brun's sum, based on the occurrence of twin primes among the first two million prime numbers. Two years later, Richard Brent at the Australian National University calculated all twin primes up to a hundred billion (224,376,048 pairs), from which he computed an estimate of 1.90216054 for Brun's sum.

And there it sat—until Nicely entered the picture. The Lynchburg math professor decided to push Brent's work into the trillions. To be on the safe side, he computed Brun's sum twice, using two different methods: the "easy" way using a computer's built-in floating point unit, which is supposed to be accurate to 19 decimal places, and the "hard" way using an extended precision arithmetic, which he set to give 26 (and later 53) digits of accuracy. (The difference can be likened to the difference between computing 1/3 + 1/7 as 0.33 + 0.14 = 0.47 and computing it as 1/3 + 1/7 = 10/21 = 0.48. The latter calculation gains accuracy by doing some exact arithmetic first.)

The comparison between the two methods is what got Intel into trouble. After Nicely added the new Pentium to his stable of computers, he found that the gap between the two results was much larger than it should have been. By trial and error and a process of elimination, he pinpointed the source of the problem: The Pentium was giving incorrect floating point reciprocals for the twin primes 824,633,702,441 and 824,633,702,443—they were wrong from the 10th digit on. Nicely still didn't know whether the error was caused by his hardware or software, in part because he'd caught an earlier error in a compiler program. "Finally, in desperation, I ran this portion of the calculation on one of the 486 [computers], rather than the Pentium," he recalls. "The error disappeared."

Even that didn't prove conclusively that it was the Pentium chip's fault; other hardware in the computer could have been responsible. But in October (4 months after he first noticed his calculations were off), Nicely nailed the culprit when he got hold of two other machines with Pentium chips and was able to reproduce the error. He notified Intel and, after getting no satisfactory answer by the end of the month, sent e-mail asking others to double-check his discovery. "I believe you are aware of events from that point on," he concludes dryly.

The Pentium's problem, as others have abundantly confirmed, lies in the way the chip does division. Although it works fine for most numbers, the chip's built-in algorithm makes mistakes in certain cases, rather like a grade-schooler who has mismemorized part of a multiplication table. Nicely estimates that the chip gets roughly one in a billion reciprocals wrong. But because the work in number theory required him to compute billions of reciprocals over a wide range, he was almost bound to run into the mistake.

"We've known for a long time that number theory computations are very helpful" for turning up computer errors, notes computational number theorist Arjen Lenstra of Bellcore, in Morristown, New Jersey. "It is useful to run number theory stuff on your processor before you sell it."

Intel hasn't decided whether to make such computations a routine part of its testing procedure, says Stephen Smith, engineering manager for the Pentium processor division. But Intel was so impressed with Nicely's work that it asked him to run further computations on a corrected chip. "We looked at him as the most thorough tester," says Smith.

—Barry Cipra

Intel's $475M mistake

Motivations for crypto verification

NIST

Information Technology Laboratory

NATIONAL VULNERABILITY DATABASE

VULNERABILITIES

CVE-2017-3736 Detail

MODIFIED

This vulnerability has been modified since it was last analyzed by the NVD. It is awaiting reanalysis which may result in further changes to the information provided.

Current Description

There is a carry propagating bug in the x86_64 Montgomery squaring procedure in OpenSSL before 1.0.2m and 1.1.0 before 1.1.0g. No EC algorithms are affected. Analysis suggests that attacks against RSA and DSA as a result of this defect would be very difficult to perform and are not believed likely. Attacks against DH are considered just feasible (although very difficult) because most of the work necessary to deduce information about a private key may be performed offline. The amount of resources required for such an attack would be very significant and likely only accessible to a limited number of attackers. An attacker would additionally need online access to an unpatched system using the target private key in a scenario with persistent DH parameters and a private key that is shared between multiple clients. This only affects processors that support the BMI1, BMI2 and ADX extensions like Intel Broadwell (5th generation) and later or AMD Ryzen.

`https://nvd.nist.gov/vuln/detail/CVE-2017-3736`

. . . and it's not just correctness

SECLISTS.ORG

Nmap Security Scanner
- Intro
- Ref Guide
- Install Guide
- Download
- Changelog
- Book
- Docs

Security Lists
- Nmap Announce
- Nmap Dev
- Bugtraq
- Full Disclosure
- Pen Test
- Basics
- More

Security Tools
- Password audit
- Sniffers
- Vuln scanners
- Web scanners
- Wireless
- Exploitation
- Packet crafters
- More

Site News
Advertising
About/Contact

Site Search

Sponsors:

oss-sec mailing list archives

By Date By Thread Search

CVE-2018-0737 OpenSSL: RSA key generation follows several non constant time code paths

From: Billy Brumley <bbrumley () gmail com>
Date: Mon, 16 Apr 2018 19:46:03 +0300

```
Hey Folks,

We discovered 3 vulnerabilities in OpenSSL that allow cache-timing
enabled attackers to recover RSA private keys during key generation.

1. BN_gcd gets called to check that _e_ and _p-1_ are relatively
prime. This function is not constant time, and leaks critical GCD
state leading to information on _p_.

2. During primality testing, BN_mod_inverse gets called without the
BN_FLG_CONSTTIME set during Montgomery arithmetic setup. The resulting
code path is not constant time, and leaks critical GCD state leading
to information on _p_.

3. During primality testing, BN_mod_exp_mont gets called without the
BN_FLG_CONSTTIME set during modular exponentiation, with an exponent
_x_ satisfying _p - 1 = 2**k * x_ hence recovering _x_ gives you most
of _p_. The resulting code path is not constant time, and leaks
critical exponentiation state leading to information on _x_ and hence
_p_.

OpenSSL issued CVE-2018-0737 to track this issue.

# Affected software

LibreSSL fixed these issues (nice!) way back when this was reported in
Jan 2017. Looks like commits

5a1bc054398ec4d2c33e5bdc3a16eece01c8901d
952c1252f58f5f57227f5efaeec0169759c77d72

We verified that with a debugger.

OTOH, OpenSSL wanted concrete evidence of exploitability. That's what
we did over the past year and a half or so.We ran with bug (1) and
recover RSA keys with cache-timings, achieving roughly 30% success
rate in over 10K trials on a cluster.
```

https://seclists.org/oss-sec/2018/q2/50

. . . and it's not just the big libraries

THE PARIS256 ATTACK

Or, Squeezing a Key Through a Carry Bit.

Sean Devlin, Filippo Valsorda

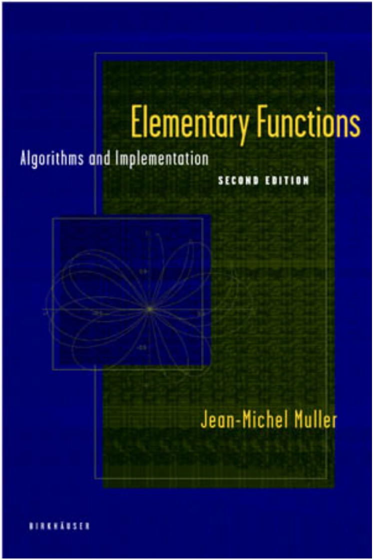
Introduction

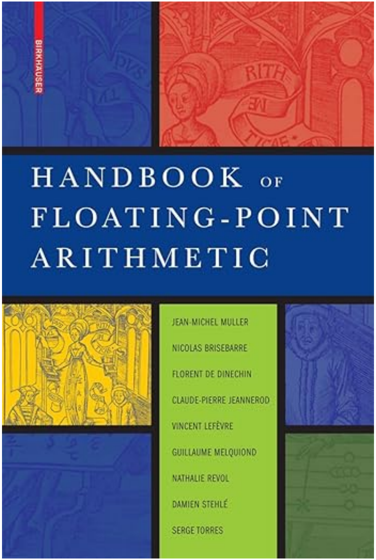
We present an adaptive key recovery attack exploiting a small carry propagation bug in the Go standard library implementation of the NIST P-256 elliptic curve, reported to the Go project as issue 20040.

Following our attack, the vulnerability was assigned CVE-2017-8932, and caused the release of Go 1.7.6 and 1.8.2.

`https://i.blackhat.com/us-18/Wed-August-8/us-18-Valsorda-Squeezing-A-Key-Through-A-Carry-Bit-wp.pdf`

From folklore to textbooks





Things were very different when I started

Results were scattered in different papaers, with some tricks hardly written down at all:

Things were very different when I started

Results were scattered in different papaers, with some tricks hardly written down at all:

- FMA-based division and square root: Markstein, *Computation of Elementary Functions on the IBM RISC System/6000 Processors*

Things were very different when I started

Results were scattered in different papaers, with some tricks hardly written down at all:

- FMA-based division and square root: Markstein, *Computation of Elementary Functions on the IBM RISC System/6000 Processors*
- Computation of transcendental functions: Tang, *Table-driven implementation of the exponential function in IEEE floating-point arithmetic*

Things were very different when I started

Results were scattered in different papaers, with some tricks hardly written down at all:

- FMA-based division and square root: Markstein, *Computation of Elementary Functions on the IBM RISC System/6000 Processors*
- Computation of transcendental functions: Tang, *Table-driven implementation of the exponential function in IEEE floating-point arithmetic*
- General floating-point "magic":
  - Sterbenz, *Floating-Point Computation*
  - Goldberg, *What every computer scientist should know about floating-point arithmetic*
  - Kahan, passim

Exact sum and exact product

The exact sum property was relatively well-known (Sterbenz, Goldberg)

```
|- a IN iformat fmt ∧ b IN iformat fmt ∧
   a / 2 <= b ∧ b <= 2 * a
   ⇒ (b - a) IN iformat fmt
```

Exact sum and exact product

The exact sum property was relatively well-known (Sterbenz, Goldberg)

```
|- a IN iformat fmt ∧ b IN iformat fmt ∧
   a / 2 <= b ∧ b <= 2 * a
   ⇒ (b - a) IN iformat fmt
```

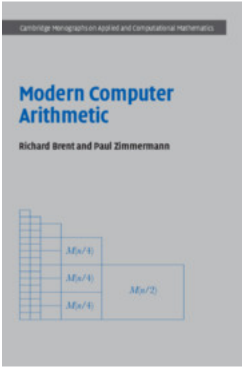
The corresponding multiplicative one was more obscure, perhaps because FMA was then not widely used or standardized:

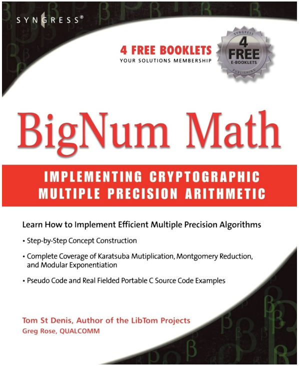
```
|- a IN iformat fmt ∧ b IN iformat fmt ∧
   2 pow (2 * precision fmt - 1) / 2 pow (ulpscale fmt)
   <= abs(a * b)
   ⇒ (a * b - round fmt Nearest (a * b)) IN iformat fmt
```

http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps

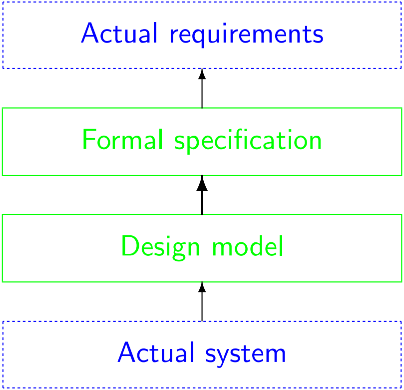
We need the textbooks for cryptographic arithmetic!

These two together probably come closest.

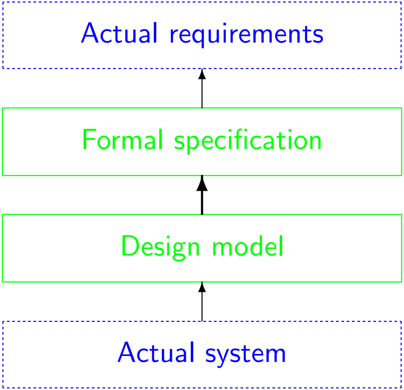




At least we have precise specifications . . .



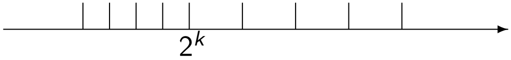
At least we have precise specifications . . .



For us, the spec is pretty easy to formalize, purely mathematical and almost formal already.

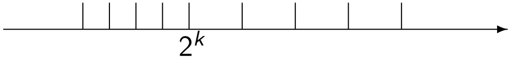
. . . or do we?

It's customary to give a bound on the error in transcendental functions in terms of 'units in the last place' (ulps), roughly the gap between adjacent floating point numbers.



. . . or do we?

It's customary to give a bound on the error in transcendental functions in terms of 'units in the last place' (ulps), roughly the gap between adjacent floating point numbers.



At the boundary 2^k between 'binades', this distance changes, which makes it tricky.

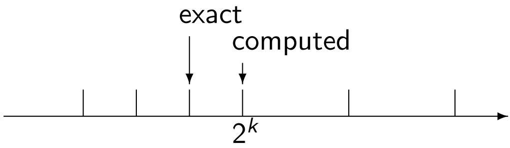
Goldberg's definition of ulp

In general, if the floating-point number $d.d \cdots d \times \beta^e$ *is used to represent* z*, it is in error by* $|d.d \cdots d - (z/\beta^e)|\beta^{p-1}e$ *units in the last place.*

Goldberg's definition of ulp

> *In general, if the floating-point number* $d.d \cdots d \times \beta^e$ *is used to represent* z, *it is in error by* $|d.d \cdots d - (z/\beta^e)|\beta^{p-1}e$ *units in the last place.*

So this is an error of $0.5ulp$ according to Goldberg, but intuitively it should be $1ulp$.



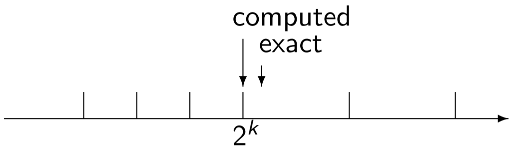
Kahan's definition of ulp

$ulp(x)$ is the gap between the two floating-point numbers nearest x, even if x is one of them.

Kahan's definition of ulp

> *$ulp(x)$ is the gap between the two floating-point numbers nearest x, even if x is one of them.*

According to that definition this is an error of $0.4ulp$, but intuitively it should be $0.2ulp$. Rounding up is worse...



Ambiguity in Ed25519 signature specification

This signature scheme is one of the most widely used for verifying the authenticity of messages, and is standardized in RFC 8032

```
Internet Research Task Force (IRTF)                          S. Josefsson
Request for Comments: 8032                                         SJD AB
Category: Informational                                      I. Liusvaara
ISSN: 2070-1721                                               Independent
                                                             January 2017


          Edwards-Curve Digital Signature Algorithm (EdDSA)

Abstract

   This document describes elliptic curve signature scheme Edwards-curve
   Digital Signature Algorithm (EdDSA).  The algorithm is instantiated
   with recommended parameters for the edwards25519 and edwards448
   curves.  An example implementation and test vectors are provided.
```

The definition of signature verification

The central operation in signature verification involves arithmetic on elliptic curve points B, R and A', with $[k]G$ denoting scalar multiplication of a group element G by an integer k, i.e $G + G + \ldots + G$ (k times)

The definition of signature verification

The central operation in signature verification involves arithmetic on elliptic curve points B, R and A', with $[k]G$ denoting scalar multiplication of a group element G by an integer k, i.e $G + G + \ldots + G$ (k times)

> *3. Check the group equation* $[8][S]B = [8]R + [8][k]A'$*. It's sufficient, but not required, to instead check* $[S]B = R + [k]A'$*.*

The definition of signature verification

The central operation in signature verification involves arithmetic on elliptic curve points B, R and A', with $[k]G$ denoting scalar multiplication of a group element G by an integer k, i.e $G + G + \ldots + G$ (k times)

> *3. Check the group equation* $[8][S]B = [8]R + [8][k]A'$. *It's sufficient, but not required, to instead check* $[S]B = R + [k]A'$.

Most implementations check yet a different equation $[S]B = R + [k \texttt{ MOD } n]A$ where n is the order of the basepoint B.

The definition of signature verification

The central operation in signature verification involves arithmetic on elliptic curve points B, R and A', with $[k]G$ denoting scalar multiplication of a group element G by an integer k, i.e $G + G + \ldots + G$ (k times)

> *3. Check the group equation* $[8][S]B = [8]R + [8][k]A'$*. It's sufficient, but not required, to instead check* $[S]B = R + [k]A'$*.*

Most implementations check yet a different equation $[S]B = R + [k \texttt{ MOD } n]A$ where n is the order of the basepoint B. These are all equivalent for well-formed signatures and do not affect the key security properties. Nevertheless, the *full* group order is $8n$, so for general points on the curve, these three equations are inequivalent in general.

New areas developing

As well as their ‘classical’ subject-matter, both floating-point arithmetic and cryptography are developing new emphases as a result of technological advances or market pressure:

New areas developing

As well as their ‘classical’ subject-matter, both floating-point arithmetic and cryptography are developing new emphases as a result of technological advances or market pressure:

- Machine learning is leading to more emphasis on the standardization of low-precision floating-point numbers and renewing interest in alternative representations.

New areas developing

As well as their ‘classical’ subject-matter, both floating-point arithmetic and cryptography are developing new emphases as a result of technological advances or market pressure:

- Machine learning is leading to more emphasis on the standardization of low-precision floating-point numbers and renewing interest in alternative representations.
- As well as classical RSA and elliptic curve techniques, post-quantum algorithms like the recently standardized ML-KEM and ML-DSA have lattice-based mathematical underpinnings.

New areas developing

As well as their ‘classical’ subject-matter, both floating-point arithmetic and cryptography are developing new emphases as a result of technological advances or market pressure:

- Machine learning is leading to more emphasis on the standardization of low-precision floating-point numbers and renewing interest in alternative representations.
- As well as classical RSA and elliptic curve techniques, post-quantum algorithms like the recently standardized ML-KEM and ML-DSA have lattice-based mathematical underpinnings.

These new areas are rich and attractive targets for formal specification and verification.

Mathematical similarity: bounds reasoning

In both domains, some formal counterpart of interval reasoning is useful, as implemented by tools like GAPPA:

Mathematical similarity: bounds reasoning

In both domains, some formal counterpart of interval reasoning is useful, as implemented by tools like GAPPA:

- In floating-point arithmetic the end result may well be stated as a bound on some overall error term, and automation can help compute it.

Mathematical similarity: bounds reasoning

In both domains, some formal counterpart of interval reasoning is useful, as implemented by tools like GAPPA:

- In floating-point arithmetic the end result may well be stated as a bound on some overall error term, and automation can help compute it.
- In *both* cases, it can be used to prove the absence of overflow/underflow so
  - Floating-point results stay finite and/or normalized giving better relative error biounds
  - Integer operations are exact because they don't wrap round.

Mathematical analogy: MSB versus LSB algorithms

Floating-point numbers, usually being normalized, naturally lend themselves to algorithms based on the ‘most significant bit’.

Mathematical analogy: MSB versus LSB algorithms

Floating-point numbers, usually being normalized, naturally lend themselves to algorithms based on the ‘most significant bit’.

In cryptography, even identifying and manipulating MSBs can be awkward or unnatural to do in constant time. More usage of LSB-based algorithms such as *Montgomery multiplication*.

Mathematical analogy: MSB versus LSB algorithms

Floating-point numbers, usually being normalized, naturally lend themselves to algorithms based on the 'most significant bit'.

In cryptography, even identifying and manipulating MSBs can be awkward or unnatural to do in constant time. More usage of LSB-based algorithms such as *Montgomery multiplication*.

Nevertheless there are often strong similarities between 'metrical' (MSB) and '2-adic' (LSB) algorithms (see table in Brent-Zimmermann).

Using Newton's method for reciprocals

Floating-point computation of $1/a$:

- Form initial approximation $y \approx \frac{1}{a}$
- Then iterate $y' = y \cdot (2 - ay) = y + y \cdot (1 - ay)$

Using Newton's method for reciprocals

Floating-point computation of $1/a$:

- Form initial approximation $y \approx \frac{1}{a}$
- Then iterate $y' = y \cdot (2 - ay) = y + y \cdot (1 - ay)$

If $y = \frac{1}{a}(1 + \epsilon)$ then $y' = \frac{1}{a}(1 - \epsilon^2)$, the classic quadratic convergence where we get twice as many bits of accuracy per iteration.

Modular inverses by Hensel lifting

Consider the 1-word (negated) modular inverse, called `word_negmodinv` in `s2n-bignum`.

Modular inverses by Hensel lifting

Consider the 1-word (negated) modular inverse, called `word_negmodinv` in `s2n-bignum`.
Given a 64-bit unsigned and *odd* integer a, returns another 64-bit integer x such that $ax \equiv -1 \pmod{2^{64}}$, i.e. that `a * x == 0xFFFFFFFFFFFFFFFF` using unsigned silently-wrapping word operations like those on C's `uint64_t`.

Modular inverses by Hensel lifting

Consider the 1-word (negated) modular inverse, called `word_negmodinv` in `s2n-bignum`.

Given a 64-bit unsigned and *odd* integer a, returns another 64-bit integer x such that $ax \equiv -1 \pmod{2^{64}}$, i.e. that `a * x == 0xFFFFFFFFFFFFFFFF` using unsigned silently-wrapping word operations like those on C's `uint64_t`.

It is implemented in a directly similar way using Hensel lifting, the p-adic analog of Newton's method.

Initial approximation

As with the floating-point inverse, we need an initial approximation to start with. The following piece of magic (in C syntax):

```
x = (a - (a<<2))^2
```

happens to give a 5-bit negated modular inverse, assuming `a` is odd.

Hensel lifting step

Given a k-bit approximation $ax \equiv -1 \pmod{2^k}$, do the same Newton step with integers, except for a sign flip because we want a *negated* inverse:

```
e = a * x + 1;
y = e * x + x;
```

Hensel lifting step

Given a k-bit approximation $ax \equiv -1 \pmod{2^k}$, do the same Newton step with integers, except for a sign flip because we want a *negated* inverse:

```
e = a * x + 1;
y = e * x + x;
```

By the initial assumption $ax = 2^k n - 1$ for some integer n

Hensel lifting step

Given a k-bit approximation $ax \equiv -1 \pmod{2^k}$, do the same Newton step with integers, except for a sign flip because we want a *negated* inverse:

```
e = a * x + 1;
y = e * x + x;
```

By the initial assumption $ax = 2^k n - 1$ for some integer n
So $e = ax + 1 = 2^k n$ and

Hensel lifting step

Given a k-bit approximation $ax \equiv -1 \pmod{2^k}$, do the same Newton step with integers, except for a sign flip because we want a *negated* inverse:

```
e = a * x + 1;
y = e * x + x;
```

By the initial assumption $ax = 2^k n - 1$ for some integer n
So $e = ax + 1 = 2^k n$ and $e + 1 = 2^k n + 1$

Hensel lifting step

Given a k-bit approximation $ax \equiv -1 \pmod{2^k}$, do the same Newton step with integers, except for a sign flip because we want a *negated* inverse:

```
e = a * x + 1;
y = e * x + x;
```

By the initial assumption $ax = 2^k n - 1$ for some integer n
So $e = ax + 1 = 2^k n$ and $e + 1 = 2^k n + 1$ so then
$ay = ax(e + 1) = (2^k n - 1)(2^k n + 1) = 2^{2k} n^2 - 1$, i.e.
$ay \equiv -1 \pmod{2^{2k}}$.

Open-source: s2n-bignum

A library of bignum arithmetic operations designed for cryptographic applications.

Open-source: s2n-bignum

A library of bignum arithmetic operations designed for cryptographic applications.

- Efficient: hand-crafted code with competitive performance

Open-source: s2n-bignum

A library of bignum arithmetic operations designed for cryptographic applications.

- Efficient: hand-crafted code with competitive performance
- Correct: every function is formally verified mathematically

Open-source: s2n-bignum

A library of bignum arithmetic operations designed for cryptographic applications.

- Efficient: hand-crafted code with competitive performance
- Correct: every function is formally verified mathematically
- Secure: all code is written in “constant-time” style

Open-source: s2n-bignum

A library of bignum arithmetic operations designed for cryptographic applications.

- Efficient: hand-crafted code with competitive performance
- Correct: every function is formally verified mathematically
- Secure: all code is written in "constant-time" style

`https://github.com/awslabs/s2n-bignum`

Open-source: s2n-bignum

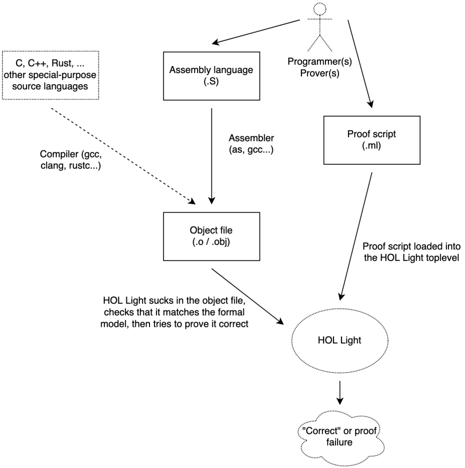
A library of bignum arithmetic operations designed for cryptographic applications.

- Efficient: hand-crafted code with competitive performance
- Correct: every function is formally verified mathematically
- Secure: all code is written in "constant-time" style

`https://github.com/awslabs/s2n-bignum`

All hand-written or specially generated 64-bit ARM and x86 machine code.

Coding and verification flow



Comparison with ther crypto verification projects

Formally verified cryptography projects can be placed on this spectrum:

Comparison with ther crypto verification projects

Formally verified cryptography projects can be placed on this spectrum:

- Correct-by-construction coding (HACL*, Jasmin)
- . . .
- A bit of both or somewhere in between (Fiat)
- . . .
- Separate verification (CryptoLine, s2n-bignum)

Pros and cons of the s2n-bignum approach

Pros and cons of the s2n-bignum approach

- ☺ Independent of compiler or even macro-assembler correctness.

Pros and cons of the s2n-bignum approach

- ☺ Independent of compiler or even macro-assembler correctness.
- ☺ Applicable to highly tuned efficient code that is hard to generate automatically as well as compiled C code etc.

Pros and cons of the s2n-bignum approach

- ☺ Independent of compiler or even macro-assembler correctness.
- ☺ Applicable to highly tuned efficient code that is hard to generate automatically as well as compiled C code etc.
- ☺ Code is conventional human-readable and human-modifiable, usage is independent of prover infrastructure.

Pros and cons of the s2n-bignum approach

- ☺ Independent of compiler or even macro-assembler correctness.
- ☺ Applicable to highly tuned efficient code that is hard to generate automatically as well as compiled C code etc.
- ☺ Code is conventional human-readable and human-modifiable, usage is independent of prover infrastructure.
- ☹ Much more work involved writing code at this level, less structured representation.

Pros and cons of the s2n-bignum approach

- ☺ Independent of compiler or even macro-assembler correctness.
- ☺ Applicable to highly tuned efficient code that is hard to generate automatically as well as compiled C code etc.
- ☺ Code is conventional human-readable and human-modifiable, usage is independent of prover infrastructure.
- ☹ Much more work involved writing code at this level, less structured representation.
- ☺/☹ Exposure of low-level details like exact stack and PC offsets and particular registers.

Implementation and verification together

The Intel work started as verifying pre-existing code, but sometimes the proof unlocked efficiency improvements.

Implementation and verification together

The Intel work started as verifying pre-existing code, but sometimes the proof unlocked efficiency improvements.

> *If q is a floating point number within 1 ulp of the true quotient a/b of two floating point numbers, and y is the correctly rounded-to-nearest approximation of the exact reciprocal $\frac{1}{b}$, then the following iteration:*
>
> $$\begin{aligned} r &= a - bq \\ q' &= q + ry \end{aligned}$$
>
> *using round-to-nearest in each case, yields the correctly rounded-to-nearest quotient q'. (Markstein)*

Implementation and verification together

The Intel work started as verifying pre-existing code, but sometimes the proof unlocked efficiency improvements.

> *If q is a floating point number within 1 ulp of the true quotient a/b of two floating point numbers, and y is the correctly rounded-to-nearest approximation of the exact reciprocal $\frac{1}{b}$, then the following iteration:*
>
> $$\begin{aligned} r &= a - bq \\ q' &= q + ry \end{aligned}$$
>
> *using round-to-nearest in each case, yields the correctly rounded-to-nearest quotient q'. (Markstein)*

Simply changing to this unlocks more efficient algorithms:

> *... y approximates $\frac{1}{b}$ to a relative error $< \frac{1}{2^p}$, ...*

Implementation and verification together

The Intel work started as verifying pre-existing code, but sometimes the proof unlocked efficiency improvements.

> *If q is a floating point number within 1 ulp of the true quotient a/b of two floating point numbers, and y is the correctly rounded-to-nearest approximation of the exact reciprocal $\frac{1}{b}$, then the following iteration:*
>
> $$\begin{aligned} r &= a - bq \\ q' &= q + ry \end{aligned}$$
>
> *using round-to-nearest in each case, yields the correctly rounded-to-nearest quotient q'. (Markstein)*

Simply changing to this unlocks more efficient algorithms:

> *. . . y approximates $\frac{1}{b}$ to a relative error $< \frac{1}{2^p}$, . . .*

And the proof is shorter!

s2n-bignum code and proof

The s2n-bignum project was conceived from the beginning as integrated design and verification

s2n-bignum code and proof

The s2n-bignum project was conceived from the beginning as integrated design and verification

- Each function has a precise specification and associated formal proof (no unverified code)

s2n-bignum code and proof

The s2n-bignum project was conceived from the beginning as integrated design and verification

- Each function has a precise specification and associated formal proof (no unverified code)
- Code written with verification in mind, emphasizing simplicity where possible.

s2n-bignum code and proof

The s2n-bignum project was conceived from the beginning as integrated design and verification

- Each function has a precise specification and associated formal proof (no unverified code)
- Code written with verification in mind, emphasizing simplicity where possible.
- Later we have sometimes taken code or algorithms from other libraries (e.g. Lenngrenn's X25519 code, `mlkem-native`) and added formal proofs post-hoc.

s2n-bignum code and proof

The s2n-bignum project was conceived from the beginning as integrated design and verification

- Each function has a precise specification and associated formal proof (no unverified code)
- Code written with verification in mind, emphasizing simplicity where possible.
- Later we have sometimes taken code or algorithms from other libraries (e.g. Lenngrenn's X25519 code, `mlkem-native`) and added formal proofs post-hoc.
- Some of the algorithms would be difficult to trust *without* a formal proof.

GCD and modular inverse using divstep

Classic binary gcd is attractive and simple

$$\begin{aligned}
\gcd(2n, 2m) &= 2\gcd(n, m) \\
\gcd(2n+1, 2m) &= \gcd(2n+1, m) \\
\gcd(2n, 2m+1) &= gcd(n, 2m+1) \\
\gcd(2n+1, 2m+1) &= gcd(min(2n+1, 2m+1), |m-n|)
\end{aligned}$$

GCD and modular inverse using divstep

Classic binary gcd is attractive and simple

$$\begin{aligned}
\gcd(2n, 2m) &= 2\gcd(n, m) \\
\gcd(2n+1, 2m) &= \gcd(2n+1, m) \\
\gcd(2n, 2m+1) &= gcd(n, 2m+1) \\
\gcd(2n+1, 2m+1) &= gcd(min(2n+1, 2m+1), |m-n|)
\end{aligned}$$

The Bernstein-Yang divstep algorithm replaces the magnitude comparison with a single-word proxy δ (assume n is odd):

$$\text{divstep}(\delta, n, m) = \begin{cases} (1-\delta, m, (m-n)/2) & \text{if } \delta > 0 \wedge \text{odd}(m) \\ (1+\delta, n, (m + (m \bmod 2)n)/2) & \text{otherwise.} \end{cases}$$

GCD and modular inverse using divstep

Classic binary gcd is attractive and simple

$$\begin{aligned}
\gcd(2n, 2m) &= 2\gcd(n, m) \\
\gcd(2n+1, 2m) &= \gcd(2n+1, m) \\
\gcd(2n, 2m+1) &= gcd(n, 2m+1) \\
\gcd(2n+1, 2m+1) &= gcd(min(2n+1, 2m+1), |m-n|)
\end{aligned}$$

The Bernstein-Yang divstep algorithm replaces the magnitude comparison with a single-word proxy δ (assume n is odd):

$$\text{divstep}(\delta, n, m) = \begin{cases} (1-\delta, m, (m-n)/2) & \text{if } \delta > 0 \land \text{odd}(m) \\ (1+\delta, n, (m + (m \bmod 2)n)/2) & \text{otherwise.} \end{cases}$$

However the bound/termination reasoning is *much* more complex.

Dan Bernstein's formal bounds proof

Fortunately the inventor of the algorithm learned HOL Light and proved the bound for us:

Side-channels and "constant-time"

There are many side-channels by which systems may 'leak' secret info (like a private key) to an observer:

- Execution time
- Memory access pattern
- Power consumption
- Electromagnetic radiation emitted
- ...
- Microarchitectural bugs

Side-channels and "constant-time"

There are many side-channels by which systems may 'leak' secret info (like a private key) to an observer:

- Execution time $\leftarrow$
- Memory access pattern $\leftarrow$
- Power consumption
- Electromagnetic radiation emitted
- ...
- Microarchitectural bugs

Main worries in typical multitasking OS on shared machine

Side-channels and "constant-time"

There are many side-channels by which systems may 'leak' secret info (like a private key) to an observer:

- Execution time $\leftarrow$
- Memory access pattern $\leftarrow$
- Power consumption
- Electromagnetic radiation emitted
- ...
- Microarchitectural bugs

Main worries in typical multitasking OS on shared machine

However, *Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86* shows how to mount power attacks remotely via frequency scaling.

How can you avoid timing/cache side-channels?

Want execution time, if not literally constant, *uncorrelated* with (secret) data being manipulated. How?

How can you avoid timing/cache side-channels?

Want execution time, if not literally constant, *uncorrelated* with (secret) data being manipulated. How?

- Add randomization or salting to the algorithm
- Balance timing of paths
- Just make it too fast to observe

How can you avoid timing/cache side-channels?

Want execution time, if not literally constant, *uncorrelated* with (secret) data being manipulated. How?

- Add randomization or salting to the algorithm
- Balance timing of paths
- Just make it too fast to observe
- Always perform exactly the same operations regardless of (secret) data. $\longleftarrow$ Our chosen solution

How can you ‘always do the same thing’?

When there is control flow depending on secret data:

```
if (n >= p) n = n - p;
```

How can you ‘always do the same thing’?

When there is control flow depending on secret data:

```
if (n >= p) n = n - p;
```

convert it into dataflow using masking, conditional moves etc.

```
b = (n < p) - 1;
n = n - (p & b);
```

What about the compiler?

The compiler may naively turn mask creation back into a branch:

```
b = (n < p) - 1;
```

What about the compiler?

The compiler may naively turn mask creation back into a branch:

```
b = (n < p) - 1;
```

Time to break out your copy of “Hacker’s Delight”:

```
b = (((~n & p) | ((~n | p) & (n - p))) >> 63) - 1;
```

What about the compiler?

The compiler may naively turn mask creation back into a branch:

```
b = (n < p) - 1;
```

Time to break out your copy of "Hacker's Delight":

```
b = (((~n & p) | ((~n | p) & (n - p))) >> 63) - 1;
```

Another motivation for working directly in machine code where flags and useful instructions like `CMOV` and `CSEL` are available.

Are the machine instructions constant-time?

- Some definitely not, e.g. division by zero is special
- General assumption that simple things like add, mul mostly are

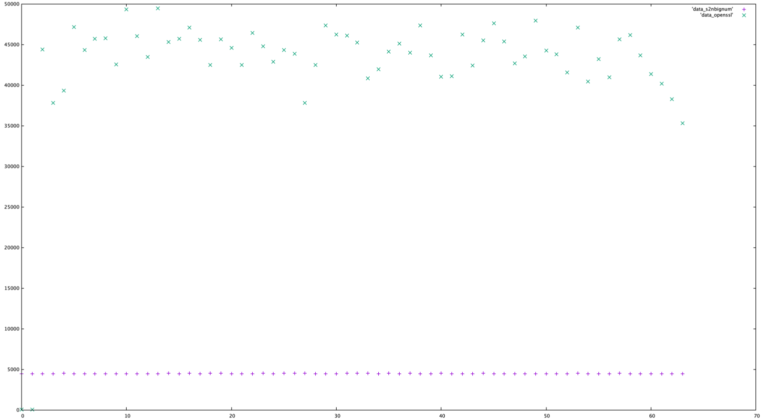
Are the machine instructions constant-time?

- Some definitely not, e.g. division by zero is special
- General assumption that simple things like add, mul mostly are

Recently CPUs have started offering *some* guarantees (DIT bit or DOIT mode).

Some empirical results on timing

Times for 384-bit modular inverse at bit densities 0–63, nanoseconds on Intel® Xeon® Platinum 8175M, 2.5 GHz.



Questions?