Fast, constant-time, correct: pick three

Daniel J. Bernstein

Is the next slide a zero-day?

Maybe this code leaks secrets through timings,
but maybe it ends up not being exploitable.

Is the next slide a zero-day?

Maybe this code leaks secrets through timings,
but maybe it ends up not being exploitable.

The proactive approach, what I recommend:
This code is dangerous, so fix it.

Is the next slide a zero-day?

Maybe this code leaks secrets through timings,
but maybe it ends up not being exploitable.

The proactive approach, what I recommend:
This code is dangerous, so fix it.

The reactive approach, common practice:
Fix the code only when an exploit is demonstrated.

Is the next slide a zero-day?

Maybe this code leaks secrets through timings, but maybe it ends up not being exploitable.

The proactive approach, what I recommend: This code is dangerous, so fix it.

The reactive approach, common practice: Fix the code only when an exploit is demonstrated.

The OpenSSL approach, at least for now: Timing attack extracting keys from “same physical system” doesn’t count as a vulnerability. Fix the code only when a *remote* exploit is demonstrated.

Does compiler use bit test + branch?

From `openssh/libcrux_mlkem768_sha3.h`
(plus extra line breaks to fit on this slide):

```
static inline uint8_t
libcrux_ml_kem_constant_time_ops_inz(uint8_t value) {
  uint16_t value0 = (uint16_t)value;
  uint16_t result = (((uint32_t)value0 |
(uint32_t)core_num__u16_7__wrapping_add(~value0, 1U))
&
                     0xFFFFU) >>
                        8U &
                    1U;
  return (uint8_t)result;
}
```

Similar cases with known branches

2024.04 Bernstein: “Tracking down some TIMECOP alerts led to a 2021 gcc patch from ARM . . . turning `(-x)>>31` into a bool, often breaking constant-time code.”

Similar cases with known branches

2024.04 Bernstein: “Tracking down some TIMECOP alerts led to a 2021 gcc patch from ARM … turning `(-x)>>31` into a bool, often breaking constant-time code.”

2024.06 Purnal: demo exploiting `&1` to extract secret keys from the reference Kyber-512 software.

Similar cases with known branches

2024.04 Bernstein: “Tracking down some TIMECOP alerts led to a 2021 gcc patch from ARM . . . turning `(-x)>>31` into a bool, often breaking constant-time code.”

2024.06 Purnal: demo exploiting `&1` to extract secret keys from the reference Kyber-512 software.

More examples are now known where compiler produces CPU branches from similar source code.

This was supposedly the state of the art

libcrux advertising: "libcrux has been formally verified. This gives you the highest level of assurance that it is safe to use and free of bugs."

This was supposedly the state of the art

libcrux advertising: “libcrux has been formally verified. This gives you the highest level of assurance that it is safe to use and free of bugs.”

Compare to 2025 Pornin, “Constant-time code: the pessimist case”: paper reviews failures; “highlights why such failures are expected to become more common, and how constant-time coding is, or will soon become, infeasible in all generality”.

This was supposedly the state of the art

libcrux advertising: "libcrux has been formally verified. This gives you the highest level of assurance that it is safe to use and free of bugs."

Compare to 2025 Pornin, "Constant-time code: the pessimist case": paper reviews failures; "highlights why such failures are expected to become more common, and how constant-time coding is, or will soon become, infeasible in all generality".

See also "Constant-time BIGNUM is bollocks" talk coming up in this conference.

C compiler can also break correctness

`gcc` repo: 223889 commits as of 2025-10-07.

`llvm-project`: 555094 commits as of 2025-10-07.

C compiler can also break correctness

`gcc` repo: 223889 commits as of 2025-10-07.
`llvm-project`: 555094 commits as of 2025-10-07.

Topics of typical commits:

- added "optimizations";
- added tests for "optimizations";
- fixes to tests for "optimizations";
- fixes for bugs in "optimizations".

C compiler can also break correctness

`gcc` repo: 223889 commits as of 2025-10-07.
`llvm-project`: 555094 commits as of 2025-10-07.

Topics of typical commits:

- added “optimizations”;
- added tests for “optimizations”;
- fixes to tests for “optimizations”;
- fixes for bugs in “optimizations”.

e.g. a gcc bug introduced in 2018 and found in 2020 incorrectly “optimizes” `memcmp` constants the same way as `strncmp` constants.

C compiler can also break correctness

`gcc` repo: 223889 commits as of 2025-10-07.
`llvm-project`: 555094 commits as of 2025-10-07.

Topics of typical commits:

- added "optimizations";
- added tests for "optimizations";
- fixes to tests for "optimizations";
- fixes for bugs in "optimizations".

e.g. a gcc bug introduced in 2018 and found in 2020
incorrectly "optimizes" `memcmp` constants
the same way as `strncmp` constants.
Similar to a gcc bug that I pointed out in 1999.

Reasonable fear of bugs holds back speed

2020 Pornin const-time inverter mod $2^{255}-19$ takes 6 kcycles on Intel Skylake. (OpenSSL has slower const-time Fermat inverter for X25519. Var-time `BN_mod_inverse` takes 53 kcycles.)

Reasonable fear of bugs holds back speed

2020 Pornin const-time inverter mod $2^{255} - 19$ takes 6 kcycles on Intel Skylake. (OpenSSL has slower const-time Fermat inverter for X25519. Var-time `BN_mod_inverse` takes 53 kcycles.)

Pornin's paper says previous version of paper had a "gap in the proof" and an example "for which the algorithm failed". Also says revised algorithm+proof "are believed correct". Would you deploy this code?

Reasonable fear of bugs holds back speed

2020 Pornin const-time inverter mod $2^{255} - 19$ takes 6 kcycles on Intel Skylake. (OpenSSL has slower const-time Fermat inverter for X25519. Var-time `BN_mod_inverse` takes 53 kcycles.)

Pornin's paper says previous version of paper had a "gap in the proof" and an example "for which the algorithm failed". Also says revised algorithm+proof "are believed correct". Would you deploy this code?

How about const-time divstep-based inverter from 2019 Bernstein–Yang where the proof involves a lengthy calculation? Would you deploy that?

Another example of the fear factor

2023 Cloudflare said it "deployed a non AVX2 implementation of Kyber, because we're more worried about implementation mistakes".

Another example of the fear factor

2023 Cloudflare said it "deployed a non AVX2 implementation of Kyber, because we're more worried about implementation mistakes".

It then turned out that the majority of Kyber implementations had divisions with secret inputs, leading to the KyberSlash timing-attack demos. The AVX2 implementation didn't use divisions, so it was immune to these attacks.

Another example of the fear factor

2023 Cloudflare said it “deployed a non AVX2 implementation of Kyber, because we’re more worried about implementation mistakes”.

It then turned out that the majority of Kyber implementations had divisions with secret inputs, leading to the KyberSlash timing-attack demos. The AVX2 implementation didn’t use divisions, so it was immune to these attacks.

So AVX2 code is magically safe? No: consider, e.g., Dilithium’s exploitable bugs in ref and AVX2 code.

Deployed crypto libraries are full of bugs

In 2016, OpenSSL claimed to be “robust”.
OpenSSL continues to claim this.

There have been **hundreds** of OpenSSL CVEs since 2016, often at the protocol layer (libssl), often at the primitives layer (libcrypto).

Wasn't this supposed to be a happy talk?

Let's take a look at s2n-bignum

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

Led by AWS's John Harrison. First release in 2021.

Let's take a look at s2n-bignum

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

Led by AWS's John Harrison. First release in 2021.
API includes various functions such as X25519:

```
extern void curve25519_x25519_byte
  (uint8_t res[32],
   const uint8_t scalar[32],
   const uint8_t point[32]);
```

Let's take a look at s2n-bignum

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

Led by AWS's John Harrison. First release in 2021.
API includes various functions such as X25519:

```
extern void curve25519_x25519_byte
  (uint8_t res[32],
   const uint8_t scalar[32],
   const uint8_t point[32]);
```

Each function has

- an asm implementation for 64-bit ARM and
- an asm implementation for 64-bit AMD/Intel.

Some X25519 speeds

Keygen+DH cost on an Intel Skylake core:

- 29+85 kcycles for s2n-bignum.
- 130+118 kcycles for OpenSSL.
- 166+166 kcycles for libcrux.

Some X25519 speeds

Keygen+DH cost on an Intel Skylake core:

- 29+85 kcycles for s2n-bignum.
- 130+118 kcycles for OpenSSL.
- 166+166 kcycles for libcrux.

Note: 2 kcycles cost about 2^{-40} USD,
as does sending a byte through the Internet.

Some X25519 speeds

Keygen+DH cost on an Intel Skylake core:

- 29+85 kcycles for s2n-bignum.
- 130+118 kcycles for OpenSSL.
- 166+166 kcycles for libcrux.

Note: 2 kcycles cost about 2^{-40} USD,
as does sending a byte through the Internet.

See the lib25519 speed page for benchmarks on
more CPU microarchitectures.

Is s2n-bignum constant-time?

2005 Bernstein: "CPU manufacturers should thoroughly document the performance of their chips. In particular, they need to highlight every variation in their instruction timings, and to guarantee that there are no other variations."

Is s2n-bignum constant-time?

2005 Bernstein: "CPU manufacturers should thoroughly document the performance of their chips. In particular, they need to highlight every variation in their instruction timings, and to guarantee that there are no other variations."

2014 Bernstein: "Please specify that various instructions keep various inputs secret."

Is s2n-bignum constant-time?

2005 Bernstein: "CPU manufacturers should thoroughly document the performance of their chips. In particular, they need to highlight every variation in their instruction timings, and to guarantee that there are no other variations."

2014 Bernstein: "Please specify that various instructions keep various inputs secret."

2020 ARM: "Data Independent Timing" (DIT). Note: instruction list changed later.

Is s2n-bignum constant-time?

2005 Bernstein: "CPU manufacturers should thoroughly document the performance of their chips. In particular, they need to highlight every variation in their instruction timings, and to guarantee that there are no other variations."

2014 Bernstein: "Please specify that various instructions keep various inputs secret."

2020 ARM: "Data Independent Timing" (DIT). Note: instruction list changed later.

2022 Intel: "Data Operand Independent Timing" (DOIT). s2n-bignum has started testing against this.

Is s2n-bignum correct?

The X25519 machine code for ARM has a theorem `CURVE25519_X25519_BYTE_SUBROUTINE_CORRECT`. There's a similar theorem for the AMD/Intel code.

Is s2n-bignum correct?

The X25519 machine code for ARM has a theorem `CURVE25519_X25519_BYTE_SUBROUTINE_CORRECT`. There's a similar theorem for the AMD/Intel code.

The theorem hypotheses include a specification of how each relevant CPU instruction works.

Is s2n-bignum correct?

The X25519 machine code for ARM has a theorem `CURVE25519_X25519_BYTE_SUBROUTINE_CORRECT`. There's a similar theorem for the AMD/Intel code.

The theorem hypotheses include a specification of how each relevant CPU instruction works.

For comparison, many formal-verification projects assume a specification of how the C language works.

Is s2n-bignum correct?

The X25519 machine code for ARM has a theorem
`CURVE25519_X25519_BYTE_SUBROUTINE_CORRECT`.
There's a similar theorem for the AMD/Intel code.

The theorem hypotheses include a specification
of how each relevant CPU instruction works.

For comparison, many formal-verification projects
assume a specification of how the C language works.
This boils down to assuming a CPU spec
if you're using CompCert to compile C code;
otherwise you're relying on the CPU *and* a compiler.

Is s2n-bignum correct? part 2

Are there mistakes in the CPU specs? Possibly, even with s2n-bignum's testing of the specs. Want more cross-checks among CPU specs.

Is s2n-bignum correct? part 2

Are there mistakes in the CPU specs? Possibly, even with s2n-bignum's testing of the specs. Want more cross-checks among CPU specs.

Are CPU designers making mistakes? Sometimes, yes, maybe including mistakes relevant to this code.

Is s2n-bignum correct? part 2

Are there mistakes in the CPU specs? Possibly, even with s2n-bignum's testing of the specs. Want more cross-checks among CPU specs.

Are CPU designers making mistakes? Sometimes, yes, maybe including mistakes relevant to this code.

Are there other deviations between the theorem statements and what we want? Maybe. Important for every aspect of the statements to be audited.

Is s2n-bignum correct? part 2

Are there mistakes in the CPU specs? Possibly, even with s2n-bignum's testing of the specs. Want more cross-checks among CPU specs.

Are CPU designers making mistakes? Sometimes, yes, maybe including mistakes relevant to this code.

Are there other deviations between the theorem statements and what we want? Maybe. Important for every aspect of the statements to be audited.

Are there gaps in the *proofs* of the theorems?

Is s2n-bignum correct? part 2

Are there mistakes in the CPU specs? Possibly, even with s2n-bignum's testing of the specs. Want more cross-checks among CPU specs.

Are CPU designers making mistakes? Sometimes, yes, maybe including mistakes relevant to this code.

Are there other deviations between the theorem statements and what we want? Maybe. Important for every aspect of the statements to be audited.

Are there gaps in the *proofs* of the theorems? Very unlikely: the proofs are checked by the small, carefully reviewed HOL Light proof-checking kernel.

Shrinking the TCB

Traditional auditing of cryptographic software:
check every line in every implementation;
check proofs that the computations work.

Shrinking the TCB

Traditional auditing of cryptographic software:
check every line in every implementation;
check proofs that the computations work.

With computer-checked proofs: auditors check

- the theorem statements
  (which are much shorter than the proofs) and
- the proof-checking tools
  (which are shared by many proofs).

Those tools then verify the theorems, *automating*
the audits of each proof line and of each code line.

Inversion modulo $2^{255} - 19$, revisited

2021 version of s2n-bignum takes 6 kcycles on Skylake. Algorithm is similar to 2020 Pornin but *with a computer-checked proof of correctness*.

Inversion modulo $2^{255} - 19$, revisited

2021 version of s2n-bignum takes 6 kcycles on Skylake. Algorithm is similar to 2020 Pornin but *with a computer-checked proof of correctness*.

Current s2n-bignum takes 4 kcycles on Skylake using a divstep-based inversion algorithm, also with a computer-checked proof of correctness.

Inversion modulo $2^{255} - 19$, revisited

2021 version of s2n-bignum takes 6 kcycles on Skylake. Algorithm is similar to 2020 Pornin but *with a computer-checked proof of correctness*.

Current s2n-bignum takes 4 kcycles on Skylake using a divstep-based inversion algorithm, also with a computer-checked proof of correctness.

X25519 auditor doesn't even have to look at these theorems: these are internal details of the proofs of `CURVE25519_X25519_BYTE_SUBROUTINE_CORRECT`.

Let’s try auditing this theorem:

`CURVE25519_X25519_BYTE_SUBROUTINE_CORRECT`

Top level of the theorem

The theorem has 18 lines. General shape:

```
!variables.
assumption1 /\
assumption2 /\
assumption3
==> conclusion
```

meaning: for all possible values of `variables`, if `assumption1` and `assumption2` and `assumption3` are true then `conclusion` is true.

The first assumption

Here's `assumption1`:

```
aligned 16 stackpointer
```

which sounds like it means:
the stack pointer has 16-byte alignment.

The first assumption

Here's `assumption1`:

```
aligned 16 stackpointer
```

which sounds like it means:
the stack pointer has 16-byte alignment.

Checking the definition of `aligned`:

```
aligned n (a:N word) <=>
  n divides 2 EXP dimindex(:N) /\
  n divides val a
```

The third assumption

Here's `assumption3`:

```
nonoverlapping (res,32) (word pc,0x27f8)
```

This means: the 32 bytes that `res` points to must not overlap the 0x27f8 bytes that `pc` points to; i.e., output array must not overlap the code. Again can check definitions.

The second assumption

Here's `assumption2`:

```
ALL (nonoverlapping
     (word_sub stackpointer (word 384),
      384))
    [(word pc,0x27f8); (res,32);
     (scalar,32); (point,32)]
```

This means: 384 bytes below `stackpointer` must not overlap code, output, first input, second input.

The conclusion

General shape of `conclusion`:

```
ensures arm pre post maychange
```

meaning: if an `arm` CPU state satisfies `pre`,
then it will evolve to a state satisfying `post`;
also, the state is unmodified beyond `maychange`.

The conclusion

General shape of `conclusion`:

```
ensures arm pre post maychange
```

meaning: if an `arm` CPU state satisfies `pre`,
then it will evolve to a state satisfying `post`;
also, the state is unmodified beyond `maychange`.

The definition of `arm` is thousands of lines,
but the work of auditing this is shared
across all ARM software in `s2n-bignum`.

A precondition on the code

```
aligned_bytes_loaded s (word pc)
  curve25519_x25519_byte_mc
```

meaning: the CPU state `s` has, at address `pc`, the machine code specified in `curve25519_x25519_byte_mc`.

A precondition on the code

```
aligned_bytes_loaded s (word pc)
  curve25519_x25519_byte_mc
```

meaning: the CPU state `s` has, at address `pc`, the machine code specified in `curve25519_x25519_byte_mc`.

One of the proof-checking tools also checks that the definition of `curve25519_x25519_byte_mc` matches the `s2n-bignum` object code on disk. There's no need for us to audit this code: the computer checks proof that the code works.

Preconditions on registers

```
read PC s = word pc
```

meaning: the program-counter register in CPU state `s` stores the same address as `pc`.

```
read SP s = stackpointer
```

meaning: the stack pointer matches `stackpointer`.

```
read X30 s = returnaddress
```

meaning: register `X30` matches `returnaddress`.

Preconditions on arguments

```
C_ARGUMENTS [res; scalar; point] s /\
read (memory :> bytes(scalar,32)) s = n /\
read (memory :> bytes(point,32)) s = X
```

meaning: variables `res` and `scalar` and `point` match arguments stored in CPU state `s` according to the C ABI; variables `n` and `X` are 32-byte integers stored at addresses `scalar` and `point`.

The postcondition

```
read PC s = returnaddress /\
read (memory :> bytes(res,32)) s
  = rfcx25519(n,X)
```

meaning: the CPU's program counter now points to `returnaddress`; the 32-byte integer stored at address `res` matches `rfcx25519(n,X)`.

Can check that the `rfcx25519(n,X)` definition matches X25519.

My reaction after auditing

Reading this was **much less work** than manually verifying an X25519 implementation.

Can still do better by eliminating boilerplate: s2n-bignum should expand `C_ARGUMENTS` to `C_CALL` encapsulating low-level details of stack, regs, etc. Would make the theorem much more concise, help auditor focus on `rfcx25519` definition.

Ecosystem evolution

Library competition

How will OpenSSL stop callers from switching to s2n-bignum's fast, constant-time, correct code?

Library competition

How will OpenSSL stop callers from switching to s2n-bignum's fast, constant-time, correct code?
"OpenSSL is robust"?

Library competition

How will OpenSSL stop callers from switching to s2n-bignum's fast, constant-time, correct code? "OpenSSL is robust"? "C is better than asm"?

Library competition

How will OpenSSL stop callers from switching to s2n-bignum's fast, constant-time, correct code?
"OpenSSL is robust"? "C is better than asm"?
"You're trapped in our API, get used to it"?

Library competition—or cooperation

How will OpenSSL stop callers from switching to s2n-bignum's fast, constant-time, correct code? "OpenSSL is robust"? "C is better than asm"? "You're trapped in our API, get used to it"?

Shouldn't OpenSSL start calling s2n-bignum?

Library competition—or cooperation

How will OpenSSL stop callers from switching to s2n-bignum's fast, constant-time, correct code? "OpenSSL is robust"? "C is better than asm"? "You're trapped in our API, get used to it"?

Shouldn't OpenSSL start calling s2n-bignum? Shouldn't other libraries also start doing this?

Supporting more cryptosystems

Adding verified asm for a new cryptosystem:

- Add pure asm implementations of the system. Maybe from an asm generator such as Jasmin; maybe from `gcc -S`; but remember that the asm will be the stable, verified, packaged code.

Supporting more cryptosystems

Adding verified asm for a new cryptosystem:

- Add pure asm implementations of the system. Maybe from an asm generator such as Jasmin; maybe from `gcc -S`; but remember that the asm will be the stable, verified, packaged code.
- Add proofs that the code works correctly. This is another toolkit to learn, but there are already some introductory examples, plus broader introductions to HOL Light.

Supporting more cryptosystems

Adding verified asm for a new cryptosystem:

- Add pure asm implementations of the system. Maybe from an asm generator such as Jasmin; maybe from `gcc -S`; but remember that the asm will be the stable, verified, packaged code.
- Add proofs that the code works correctly. This is another toolkit to learn, but there are already some introductory examples, plus broader introductions to HOL Light.

AWS reported 1 person-year for X25519.

Supporting more CPUs

Remember that s2n-bignum is only for 64-bit ARM and 64-bit AMD/Intel. On other platforms, libraries will fall back to other code—maybe bad code.

Supporting more CPUs

Remember that s2n-bignum is only for 64-bit ARM and 64-bit AMD/Intel. On other platforms, libraries will fall back to other code—maybe bad code.

Adding verified asm for, e.g., 32-bit ARM:

- Add+test central specification of the 32-bit ARM instruction set.
- Add pure asm for 32-bit ARM for each function. Every new addition helps!
- Add proofs that the code works correctly.

The bigger picture

Verification has moved from theory to practice.

Some related talks coming up at this conference:

- "PQConnect: automated post-quantum end-to-end tunnels"—the handshake protocol inside PQConnect is formally verified.
- "High-assurance post-quantum cryptography"—broader view of the ecosystem, including libcrux.

More projects: Cryptol/SAW/hacrypto, Cryptoline, Fiat-Crypto, HACL*, Libjade, ValeCrypt, VST.