

Fast verified post-quantum software

Daniel J. Bernstein

SymCrypt: failures for rare inputs

It's actually a bug within SymCrypt, the core cryptographic library responsible for implementing asymmetric crypto algorithms in Windows 10 and symmetric crypto algorithms in Windows 8.

—“Warning: Google Researcher Drops Windows 10 Zero-Day Security Bomb”,
Forbes, <https://tinyurl.com/y69fx3nh>

Falcon software: skewed randomness

Produced signatures were valid but leaked information on the private key. . . . The fact that these bugs existed in the first place shows that the traditional development methodology (i.e. ‘being super careful’) has failed.

—“OFFICIAL COMMENT” within NISTPQC (NIST Post-Quantum Cryptography Standardization Project), <https://tinyurl.com/y5w46bde>

Minerva: timing attack

Libgcrypt, wolfSSL, and Crypto++ have issued patches over the summer to fix this bug.

Maintainers of MatrixSSL fixed some issues, but the library remains vulnerable. Oracle's SunEC library remains open to attacks.

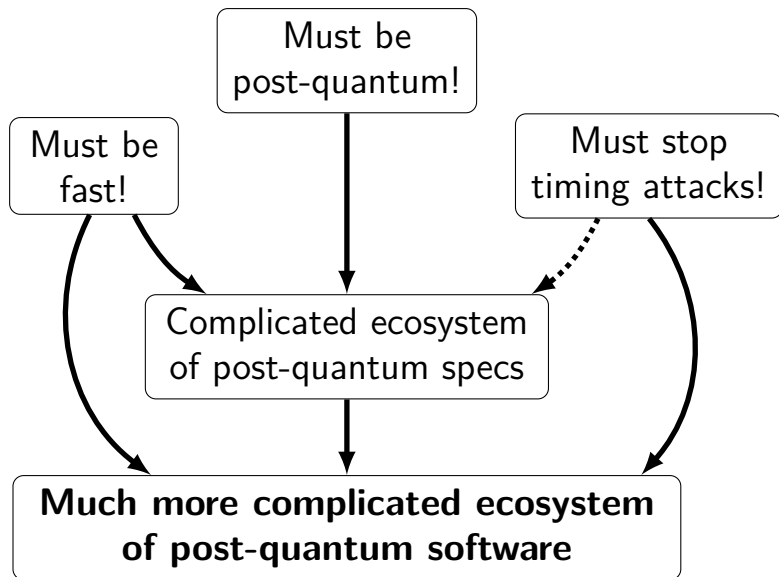
—“Minerva attack can recover private keys from smart cards, cryptographic libraries”, ZDNet, <https://tinyurl.com/y6rlkov4>

Cryptographic software has a problem . . .

2021.07 Blessing–Specter–Weitzner “You really shouldn’t roll your own crypto: an empirical study of vulnerabilities in cryptographic libraries”:

73 “actual” cryptographic vulnerabilities, including
11 “severe” cryptographic vulnerabilities, among
OpenSSL, GnuTLS, Mozilla TLS, WolfSSL, Botan,
Libgcrypt, LibreSSL, BoringSSL post-2010 CVEs.

... and the complexity is getting worse



Examples of the complications

Official Keccak (SHA-3) code package:

- `KeccakP-1600-reference.c`,
- `KeccakP-1600-x86-64-shld-gas.s`,
- `KeccakP-1600-AVX2.s`,
- `KeccakP-1600-AVX512.s`,
- `KeccakP-1600-times8-SIMD512.c`,
- ...

Much better speeds than using just
reference + “optimizing” compiler.

Examples of the complications

Official Keccak (SHA-3) code package:

- `KeccakP-1600-reference.c`,
- `KeccakP-1600-x86-64-shld-gas.s`,
- `KeccakP-1600-AVX2.s`,
- `KeccakP-1600-AVX512.s`,
- `KeccakP-1600-times8-SIMD512.c`,
- ...

Much better speeds than using just
reference + “optimizing” compiler.

Each NISTPQC candidate includes hand-optimized
software faster than state-of-the-art compiled code.

The good news: symbolic testing

Symbolic-testing tools check that optimized software equals reference software.

“Equals”: gives the same outputs **for all inputs**.

Today's tools are surprisingly easy to use and quickly handle many post-quantum subroutines.

The good news: symbolic testing

Symbolic-testing tools check that optimized software equals reference software.

“Equals”: gives the same outputs **for all inputs**.

Today's tools are surprisingly easy to use and quickly handle many post-quantum subroutines.

This talk: new saferewrite symbolic-testing tool.
Open source from <https://pqsrc.cr.yp.to>.

The good news: symbolic testing

Symbolic-testing tools check that optimized software equals reference software.

“Equals”: gives the same outputs **for all inputs**.

Today's tools are surprisingly easy to use and quickly handle many post-quantum subroutines.

This talk: new saferewrite symbolic-testing tool.
Open source from <https://pqsrc.cr.yp.to>.

Under the hood, doing most of the work:
valgrind; its VEX library; Z3 theorem prover;
`angr.io` binary-analysis/symbolic-execution toolkit.

Case study: `int16[64]` comparison

Subroutine used inside Frodo post-quantum KEM.
My ref version, `cmp_64xint16/ref/verify.c`:

```
#include <stdint.h>

int cmp_64xint16(const uint16_t *x,
                 const uint16_t *y)
{ for (int i = 0; i < 64; ++i)
    if (x[i] != y[i])
        return -1;
    return 0;
}
```

Automatic saferewrite analysis

Using `clang -O1 -fwrapv -march=native`:

- `saferewrite` says `unsafe-valgrindfailure`:
Code has variable branches/indices,
violating constant-time coding discipline.
- And `unsafe-unrollsplit-65`:
Unrolling split the code into 65 cases.

Automatic saferewrite analysis

Using `clang -O1 -fwrapv -march=native`:

- saferewrite says `unsafe-valgrindfailure`:
Code has variable branches/indices,
violating constant-time coding discipline.
- And `unsafe-unrollsplit-65`:
Unrolling split the code into 65 cases.

Using `gcc -O3 -march=native -mtune=native`:

- `unsafe-valgrindfailure`
- `unsafe-unrollsplit-65`
- `equals-ref-clang_-O1_...`:
`cmp_64xint16` binaries give same outputs.

Automatic analysis of a rewrite

```
#include <stdint.h>
#include <string.h>

int cmp_64xint16(const uint16_t *x,
                 const uint16_t *y)
{
    return memcmp(x,y,128);
}
```

Automatic analysis of a rewrite

```
#include <stdint.h>
#include <string.h>

int cmp_64xint16(const uint16_t *x,
                 const uint16_t *y)
{
    return memcmp(x,y,128);
}
```

Again unsafe-valgrindfailure: variable time.
Also unsafe-differentfrom-ref-clang_....
Why? Nonzero memcmp output isn't always -1.

Automatic analysis of another rewrite

```
#include <stdint.h>
#include <string.h>
int cmp_64xint16(const uint16_t *x,
                 const uint16_t *y)
{ int r = memcmp(x,y,128);
  if (r != 0) return -1;
  return 0;
}
```

Automatic analysis of another rewrite

```
#include <stdint.h>
#include <string.h>
int cmp_64xint16(const uint16_t *x,
                 const uint16_t *y)
{ int r = memcmp(x,y,128);
  if (r != 0) return -1;
  return 0;
}
```

Now equals-ref-clang_... but still
unsafe-valgrindfailure. 2017 Frodo software
used memcmp; broken by 2020.06 timing attack.

2020.06 Frodo official constant-time code

```
int8_t ct_verify(const uint16_t *a,
                  const uint16_t *b, size_t len)
{ // Compare two arrays in constant time.
  // Returns 0 if the byte arrays are equal,
  // -1 otherwise.
  uint16_t r = 0;
  for (size_t i = 0; i < len; i++) {
    r |= a[i] ^ b[i];
  }
  r = -(int16_t)r >> (8*sizeof(uint16_t)-1);
  return (int8_t)r;
}
```

Use saferewrite to analyze this ...

Add wrapper to fit the `cmp_64xint16` interface:

```
int cmp_64xint16(const uint16_t *x,
                 const uint16_t *y)
{ return ct_verify(x,y,64); }
```

`saferewrite` focuses on constant lengths.
(Frodo uses `int16[N]` for a few choices of `N`.)

Use saferewrite to analyze this ...

Add wrapper to fit the `cmp_64xint16` interface:

```
int cmp_64xint16(const uint16_t *x,
                 const uint16_t *y)
{ return ct_verify(x,y,64); }
```

`saferewrite` focuses on constant lengths.
(Frodo uses `int16[N]` for a few choices of `N`.)

Feed `ct_verify` and wrapper to `saferewrite`:

- No more `unsafe-valgrindfailure`: Great.

Use saferewrite to analyze this ...

Add wrapper to fit the `cmp_64xint16` interface:

```
int cmp_64xint16(const uint16_t *x,
                 const uint16_t *y)
{ return ct_verify(x,y,64); }
```

`saferewrite` focuses on constant lengths.
(Frodo uses `int16[N]` for a few choices of `N`.)

Feed `ct_verify` and wrapper to `saferewrite`:

- No more `unsafe-valgrindfailure`: Great.
- `unsafe-differentfrom-ref-...`: Oops!

Bug discovered 2020.12 by Saarinen; easy to exploit.

A safe rewrite: correct constant-time code

```
#include <stdint.h>
int cmp_64xint16(const uint16_t *x,
                 const uint16_t *y)
{ uint32_t differences = 0;
  for (long long i = 0; i < 64; ++i)
    differences |= x[i] ^ y[i];
  return (1 & ((differences - 1) >> 16)) - 1;
}
```

Now saferewrite analysis with both compilers
says equals-ref-... and no more unsafe.

Examples in saferewrite package

10 sample implementations of `cmp_64xint16`.
One uses OpenSSL's `CRYPTO_memcmp` Intel asm;
see CVE-2018-0733 re `CRYPTO_memcmp` HP asm.

Examples in saferewrite package

10 sample implementations of `cmp_64xint16`.

One uses OpenSSL's `CRYPTO_memcmp` Intel asm;
see CVE-2018-0733 re `CRYPTO_memcmp` HP asm.

103 sample implementations of 39 other functions.

Some functions much bigger than `cmp_64xint16`.

Some simple functions for exercising `saferewrite`.

Examples in saferewrite package

10 sample implementations of `cmp_64xint16`.

One uses OpenSSL's `CRYPTO_memcmp` Intel asm;
see CVE-2018-0733 re `CRYPTO_memcmp` HP asm.

103 sample implementations of 39 other functions.

Some functions much bigger than `cmp_64xint16`.

Some simple functions for exercising `saferewrite`.

`unsafe-differentfrom` automatically includes
example of an input triggering the difference.

Can be hard to find by traditional testing/fuzzing!

Examples in saferewrite package

10 sample implementations of `cmp_64xint16`.

One uses OpenSSL's `CRYPTO_memcmp` Intel asm;
see CVE-2018-0733 re `CRYPTO_memcmp` HP asm.

103 sample implementations of 39 other functions.
Some functions much bigger than `cmp_64xint16`.
Some simple functions for exercising `saferewrite`.

`unsafe-differentfrom` automatically includes
example of an input triggering the difference.

Can be hard to find by traditional testing/fuzzing!

Beware: automatically uses many cores, big RAM.

Tip: `chmod +t src/*; chmod -t src/cmp*`

Example: integer-sequence encoders

Existing optimized code from NTRU Prime,
with heavy use of Intel AVX2 vector instructions:

- 245-line `encode_761x1531/avx/encode.c`
`encode.c` and similar encoders for other sizes are
automatically generated by 239-line Python script.

Example: integer-sequence encoders

Existing optimized code from NTRU Prime, with heavy use of Intel AVX2 vector instructions:

- 245-line `encode_761x1531/avx/encode.c`

`encode.c` and similar encoders for other sizes are automatically generated by 239-line Python script.

Existing reference code, much simpler:

- 38-line `encode_761x1531/ref/Encode.c`
- 18-line `encode_761x1531/ref/wrapper.c`

Example: integer-sequence encoders

Existing optimized code from NTRU Prime,
with heavy use of Intel AVX2 vector instructions:

- 245-line `encode_761x1531/avx/encode.c`
`encode.c` and similar encoders for other sizes are
automatically generated by 239-line Python script.

Existing reference code, much simpler:

- 38-line `encode_761x1531/ref/Encode.c`
- 18-line `encode_761x1531/ref/wrapper.c`

“Is the optimized code a safe rewrite of ref?”

Automatic saferewrite analysis: `equals-ref`.

Excerpt from avx/encode.c

```
x = _mm256_loadu_si256((__m256i *) reading);
x = _mm256_add_epi16(x, _mm256_set1_epi16(2295));
x &= _mm256_set1_epi16(16383);
x = _mm256_mulhi_epi16(x, _mm256_set1_epi16(21846));
y = x & _mm256_set1_epi32(65535);
x = _mm256_srli_epi32(x, 16);
x = _mm256_mullo_epi32(x, _mm256_set1_epi32(1531));
x = _mm256_add_epi32(y, x);
x = _mm256_shuffle_epi8(x, _mm256_set_epi8(
    12, 8, 4, 0, 12, 8, 4, 0, 14, 13, 10, 9, 6, 5, 2, 1,
    12, 8, 4, 0, 12, 8, 4, 0, 14, 13, 10, 9, 6, 5, 2, 1
));
x = _mm256_permute4x64_epi64(x, 0xd8);
_mm_storeu_si128((__m128i *) writing,
    _mm256_extractf128_si256(x, 0));
*((uint32 *) (out+0)) = _mm256_extract_epi32(x, 4);
*((uint32 *) (out+4)) = _mm256_extract_epi32(x, 6);
```


saferewrite package is available now from
<https://pqsrc.cr.yp.to>. Work in progress:

- More post-quantum case studies.
- More pre-quantum case studies: e.g., Ed25519.
- More languages: e.g., support Python ref.
- Developer integration: incremental testing etc.
- “Cuts”: subroutine swaps etc. for faster testing.
- Plugins for dedicated equivalence testers.
- Higher assurance for the entire toolchain.

Related work: [Cryptol](#)/[SAW](#)/[hacrypto](#), [Cryptoline](#),
[Fiat-Crypto](#), [HACL*](#), [Jasmin](#), [ValeCrypt](#), [VST](#).