Cryptographic code snippets

Daniel J. Bernstein

Timing attacks

Example of exploiting timing variations

June 2024 paper from Bernstein, Bhargavan, Bhasin, Chattopadhyay, Chia, Kannwischer, Kiefer, Paiva, Ravi, Tamvada: "KyberSlash: Exploiting secret-dependent division timings in Kyber implementations."

https://kyberslash.cr.yp.to/papers.html

Demos: exploiting KyberSlash1/2 to extract secret key from reference Kyber-512/Kyber-768 software on a Cortex-A7/Cortex-M4 in a few hours/minutes.

The code that was exploited

```
// Replaced KYBER_Q with Q to fit on slide.
// KyberSlash1:
t=(((t<<1)+Q/2)/Q)&1;
...
// KyberSlash2:
t[j]=((((uint16_t)u<<4)+Q/2)/Q)&15;
...
t[j]=((((uint32_t)u<<5)+Q/2)/Q)&31;
...
t[k]=((((uint32_t)t[k]<<11)+Q/2)/Q)&0x7ff;
...
t[k]=((((uint32_t)t[k]<<10)+Q/2)/Q)&0x3ff;
```

More examples this year

May 2024: I posted a demo that often recovers keys from the official optimized software for SMAUG-T on an Intel Skylake in minutes.

(Context: SMAUG-T is a smaller KEM than Kyber.)

June 2024: Purnal posted a demo that reportedly recovers secret key from reference Kyber-512 software in minutes on a laptop, *if* the software is compiled with clang 15 (2022) or newer.

One of many older examples

2020 Guo–Johansson–Nilsson FrodoKEM attack paper "A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM": "Experiments show that the attack code is able to extract the secret key for all security levels using about 2^{30} decapsulation calls."

One of many older examples

2020 Guo–Johansson–Nilsson FrodoKEM attack paper "A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM": "Experiments show that the attack code is able to extract the secret key for all security levels using about 2^{30} decapsulation calls."

This was a timing leak from `memcmp`.

Blame the protocols?

"Protocols should never use long-term keys! Do whatever you have to do to keep switching keys!"

Blame the protocols?

"Protocols should never use long-term keys! Do whatever you have to do to keep switching keys!"

— This stops many timing-attack demos but doesn't guarantee security.

See, e.g., 2018 "Single trace attack against RSA key generation in Intel SGX SSL".

Blame the protocols?

"Protocols should never use long-term keys! Do whatever you have to do to keep switching keys!"

— This stops many timing-attack demos but doesn't guarantee security.

See, e.g., 2018 "Single trace attack against RSA key generation in Intel SGX SSL".

We know one way to systematically stop all timing attacks: eliminate data flow from secrets to timings.

Writing constant-time code

Multiple levels of strategies

My low-level focus for the rest of the talk:

- Take snippets of code with timing variations.
- Rewrite to avoid timing variations.
- Make sure the rewrite is correct.

Multiple levels of strategies

My low-level focus for the rest of the talk:

- Take snippets of code with timing variations.
- Rewrite to avoid timing variations.
- Make sure the rewrite is correct.

Higher-level strategies that I won't cover today:

- Reorganizing higher-level computations (e.g., bitslicing, or changing algorithms for sorting and invmod) to streamline constant-time code.

Multiple levels of strategies

My low-level focus for the rest of the talk:

- Take snippets of code with timing variations.
- Rewrite to avoid timing variations.
- Make sure the rewrite is correct.

Higher-level strategies that I won't cover today:

- Reorganizing higher-level computations (e.g., bitslicing, or changing algorithms for sorting and invmod) to streamline constant-time code.
- Switching to cryptosystems that simplify constant-time code. Examples: X25519, Ed25519, Salsa20, ChaCha20, and more.

Case study: incrementing an array

Let's start with this self-contained function:

```
#include <stdint.h>
void inc128big(int8_t x[16])
{
  for (int i = 15;i >= 0;--i)
    if (++x[i])
      break;
}
```

Note: Always compile with `-fwrapv` (or equivalent) to guarantee twos-complement arithmetic.

First rewrite

Always run loop to the maximum length:

```
#include <stdint.h>
void inc128big(int8_t x[16])
{
  int8_t mask = -1;
  for (int i = 15;i >= 0;--i) {
    x[i] -= mask;
    if (x[i] != 0) mask = 0;
  }
}
```

“Mask” convention: -1 for true, 0 for false.

Second rewrite

Always update mask using logic operation:

```
#include <stdint.h>
void inc128big(int8_t x[16])
{
  int8_t mask = -1;
  for (int i = 15;i >= 0;--i) {
    x[i] -= mask;
    mask &= -(x[i] == 0);
  }
}
```

Constant-time for some CPUs and compilers.

Third rewrite

```
#include "crypto_int8.h"
#include <stdint.h>
void inc128big(int8_t x[16])
{
  int8_t mask = -1;
  for (int i = 15;i >= 0;--i) {
    x[i] -= mask;
    mask &= crypto_int8_zero_mask(x[i]);
  }
}
```

Uses constant-time subroutine from SUPERCOP.

Do the rewritten snippets work?

Maybe even more rewrites in asm.
Probably `inc128big` speed doesn't matter,
but asm is the best defense against compilers.

With or without asm, rewrites risk introducing bugs.

Do the rewritten snippets work?

Maybe even more rewrites in asm.
Probably `inc128big` speed doesn't matter,
but asm is the best defense against compilers.

With or without asm, rewrites risk introducing bugs.

e.g. CVE-2018-0733 for OpenSSL:
"Because of an implementation bug the PA-RISC `CRYPTO_memcmp` function is effectively reduced to only comparing the least significant bit of each byte." Bug introduced May 2016.

Do the rewritten snippets work?

Maybe even more rewrites in asm.
Probably `inc128big` speed doesn't matter,
but asm is the best defense against compilers.

With or without asm, rewrites risk introducing bugs.

e.g. CVE-2018-0733 for OpenSSL:
"Because of an implementation bug the PA-RISC `CRYPTO_memcmp` function is effectively reduced to only comparing the least significant bit of each byte." Bug introduced May 2016.

e.g. FrodoKEM replaced `memcmp` with a buggy constant-time rewrite, allowing a faster attack.

Obvious response: test, test, test

Testing millions of random inputs for `inc128big` feels like it *should* trigger any bugs.

Also use fuzzing techniques. Try 2^{16} `inc128big` inputs where each input byte is -1 or 0. Try arrays for `CRYPTO_memcmp` that differ in just a few bits.

Knuth's *Art of Computer Programming*: "it is also necessary to find some test cases that cause the rarely executed parts of the program to be exercised".

Obvious response: test, test, test

Testing millions of random inputs for `inc128big` feels like it *should* trigger any bugs.

Also use fuzzing techniques. Try 2^{16} `inc128big` inputs where each input byte is -1 or 0. Try arrays for `CRYPTO_memcmp` that differ in just a few bits.

Knuth's *Art of Computer Programming*: "it is also necessary to find some test cases that cause the rarely executed parts of the program to be exercised".

But there are endless snippets being rewritten. Some bugs are going to slip through.

More powerful: symbolic testing

Download `saferewrite` and create `src/inc128big/api` with these two lines:

```
inout int8 x 16
call inc128big
```

Create `src/inc128big/ref/inc.c`, `src/inc128big/mask1/inc.c`, etc., with the `inc128big` implementations from my slides.

Then follow the `saferewrite` instructions.

Results of symbolic testing

Outputs in `build/inc128big` within a few minutes:

```
mask1/.../analysis/equals-ref-...
mask2/.../analysis/equals-ref-...
mask3/.../analysis/equals-ref-...
```

This analysis is unrolling the compiled binaries and using an “SMT solver” to show that the outputs are the same for *all* inputs.

Results of symbolic testing

Outputs in `build/inc128big` within a few minutes:

```
mask1/.../analysis/equals-ref-...
mask2/.../analysis/equals-ref-...
mask3/.../analysis/equals-ref-...
```

This analysis is unrolling the compiled binaries and using an "SMT solver" to show that the outputs are the same for *all* inputs.

Of course, `saferewrite` or the SMT solver could have bugs. Don't skip conventional tests!

Inside crypto_int8_zero_mask

```
crypto_int8 crypto_int8_zero_mask
  (crypto_int8 x) {
#if defined(__GNUC__) && defined(__aarch64__)
  crypto_int8 z;
  __asm__ ("tst %w1,255\n csetm %w0,eq" :
    "=r"(z) : "r"(x) : "cc");
  return z;
#else
  return ~crypto_int8_nonzero_mask(x);
#endif
}
```

Inside crypto_int8_nonzero_mask

```
crypto_int8 crypto_int8_nonzero_mask
  (crypto_int8 x) {
#if defined(__GNUC__) && defined(__aarch64__)
  crypto_int8 z;
  __asm__ ("tst %w1,255\n csetm %w0,ne" :
    "=r"(z) : "r"(x) : "cc");
  return z;
#else
  x |= -x;
  return crypto_int8_negative_mask(x);
#endif
}
```

Inside crypto_int8_negative_mask

```
crypto_int8 crypto_int8_negative_mask(crypto_int8 x) {
#if defined(__GNUC__) && defined(__x86_64__)
  __asm__ ("sarb $7,%0" : "+r"(x) : : "cc");
  return x;
#elif defined(__GNUC__) && defined(__aarch64__)
  crypto_int8 y;
  __asm__ ("sbfx %w0,%w1,7,1" : "=r"(y) : "r"(x) : );
  return y;
#else
  x >>= 8-6;
  x ^= crypto_int8_optblocker;
  x >>= 5;
  return x;
#endif
}
```

Why not just shift right by 7 in C?

Standard advice for many years:
don't use secret `bool` in constant-time code;
e.g., don't use secret `!`, `<`, `&&`, etc. in C or C++.
(The C language turns those into `int`, but the programmer and compiler know that they're `bool`.)

When papers complain about compilers producing branches, one finds `bool` in the original code;
BearSSL says "Avoid boolean types"; etc.

Why not just shift right by 7 in C?

Standard advice for many years:
don't use secret `bool` in constant-time code;
e.g., don't use secret `!`, `<`, `&&`, etc. in C or C++.
(The C language turns those into `int`, but the programmer and compiler know that they're `bool`.)

When papers complain about compilers producing branches, one finds `bool` in the original code;
BearSSL says "Avoid boolean types"; etc.

April 2024: I pointed out that current compilers are sometimes "optimizing" arithmetic into `bool`.

What's `optblocker`?

`optblocker` is a `volatile` variable set to 0.

The usage of `optblocker` is designed to prevent compilers from seeing that there's a 1-bit result.

What's `optblocker`?

`optblocker` is a `volatile` variable set to 0.

The usage of `optblocker` is designed to prevent compilers from seeing that there's a 1-bit result.

e.g. `crypto_int8_negative_mask(x)` returns `((x>>2)^crypto_int8_optblocker)>>5`.

What's `optblocker`?

`optblocker` is a `volatile` variable set to 0.

The usage of `optblocker` is designed to prevent compilers from seeing that there's a 1-bit result.

e.g. `crypto_int8_negative_mask(x)` returns `((x>>2)^crypto_int8_optblocker)>>5`.

e.g. `crypto_int8_bottombit_mask(x)` returns `-(x&(1^crypto_int8_optblocker))`.

Do these functions actually work?

There are more of these functions,
times `int` vs. `uint`,
times 8 vs. 16 vs. 32 vs. 64.
Overall 144 `crypto_{int,uint}` functions.

Do these functions actually work?

There are more of these functions,
times `int` vs. `uint`,
times 8 vs. 16 vs. 32 vs. 64.
Overall 144 `crypto_{int,uint}` functions.

All checked against reference implementations
by `saferewrite` on various platforms,
plus various conventional tests in SUPERCOP.
Better to centralize on these functions
than have everyone reinventing the wheel.

Extra concerns for inline asm

There are 196 `__asm__` lines in these files, each with input-output declarations such as `"+r"(x) : : "cc"`. Declaration errors can produce bugs in *some* callers despite passing tests.

The rules here are more complicated and error-prone than the “restore all callee-save registers” rule for separate asm functions; but I want these `.h` files usable in programs that don’t want separate asm.

Extra concerns for inline asm

There are 196 `__asm__` lines in these files, each with input-output declarations such as `"+r"(x) : : "cc"`. Declaration errors can produce bugs in *some* callers despite passing tests.

The rules here are more complicated and error-prone than the “restore all callee-save registers” rule for separate asm functions; but I want these `.h` files usable in programs that don’t want separate asm.

So I actually auto-generate these `__asm__` lines from lines in a simpler `readasm` language.

The actual source code

This is the input used to auto-generate `crypto_{int,uint}{8,16,32,64}_zero_mask`:

```
TYPE TYPE_zero_mask(TYPE X) {
#if arm64
  TYPE Z;
 8:  readasm("arm64; int8 X Z; X & 255; Z = -1 if = else 0");
16:  readasm("arm64; int16 X Z; X & 65535; Z = -1 if = else 0");
32:  readasm("arm64; int32 X Z; X - 0; Z = -1 if = else 0");
64:  readasm("arm64; int64 X Z; X - 0; Z = -1 if = else 0");
  return Z;
#else
  return ~TYPE_nonzero_mask(X);
#endif
}
```