Software analysis of the KpqC candidates

(+ bonus slides on patents)

Daniel J. Bernstein

Objectives for cryptographic software

Assume specified cryptosystem X meets its security goals: e.g., IND-CCA2 security 2^{256} for a KEM.

Basic questions about software for X:

- Is the software correct, i.e., does it compute the specified functions for all inputs?
- Is the software constant-time, i.e., does it avoid leaking secrets through timings?
- Is the software efficient enough to meet the user's cost constraints?

Bugs in software can damage security

From the NCC-Sign code (minus braces):

```
if (t0 < 3)
        a[ctr++] = 1 - t0;
if (t1 < 3 && ctr < len)
        a[ctr++] = 1 - t0;
if (t2 < 3 && ctr < len)
        a[ctr++] = 1 - t0;
if (t3 < 3 && ctr < len)
        a[ctr++] = 1 - t0;
```

Similar randomness-reuse bugs were announced in 2018+2019 in the official Dilithium+Falcon code.

Some advice on catching bugs

Compile and test with `-fsanitize=address`.

Some advice on catching bugs

Compile and test with `-fsanitize=address`.

Test bad inputs: e.g., modified ciphertexts.

Some advice on catching bugs

Compile and test with `-fsanitize=address`.

Test bad inputs: e.g., modified ciphertexts.

Follow https://bench.cr.yp.to/tips.html to integrate your software into SUPERCOP. See if SUPERCOP's tests pass. See if multiple compiler options produce identical SUPERCOP checksums.

Some advice on catching bugs

Compile and test with `-fsanitize=address`.

Test bad inputs: e.g., modified ciphertexts.

Follow https://bench.cr.yp.to/tips.html to integrate your software into SUPERCOP. See if SUPERCOP's tests pass. See if multiple compiler options produce identical SUPERCOP checksums.

Have someone else use the specification to write Python software. Compute checksums from Python and see if those match the checksums from C code.

Some advice on catching bugs

Compile and test with `-fsanitize=address`.

Test bad inputs: e.g., modified ciphertexts.

Follow https://bench.cr.yp.to/tips.html to integrate your software into SUPERCOP. See if SUPERCOP's tests pass. See if multiple compiler options produce identical SUPERCOP checksums.

Have someone else use the specification to write Python software. Compute checksums from Python and see if those match the checksums from C code.

Expensive but convincing: computer-checked proofs.

Timing variations can damage security

In 6 experiments out of 10, my 2024-05-11 attack script recovered a long-term SMAUG-T128 secret key from $<$10 minutes of decapsulation timings of the SMAUG-T optimized code.

Timing variations can damage security

In 6 experiments out of 10, my 2024-05-11 attack script recovered a long-term SMAUG-T128 secret key from $<$10 minutes of decapsulation timings of the SMAUG-T optimized code.

"What if we use keys just once?" — 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".

Some advice on catching timing variations

After integrating software into SUPERCOP:
mark `goal-constbranch` and `goal-constindex`;
then run TIMECOP 2, which is part of SUPERCOP.

Some advice on catching timing variations

After integrating software into SUPERCOP:
mark `goal-constbranch` and `goal-constindex`;
then run TIMECOP 2, which is part of SUPERCOP.

When the code passes all tests on your machine,
submit the software to SUPERCOP: e.g.,

- AIMER is in supercop-20240716,
- HAETAE is in supercop-20240625,
- MQ-Sign is in supercop-20240625,
- NTRU+ is in supercop-20240625.

Then `https://bench.cr.yp.to` shows tests,
including TIMECOP results, from many machines.

What TIMECOP checks for

TIMECOP runs your code, checking whether there is any data flow from secret inputs (including randomness) to branch conditions or array indices.

What TIMECOP checks for

TIMECOP runs your code, checking whether there is any data flow from secret inputs (including randomness) to branch conditions or array indices.

Limitations: TIMECOP will not

- catch data flow not visible in these runs (e.g., does signing a long message trigger a branch?),

What TIMECOP checks for

TIMECOP runs your code, checking whether there is any data flow from secret inputs (including randomness) to branch conditions or array indices.

Limitations: TIMECOP will not

- catch data flow not visible in these runs (e.g., does signing a long message trigger a branch?),
- run perfectly on all computers (e.g., it gives up on AMD XOP instructions),

What TIMECOP checks for

TIMECOP runs your code, checking whether there is any data flow from secret inputs (including randomness) to branch conditions or array indices.

Limitations: TIMECOP will not

- catch data flow not visible in these runs (e.g., does signing a long message trigger a branch?),
- run perfectly on all computers (e.g., it gives up on AMD XOP instructions),
- catch variable-time instructions other than branches and memory accesses, or

What TIMECOP checks for

TIMECOP runs your code, checking whether there is any data flow from secret inputs (including randomness) to branch conditions or array indices.

Limitations: TIMECOP will not

- catch data flow not visible in these runs (e.g., does signing a long message trigger a branch?),
- run perfectly on all computers (e.g., it gives up on AMD XOP instructions),
- catch variable-time instructions other than branches and memory accesses, or
- fix the problems it finds.

Declassification

Sometimes you need data flow from randomness to branches for rejection sampling.
e.g., to generate a prime p for an RSA key:
generate an integer p; if p is not prime, start over.

Declassification

Sometimes you need data flow from randomness to branches for rejection sampling.
e.g., to generate a prime p for an RSA key: generate an integer p; if p is not prime, start over.

To tell TIMECOP to stop tracing data flow from a variable, use `crypto_declassify`:

```
#include "crypto_declassify.h"
...
do {
  ...
  crypto_declassify(&rej,sizeof rej);
} while (rej);
```

Divisions involving secrets

HAETAE code uses this function for secret inputs:

```
int32_t fix_round(int32_t num) {
    num += (num >> 31) & (-LN + 1);
    num += LN / 2;
    return num / LN;
}
```

`LN` is a power of 2. Easy to avoid division here:

```
int32_t fix_round(int32_t num) {
  return (num + LNHALF) >> LNBITS;
}
```

Divisions can create timing variations

Compilers will often replace “divide by constant” with an appropriate multiplication, but not always. Depends on platform and compiler options.

Divisions can create timing variations

Compilers will often replace "divide by constant" with an appropriate multiplication, but not always. Depends on platform and compiler options.

Discovery in December 2023: divisions in the Kyber reference code are exploitable on some platforms.

Divisions can create timing variations

Compilers will often replace "divide by constant" with an appropriate multiplication, but not always. Depends on platform and compiler options.

Discovery in December 2023: divisions in the Kyber reference code are exploitable on some platforms.

New paper "KyberSlash: Exploiting secret-dependent division timings in Kyber implementations" also reports results of a TIMECOP patch to look for secret divisions.

New compilers can add timing variations

April 2024: "Tracking down some TIMECOP alerts led to a 2021 gcc patch from ARM ... turning `(-x)>>31` into a bool, often breaking constant-time code. Can often work around with `(-x)>>30`, and asm is safer anyway, but for portable fallbacks we need security-aware compilers."

New compilers can add timing variations

April 2024: "Tracking down some TIMECOP alerts led to a 2021 gcc patch from ARM ...
turning `(-x)>>31` into a bool, often breaking constant-time code. Can often work around with `(-x)>>30`, and asm is safer anyway, but for portable fallbacks we need security-aware compilers."

Starting with version 15 in 2022,
clang often turns `x&1` into variable branches.

New compilers can add timing variations

April 2024: "Tracking down some TIMECOP alerts led to a 2021 gcc patch from ARM ... turning `(-x)>>31` into a bool, often breaking constant-time code. Can often work around with `(-x)>>30`, and asm is safer anyway, but for portable fallbacks we need security-aware compilers."

Starting with version 15 in 2022,
clang often turns `x&1` into variable branches.

TIMECOP catches these timing variations
if it's run with these compilers.

Constant-time subroutines in SUPERCOP

```
#include "crypto_int32.h"
crypto_int32_positive_mask(x);
crypto_int32_negative_mask(x);
crypto_int32_zero_mask(x);
crypto_int32_nonzero_mask(x);
crypto_int32_equal_mask(x,y);
crypto_int32_unequal_mask(x,y);
crypto_int32_smaller_mask(x,y);
crypto_int32_leq_mask(x,y);
crypto_int32_bottombit_mask(x);
crypto_int32_bitmod_mask(x,i);
...
```

Constant-time array lookups

```
uint8_t loadbyte(const uint8_t *in,
  crypto_int64 inlen,crypto_int64 inpos)
{
  uint8_t result = 0;
  while (inlen > 0) {
    result |=
      *in & crypto_int64_zero_mask(inpos);
    ++in;
    --inlen;
    --inpos;
  }
  return result;
}
```

Variable-time Fisher–Yates shuffling

In the Paloma code, (somewhat) random shuffling:

```
for (i = set_len - 1; i > 0; i--)
{
    j = ((seed[w % 16]) % (i + 1));
    /* Swap */
    tmp = shuffled_set[j];
    shuffled_set[j] = shuffled_set[i];
    shuffled_set[i] = tmp;

    w = (w + 1) & 0xf;
}
```

Shuffling, continued

Shuffling is also used in many other cryptosystems: e.g., NCC-Sign, SMAUG-T.

Often specifications ask for Fisher–Yates, and all implementations of the cryptosystems have to use Fisher–Yates for interoperability.

Implementors can use constant-time array lookups, but then total Fisher–Yates time is n^2.

I recommend changing the specifications to use cr.yp.to/papers.html#divergence or eprint.iacr.org/2024/548 or cr.yp.to/2024/insertionseries-20240515.py.

Example of constant-time shuffling

`sntrup` generating $r \in \{-1, 0, 1\}^n$ of weight w:

- generate array of n random 32-bit integers;
- `for(i=0;i<w;++i) r[i] = (r[i]&-2);`
- `for(i=w;i<n;++i) r[i] = (r[i]&-3)|1;`
- `crypto_sort_uint32(r,n);`
- `for(i=0;i<n;++i) r[i] = (r[i]&3)-1;`

SUPERCOP provides `crypto_sort_uint32`.

Example of constant-time shuffling

`sntrup` generating $r \in \{-1, 0, 1\}^n$ of weight w:

- generate array of n random 32-bit integers;
- `for(i=0;i<w;++i) r[i] = (r[i]&-2);`
- `for(i=w;i<n;++i) r[i] = (r[i]&-3)|1;`
- `crypto_sort_uint32(r,n);`
- `for(i=0;i<n;++i) r[i] = (r[i]&3)-1;`

SUPERCOP provides `crypto_sort_uint32`.

cr.yp.to/papers.html#divergence proves that, for search problems (not distinguishing problems!), this is close enough to uniform random if $n \leq 2^{13}$. With more work, can do exactly uniform random.

Efficiency

See bench.cr.yp.to/results-kem.html
for measurements so far for NTRU+;
bench.cr.yp.to/results-sign.html
for measurements so far for HAETAE, MQ-Sign,
and (starting to appear) AIMER.

Also click on system names from
bench.cr.yp.to/primitives-kem.html and
bench.cr.yp.to/primitives-sign.html
to find implementation notes:
per-implementation speeds, test failures,
checksum failures, TIMECOP failures, etc.

Bonus slides: Patents

What is a patent?

Patent: government-issued monopoly on an "invention". Using the "invention" without authorization from the patent holder is unlawful.

I'll take the U.S. as a running example.
U.S. law says that anyone who "without authority makes, uses, offers to sell, or sells any patented invention, within the United States or imports into the United States any patented invention during the term of the patent therefor, infringes the patent".

How patent infringement is penalized

Example of the process:

- You distribute free software.
- 6 years later, patent holder files a lawsuit.
- Patent holder says: people using your software used the patented invention; this reduced the patent holder's income by 10 million USD.
- Court agrees.
- Court chooses N with $10 \leq N \leq 30$ and forces you to pay N million USD to patent holder.

How courts evaluate patents, part 1

Each patent has a series of "claims".
The patent is infringed if any claim is infringed.

e.g. claim 1 of the RSA patent: "A cryptographic communications system comprising: ... a communications channel ... M corresponds to a number representative of a message ... $n = p \cdot q$ ... $C \equiv M^e \pmod{n}$" etc.

Court asks: What is a "channel"? What is a "message"? What is "representative"? etc.

How courts evaluate patents, part 2

"The scope of a patent **is not limited to its literal terms** but instead embraces all equivalents to the claims described". (Unanimous 2002 U.S. Supreme Court decision; emphasis added. There are many previous court cases like this.)

Patent holder wins if "the accused product performs substantially the same function in substantially the same way with substantially the same result".

Example of equivalents

The RSA patent's *introduction* says "the present invention may use a modulus n which is a product of three or more primes (not necessarily distinct)".

But the *claims* say specifically "$n = p \cdot q$ where p and q are prime numbers".

Example of equivalents

The RSA patent's *introduction* says "the present invention may use a modulus n which is a product of three or more primes (not necessarily distinct)".

But the *claims* say specifically "$n = p \cdot q$ where p and q are prime numbers".

— "Aha, they made a mistake in writing the patent claims! I can take $n = 3pq$ and avoid the patent!"

Example of equivalents

The RSA patent's *introduction* says "the present invention may use a modulus n which is a product of three or more primes (not necessarily distinct)".

But the *claims* say specifically "$n = p \cdot q$ where p and q are prime numbers".

— "Aha, they made a mistake in writing the patent claims! I can take $n = 3pq$ and avoid the patent!"

— No. This is performing *substantially* the same function in *substantially* the same way with *substantially* the same result, so it infringes.

Example of RSA patent enforcement

May 1991: Mark Riordan released
free "rpem" encryption software
using Rabin ciphertexts $C = M^2 \bmod pq$.

Patent holders sent a letter saying that Riordan was infringing the RSA patent (and the DH patent).
Riordan withdrew the software.

The claims said M^e where "e is a number relatively prime to 1 cm(p-1,q-1)", but M^2 accomplishes substantially the same thing.

More patent-enforcement examples

July 2016: Google rolled out post-quantum experiment: "we plan to discontinue this experiment within two years, hopefully by **replacing it with something better**" (emphasis added); "ensure our users' data will remain secure long into the future".

A patent holder asked Google for money. November 2016: Google said it was **ending** the experiment.

More patent-enforcement examples

July 2016: Google rolled out post-quantum experiment: "we plan to discontinue this experiment within two years, hopefully by **replacing it with something better**" (emphasis added); "ensure our users' data will remain secure long into the future".

A patent holder asked Google for money. November 2016: Google said it was **ending** the experiment.

Another example, 2015: "Patent troll claims HTTPS websites infringe crypto patent, sues everybody ... Netflix and others are fighting back while Scotttrade and others are settling".

Myth: patents promote progress

The U.S. constitution gives the legislature the power to "promote the Progress of Science and useful Arts, by securing for limited Times to Authors and Inventors the exclusive Right to their respective Writings and Discoveries".

Patents are granted only to "inventors", and expire after 20 years.

Reality: patents damage progress

2012 Lemley "The myth of the sole inventor":
"Surveys of hundreds of significant new technologies show that almost all of them are invented simultaneously or nearly simultaneously by two or more teams working independently of each other."

Reality: patents damage progress

2012 Lemley "The myth of the sole inventor": "Surveys of hundreds of significant new technologies show that almost all of them are invented simultaneously or nearly simultaneously by two or more teams working independently of each other."

1978 Rabin: a "public-key system employing large prime numbers was discovered by the author ... and independently by Rivest, Adleman and Shamir".

The lack-of-novelty defense

You win if you can find **one reference** that contains **every** element of the claim and was published before the patent was filed.

The lack-of-novelty defense

You win if you can find **one reference** that contains **every** element of the claim and was published before the patent was filed.

Logical consequence of "every": a patent doesn't stop subsequent patents on specializations.

The lack-of-novelty defense

You win if you can find **one reference** that contains **every** element of the claim and was published before the patent was filed.

Logical consequence of "every": a patent doesn't stop subsequent patents on specializations.

e.g. The RSA patent doesn't prevent a subsequent patent on "RSA with $p \equiv q \equiv 1 \pmod{2^{64}}$".
If you then use RSA with $p \equiv q \equiv 1 \pmod{2^{64}}$, you're infringing *both* patents.

The obviousness defense

You win if you can convince a court that,
before the patent was filed,
the “invention” was already obvious
to someone of “ordinary skill in the art”.

The obviousness defense

You win if you can convince a court that,
before the patent was filed,
the "invention" was already obvious
to someone of "ordinary skill in the art".

Your expert witnesses say it's obvious. Expert witnesses for the patent holder say it isn't obvious. You were sued in a court in Texas, and the decision is made by a jury of random citizens of Texas, usually concluding that it isn't obvious.

The ensnarement defense

When patent holder says you're using
something equivalent to the claimed "invention",
you can respond that "equivalent" is too broad
since it covers pre-patent publications too.
e.g. argue that taking $e = (p-1)(q-1)$
includes something published previously.

The ensnarement defense

When patent holder says you're using something equivalent to the claimed "invention", you can respond that "equivalent" is too broad since it covers pre-patent publications too. e.g. argue that taking $e = (p-1)(q-1)$ includes something published previously.

Court then requires the patent holder to state a "hypothetical" claim that literally covers what you're doing without covering those publications. e.g. "$\gcd\{e, \operatorname{lcm}\{p-1, q-1\}\}$ is below 1000".

Assessing post-quantum patents

A Google Patents search for "post-quantum" finds

- 100 results filed before 2017-06-29,
- 100 more results filed before 2018-07-27,
- 100 more results filed before 2019-03-18,
- 100 more results filed before 2019-04-17,
- etc.

Of course, a patent relevant to post-quantum cryptography doesn't have to say "post-quantum".

Analyzing a patent threat can be difficult.
What a court will do is often clear, often not.
Critical to understand the rules: equivalents etc.

Some patents that worry me

2010-02-18, US 9094189 (and corresponding patents in other countries), GAM/LPR.

2011-02-09, US 8522033, secret-sharing signatures.

2012-04-12, US 9246675, smaller GAM/LPR ct.

2016-11-18, US 11329799, GAM/LPR + rounding.

2016-11-18, CN 107566121, GAM/LPR variants.

2017-02-15, US 11070367, GAM/LPR + rounding.

2017-06-09, US 9912479, QC-MDPC.

2018-09-12, US 11798435, anti-SCA poly mult.