Constant time algorithms in PQC

`https://lukas-prokop.at/talks/2022-01-26_rustgraz-const-time`

Lukas Prokop

2022-01-27

RustGraz community

Introduction

Cryptography status-quo

```
$ nmap --script ssl-enum-ciphers -p 443 rust-lang.org
443/tcp open  https
| ssl-enum-ciphers:
|   TLSv1.1:
|     ciphers:
|       TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (ecdh_x25519) - A
|       TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (ecdh_x25519) - A
|       TLS_RSA_WITH_AES_256_CBC_SHA (rsa 2048) - A
|       TLS_RSA_WITH_AES_128_CBC_SHA (rsa 2048) - A
|       TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (ecdh_x25519) - A
|       TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (ecdh_x25519) - A
|       TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (ecdh_x25519) - A
|       TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (ecdh_x25519) - A
|       TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (ecdh_x25519) - A
|       TLS_RSA_WITH_AES_128_GCM_SHA256 (rsa 2048) - A
|       TLS_RSA_WITH_AES_256_GCM_SHA384 (rsa 2048) - A
|       TLS_RSA_WITH_AES_128_CBC_SHA256 (rsa 2048) - A
|     compressors:
|       NULL
|     cipher preference: server
|   TLSv1.2:
|     ciphers:
|       TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (ecdh_x25519) - A
|       TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (ecdh_x25519) - A
|       TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (ecdh_x25519) - A
```

Post-quantum cryptography (PQC)

Fact
Our current cryptographic infrastructure is built on top of RSA and elliptic cryptography. Their security is based on the integer factorization (RSA) and discrete logarithm problem (ECC).

Assumption
A sufficiently large quantum computer can solve integer factorization and the discrete logarithm problem in polynomial time (c.f. Shor's algorithm, Grover's algorithm).

Assumption
No sufficiently large quantum computer exists, but we should protect current communication against future decryption.

Post-quantum cryptography (PQC)

Research questions:

1. Which problems provide security under the QROM model?
2. Which cryptographic primitives do we need?
3. What are algorithmic candidates for cryptographic primitives?
4. Which algorithms can be implemented [securely and efficiently] in software and hardware?

Post-quantum cryptography (PQC)

Basic answers:

problems Speculation. Ask theoretical computer scientists about the presumed difficulty to solve computational problems.

primitives Symmetric cryptographic primitives can be used with doubled key sizes. Asymmetric cryptographic primitives like public key encryption schemes and digital signatures need to be replaced.

candidates Ask cryptographic community for proposals. Ask everyone to break security.

implementation Ask for feedback within a time frame.

Post-quantum cryptography (PQC)

Initiate cryptograpic competition[1] similar to SHA-3, CAESAR, and NISTLWC.

Competition by National Institute for Standards and Technology, USA (NIST)

2016-02	Announcement for “Post-Quantum Cryptography Standardization Effort”
2017-11	Deadline for submissions
2017-12	Round 1 algorithms announced (69)
2019-01	Round 2 algorithms announced (26)
2020-07	Round 3 algorithms announced (7)
2022-01	Schemes to standardize to be announced
2024	Expected end of standardization

[1] “Cryptographic competitions” by Daniel J. Bernstein (2020)

Post-quantum cryptography (PQC)

What are candidates for post-quantum cryptography NOT?

- New algorithms for symmetric algorithms
- Quantum cryptography: Industry won’t be able to deploy quantum co-processors in the next few years.

Common approach:

1. Pick an NP-hard problem, where no advantage is known for quantum computers.
2. Design cryptographic scheme on top of the problem
3. Reiterate over implementations to improve efficiency and security

Cryptographic primitives

Key Encapsulation Mechanism:

1. KeyGen() → (pk, sk)
2. Encapsulate(pk) → (ct, ss)
3. Decapsulate(pk, sk, ct) → (ss)

Digital signatures:

1. KeyGen() → (pk, sk)
2. Sign(sk, msg) → (sig)
3. Verify(sig, msg, pk) → (msg)

PQC Round 3 finalists

https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions

History of Round 3 Updates

Round 3 Finalists: Public-key Encryption and Key-establishment Algorithms

Algorithm	Algorithm Information	Submitters	Comments
Classic McEliece *(merger of Classic McEliece and NTS-KEM*	Zip File (97MB) IP Statements Website	Martin R. Albrecht Daniel J. Bernstein Tung Chou Carlos Cid Jan Gilcher Tanja Lange Varun Maram Ingo von Maurich Rafael Misoczki Ruben Niederhagen Kenneth G. Paterson Edoardo Persichetti Christiane Peters Peter Schwabe Nicolas Sendrier Jakub Szefer Cen Jung Tjhai Martin Tomlinson Wen Wang	Submit Comment View Comments
CRYSTALS-KYBER	Zip File (7MB) Website	Peter Schwabe Roberto Avanzi Joppe Bos Leo Ducas Eike Kiltz	Submit Comment View Comments

PQC Round 3 finalists

Round 3 finalists:

1. Classic McEliece (KEM, code)
2. CRYSTALS-KYBER (KEM, lattice, MLWE)
3. NTRU (KEM, lattice, NTRU)
4. SABER (KEM, lattice, MLWR)
5. CRYSTALS-DILITHIUM (sig, lattice, Fiat-Shamir)
6. FALCON (sig, lattice, NTRU)
7. Rainbow (sig, multi-variate, Oil-Vinegar)

(Alternate candidates neglected)

General categories: lattice-based, code-based, multivariate, hash-based, braid group, supersingular elliptic curve cryptography

Usecase

Security

Theoretical security choice of parameters

Software security timing, caches, memory management, misuse prevention by API design

Hardware security EM emission, power analysis, fault attacks

Security

Theoretical security choice of parameters

Software security timing, caches, memory management, misuse prevention by API design

Hardware security EM emission, power analysis, fault attacks

Constant time

- On our hardware, assembly instructions are run
- Independent of the values, the algorithm should take the same amount of time (i.e. constant time)
- Classic counterexample: Exponentiation by squaring
- Sorry, most code snippets are in C

Assembly in rust

Blog article “Intel’s RDTSC instruction with rust’s RFC-2873 asm! macro” (2021)

```
#![feature(asm)]

#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn has_rdtsc_support() -> bool {
  // Step 1: ask for generic information and print it to stdout
  {
    let ebx: u32;
    let ecx: u32;
    let edx: u32;
```

Assembly in rust

Blog article “Intel’s RDTSC instruction with rust’s RFC-2873 asm! macro” (2021)

```
unsafe {
  asm!(
    "cpuid",
    "mov {bx:e}, ebx",
    // “output operands” following
    bx = lateout(reg) ebx,
    lateout("ecx") ecx,
    lateout("edx") edx,
    // “input operands” following
    in("eax") 0,
    // “clobbers” list
    lateout("eax") _,
    // “options” ⊠ {"pure", "nomem", "nostack"}
    options(nomem, nostack)
  );
}
```

Software security

- branch instructions independent of the secret
- memory access pattern independent of the secret
- runtime independent of the secret (constant time algorithms)

Various countermeasures in physical security:

Masking, shuffling, randomized instruction order, …

Various countermeasures in software security:

Branch independence, indexing independent of secret, prevention of data races, …

Constant time algorithms

Problem: Let a be integer $\in \{0, \ldots, 15\}$. Return 0 if $a = 0$ else 1.

```
static inline uint8_t gf16_is_nonzero(uint8_t a) {
  unsigned a4 = a & 0xf;
  unsigned r = ((unsigned) 0) - a4;
  r >>= 4;
  return r & 1;
}
```

via Rainbow round 3 reference implementation, `gf16.c`

Reminder: two's complement

How does one represent non-negative integer i bitwise?

Value 0 is a sequence of zeros. Value 1 has the least-significant bit (LSB) zero but others set to one (i.e. …0001). Value 2 is represented as …0010. And so on …

How does one represent $-i$ for non-negative integer i bitwise?

The most common encoding used on all platforms is the two's complement: If we map value *i* to the negative space, we need to invert all bits and add 1. Example: …0001 inverted, gives …1110 and adding 1 gives …1111. Thus, -1 is a sequence of ones.

argument is non-zero

Description:

- recognize that $a4$ (unlike a) has more than 8 bits.
- if a is zero
  - then $0 - 0$ yields zero for r
  - the fourth bit of r is zero
  - returns 0
- else
  - $a4$ has 4 bits set
  - the two's complement by $0 - a4$ sets the fifth, sixth, $\ldots$ bits to one
  - the fourth bit of r is one
  - returns 1

conditional move

Problem: If $b = 1$, copy len elements from x to r. If $b = 0$, don't do anything.

```
/* b = 1 means mov, b = 0 means don't mov*/
void cmov(unsigned char *r, const unsigned char *x,
          size_t len, unsigned char b)
{
  size_t i;

  b = (~b + 1);
  for(i=0;i<len;i++)
    r[i] ^= b & (x[i] ^ r[i]);
}
```

via Classic McEliece round 3 reference implementation, `int32_sort.c`

Reminder: XOR

- XOR is denoted by the ^ operator
- XOR is a bitwise binary operator and returns 1 iff both bits are different
- `a ^ b` for some integers *a* and *b* is zero iff *a* equals *b*
- `a ^ a` for some integer *a* is always zero

conditional move

Description:

- recognize that b is (two's complement) negated first.
- thus, if b is zero, it retains zero. Otherwise b becomes a sequence of ones bits (-1).
- if b is zero
  - we compute `r[i] = r[i] ^ 0`
- if b is a sequence of ones bits
  - we compute `r[i] = r[i] ^ (x[i] ^ r[i])`
  - this equals `r[i] = (r[i] ^ r[i]) ^ x[i]`
  - this equals `r[i] = 0 ^ x[i] = x[i]`

modulo 3

Problem: Compute $a \bmod 3$ of some integer $a \in \{0, 1, \ldots, 2^{13} - 1\}$.

Blog article “Deriving algorithms for computing modulo constant n” (2021)

Blog article “mod3 of NTRU’s reference implementation” (2021)

```c
static uint16_t mod3(uint16_t a)
{
  uint16_t r;
  int16_t t, c;
  r = (a >> 8) + (a & 0xff); // r mod 255 == a mod 255
  r = (r >> 4) + (r & 0xf); // r' mod 15 == r mod 15
  r = (r >> 2) + (r & 0x3); // r' mod 3 == r mod 3
  r = (r >> 2) + (r & 0x3); // r' mod 3 == r mod 3

  t = r - 3;
  c = t >> 15;
  return (c&r) ^ (~c&t);
}
```

modulo 3

Rough description (blog articles contain details):

- in general, ‘$(a \bmod kp) \bmod p$’ equals ‘$a \bmod p$’ where $a, k, p \in \mathbb{Z}$
- 15 and 255 are multiples of 3
- the first three assignments r use this principle to reduce mod 3, but not completely
- after the first three assignments, $r \in \{0, 1, \ldots, 5\}$ with $r \equiv a \bmod 3$
- so, $t \in \{-3, -2, \ldots, 2\}$
- c is 0 if t is negative and 1 otherwise
- we return r if c is zero, otherwise t

0,1,2 to 0,1,q-1

Problem: Given a polynomial r with coefficients $\in \{0, 1, 2\}$. Map them to $\{0, 1, \text{NTRU\_Q} - 1\}$ assuming the three LSBs of $\text{NTRU\_Q} - 1$ are ones.

```c
/* Map {0, 1, 2} -> {0,1,q-1} in place */
void poly_Z3_to_Zq(poly *r)
{
  int i;
  for(i=0; i<NTRU_N; i++)
    r->coeffs[i] = r->coeffs[i]
                 | ((-(r->coeffs[i]>>1)) & (NTRU_Q-1));
}
```

via NTRU round 3 reference implementation `poly.c`

0,1,2 to 0,1,q-1

Description:

- `r->coeffs[i]>>1` is 0 for values $\{0, 1\}$ and 1 for $\{2\}$
- `-(r->coeffs[i]>>1)` is 0 for values $\{0, 1\}$ and a sequence of ones for $\{2\}$
- if the value is 0 or 1, we apply AND to 0 and NTRU_Q $-$ 1 which is 0
- if the value is 2, we apply AND to -1 and NTRU_Q $-$ 1 which is NTRU_Q $-$ 1

minmax

Problem: Given 31-bit integers a and b. Assign $a = \min(a, b)$ and $b = \max(a, b)$

```
#define int32_MINMAX(a,b) \
do { \
  int32_t ab = b ^ a; \
  int32_t c = b - a; \
  c ^= ab & (c ^ b); \
  c >>= 31; \
  c &= ab; \
  a ^= c; \
  b ^= c; \
} while(0)
```

Description:

- c is positive if $b \geq a$ and negative otherwise
- 32nd bit of c indicates whether we need to swap
- A right-shift operator for signed integers replicates the most-significant bit. So, `0b1000_0000i8 >> 7 == 0b1111_1111`
- so we apply AND between c and `b ^ a`
- If $a \geq b$, `a = a ^ ab = a ^ b ^ a = b`
  else $a < b$, `a = a ^ ab = a ^ 0 = a`

This code is the fundamental routine for conditional swaps to implement sorting algorithms. To implement sorting algorithms in constant time, you need algorithms similar to merge sort (c.f. TAOCP, Vol 3, sorting networks).

number of trailing zeros of the non-zero input in

Description: Count the number of zero bits next to the LSB (“trailing zeros”)

```c
static inline int ctz(uint64_t in) {
  int i, b, m = 0, r = 0;

  for (i = 0; i < 64; i++) {
    b = (in >> i) & 1;
    m |= b;
    r += (m^1) & (b^1);
  }

  return r;
}
```

number of trailing zeros of the non-zero input in

Description:

- *b* contains the *i*-th bit of variable in
- *m* is one iff *b* is one or any bit before is one
- *r* is a counter
- Assume the LSB is zero, then *b* is zero, *m* is zero.
  `(0^1) & (0^1) = 1 & 1 = 1.`
- Assume the LSB is one, then *b* is one, *m* is one.
  `(1^1) & (1^1) = 0 & 0 = 0.`
- If a consecutive bit is zero, but a previous bit was one, one element of the AND operation becomes zero and thus zero will be added to *r*.

Audience quiz

Problem statement

```c
static inline unsigned char same_mask(uint16_t x, uint16_t y);
```

Problem: If x equals y, return non-zero. If x does not equal y, return zero.

Goal: write a constant time algorithm.

Solution

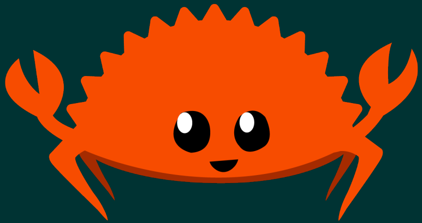
```c
static inline unsigned char same_mask(uint16_t x, uint16_t y)
{
  uint32_t mask;

  mask = x ^ y;
  mask -= 1;
  mask >>= 31;
  mask = -mask;

  return mask & 0xFF;
}
```

via Classic McEliece round 3 reference implementation, `pk_gen.c`

Thank you! Q/A?



Advertisment

Grazer Linuxtage:

- https://www.linuxtage.at/en/
- Fri, 2022-04-22 and Sat, 2022-04-23
- On-site event at TU Graz Inffeld
- Please submit your proposal!