CWE-208: Constant-time Implementation to Lattice-based Post-Quantum Cryptography

Duc Tri Nguyen

Geogre Mason University, Fairfax, VA, 22030

Abstract. Very soon, when a large scale quantum computer is built, the modern public-key cryptography we know today will soon be broken by Shor's algorithm[Sho99]. Follow NIST's standardization call, many post-quantum cryptography software submissions are implemented in a secure way to thwart side-channel attack. In this paper, we discuss the techniques used in Post-Quantum Cryptography implementation to counter CWE-208.

Keywords: Post-Quantum Cryptography, Lattice-based Cryptography, Timing Attack, Cache-timing attack, Constant-time Implementation.

1 CWE-208: Observable Timing Discrepancy

According to CWE-208 description in [cwe]: "Two separate operations in a product require different amounts of time to complete, in a way that is observable to an actor and reveals security-relevant information about the state of the product, such as whether a particular operation was successful or not."

Timing attack in Cryptography is a side-channel attack, in which attacker can predict or estimate either private key or plaintext via timing leakage. A secure cryptography implementation must have constant time execution, thus, this is true in mature cryptography implementation, which has been around for a long period of time, e.g TLS, openSSL, PyCrypto, ...

In the area of Post-Quantum Cryptography, many of the schemes are new, many implementations have not been analyzed thoroughly, due to the threat of Quantum Computer, modern cryptography needs to be replaced in the very near future. To emerge secure cryptography implementation, timing attacks must be eliminated. Timing attacks are often overlooked in the design phase because they are so dependent on the implementation and can be introduced inadvertently with compiler optimization. Therefore, constant-time cryptography implementation is required.

A constant-time implementation that do not use conditional jumps on secret data (branch-free)[Koc96] no-local memory access (cache-free)[Ber05], depend on CPU architecture, the micro-architecture must run low-level instructions in constant-time as well.

2 Post-Quantum Cryptography

A key encapsulation mechanism (KEM) is a scheme including public and private keys, where the public key is used to create a ciphertext (encapsulation) containing a randomly chosen symmetric key. The private key is used to decapsulate the ciphertext. This allows two parties to share a secret key. Traditional KEMs such as RSA rely on the difficulty of factoring large integer numbers. This problem is widely regarded to be infeasible for large numbers with classical computers. The factoring problem can be solved in polynomial time with quantum computers. It is, however, not yet clear, whether quantum computers with enough computation power to break current cryptographic schemes may ever be built. However, the sole risk that such a machine may eventually be built justifies the effort in finding alternatives to today's cryptography.

In 2017, the National Institute of Standards and Technology (NIST) started a standardization process for post-quantum algorithms, i.e. cryptographic algorithms able to withstand attacks that would benefit from the processing power of quantum computers. Proposed algorithms in this process include digital signature schemes, key exchange mechanisms, and asymmetric encryption. In 2019, 26 of the primary 69 candidates from the first round were selected to move to the second round. In 2020, NIST proposes finalist candidates, only 7 candidates are selected as finalists, among them, there are 5 lattice-based cryptographic submissions.

In this paper, we only focus on the software implementation aspects, since the security of lattice-based submissions is evaluated each round by experts, thus, the cryptographic security such as security bits, quantum security bits are considered as out of the scope of this paper.

3 Background

This section provides background context and requirements in the implementation of lattice-based cryptography.

3.1 Lattice-based Cryptography

Without going into mathematical details of lattice-based Cryptography, in the Post-Quantum Cryptography (PQC) 3rd round, there are 3 current constructions of lattice-based cryptography:

- *Module Learning With Errors* (M-LWE): Vector and Matrix construction style, adding random small errors to hide their exact value
- *Ring Learning With Errors* (R-LWE): Vector and Vector construction style, adding random small errors to hide their exact value
- *Module Learning With Rounding* (M-LWR): Vector and Matrix construction style, instead of adding errors, only output the most significant bits, create deterministic encryption.

3.2 Theoretical Implementation

Rejection Sampling To generate random numbers for a set in constant-time, the sampling step requires rejection. If a random number is in the desired range, then it is *accepted*, otherwise, it is *rejected*. This lead to a problem in constant-time implementation, if too many numbers is *rejected*, the time it takes for sampling *increase*. If each sample acceptation or rejection is time-varies, then adversaries can guess the randomness used is accepted or rejected branch by profiling CPU cycles count either *accept* or *reject* branch.

Hashing In many PQC submissions, the selected hashing algorithm is SHA3 and SHAKE128/256 [Dwo15] (Extendable Output Function in SHA3-family), to decrypt the ciphertext correctly (there is a slight chance that decryption error may happen), the receiver has to encrypt the plaintext again to compare with ciphertext. If it is a match, then the decryption is a success, otherwise, it is a failure. Why is hashing matter here? To compare ciphertext and encrypted plaintext securely, we hash data to avoids cache-timing attack to plain data and then do constant-time comparison.

Encoding/Decoding In this phase, if the number of errors is less than γ, then the noise is removed, the original message is recovered, otherwise it is a failure, this lead to a ciphertext can be modified to fail decoding routine, if the decoding routine is not constant-time, attackers can gradually recover plaintext by byte-by-byte fashion, using technique explain above, we hash data and do constant-comparison.

4 Challenge in Constant-time Implementation

Section 4. lists challenges in constant-time implementation of lattice-based cryptography, each problem are extracted from the specification of CRYSTAL-Kyber[BDK+18], SABER[DKRV18] and NTRU[HRSS17].

4.1 Effect of Compiler Optimization

Most of the time, compiler optimization can bring tremendous effect on the table in terms of speed, memory, and code size with many trade-offs.

The assembly version of 4 conditional selections can be seen in *godbolt*[1], *gcc* does not optimize the code while *clang* is able to convert 4 conditional selections to the same assembly instructions.

[1] https://godbolt.org/z/za63zf

```
int select(bool b, int x, int y) {
    return b ? x : y;
}

int ifelse(bool b, int x, int y) {
    if (b)          return x;
    else            return y;
}

int mul(bool b, int x, int y) {
    return (x*b) | (1-b)*y;
}

int and_inverse(bool b, int x, int y) {
    return (x & (-b)) | (~(-b)) & y;
}
```

However, ***cmove*** is selected on ***x86, amd64***, ***csel*** on ***arm*** platform, but it is not always work on non-support conditional move platform, for instance: ***avr, risc-v***[2]. It is obvious that the usage of PQC software is not only used in ***x86, amd, arm*** but also future platform like ***risc-v*** and IoT device like ***avr, mips***, etc...

4.2 Effect of Cache

In WeiB et. al[WHS12], multiple C reference implementations of AES are analyzed in a virtual environment, an adversary can observe a portion of ***S-box*** table in CPU cache, then apply correlation attack to recover full private key from million samples. Briefly talk, the setting is similar to PQC case, whether an attacker can access to a trusted environment, he can mount side-channel attack. The result section demonstrates full key recovery in multiple implementations within million of CPU cycles.

Different from symmetric cryptography, PQC does not suffer directly from cache-timing attack since all operations can be calculated on-fly, even if there is a pre-computed table, it is public data.

4.3 Non-constant time comparison

The *strcmp* or *memcmp* function can be used to effectively compare a large chunk of memory, it's simple and fast to use, however, this creates early break out of the loop. The code below illustrates *strcmp* and *memcmp*.

```
// Input: *a, *b
for (int i = 0; i < length; i++) {
  if (a[i] != b[i]) break;
}
// Output: 0 or 1
```

[2] https://godbolt.org/z/qeTh47

4.4 Decryption Failure

In Lattice-based Cryptography, decryption failure happen when an adversary finds an input vector that $\boldsymbol{Dec(Enc(m))} \neq \boldsymbol{m}$, where $\boldsymbol{m}$ is plaintext.

Decryption failure has highly impact on security of lattice-base cryptography scheme, it can be used to recover the private key[HGNP+03][DVV18].

To reduce the decryption failure rate, The Fujisaki–Okamoto transform [FO99] can be used to construct secure Key Encapsulation Mechanism (KEM) from Public Key Encryption (PKE).

For short, Fujisaki–Okamoto transform introduces re-encryption, check if $\boldsymbol{Enc(Dec(c))} \neq \boldsymbol{c}$, if the check is true, then decryption failure happens. The following code demonstrates how a shared secret is assigned depend on the result of the check.

```
1 // Input: check, buffer SHAKE256
2 if (!check)
3     secret = SHAKE256(K || H(c))
4 else
5     secret = SHAKE256(z || H(c))
6 // Output: *secret
```

Line 2 and 4 compute Extendable Output Function (XOF) in SHA-3 family, $\boldsymbol{K}$ and $\boldsymbol{z}$ are ***secret*** buffer is assigned by output of hash function, in which, this is conditional move, explains in the next section.

4.5 Conditional Move

To avoid cache-timing attack, when query valid data $\boldsymbol{a}$ and invalid data $\boldsymbol{b}$, depend on the secret conditional byte, either valid data a or invalid data $\boldsymbol{b}$ will be loaded into L1 cache, attacker can guess the conditional byte by cache-timing data $\boldsymbol{a}$ or $\boldsymbol{b}$.

```
1 // Input: check, *a, *b
2 for (int i = 0; i < length; i++) {
3     if (!check)
4         secret[i] = a[i];
5     else
6         secret[i] = b[i];
7 }
8 // Output *secret
```

4.6 Rejection Sampling

```
1 // Input: *a
2 int i = 0, j = 0;
3 while (j < 256) {
4   d = a[i] | (a[i+1] << 8);
5   if (d < upperbound)
6       secret[j++] = d;
7   i += 2;
8 }
9 // Output: *secret
```

As we can see at line 7, the rejection sampling will process 2 bytes per iteration, and do a conditional check depends on the value of 2 random bytes in input byte stream ***a***. By timing CPU cycles in each iteration,if the value ***d*** is less than ***upperbound***, then ***if*** branch is taken, lead to longer CPU cycles than the ***else*** branch.

4.7 Conditional Subtraction

A quick way to reduce the number to a desired range is to do conditional subtraction, if the value is greater than ***bound***, then subtract the value, otherwise, leave the value as it is.

```
1 // Input: value
2 if (value > bound)
3     value -= bound
4 // Output: value
```

The code below is common pitfalls due to the effect of compiler optimization.

```
1 // Input: value
2 if (value > bound)
3     value -= bound
4 else
5     value = value;
6 // Output: value
```

As shown in *godbolt*[3], the compiler will automatically remove ***line 4*** , due to *-Wself-assign* flag enable in **-O1, -O2, -O3** optimization.

5 Constant-time Implementation

This section address problem listed in Section 4.

5.1 Constant-time Comparison

To replace *strcmp* and *memcmp*, we need constant-time comparison, the code must survive compiler optimization.

[3] https://godbolt.org/z/e3j17h

```
// Input: *a, *b, len
uint8_t compare(uint8_t *a, uint8_t *b, int len) {
  int i;
  uint8_t r = 0;
  for(i=0;i<len;i++)
    r |= a[i] ^ b[i];
  return (-r) >> 7;
}
// Output: 0 or 1
```

As show in *godbolt*[4], output of compare function is either 0 or 1. If array $\boldsymbol{a}$ identical to array $\boldsymbol{b}$, then $\boldsymbol{r=0}$, else $\boldsymbol{r \neq 0}$, sign bit of $\boldsymbol{(-r)}$ is always $\boldsymbol{1}$.

5.2 Constant-time Conditional Move

The code demonstrate below assume ***secret*** buffer is initialized to $\boldsymbol{0}$. ***sel*** is input which is either $\boldsymbol{0}$ or $\boldsymbol{1}$, if ***sel*** is $\boldsymbol{0}$ then ***secret*** is assigned to itself, unchanged, if ***sel*** is $\boldsymbol{1}$, then $\boldsymbol{a}$ is assigned to ***secret***.

```
// Input: *a, sel in {0, 1}
for (i = 0; i < length; i++)
    secret[i] ^= (-sel) & (a[i] ^ secret[i]);
// Output: *secret
```

As shown in godbolt[5], the code above is not optimized by compiler.

5.3 Constant-time Rejection Sampling

To remove timing discrepancy in Rejection Sampling, we must balance ***if-else*** branch, depend on value of $\boldsymbol{d}$, ***sel*** will be either $\boldsymbol{0}$ or $\boldsymbol{1}$. If ***sel*** is $\boldsymbol{1}$, then ***-sel*** is equal to 1111111111111111_2, $\boldsymbol{d}$ is assigned to ***secret*** at index $\boldsymbol{j}$, if ***sel*** is 0, then $\boldsymbol{d}$ is rejected by bitwise ***AND***. Index $\boldsymbol{j}$ will only increase if and only if $\boldsymbol{d}$ is selected.

```
// Input: *a
uint16_t i, j, sel, secret[256] = {0};
i = 0; j = 0;
while (j < 256) {
    d = a[i] | (a[i+1] << 8);
    if (d < upperbound)
        sel = 1;
    else
        sel = 0;
    secret[j] ^= (-sel) & (d ^ secret[j]);
    j += sel;
    i += 2;
}
// Output: *secret
```

[4] https://godbolt.org/z/hTxbvb
[5] https://godbolt.org/z/4xEqxo

As a result, ***if*** and ***else*** branch run in constant-time. At line 4, timing cannot be easily leaked since the byte stream a is the output of SHAKE128 (XOF), the output of a XOF is deterministic but unpredictable base on the security assumption of hash. Given a hash $\boldsymbol{h}$, it is difficult to find $\boldsymbol{m}$ that $\boldsymbol{hash(m) = h}$, called **Pre-image resistance**[GLL+20].

5.4 Constant-time Conditional Subtraction

This time, the constant-time conditional subtraction must outsmart compilers.

```
1 // Input: value, bound
2 int16_t csubq(int16_t value, const int16_t bound) {
3   value -= bound;
4   value += (value >> 15) & bound;
5   return value;
6 }
7 // Output: value
```

The idea can be summarized as follow, calculate subtraction anyway, if ***value***<***bound***, the result is negative, which enable sign bit to ***1***, notice here we use integer type is *int16_t*, shifting by 15 is *arithmetic shift* rather than *logical shift*, under the hood, *arithmetic shift* use ***sar*** assembly instruction, while ***shr*** is for *logical shift*.

The full assembly code at optimization level ***-O3*** can be seen at *godbolt*[6].

6 Constant-time Implementation of Kyber, NTRU, and Saber

This section compares approaches in Section 5 to current implementation in CRYSTAL-Kyber, SABER, and NTRU. The reference software of 3 lattice-based PQC finalist can be found at [BDK+18], [HRSS17], [DKRV18].

Constant-time Comparison, Conditional Move, and Subtraction , in Section 5.1,5.2,5.4 are used in CRYSTAL-Kyber, SABER and NTRU.

Rejection Sampling The Section 5.3 can be applied to the reference code of CRYSTAL-Kyber[7] to balance ***if-else*** branch. NTRU and SABER specifications do not mention Rejection Sampling.

7 Conclusion

This paper proposes C coding style that **outsmart** compiler to implement secure constant-time lattice-based Post-Quantum Cryptography. In addition, the paper contributes to lattice-based Cryptography CRYSTAL-Kyber somewhat balance ***if-else*** branch.

[6] https://godbolt.org/z/czEs6b
[7] https://github.com/pq-crystals/kyber/blob/master/ref/indcpa.c#L149

References

BDK+18. Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

Ber05. Daniel J Bernstein. Cache-timing attacks on aes. 2005.

cwe. Observable timing discrepancy. https://cwe.mitre.org/data/definitions/208.html. Accessed: 2020-10-20.

DKRV18. Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem. In *International Conference on Cryptology in Africa*, pages 282–305. Springer, 2018.

DVV18. Jan-Pieter D'Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. On the impact of decryption failures on the security of lwe/lwr based schemes. *IACR Cryptol. ePrint Arch.*, 2018:1089, 2018.

Dwo15. Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. Technical report, 2015.

FO99. Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual International Cryptology Conference*, pages 537–554. Springer, 1999.

GLL+20. Jian Guo, Guohong Liao, Guozhen Liu, Meicheng Liu, Kexin Qiao, and Ling Song. Practical collision attacks against round-reduced sha-3. *Journal of Cryptology*, 33(1):228–270, 2020.

HGNP+03. Nick Howgrave-Graham, Phong Q Nguyen, David Pointcheval, John Proos, Joseph H Silverman, Ari Singer, and William Whyte. The impact of decryption failures on the security of ntru encryption. In *Annual International Cryptology Conference*, pages 226–246. Springer, 2003.

HRSS17. Andreas Hülsing, Joost Rijneveld, John M Schanck, and Peter Schwabe. Ntru-hrss-kem. *NIST submissions*, 2017.

Koc96. Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

Sho99. Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

WHS12. Michael Weiß, Benedikt Heinz, and Frederic Stumpf. A cache timing attack on aes in virtualization environments. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security*, pages 314–328, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.