Cache-timing attacks on cryptographic libraries

Sylvain Guilley, CTO.
Presentation Outline

Introduction

Post-Quantum Cryptography Algorithms

Cache-Timing Attacks

Vulnerabilities of Post-Quantum Cryptography Algorithms to Cache-Timing Attacks

Vulnerabilities of a Cryptographic Library to Cache-Timing Attacks

Countermeasures
Presentation Outline

Introduction

Post-Quantum Cryptography Algorithms

Cache-Timing Attacks

Vulnerabilities of Post-Quantum Cryptography Algorithms to Cache-Timing Attacks

Vulnerabilities of a Cryptographic Library to Cache-Timing Attacks

Countermeasures
WHAT DO WE DO?

SECURITY TECHNOLOGIES

FOR EMBEDDED SYSTEMS

CHIPSET/DEVICE VENDORS

IC DESIGN HOUSES

CERTIFICATION LABS

GOVERNMENTAL AGENCIES

FOR WHOM?

FOR WHICH MARKETS?

OUR VISION

Going forward, there will be more and more interconnected devices or objects in various market verticals, this is what we call Internet of Things or Internet of Everything. All those objects being interconnected to the cloud, each and every object could be a threat for the whole network. Therefore, the security of the objects or the devices is key. Even more, security will become one of the most important assets of the digital world.
CORPORATE PRESENTATION

YOUR SECURITY PARTNER

A SOLID BACKGROUND
MORE THAN 15 YEARS OF RESEARCH
MORE THAN 200 PUBLICATIONS

A STRONG RESEARCH FORCE
RESEARCH CENTRES IN FRANCE AND SINGAPORE
SPIN-OFF FROM TELECOM PARISTECH

A THOUGHT LEADERSHIP POSITION IN THE SECURITY WORLD
MEMBERSHIP OF ISO COMMITTEES
3 WORLD-RENOWNED SCIENTIFIC ADVISORS
WORLDWIDE PRESENCE

A COMPLETE PORTFOLIO OF SOLUTIONS
A FULL SET OF ANALYSIS PLATFORMS
A COMPREHENSIVE SET OF SECURITY TECHNOLOGIES
CORPORATE PRESENTATION

BUSINESS LINES

PROTECT

THREAT PROTECTION
Business Line

SECURYZR

COMBINATION OF SMART UNITS AND EXPERTISE RESULTS

EVALUATE

THREAT ANALYSIS
Business Line

LABORYZR

READY-TO-USE PRE AND POST-SILICON ANALYSIS PLATFORMS

SERVICE & CERTIFY

THINK AHEAD
Business Line

EXPERTYZR

THE NEXT STEPS TOWARDS SECURITY CHALLENGES
Presentation Outline

Introduction

Post-Quantum Cryptography Algorithms

Cache-Timing Attacks

Vulnerabilities of Post-Quantum Cryptography Algorithms to Cache-Timing Attacks

Vulnerabilities of a Cryptographic Library to Cache-Timing Attacks

Countermeasures
NIST PQC Contest  
(replacement of RSA, ECC, DH)

1. Several proposals: need to evaluate them (security, performance, etc.)
2. Complete submissions published in December 2017: 69 submissions revealed
3. July 2018: 5 submissions retracted, about a dozen need parameter changes
4. Merges... eventually, Feb 2019: round 2
   - Digital Signature Algorithms (9 remaining): CRYSTALS-DILITHIUM, FALCON, GeMSS, LUOV, MQDSS, Picnic, qTESLA, Rainbow, SPHINCS+.

http://csrc.nist.gov/groups/ST/post-quantum-crypto/
Presentation Outline

Introduction

Post-Quantum Cryptography Algorithms

Cache-Timing Attacks

Vulnerabilities of Post-Quantum Cryptography Algorithms to Cache-Timing Attacks

Vulnerabilities of a Cryptographic Library to Cache-Timing Attacks

Countermeasures
Cache Architecture

Figure: High-level overview of cache architecture (here: modern quadcore Intel processor with Hyperthreading)
Constant-time operation — detailed investigation

Algorithm 1: RSA implementation (sq. & mult. always) protected against SCA and FIA [1]

Input : $M \in \mathbb{Z}_N$, $d = (d_{n-1}, \cdots, d_0)_2$
Output: $M^d \in \mathbb{Z}_N$ or “Error”

1. $R[0] \leftarrow r$ (r is a uniform random $\in \mathbb{Z}_N^*$)
2. $R[1] \leftarrow r^{-1}$
3. $R[2] \leftarrow M$
4. for $i \in \{0, \ldots, n-1\}$ do
5.   $R[d_i] \leftarrow R[d_i] \cdot R[2]$
7. if $R[0] \cdot R[1] \cdot M = R[2]$ then
8.   return $r \cdot R[1]$
9. else
10. return “Error”

Algorithm 2: RSA implementation (Montgomery ladder) [2]

Input : $M \in \mathbb{Z}_N$, $d = (d_{n-1}, \cdots, d_0)_2$
Output: $M^d \in \mathbb{Z}_N$

1. $R[1] \leftarrow 1$
2. $R[2] \leftarrow M$
3. for $i \in \{0, \ldots, n-1\}$ do
4.   if $d_i = 1$ then
7.   else
10. return $R[1]$

Not constant time due to (cached or predicted) table accesses.
Constant-time operation — detailed investigation

Algorithm 1: RSA implementation (sq. & mult. always) protected against SCA and FIA [1]

Input : \( M \in \mathbb{Z}_N, d = (d_{n-1}, \ldots, d_0) \)

Output: \( M^d \in \mathbb{Z}_N \) or “Error”

1. \( R[0] \leftarrow r \) (\( r \) is a uniform random \( \in \mathbb{Z}_N^* \))
2. \( R[1] \leftarrow r^{-1} \)
3. \( R[2] \leftarrow M \)
4. for \( i \in \{0, \ldots, n-1\} \) do
   5. \( R[d_i] \leftarrow R[d_i] \cdot R[2] \)
5. if \( R[0] \cdot R[1] \cdot M = R[2] \) then
   6. return \( r \cdot R[1] \)
5. else
   7. return “Error”

Algorithm 2: RSA implementation (Montgomery ladder) [2]

Input : \( M \in \mathbb{Z}_N, d = (d_{n-1}, \ldots, d_0) \)

Output: \( M^d \in \mathbb{Z}_N \)

1. \( R[1] \leftarrow 1 \)
2. \( R[2] \leftarrow M \)
3. for \( i \in \{0, \ldots, n-1\} \) do
   4. if \( d_i = 1 \) then
   7. else
   10. return \( R[1] \)

Not constant time due to (cached or predicted) table accesses.
Algorithm 3: Constant-time RSA implementation

Input: $M \in \mathbb{Z}_N$, $d = (d_{n-1}, \ldots, d_0)_2$
Output: $M^d \in \mathbb{Z}_N$

1. $R[1] \leftarrow 1$
2. $R[2] \leftarrow M$
3. for $i \in \{0, \ldots, n-1\}$ do
   4. $\text{mask} \leftarrow -!!(d_i)$
   5. $R[1] \leftarrow R[1] \cdot ((R[2] \land \text{mask}) \lor (1 \land \neg \text{mask}))$
4. return $R[1]$

Notice that $\text{mask} = -!!(d_i)$ is equal to either $0x0000\ldots00$ or $0xFFFF\ldotsFF$

Horizontally aligned, thus subsequently attacked based on values:

Not vertically secure.

See attack [3]: One&Done: A Single-Decryption EM-Based Attack on OpenSSL’s Constant-Time Blinded RSA, by Monjur Alam et al. (same attack on windowed implem.)
### Measurement Techniques — Contention

<table>
<thead>
<tr>
<th>Feature</th>
<th>Attack</th>
<th>PRIME PROBE(^1)</th>
<th>+</th>
<th>EVICT + TIME (^2)</th>
<th>FLUSH RELOAD (^2)</th>
<th>+</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target</strong></td>
<td>L1 / L3(^3)</td>
<td>L1</td>
<td></td>
<td>LLC (L3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| **Behavior**| \(\neg (V \text{ access})\) (A fast) | \(\Rightarrow (V \text{ access})\) (A fast) | \(\Rightarrow (V \text{ access})\) (A fast) | \(\Rightarrow (V \text{ access})\) (A fast) | \(\Rightarrow (V \text{ access})\) (A fast) | \(\Rightarrow (V \text{ access})\) (A fast) | \(\Rightarrow (V \text{ access})\) (A fast) | \(\Rightarrow (V \text{ access})\) (A fast) |}
| **Accuracy**| small           | small             |   | high              |                   |   |
| **False positives** | many         | many              |   | few               |                   |   |


Taxonomy [9]

- EVICT + TIME
- PRIME + PROBE
- FLUSH + RELOAD
- FLUSH + TIME + FLUSH
- FLUSH + GAUSS + RELOAD
- CacheBleed

Best state-of-the-art:

*Timing Channels in Cryptography — A Micro-Architectural Perspective*
Authors: Rebeiro, Chester, Mukhopadhyay, Debdeep, Bhattacharya, Sarani.
Timing attacks

- Branches
- Caches (data, code)

<table>
<thead>
<tr>
<th>Feature</th>
<th>External side-channel</th>
<th>Timing side-channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Possibility to profile</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sampling rate</td>
<td>10 Gsample/s</td>
<td>10 Msample/s</td>
</tr>
<tr>
<td>Diversity</td>
<td>Monovariate</td>
<td>Multivariate*</td>
</tr>
<tr>
<td>Signal-to-noise ratio</td>
<td>$10^{-3}$</td>
<td>10</td>
</tr>
</tbody>
</table>

*: probing of I/D cache of various levels, MMU, branch prediction registers, HPC (Hardware Perf Counters), etc.
Vulnerabilities List

<table>
<thead>
<tr>
<th>How</th>
<th>What</th>
<th>Data</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Write</td>
<td></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Exploit</td>
<td>Sensitive indirections</td>
<td>Conditional jump/call</td>
<td></td>
</tr>
</tbody>
</table>

Note: FLUSH + RELOAD only applicable to **shared** data or code (static arrays, code in shared dynamic libraries, etc.)
user code
example in C language:

```
if(s){} or s?{}
for(i=0;i<s;++i){}
while(s){}
switch(s){}
```

conditional control flow

conditional table lookup

```
y=T[s] or y=*(ptr+s) 
T[s]=y or *(ptr+s)=y
```

sensitive variable s

tainting

tainting

computer architecture

conditional behavior:

Hyper-Thread/SMT branch prediction
speculation
out-of-order
pipeline ports
TLB (in MMU)
HW perf. counter
L1 I$ hit/miss
L2$ hit/miss
LLC$ hit/miss
DRAM banks

L1 D$ hit/miss
L2$ hit/miss
LLC$ hit/miss
DRAM banks

attacks:

Spectre
Meltdown
Ryzenfall
Masterkey
Fallout
Chimera
Portsmash
TLBleed
Cachebleed
Timing attack
Extra-reductions
Prime & Probe
Evict & Time
Flush & Reload
Flush & Flush
Template timing attack
Presentation Outline

Introduction

Post-Quantum Cryptography Algorithms

Cache-Timing Attacks

Vulnerabilities of Post-Quantum Cryptography Algorithms to Cache-Timing Attacks

Vulnerabilities of a Cryptographic Library to Cache-Timing Attacks

Countermeasures
Vulnerability Research
Possible Approaches

- Dynamic analysis: valgrind "abuse" (sensitive = non-initialized variables)
- Static analysis:
  - On assembly code: cacheaudit (only a subset of 32-bit assembler)
  - On intermediary language: ct-verif (safe and sound, but not easy to use)
  - On C source code:
    - tis-ct (but no way to tag variables as sensitive)
    - Catalyzr™ tool [10], based on Alexander Schaub’s Stanalyzr
Potential vulnerabilities if sensitive value:
- Used to index in array (or in general, to compute address in memory to be addressed)
- Used in a branching operation

Thus, vulnerability research needs to:
- Propagate sensitive values,
- Determine if value used in leaking operation
Vulnerability Research Methodology

Catalyzr™ tool

Figure: Static analysis tool principle

1. Input Preparation
   - Tag of Sensitive Variable

2. Dependency Analysis
   - Build of AST
   - Extraction of sensitive paths

3. Vulnerability Research
   - Identification of leakage patterns

4. Vulnerability Analysis
   - Categorization
   - Vulnerability Refining
Vulnerability Research
Methodology

- All submissions must implement a function to generate KAT files (for signature, encryption or encapsulation)
- Tagged as sensitive: randomness used in generating KAT files (thus also secret keys)
- KAT-generating function (for the Reference Implementation) were analyzed

Note: this might lead to false positives (leakage of non-sensitive values possibly reported)
Results
Vulnerable Implementations

Figure: Total number of potential vulnerabilities found for each analyzed round #1 candidate

Note: 52 out of the 69 submissions were analyzed.
Results

Vulnerable Implementations

Out of 52 analyzed candidates:

- Potential vulnerabilities in 42 submissions (80.8%)
  - More than 100 reported vulnerabilities in 17 submissions
  - More than 1000 reported vulnerabilities in 3 submissions

- 4 submissions with easily fixable / probably not exploitable vulnerabilities (EMBLEM, Lima, Giophantus, OKCN-AKCN in the MLWE variant)

- 10 submissions without detected vulnerabilities (Frodo, Rainbow, Hila5, Saber, CRYSTALS-Kyber, LOTUS, NewHope, ntruprime, ThreeBears and Titanium)
Results
Statistics
(raw data—not divided by the number of submissions)

- 54.53% (6524)
- 12.71% (1520)
- 32.76% (3919)
Results

Statistics

(raw data—not divided by the number of submissions)

- Lattice-based (4141)
- Code-based (4592)
- Finite Automata (357)
- Hash-based (37)
- Hypercomplex Numbers (85)
- LPN (666)
- Mersenne (33)
- Multi-variate (287)
- Rand Walk (27)
## Results

Overview Per Type of Vulnerabilities

<table>
<thead>
<tr>
<th>Vulnerability Type</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian sampling</td>
<td>3</td>
</tr>
<tr>
<td>Other sampling</td>
<td>13</td>
</tr>
<tr>
<td>Use of GMP</td>
<td>4</td>
</tr>
<tr>
<td>Insecure GF operations</td>
<td>12</td>
</tr>
<tr>
<td>Other leakage sources (incl. error correction)</td>
<td>31</td>
</tr>
</tbody>
</table>

Table: Breakdown of vulnerabilities per type.

**Note:**
the 10 constant-time submissions of round 1 are:

- 7 (out of 17 of Round 2 remaining candidates) for Encryption and KEM;
- 1 (out of 9 of Round 2 remaining candidates) for Digital Signature;
- 2 (namely: LOTUS and Titanium) have been rejected from the competition.
Presentation Outline

Introduction

Post-Quantum Cryptography Algorithms

Cache-Timing Attacks

Vulnerabilities of Post-Quantum Cryptography Algorithms to Cache-Timing Attacks

Vulnerabilities of a Cryptographic Library to Cache-Timing Attacks

Countermeasures
When, how many times? How often?

Where in the source code?

Where in the assembly code?

Are the leakages exploitable?
Sensitive RSA function, mbedtls-2.14.0/library/rsa.c

/*
 * Exponent blinding supposed to prevent side-channel attacks using multiple
 * traces of measurements to recover the RSA key. The more collisions are there,
 * the more bits of the key can be recovered. See [3].
 *
 * Collecting n collisions with m bit long blinding value requires 2^(m-m/n)
 * observations on average.
 *
 * For example with 28 byte blinding to achieve 2 collisions the adversary has
 * to make 2^112 observations on average.
 *
 * (With the currently (as of 2017 April) known best algorithms breaking 2048
 * bit RSA requires approximately as much time as trying out 2^112 random keys.
 * Thus in this sense with 28 byte blinding the security is not reduced by
 * side-channel attacks like the one in [3])
 *
 * This countermeasure does not help if the key recovery is possible with a
 * single trace. */
#define RSA_EXPONENT_BLINDING 28

/*
 * Do an RSA private key operation
 */
int mbedtls_rsa_private( mbedtls_rsa_context *ctx,
                        int (*f_rng)(void *, unsigned char *, size_t),
                        void *p_rng,
                        const unsigned char *input,
                        unsigned char *output )
MbedTLS leakage in RSA signature function [11]
Most of the leakages are in file bignum.c (output of Catalyzr™ tool)

There are multiple paths to activate a vulnerability
MbedTLS leakage in RSA signature function [11]
Most of the leakages are in file bignum.c (output of Catalyzr™ tool)

There are multiple paths to activate a vulnerability
Symbolic static over-estimated analysis

No undetected leakages, but some false positives

False positives are rare because the assumptions made in static analysis are tight in cryptography. Indeed, intermediate values are randomized. Example of unlikely false positives are:

- sensitive annihilation is not understood, e.g., \( s \oplus s \) is null but still considered sensitive
- dead code can be analyzed if present in obfuscated blocks, e.g.,
  
  ```c
  if(tricky_condition_that_happens_to_be_always_zero) {
    y=f(s)...
  }
  ```
- output of a hash function is sensitive but unexploitable
- conditional code turned into constant-time assembly
- let \( u \) be unsensitive and \( s \) be sensitive, then \( *(s+u) = s[u] = u[s] \) is sensitive, but
  - \( u[s] \) is sensitive (e.g., SubBytes table lookup), and
  - \( s[u] \) is unsensitive (e.g., \( s[7]\gg31 \) is the unsensitive evaluation of the MSB of a 256-bit nonce \( s \) to be used for ECDSA).
Translation of cache-timing vulnerable C operations into assembly

<table>
<thead>
<tr>
<th>C construct</th>
<th>Pseudo-assembly construct</th>
<th>Vulnerable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(s){}</td>
<td>cmov s or setcc s</td>
<td>no</td>
</tr>
<tr>
<td>if(s){}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>for(i=0;i&lt;s;++i){}</td>
<td>test s</td>
<td>yes</td>
</tr>
<tr>
<td>while(s){}</td>
<td>jump address</td>
<td></td>
</tr>
<tr>
<td>switch(s){}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y=T[s] or y=* (ptr+s)</td>
<td>load s</td>
<td>yes</td>
</tr>
<tr>
<td>T[s]=y or *(ptr+s)=y</td>
<td>store s</td>
<td>yes</td>
</tr>
</tbody>
</table>
Dynamic profiling with Catalyzr™ of the 160+ vulnerable LoC in bignum.c ................. [12]

- Functions and loops can be identified → useful feedback to developer.
/* 
 * Conditionally assign X = Y, without leaking information 
 * about whether the assignment was made or not. 
 * (Leaking information about the respective sizes of X and Y is ok however.) 
 */

int mbedtls_mpi_safe_cond_assign( mbedtls_mpi *X, const mbedtls_mpi *Y, 
        unsigned char assign )
{
    int ret = 0;
    size_t i;

    /* make sure assign is 0 or 1 in a time-constant manner */
    assign = (assign | (unsigned char)-assign) >> 7;

    MBEDTLS_MPI_CHK( mbedtls_mpi_grow( X, Y->n ) );

    X->s = X->s * ( 1 - assign ) + Y->s * assign;

    for ( i = 0; i < Y->n; i++ )

    for ( ; i < X->n; i++ )
        X->p[i] *= ( 1 - assign );

    cleanup:
        return( ret );
}
Signed addition: \( X = A + B \)

```c
int mbedtls_mpi_add_mpi( mbedtls_mpi *X, const mbedtls_mpi *A, const mbedtls_mpi *B )
{
    int ret, s = A->s;

    if( A->s * B->s < 0 )
    {
        if( mbedtls_mpi_cmp_abs( A, B ) >= 0 )
        {
            MBEDTLS_MPI_CHK( mbedtls_mpi_sub_abs( X, A, B ) );
            X->s = s;
        }
        else
        {
            MBEDTLS_MPI_CHK( mbedtls_mpi_sub_abs( X, B, A ) );
            X->s = -s;
        }
    }
    else
    {
        MBEDTLS_MPI_CHK( mbedtls_mpi_add_abs( X, A, B ) );
        X->s = s;
    }

    cleanup:

    return( ret );
}
```
/*
 * Montgomery multiplication: A = A * B * R^-1 mod N (HAC 14.36)
 */
static int mpi_montmul ( mbedtls_mpi *A, const mbedtls_mpi *B, const mbedtls_mpi *N,
                        mbedtls_mpi_uint mm, const mbedtls_mpi *T )
{
    // [...]

    if( mbedtls_mpi_cmp_abs( A, N ) >= 0 )
    {
        mpi_sub_hlp( n, N->p, A->p );
    }
    else
    {
        /* prevent timing attacks */
        mpi_sub_hlp( n, A->p, T->p );
    }

    return( 0 );
}
Presentation Outline

Introduction

Post-Quantum Cryptography Algorithms

Cache-Timing Attacks

Vulnerabilities of Post-Quantum Cryptography Algorithms to Cache-Timing Attacks

Vulnerabilities of a Cryptographic Library to Cache-Timing Attacks

Countermeasures
Issues with multiplication in $GF(2^N)$ extensions [13]

Table 1: PQC submissions at NIST using vulnerable $GF(2^N)$ operations

<table>
<thead>
<tr>
<th>Submission</th>
<th>Type</th>
<th>Finite Group</th>
<th>Tower fields used</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIG QUAKE</td>
<td>Code-based</td>
<td>$GF(2^N), N = 12, 18$</td>
<td>No</td>
</tr>
<tr>
<td>DAGS</td>
<td>Code-based</td>
<td>$GF((2^5)^2), GF((2^6)^2)$</td>
<td>Yes</td>
</tr>
<tr>
<td>EdonK</td>
<td>Code-based</td>
<td>$GF(2^N), N = 128, 192$</td>
<td>No</td>
</tr>
<tr>
<td>Ramstake</td>
<td>Code-based</td>
<td>$GF(2^8)$</td>
<td>No</td>
</tr>
<tr>
<td>RLCE</td>
<td>Code-based</td>
<td>$GF(2^N), N = 10, 11$</td>
<td>No</td>
</tr>
<tr>
<td>LAC</td>
<td>Lattice-based</td>
<td>$GF(2^N), N = 9, 10$</td>
<td>No</td>
</tr>
<tr>
<td>DME</td>
<td>Multivariate</td>
<td>$GF(2^N), N = 24, 48$</td>
<td>No</td>
</tr>
<tr>
<td>HIMQ-3</td>
<td>Multivariate</td>
<td>$GF(2^8)$</td>
<td>No</td>
</tr>
<tr>
<td>LUOV</td>
<td>Multivariate</td>
<td>$GF(2^8), GF((2^{16})^l), l = 3, 4, 5$</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Vulnerable $a \cdot b$ in C: $a ? \text{ antilog}[\log[a]+\log[b]] : 0$. 
Issues with multiplication in $GF(2^N)$ extensions [13]

Table 3: Performances (in clock cycles) of several multiplication algorithms

<table>
<thead>
<tr>
<th>Multiplication algorithm</th>
<th>Alg.</th>
<th>$GF(2^6)$</th>
<th>$GF(2^{12})$ (*)</th>
<th>$GF((2^6)^2)$</th>
<th>Constant-time?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tabulated log/antilog (**)</td>
<td>3, 4</td>
<td>8</td>
<td>11</td>
<td>20</td>
<td>No</td>
</tr>
<tr>
<td>Iterative, conditional reduction</td>
<td>5</td>
<td>27</td>
<td>51</td>
<td>133</td>
<td>No</td>
</tr>
<tr>
<td>Iterative, ASM with PCLMUL, conditional reduction</td>
<td>5</td>
<td>29</td>
<td>41</td>
<td>146</td>
<td>No</td>
</tr>
<tr>
<td>Iterative, unconditional reduction</td>
<td>6</td>
<td>30</td>
<td>58</td>
<td>155</td>
<td>Yes</td>
</tr>
<tr>
<td>Iterative, ASM with PCLMUL, unconditional reduction</td>
<td>6</td>
<td>35</td>
<td>65</td>
<td>225</td>
<td>Yes</td>
</tr>
<tr>
<td>Iterative, unconditional reduction, 1-bit-sliced (***) 64 computations in parallel</td>
<td>7</td>
<td>55/64</td>
<td>335/64</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>Iterative, ASM with PCLMUL, unconditional reduction, bit-sliced (****) 2 computations in parallel</td>
<td>8</td>
<td>55/2</td>
<td>95/2</td>
<td>-</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Example of implementation in bitslice

```c
/* 7.2: 1-bit-sliced multiplication (SIMP code) */

void gf_multsubTab(gf_t *x, gf_t *y, gf_t *z)
{
    uint64_t xbin[6];
    uint64_t ybin[6];
    uint64_t res[6];


    res[0] = (xbin[0] & ybin[0]));
    res[1] = (xbin[1] & ybin[0]));
    res[0] ^= (xbin[5] & ybin[1]));
    res[1] ^= (xbin05 & ybin[1]));
    ...
Conclusions and Perspectives

Conclusions

- We presented a static analysis tool (Catalyzr™) that allows to determine whether the implementation of a cryptographic algorithm is susceptible to cache-timing side-channel leaks.
- NIST post-quantum project candidates: potential leaks in a vast majority of the candidates.
- Analysis of the severity of leakages.

Perspectives

- Formalize the analysis carried out by Catalyzr™ core tool (namely Alexander Schaub’s Stanalyzr)
- Extend the tool to RowHammer [14], [15]
Bibliographical references


Bibliographical references II


Bibliographical references III


THANKS FOR YOUR ATTENTION

CONTACT

EUROPE       sales-EU@secure-ic.com
APAC         sales-APAC@secure-ic.com
JAPAN        sales-JAPAN@secure-ic.com
AMERICAS     sales-US@secure-ic.com