Detecting cache-timing vulnerabilities in post-quantum cryptography algorithms

Adrien Facon†‡, Sylvain Guilley∗†‡, Matthieu Lec‘Hvien†, Alexander Schaub∗ and Youssef Souissi†

* LTCI, Telecom ParisTech
† École normale supérieure
‡ Secure-IC S.A.S.
75013 Paris, France
75005 Paris, France
35510 Cesson-Sévigné, France

Corresponding author: Alexander Schaub (alexander.schaub@telecom-paristech.fr)

Abstract—When implemented on real systems, cryptographic algorithms are vulnerable to attacks observing their execution behavior, such as cache-timing attacks. Designing protected implementations must be done with knowledge and validation tools as early as possible in the development cycle. In this article we propose a methodology to assess the robustness of the candidates for the NIST post-quantum standardization project to cache-timing attacks. To this end we have developed a dedicated vulnerability research tool. It performs a static analysis with tainting propagation of sensitive variables across the source code and detects leakage patterns. We use it to assess the security of the NIST post-quantum cryptography project submissions. Our results show that more than 80% of the analyzed implementations have at least one potential flaw, and three submissions total more than 1000 reported flaws each. Finally, this comprehensive study of the competitors security allows us to identify the most frequent weaknesses amongst candidates and how they might be fixed.

I. INTRODUCTION

The NIST post-quantum call for proposals aims at evaluating new cryptographic algorithms not only for their theoretical security, but also their practical security. In this respect, parameter numerical choices are the main discussed topic. Regarding security at implementation-level, side-channels are considered some of the most efficient ways of breaking even strong cryptographic standards. These attacks exploit unintentional but observable effects of the execution of a program, in order to gain knowledge about its internal state and recover sensitive information it processes (e.g., the secret key). These attacks are applicable both for hardware and software implementations. For hardware implementation, classical attacks include electromagnetic [1] and power analysis [9]. For software implementations such as the NIST competitors, cache attacks are among the most threatening as they are passive (the attacked code is not aware it is under attack), can succeed key extraction with one measure only and target easily modern, high-frequency CPUs.

Outline. This paper is organized as follows: in Section II, we provide a summary of the existing works in cache exploitation attacks, as well as applicable static and dynamic analysis frameworks. In Section III, we describe our vulnerability research methodology and associated tool. In Section IV we provide analysis results on the NIST post-quantum submissions. In Section V we study the most frequent weaknesses and associated programming patterns, and provide possible corrections. Conclusions are given in Section VI.

II. PREVIOUS WORKS

A. Cache-timing attacks

Cache-timing attacks exploit the access time difference between the slow main memory, or RAM, and the much faster processor cache. Monitoring these cache access patterns during the execution of cryptographic code allows an attacker to efficiently recover the values of internal variables. If the code is not correctly protected, the secret key can potentially be recovered.

Measurement techniques. Several techniques to spy the cache access have been discovered, such as PRIME+PROBE [10], EVICT+TIME [15], or recently the more efficient FLUSH+RELOAD [19] attack. All of them provide a way to gain information about the value of variables used to index an array or to determine the test binary outcome of conditional jumps. PRIME+PROBE and EVICT+TIME attempt to recover access patterns in the L1 cache (L1I for conditional control-flow and L1D for conditional table accesses). However, this cache has long lines, meaning that the accuracy of the spied data is limited, and many processes use the L1 cache, hence many false positives. On the contrary, FLUSH+RELOAD targets the last-level cache (LLC), which is both more accurate in data (granularity is that of the machine word) and in location (the LLC is large, hence limited conflicts occur there).

Exploitation of the leakage. These techniques have been used to break popular implementations of RSA, AES, ECDSA [18] and more recently, the post-quantum signature scheme BLISS [4]. This is possible by relating the cache access pattern to the executed code.

For example, consider the implementation of a simple square-and-multiply modular exponentiation algorithm given in Algorithm 1.

The code has to branch in order to decide whether to perform a multiplication, depending on the values of
the successive bits of \( d \). Therefore, probing whether the instructions of that branch are loaded into the instruction cache allows an attacker to learn the value of \( d \). This is a fatal flaw for RSA decryption, as this would allow the attacker to learn the value of the secret key.

**Other exploitable leakages.** However, even codes that do not make use of conditional branches might be vulnerable. Consider Algorithm 2, used for implementing multiplications in finite fields.

Ignoring the fact that this code is not correct if one of the arguments is equal to 0, this code computes the multiplication of \( a \) and \( b \) in a finite field without using conditional expressions. However, the table accesses potentially allow an attacker to recover the values of \( a \) and \( b \), as well as the product \( a \times b \).

### Algorithm 2: Table-based finite-field multiplication in \( GF(q + 1) \)

**input:**
- \( a \): factor \( (a \in GF(q + 1)) \)
- \( b \): factor \( (b \in GF(q + 1)) \)

**output:** \( a \times b \) in \( GF(q+1) \)

// \texttt{gf\_log} and \texttt{gf\_antilog} are arrays containing the logarithm (resp the anti-logarithm) of the elements in \( GF(q+1) \)

1. return \( \texttt{gf\_antilog}[(\texttt{gf\_log}[a] + \texttt{gf\_log}[b]) \mod q] \)

Implementing protections against cache-timing attacks can be difficult, especially on large software codes dispatched in many files, with many indirection levels. A sensitive variable can be spread across several new variables, recombined. All of the new sensitive variable can create new entry points for attacks. It is therefore difficult to determine the critical code sections of a program, as numerous functions may handle sensitive data. This sensitive data needs to be tracked across all the variable assignments and function calls, to ensure that no conditional jump or array access depends on it. This highlights the need for automatic tools to check exhaustively the code.

### III. FOLLOWED METHODOLOGY

#### A. Leakage types

We distinguish three types of leakages:

1. Secret-dependent conditional instructions: This includes conditional branching as well as loops with a variable number of iterations. In both cases, some instructions will be executed (or not) depending on the sensitive variable value, allowing attacks based on timing and instruction counting, as well as potential cache-attacks on the instruction cache;

2. Secret used to index in an array: this includes accesses to memory addresses depending on the sensitive variable value. This allows attacks based on tracking the addresses values;

### B. Existing tools

Our goal is to identify the instructions and code lines which can produce cache-timing leakages when executed on a processor. We want to capture as many vulnerabilities as possible, with a low dependence to the CPU and with a detailed feedback to the designer.

Cache-timing attacks apply to the binary code. However, most of the time, the vulnerabilities are already present in the source code, and it is very generic to catch the leakage from there.

Dynamic Analysis tools such as the Valgrind debugger framework\(^1\) recently used for Side-Channel attacks [3], were not satisfying for our leakage research. Indeed, this type of analysis has a limited code coverage, as some portions of the code may not be part of control flow graph depending on the parameters values. This is especially true in the type of code targeted. We also wanted to have all the dependency information and a direct mapping with the code lines, which can be challenging after compilation and optimization.

Other static analysis tools do exist. Almedia et al \([2]\) provide such a tool as well as a survey of other existing alternatives. Their approach is sound and complete and serves well to prove that a given implementation runs in constant time. However, it does not seem to provide useful information for non-constant time implementations, and thus is probably not suitable for pinning down leakages that might be exploited for attacks. Furthermore, it is relatively complex to use. \textsc{tis-ct}\(^2\) is another static analysis tool working at the source code level. However, it lacks some flexibility: filtering of false positives needs to be done after the analysis, and it does not seem to be easily available.

---

\(^1\)\url{http://valgrind.org/}

\(^2\)\url{http://trust-in-soft.com/tis-ct}
3) Potential vulnerabilities in unknown code: this mainly includes calls to function pointers that are passed at runtime. These might be refined by hand if it is clear what function is passed as an argument at runtime.

Our approach covers a wide range of cache-timing leakages sources but may be too cautious. The goal is to assist the designer when implementing security critical libraries by highlighting immediately the potential vulnerabilities. Therefore false positive might be reported. For example the last leakage type makes the assumption that unknown code could be dangerous. Similarly the report might include leaking variables that are not critical for security, or vulnerabilities hardly exploitable by an attacker.

B. Principle of the tool

We have built a static analysis tool to scan C source code and identify vulnerabilities to cache-timing attacks. The tool produces reporting which allows determining which functions contain leaks, what sensitive variables are leaked, the dependency chain between variables, etc.

The workflow is illustrated in Figure 1. It works in 4 steps:

1) Input Preparation. The only required preparation of the code is to tag the sensitive variable(s). In order to minimize the impact on the code, we choose to use #pragma directive with a STA tag specific to our tool. The main advantage of using pragma directives is that unknown directives are ignored by the compiler, thus the behavior of the C program, augmented with our tags, does not change. It might be compiled, executed, and even shipped with the pragma directives still present. Then, with the keyword secret followed by the variable name, we specify to the tool to consider the variable as sensitive and track its dependencies across the rest of the code. Sensitive data might be the secret key of the cryptographic algorithm, but in many cases also the randomness used during encryption, decryption or signature.

2) Dependency Analysis. The tool builds the Abstract Syntax Tree (AST) of the whole code and performs a dependency analysis of the code, specifically tracking sensitive variables. The dependency analysis is kept in memory, so that it can be output later for verification and for eliminating false positives.

3) Vulnerability Research. At the same time the AST is walked down and a dependency analysis is performed, we study the code portions manipulating sensitive data and detect potential leakage patterns (conditional branching and array accesses). When a variable depending on sensitive values is involved in a leakage pattern, a warning is emitted.

4) Vulnerability Analysis. A post-processing is then applied on the reported leakages to classify them, remove the potential false positive and produce a report readable to the designer.

C. Evaluating NIST contest submissions

The submissions to the NIST contest have mostly similar structures, owing to predefined templates for implementation. In particular, they all need to implement a main function which creates and verifies Known Answer Tests (KATs). This function thus performs either a signature and verification, or an encryption and decryption, with randomly generated key pairs. Executing this function therefore provides a relatively good code coverage. For the tagged variables, we simply choose to taint the randomness sources (which are also required to respect some specific format), as these sources are used to generate the secret keys as well as the randomness used during signature or encryption, which is probably sensitive in most cases. An issue is that the public key would also be considered sensitive. We mitigate this issue by automatically declaring safe any variable called pk. Of course, a more individual analysis would be necessary if one wants to guarantee the constant time property of a given implementation or, on the contrary, build a side-channel attack. However, this first analysis allows us to gain first insights about the existing and probable future difficulties of implementing constant time post-quantum schemes.

IV. RESULTS

A. NIST post-quantum Cryptography project

Integer factorization and the discrete logarithm problem can be efficiently solved by quantum computers with a large enough number of qubits using Shor’s algorithm [17]. Thus, public-key cryptographic algorithms that are meant to be resistant against quantum computers need to rely on different hard problems. The most promising ones seem to be:

1) LWE (Learning with Errors) [16], which gives rise to the family of lattice-based proposals;
2) the decoding problem, used in the code-based family of proposals, initiated by the McEliece cryptosystem [13] and
3) multivariate polynomial systems solving [12], which is the cornerstone of the multivariate cryptography family.

A small number of submissions explore different directions, such as elliptic curve isogenies [7] or hash-based signatures [14], [5]. Almost none of these proposals have been deployed in the wild yet (with maybe one exception, NewHope3). The security of the implementations thus remains largely unchallenged for now.

B. Overview

We present on the figure 4 the results for the 52 submissions we were able to perform static analysis on. Out of these 52 candidates, vulnerabilities were found in 42 of them (80.8%). 17 candidates have more than 100...
vulnerabilities reported and 3 have more than 1000 vulnerabilities reported. Ten submissions (Frodo, Rainbow, Hila5, Saber, CRYSTALS-Kyber, LOTUS, NewHope, ntruprime, ThreeBears and Titanium) were found to be correctly protected. A few submissions were almost perfectly constant time, and replacing, for example, a small number of conditional branches with conditional move operations would render these implementations perfectly constant-time (EMBLEM, Lima, Giophantus, OKCN-AKCN in the MLWE variant). The vast majority of the submissions, however, is not correctly protected against cache attacks, due to recurrent programming oversights in the portions of code handling sensitive data.

Figure 2 provides the repartition of leakages per algorithm class.

V. ANALYSIS OF VULNERABILITIES

The detailed reports provided by the tool allowed us to classify the vulnerabilities into several categories, which seem to occur frequently among the submissions:

1) **Gaussian sampling leaks:** similar to the issues reported with BLISS [4], implementing a side-channel leakage free Gaussian sampler is not trivial to achieve. Some proposals, such as Frodo, manage to avoid this issue by slightly modifying the distribution being sampled and implementing this sampling in a constant-time fashion. If the use of a discrete Gaussian sampler is required, constant time implementations [8] should be used.

2) **Other sampling leaks:** in general, when specific distributions need to be sampled, one has to take care to avoid conditional branches and array accesses that depend on the randomness source being used. There is no general solution to these type of leaks, as every submission implements samplers for potentially different kinds of distributions.

3) **GMP library use:** some submissions use GMP. This library does not seem to implement operations...
in constant time (at least, according to the C code implementing the version we used for analysis, which is the latest version, at the time of writing, of the mini-gmp variant), and thus functions defined by this library should not be used on sensitive data without further inspection of their assembly level implementation. In any case, the portable mini-gmp implementation should probably not be used in implementations desiring to achieve the constant-time property.

4) **Operations in finite fields:** several implementations need to handle operations in finite fields, notably multiplications. For small groups, this is often done via log/anti-log tables, which might open these implementations to cache-attacks. Other ways to implement these operations must be considered, such as bit-slicing [11].

5) **Other:** there were some potential leaks that we were not able to fit into one of the other categories. For instance, some submissions provide their own implementation of AES that use table accesses, or perform data-dependent branching for matrix operations such as Gaussian elimination. Custom implementations of AES will probably be replaced with calls to hardware-optimized instructions (such as AES-NI [6]) on the platforms most commonly targeted by cache-timing attacks.

Code-based schemes are mainly affected by leakages due to finite field operations, as they need to perform multiplications in extensions of GF(2). Additions in these groups are less likely to cause side-channel leakages, as they can be implemented using constant time bit shifts and XOR-operations. However a naïve implementation might be vulnerable, as the conditional XOR, used during reduction, must also be executed in constant time, for example by using a conditional move (cmove) instruction.

On the other hand, vulnerabilities in randomness sampling are mainly encountered among lattice-based proposals. Perhaps surprisingly, vulnerabilities in Gaussian samplers are relatively rare. The publication of attacks against BLISS [4] might have forced some candidates to take special care in implementing the Gaussian sampler (DRS, qTesla, although the implementation is not entirely constant-time), while others avoided discrete Gaussian distributions altogether (CRYSTALS-Dilithium). However, other sampling routines were not sufficiently protected, for example Bernoulli sampling.

### VI. Conclusion

In this paper, we presented a static analysis tool that allows to determine whether the implementation of a cryptographic algorithm is susceptible to cache-timing side channel leaks. When applied to the candidates for the NIST post-quantum project, our tool allows us to find potential leaks in a vast majority of the candidates. The sources of leakage were classified into several categories, as some sources of leakage, such as finite field operations, were common among the submissions. This paper thus gives a first overview of the state of post-quantum cryptography implementations. In the future, we would like to perform an analysis of the optimized implementations as well, and also provide a more precise breakdown of the sources of leakage for every candidate. This would provide useful insight on the changes that are needed in order to obtain constant time implementations, but it also requires to manually filter out false positives.

As shown in the Table I, vulnerable code appears relatively often when implementing finite field operations and sampling specific distributions.

<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>Unsecure GF operations</td>
<td>12</td>
</tr>
<tr>
<td>Other leakage sources</td>
<td>31</td>
</tr>
</tbody>
</table>

Table I

**BREAKDOWN OF VULNERABILITIES PER TYPE.**
VII. ACKNOWLEDGEMENTS

We acknowledge the support of the French Programme d’Investissements d’Avenir under the national project RISQ. Alexander Schaub acknowledges the financial support from the Direction Générale de l’Armement (DGA).

REFERENCES


