cryptography

MDPI

Article

Side-Channel Evaluation Methodology on Software

Sylvain Guilley [1,2,*], Khaled Karray [1], Thomas Perianin [3], Ritu-Ranjan Shrivastwa [1,2,*], Youssef Souissi [1] and Sofiane Takarabt [1,2]

1 Secure-IC S.A.S., Tour Montparnasse, 75015 Paris, France; khaled.karray@secure-ic.com (K.K.); youssef.souissi@secure-ic.com (Y.S.); sofiane.takarabt@secure-ic.com (S.T.)
2 Telecom-ParisTech, 91120 Palaiseau, France
3 Secure-IC K.K., Hirakawa-cho, Chiyoda-ku, Tokyo 102-0093, Japan; thomas.perianin@secure-ic.com
* Correspondence: sylvain.guilley@secure-ic.com or sylvain.guilley@telecom-paristech.fr (S.G.); ritu-ranjan.shrivastwa@secure-ic.com (R.-R.S.)

Received: 2 August 2020; Accepted: 21 September 2020; Published: 25 September 2020

Abstract: Cryptographic implementations need to be robust amidst the widespread use of crypto-libraries and attacks targeting their implementation, such as side-channel attacks (SCA). Many certification schemes, such as Common Criteria and FIPS 140, continue without addressing side-channel flaws. Research works mostly tackle sophisticated attacks with simple use-cases, which is not the reality where end-to-end evaluation is not trivial. In this study we used all due diligence to assess the invulnerability of a given implementation from the shoes of an evaluator. In this work we underline that there are two kinds of SCA: horizontal and vertical. In terms of quotation, measurement and exploitation, horizontal SCA is easier. If traces are constant-time, then vertical attacks become convenient, since there is no need for specific alignment ("value based analysis"). We introduce our new methodology: Vary the key to select sensitive samples, where the values depend upon the key, and subsequently vary the mask to uncover unmasked key-dependent leakage, i.e., the flaws. This can be done in the source code (pre-silicon) for the designer or on the actual traces (post-silicon) for the test-lab. We also propose a methodology for quotations regarding SCA unlike standards that focus on only one aspect (like number of traces) and forgets about other aspects (such as equipment; cf. ISO/IEC 20085-1.

Keywords: side-channel evaluation; cryptographic implementation; cybersecurity; AES; RSA

1. Introduction

Many papers deal with side-channel attacks ([1–4]). However, the digital systems have become so complex that one cannot speak about only one side-channel, but many of them, such as: protocol-level (e.g., error rates in post-quantum cryptography (PQC), code-based), cache-attack, simple power analysis (SPA), differential power analysis (DPA), etc. They are generally addressed individually. For example, a scientific paper will show how to best break one given countermeasure using one attack, in a precise context. However, seldom is there a big picture of evaluating a fully-fledged implementation end-to-end.

Now, from a designer's or an evaluator's standpoint, the goal is to get rid of all the leakages, and/or have full coverage. In practice, leakage detection can appear in two flavors:

1. Formal verification of the absence of leakage. The analysis outcome is binary, and therefore unambiguous to interpret.
2. Trace-driven leakage detection. In this case, the tests feature true/false positive/negative probabilities. Detection metrics shall therefore be analyzed in great detail, as, for instance, dramatized in [5].

There can be two scenarios for verification that can be distinguished as:

- Developer: has at his disposal the entirety of the source code;
- Lab evaluator: can only resort to machine code.

The developer has the two views: correspondence is ensured with a DWARF (Debugging With Attributed Record Formats) file.

It is useful to consider side-channel mitigation under the prism of the SNR (signal-to-noise ratio) metric, defined in [6] (§ 4.3.2, p. 73): either a signal is reduced or noise is added. Both are not exclusive. Supportive technologies are balancing and random masking. The current consensus is that:

- Biases in the control flow can be spotted very easily; hence, control flow (instructions) shall not depend on the secret;
- Whereas data leakage cannot usually be extracted in one go; therefore, random masking is suitable.

One apparent drawback of this approach is that once the control flow has been balanced, the traces are already well aligned for subsequent statistical analysis of the data leakage. However, this issue shall rather be considered as an advantage: from the developer's standpoint the burden of trace alignment is relieved, and therefore the developer can focus on real activity, namely, leakage analysis, in ideal conditions. From the lab evaluator's standpoint, the alignment is indeed an issue, but it is not the core protection, and resynchronization techniques do exist (cross-correlation, dynamic time warping (DTW), etc.) and/or some analyses are invariant in the time offsets (frequency domain analysis, convolutional neural networks or CNN [7], recurrent neural networks with connectionist temporal classification loss [8], etc.). Therefore, in the rest of the paper, we assume a progression of the analysis in two steps: horizontal and then vertical.

1.1. Illustrations

We illustrate the article briefly with symmetric cryptography (AES), and in more detail with asymmetric cryptography (RSA). The RSA implementation is protected with masking, namely, exponent blinding and base blinding (also illustrated in Figure 1).

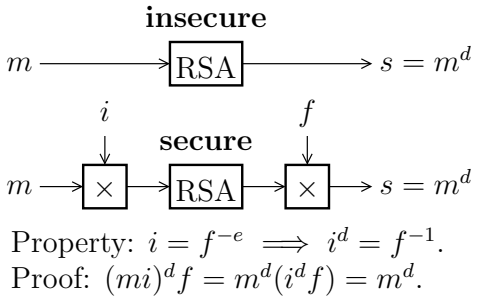

Figure 1. *Top*: Insecure RSA. *Bottom*: Plaintext masking of RSA (note that this countermeasure has been initially proposed against timing attacks [9], at the time where vertical attacks where not known—however it fits the purpose of protecting against such attacks) built on top of an insecure RSA.

1.2. Contributions

As our contributions in this paper, we propose:

- A methodology to analyze leakages based on a partitioning of the studied algorithm.
- Symbolic (in whitebox context) and dynamic (in blackbox context) horizontal leakage detection, and their repair (a topic that is seldom addressed).
- A new strategy for vertical leakage detection (in the case of aligned traces), which does not need any determination of sensitive variables to some constant. Additionally, this strategy leverages a two-step algorithm which first selects the points of interests (that depend on the key), and second, checks whether they are properly masked.

This paper is basically revisiting comprehensive side-channel analysis on software implementations, showing how to detect, diagnose and then repair them in an interactive manner. Typically, we show that the number of fixes to apply to an implementation of `mbedTLS` RSA is such that the final overhead in terms of performances is about +40% clock cycles. Only after this fix is applied, vertical analyses can follow (indeed, vertical analyses assume that traces are aligned). When traces are not aligned, blinding is ineffective. In this respect, the novel method we put forward allows one to detect sensible samples which are unmasked, with an algorithm that is universal, in that it does not require the tester to set input parameters to some arbitrary constant values (which is the state-of-the-art in ISO/IEC 17825).

1.3. Related Works

There have been some works in this direction. In [2], the authors explain how it is possible to automatically fix detected timing and cache-timing vulnerabilities in order to reach a constant time implementation of the code-under-test through a series of transformations that operate on the basic blocks. This approach seems interesting; however, the sensitivity propagation method would inevitably catch false positives, for which fixes will be automatically deployed and will add unnecessary overhead to the code.

In [3], the authors present a tool, which they call SLEAK, whose goal is to automate the analysis against side-channel attack (SCA) vulnerabilities of software implementations. They present a case study on a symmetric algorithm (AES) against vertical attacks. The paper, however, does not address the constant-time feature of the algorithm under test or how to deal with non-constant time implementations (which is challenging, e.g., for the implementations that use shuffling countermeasures). Besides, the presented approach is based on iterations that consider leakages related to each bit of the secret. This may decrease the performance of the evaluation.

In [4], the authors present a "DATA" framework, whose goal is to detect attacks that exploit cache, DRAM and branch predictions. Their approach consists of recording address access patterns in software with different inputs, and performing a differential analysis in order to find dependency on the secret.

1.4. Scope of This Paper

The goal of this paper is to present an extensive methodology to evaluate cryptographic software in front of horizontal and vertical attacks. Those are threats for software that is designed to conceal secrets. Notice that we aim to detect vulnerabilities in such a way that the implementation can be fixed. Therefore, we will be considering an iterative approach whereby the evaluation results allow one to fix the identified vulnerabilities. We are not interested in attacks, but rather in a methodology to either pinpoint issues or to prove that the software is free from flaws.

1.5. Assumptions

Firstly, we assume that the studied code is correct, i.e., that it contains no bugs. For instance, OpenSSL has several CVEs (common vulnerabilities and exposures), including buffer overflows, etc. Even post-quantum cryptography is prone to bugs, such as the underflow in the BIKE decapsulation algorithm (327 CVEs found in https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=openssl).

Secondly, we assume that the secrets to be protected are the inputs to the algorithm, and not the outputs. In some (rare) cases, the secrets are the outputs, which makes the analysis more complex (the tainting especially cannot be achieved). Examples are the key generation algorithms, or even the encapsulation algorithms, which yield a so-called shared secret.

1.6. Methodology

The search for vulnerability unfolds in three steps:

1. Partitioning of inputs into four classes: inputs from the user, algorithm constants, private keys and randomization.
2. Identification of horizontal leakages, which enables their correction.
3. Identification of vertical leakages, which enables their correction.

2. Input Partitioning

The inputs of any cryptographic algorithm can be classified into four categories, depending on whether they are public or private, and whether they are fixed or variable. The taxonomy is provided in Table 1.

Table 1. Partitioning of studied cryptographic algorithm inputs.

Parameter	Public/Private	Fixed/Variable	Comment
p	Public	Variable	Is known to the attacker, and can possibly be chosen
n	Public	Fixed	Ignored in the sequel, because it does not impact the security
k	Private	Variable	Target of the side-channel attacks
m	Private	Fixed	Functional randomness in algorithms or non-functional entropy added to thwart side-channel attacks

Let us notice that the randomness (or **m**asks) can be part of the algorithm's specifications, such as in digital signatures (which shall not yield twice the same signature, even when signing twice the same message). However, the randomness can also be a means to implement the algorithm in an unpredictable way, so that the attacker fails to correlate a (secret dependent) model to internal values. Such randomness is also referred to as random masking or blinding, and is usually considered only a little effective against effective attacks (which are better off protected by balancing the operations). On the contrary, masking is the preferred technique to protect against vertical attacks, where intermediate values shall be protected.

Examples of parameters are provided in Table 2.

Table 2. Examples of some parameters used in different algorithms.

Algorithm	p	n	k	m	Observations
AES	Plaintext, initialization vector	None	Secret key	Random masks	Examples are [10] (masked computation) or [11,12] (masked tables)
RSA no CRT	Base m	Modulus N, public exponent e	Private exponent d	Base blinding f & exponent blinding r	Computation $c = (mf^{-e})^{d+r\phi(n)} f \bmod n$, as per [9]
RSA CRT	Base m	Modulus N, public exponent e	Private exponent d, prime number p	Base blinding f & exponent blinding r	Computation $c = (mf^{-e})^{d+r(p-1)} f \bmod p$, as per [9]

3. Horizontal Leakage

3.1. Description of the Leak

Horizontal leakage consists of temporal variations which can be monitored while the algorithm is running. The observation can be external, e.g., by monitoring the time or even the power profile. Alternatively, it can be internal, by checking whether a line of cache is required by the (victim) cryptographic program, which can be asserted by concurrently trying to access the same line of cache.

The time taken by the attacker process depends upon whether the victim is actually using it or not. Notice that cache-related attacks are preferably executed on a platform with an operating system, since the attacker can deploy, in parallel, one or several attacks to probe the shared cache (at least its timing behavior). However, this situation is not required. Indeed, the attacker can measure externally that the cryptographic code evicts itself (or not) while trying to load far (or close) data/code, relative to the current position.

There are two reasons for horizontal leakage: conditional code and conditional data access (read or write). Control-flow (resp. data-flow) leakage can be prevented by disabling the instruction (resp. data) cache. Indeed, without cache, there is no longer any observable hit/miss pattern in terms of time, nor is there the possibility for an attacker to flush lines of cache to test the time it takes to access any address.

3.2. Identification of the Leak

In a whitebox scenario, leaks are identified by a traversal of the source code abstract syntax tree (AST). The AST vertices are tainted: instructions are termed sensitive if they manipulate a sensitive variable `s`—that is, a variable which depends on any secret **k**.

The tainting algorithm analyzes only a dependence relationship, but can be refined to analyze values. For instance, when a sensitive variable is affecting a constant or a non-sensitive variable, then it is no longer sensitive. Some user-level annotations can also help, for instance, removing the sensitivity from a variable which is the output of a hash function, since there is no way to recover a preimage (computationally speaking). Still, such a sensitive variable equal to a hash value shall remain sensitive if the attack can be perpetrated only knowing the hash value (it is useless for the attacker to know the preimage). Such a situation occurs while analyzing HMAC (hashed message authentication code) functions [13].

Vulnerabilities are merely identified as the encounter of a sensitive variable `s` with:

- A conditional instruction, such as `if`(s), or
- A conditional indirection, such as `tab`$[s]$.

In the context of blackbox analysis, the binary code is exercised under a debugger (GNU Debugger or GDB in our case) under constants **p**, **n** and **m**, but varying **k**. Then, the detection occurs as follows:

- A conditional instruction is revealed by varying the instruction pointer;
- A conditional indirection is revealed by varying the address in an indirect load or store operation.

We refer to this as the GDB methodology. Notice that this methodology is the same as that already employed by practitioners, using `valgrind`. In this methodology, instead of varying **k**, it is left uninitialized. This is possible in C language. By default, compilers might assign a zero value to **k**, but in general the actual initialization is undefined. Such code is not executed in the nominal environment, but is handed over to `valgrind`. The tool will tag specifically uninitialized variables, and will precisely report warning upon:

- A conditional instruction or
- A conditional indirection.

whose condition is uninitialized. Thus, assuming that the only uninitialized variables in the code are **k**, namely, the secrets intentionally not set, the warnings reported by `valgrind` will exactly coincide with those emitted by the proposed GDB methodology.

3.3. Examples on mbedTLS

3.3.1. AES

It is well-known that the vanilla AES exhibits both control-flow and data-flow leakages. Namely:

- The `xtime` function contains an `if`(s) statement on the MSB (most significant bit) of the output of SubBytes. This leak is traditionally fixed by replacing the test with a **Boolean selection**;

- The SubBytes look-up can be resolved by an exhaustive access.

Vulnerable code and repaired code (constant-time code) are demonstrated in Algorithms A1 and A2 (Appendix A).

3.3.2. RSA

Two variants of RSA are shown in Table 2: either with or without CRT (Chinese remainder theorem). We note that with more secret variables, more vulnerabilities are found.

The initial list of vulnerabilities is fairly large, as shown in Figure 2 (using a representation as that already introduced in [14]). The leakage graph in the figure reads as follows. The entry point is the function represented at the top of the tree. Internal sub-function calls are indicated as ovals below it. The annotations on the edges represent the propagation path of the master secret $\mathbf{k}$ to the sensitive variable $\mathtt{s}$ which triggers the non-constant timing issue. The rectangle boxes contain all the lines of code within one function which are vulnerable.

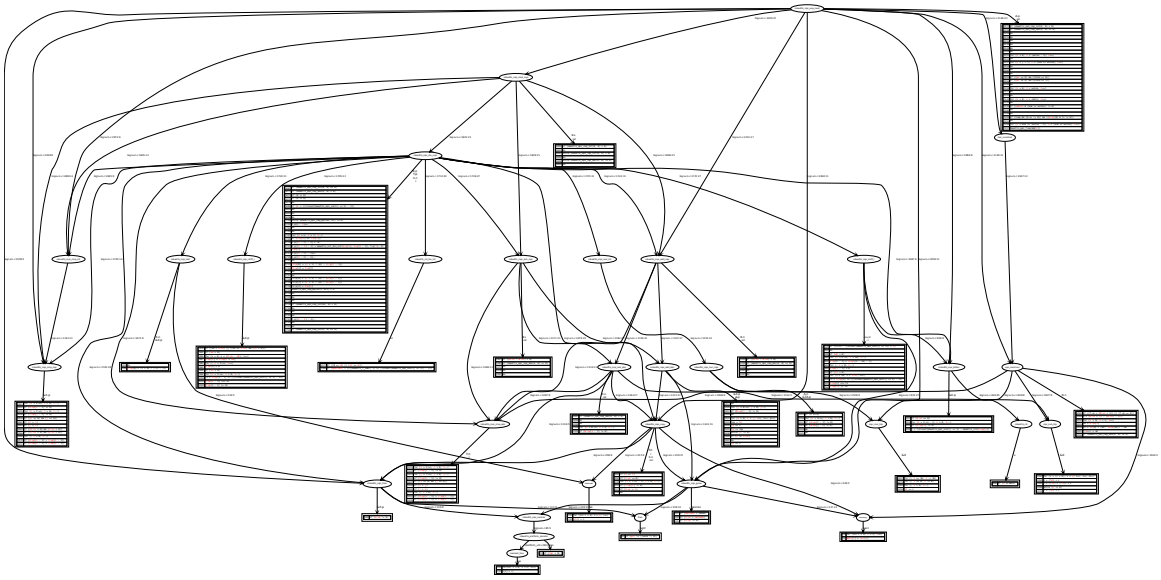

Figure 2. Horizontal leakages of MbedTLS RSA (full – base and exponent).

The static analysis tool allows one to detect lines of code that might leak horizontally which might otherwise be exploited by SPA or cache-timing attacks. Fixing those vulnerabilities results in a constant-time implementation.

In order to illustrate this, we take the example of an RSA signature in which the target security-sensitive function is the modular exponentiation. Tagging the exponent (respectively the base) uncovers potential vulnerabilities that induce non-constant time behavior of modular exponentiation that depends on the exponent (and respectively the base). Non-constant-time vulnerabilities that were revealed by the static analysis tool applied to sliding window modular exponentiation of `mbedTLS` for the exponent are illustrated in Figure 3 and are summarized as follows:

- V1: Conditional branches that depend on the length of the exponent. The exponent length is used in order to compute the width of the window to be used in the computation.
- V2: Conditional branch (while loop) that depends upon the length of the exponent.
- V3: Conditional branches that depend upon the ith bit of the exponent in order to skip the leading zeros of the exponent (leading zeros do not have to be processed). This approach allows one to optimize the execution time of the modular exponentiation by simply skipping all MSB set to zero.
- V4: Conditional branches that depend upon the ith bit of the exponent in order to slide the window and always start the window with a MSB set to one. This approach make available two

optimizations: a first optimization in the execution time of the modular exponentiation, and a second optimization in the precomputation of the windows (since we always start with the MSB set to one, we only need to precompute half of the windows).

- V5: Table access that depends upon the window value.
- V6: Conditional branch in the processing of the remaining bits (out of window bits) in a square and multiply fashion.

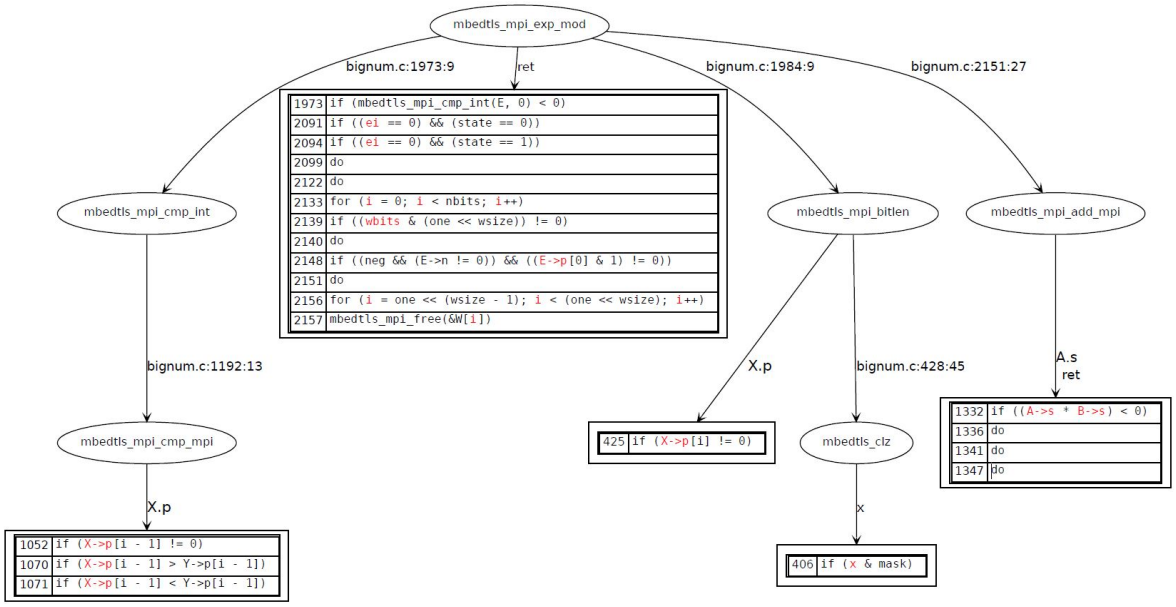

Figure 3. Horizontal leakages of MbedTLS RSA (modular exponentiation only).

In order to make the `mbedTLS` implementation of modular exponentiation we address these vulnerabilities one by one (in the above order).

- Fix for V1 and V2: In order to make the modular exponentiation constant time, the solution would be to fix the length of the exponent.
- fix for V3 and V4: The solution for V3 and V4 is to process the leading zeros and the sliding of the window, and process all windows in the same way. This results in dropping the mentioned optimizations (that makes the implementation non-constant time) in favor of a fixed exponent length and fixed window implementation. The costs of the fix are to spend time processing the windows that does not impact the final result and to precompute all windows (as a window value in this case will not necessarily begin with an MSB equal to one).
- Fix for V5: In order to make the table access indistinguishable from an attacker, the solution would be to access all the elements and keep only the desired ones. This comes at a huge cost, as one has to access all precomputed windows before performing the multiplication.
- Fix for V6: The square and multiply algorithm is used to deal with the remaining bits. From one SPA trace, an attacker can deduce the value of less than *wsize* bits of the exponent. Knowing only *wsize* of the exponent is not critical. However, if the attacker repeats the attack many times, he will collect a set of n equations that gives some information about the secret. The relations are of the form:

$$d + r_i \times \phi(N) = k_i, \qquad i \in \{1, \ldots, n\}$$

where r_i and k_i are the unknown random value (of 224 bits) and the recovered exponent at the step where the remaining bits are processed. To our best knowledge, no algebraic attack has been published in this sense.

Not all those vulnerabilities are called the same amount of times when the code is executed dynamically. A count of all occurrences is represented in Figure 4, obtained by a concrete evaluation under a debugger.

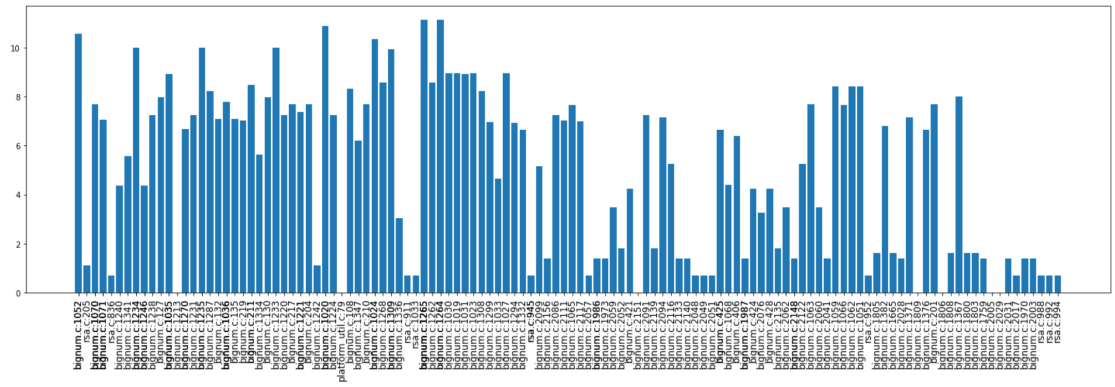

Figure 4. Frequency analysis of vulnerabilities with GDB (logarithmic scale).

An analysis of the vulnerabilities has been conducted and the results are summarized in Table 3. Our selected repair methods are also indicated. They can be divided into two categories:

1. Automatable countermeasures, which apply a stereotyped strategy (here: Boolean selection);
2. Non-automatable countermeasures, which require an algorithmic change (contrast Algorithm A5 to Algorithm A6).

The asterisk (*) in Table 3 indicates that this is not our preferred option. Indeed, if the code still searches for the secret exponent MSB position, then, for subsequent vertical leakage analysis, traces are not aligned. Therefore, we opt to have the exponentiation be fixed/constant time.

Table 3. Vulnerabilities and implemented repairs.

	Vulnerability	Exploitable?	Repair	Automatable?
Exponentiation	Sliding window	✓ [15]	Fixed window	✗
	Look-up in precomputed windows	✓(cache attack)	All lookups + Boolean selection	✗
	Find the secret exponent MSB (Most Significant Bit)	✗	Either force the operations to start at the worst case position (fixed) or live with the leakage (*)	✗
	Value of the LSB (Least Significant Bit)	✗	Either remove the test or live with the leakage (**)	✗
Arithmetic	Extra reductions	✓ [16]	Boolean selection	✓
	Carries	✓ [17]	Remove carries, or protect them by Boolean selection, or process them in constant-time using assembly language instructions	✓

(*) and (**) indicate non-preferred options.

The asterisks pair (**) in Table 3 suggests that the test is removed. Indeed, it serves the purpose of the determination of the sign of the result, in the case where the input is negative. Now, in RSA, all the computations can be carried out on positive numbers; hence, the elimination of the test is harmless.

Instead, keeping the test would also have been fine, as our testbenches never call RSA on a negative message (=basis).

The vulnerabilities listed in Table 3 are classified as pertaining to "exponentiation" or "arithmetic". Big number computation is indeed structured as a stack, where exponentiation is built on top of some basic arithmetic operations. The leaks occurring in the exponentiation are the most straightforward to flags by the attacker (attacks including SPA, machine-learning, etc.). Attacks at the arithmetic level are more complex, and require a precise analysis of the underlying mathematics. Nonetheless, despite the leakage in the arithmetic code, it is only indirectly linked to the leakage of the secret exponent, of which some exploits are known, such as extra-reduction [18]. Still, it is an open problem to know whether the amount of carries in a multiplication allows one to recover information about the secret exponent.

3.4. Performance

In this section, we study the impacts of the non-constant timing vulnerabilities on the performances. Those are illustrated in Table 4.

Table 4. Impacts of fixing vulnerabilities on the mbedtls modular exponentiation implementation measured with mean clock cycles (10,000 runs).

Implementation	Ref	F_1	$F_{1,4}$	$F_{1,4,3}$	$F_{1,4,3,5}$	$F_{1,4,3,5,6}$
Mean clock cycles	19,258,581	15,658,346	16,472,422	17,101,948	23,338,141	27,088,811
Added mean clock cycles	0	−3,600,235	+814,076	+629,526	+6,236,193	+3,750,670

In Table 4 the impact on the execution time of the modular exponentiation is shown (in mean clock cycles). The impact was measured and compared on six different versions, each one implementing more protections than the others (F_i refers to a version implementing fixes against the vulnerability V_i), except for F_1 (for which fixing the vulnerability V_1 allows one to skip a call to the function "mbedtls_mpi_bitlen" which results in a faster implementation). All subsequent versions of the modular exponentiation were more time consuming. This came as no surprise, as fixing some of the vulnerabilities (e.g., V_5 and V_6) requires useless access to some elements of a table (for which we observe the most impact).

4. Vertical Leakage

4.1. Description of the Leak

Vertical leakages are side-channel attacks which attempt to collect information about the software code internal values. Such side-channels are typically power or electromagnetic traces. However, for the sake of whitebox analysis, they also consist of any execution trace which can be obtained by simulation.

Implicitly, they assume that traces are well aligned, in time, so that statistics about the values can be collected in a consistent manner. This assumption was fulfilled as, in the previous section, we exposed not only the vulnerabilities but also the means to plug them.

4.2. Identification of the Leak

We leverage the following two-step algorithm.

Example on RSA:

On RSA, we got the following results:

- Number of sensitive samples vs. T:
- Leakage—following t-test or even improved t-test, as per [19].

The leakage is denoted as $l_v[t]$, where v is the leaking resource, and where t is the time index ($1 \leq t \leq T$). Some examples of leakage functions are depicted in Figure 5:

- At the lowest possible level, namely, the hardware level, the quantum of information is the bit. They are carried either by a memory element (such as a `register`) or by a logic gate (termed `combi`national resource).
- At the software level, the information is represented by values in registers, which consists of fixed size arrays of bits (e.g., registers are typically named `ax`, `bx`, `cx`, `dx`, etc.).
- At the concrete level, a leakage can only by captured by one probe, and consists of a real-valued signal, typically sampled by an ADC (analog-to-digital converter) such as an oscilloscope.

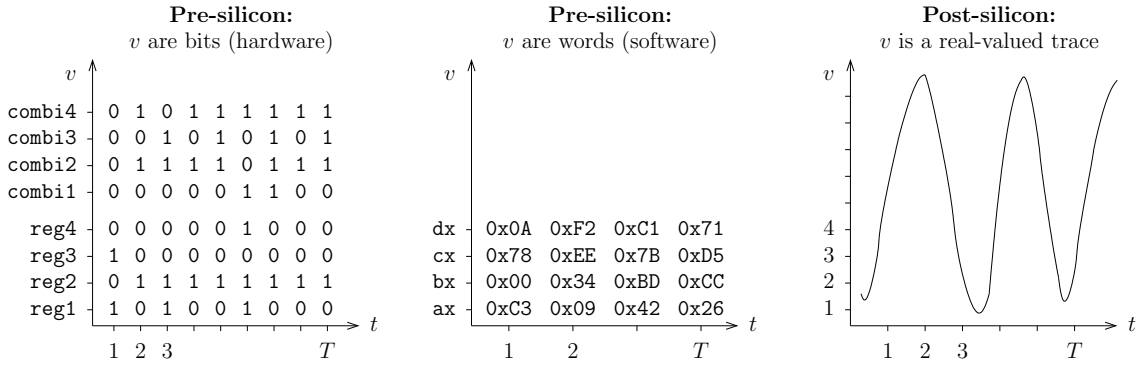

Figure 5. Some typical leakage functions $l_v[t]$, shown as matrices of values.

For the sake of the leakage detection, one needs a correlation function (**Corr**), which correlates bits, words or real values (depending on the three situations represented in Figure 5). Notice that words can be exploded into bits; therefore, the first two analyses (termed pre-silicon) follow the same modus operandi.

Algorithm 1 operates in two steps:

- Lines 4–6: Selection of sensitive samples. The remaining values depend on the key. After this "collapse", the points are all so-called "points-of-interest" (PoIs). As an example, in an AES, the computation and the update of the round counter shall be removed from the PoIs.
- Lines 7–9: The leakage detection can operate globally on the remaining time samples. However, also, it is possible to perform the **Corr** tests for each selected sample t, which will allow for an attribution of the leakage (whose instruction is leaking, and how much).

Notice that those two steps could as well be executed in the reversed order: first, all non-masked samples are listed, and second, this list is narrowed down to samples which in addition depend on the key. On secure designs, this final list is empty. Otherwise, it gathers all leaking samples. Interestingly, the two steps consist of non-interference tests, as coined historically by Goguen and Meseguer [20]. Distinctive features are that this vertical leakage detection methodology allows one to get rid of the "constants" in specific tests (T-Test). In our methodology, Algorithm 1 does not need to select constants, so we evade this question.

The `Corr` function can be a t-test specifically (basic though efficient linear two-class metric). However, the formulation in Algorithm 1 also opens the opportunity to do detection in one go across all key-sensitive samples (belonging to the set SensitiveSamples), for instance, by leveraging machine learning, which is naturally proficient in handling vectorial datasets.

Algorithm 1: Detection of vertical leakage.

input :

- A leakage function l_v for a given value v (either the full trace or each node inside of the algorithm), which belongs to $\{1, \ldots, T\}$ (T time samples)
- q: the number of side-channel traces required to make the decision

output: Raise errors if a variable v is unmasked though it depends on the key

$\mathbf{k}, \mathbf{k}', \mathbf{p}, \mathbf{m}, \mathbf{m}' \leftarrow \mathsf{RandomVector}(q)$;
for *v amongst all leakages* **do** `// Values (e.g., single trace if DPA, or sequence of values for each internal bit if simulation)`
  $\mathsf{SensitiveSamples} \leftarrow \emptyset$;
  for $t \in \{1, \ldots, T\}$ **do** `// Timing samples`
    if $\mathbf{Corr}(l_v(\mathbf{k}, \mathbf{p}, \mathbf{m})[t], l_v(\mathbf{k}', \mathbf{p}, \mathbf{m})[t]) \approx 0$ **then**
      $\mathsf{append}(t, \mathsf{SensitiveSamples})$; `//` v `depends on the key at` t
  for $t \in \mathsf{SensitiveSamples}$ **do** `// Sensitive timing samples`
    if $\mathbf{Corr}(l_v(\mathbf{k}, \mathbf{p}, \mathbf{m})[t], l_v(\mathbf{k}, \mathbf{p}, \mathbf{m}')[t]) \approx 1$ **then**
      Error: unmasked key-dependent leakage v at time $t \implies$ flaw

4.3. Lab Evaluator View

Then in practice, we must do the analysis on real traces. Some questions regarding the ISO/IEC 17825 [21] were raised by [19]. The ISO/IEC 17825 [21], which is currently in the update phase—one of the reasons being that it encountered some questions raised by [19] aimed the security levels 3 and 4 of the standard—provides thorough guidelines to mitigate non-invasive attacks on cryptographic modules. The major concern raised in the challenger article was that the Test Vector Leakage Assessment (TVLA) guidelines provided, in the standard targets, leakage detection in full first-order form as the only required measure for testing against differential side channel attacks on symmetric key cryptosystems. Additionally, the α or the significance level or the false positives threshold in the ISO/IEC 17825 is defined to be 0.05 which is significantly higher than the original value implied in the TVLA design by Goodwill et al. as 0.00001 with t-value threshold being 4.5.

To gain some evidence for the proposed side channel evaluation methodology we performed a simple machine learning experiment based on binary classification on a trace set of SM4 encryption for both leakage and obfuscated implementations on software. The non-leakage dataset has been recorded with a fixed key and fixed plaintext for half of the recordings (which is 500,000) and the other half with fixed key and random plaintext (fix-versus-random). An auxiliary study based on the guidelines provided in [19], to perform correction to the significance criterion α in order to minimize the false positive rate for multiple tests in the TVLA based approach provided in ISO/IEC 17825 wherein the initial value of α was kept at 0.00001, was conducted, validating the proposed methodology. In the machine learning approach we tried to divide the non-leakage dataset into two sets, one with random plaintext and the other with fixed plaintext, and then perform supervised learning to classify the two sets. However, the ML classification approach does not provide sufficient accuracy, even with the full-size dataset because the calculated effect size in this trace set using the TVLA approach with Bonferroni correction is 0.0068, which falls under the "very small" category, as categorized in [22]. However, the t-test statistics give a maximum value of 6.2449 for one iteration over 100 iterations in the multiple Welch's t-test evaluation when Bonferroni correction is used over the significance criterion. Also, the recommendations enacted in that sense [19] take into consideration the attack setup (cf. ISO/IEC 20085-1 [1]).

5. Discussion

The existing CVEs regarding the implementation of `mbedTLS` target mostly the protocol stack and not the cryptographic implementation at the algorithm level. The uncovered vulnerabilities in this paper are not new in the sense that they existed earlier, but they have not been reported until now as CVEs or in any other format, to the best of the knowledge of the authors, and subsequently they have not been formally fixed in the recent releases of `mbedTLS`. Therefore, it gives more relevance to this work to be considered as presenting a new CVE, with a fix, for the algorithm-level implementation of `mbedTLS` RSA.

6. Conclusions and Perspectives

We have demonstrated a comprehensive flow for a cryptographic software evaluation, mostly using whitebox—but also applicable in the context of blackbox scenarios.

Such a method is needed to test for leakages among the numerous PQC algorithms, and doing so on an equal footing. Our approach enables this assessment.

Author Contributions: Conceptualization, S.G.; methodology, R.-R.S.; software, S.T.; validation, T.P.; formal analysis, S.T.; investigation, K.K.; resources, K.K.; data curation, T.P.; writing, R.-R.S.; supervision, Y.S.; project administration, Y.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received external funding from TeamPlay, a Project from European Union's Horizon 2020 research and innovation program, under grant agreement N° 779882.

Acknowledgments: We are thankful to the reviewers of MDPI Cryptography.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

In this appendix, we provide the pseudo-code for both genuine code and our repaired version. The syntax is close to that of C language; for instance, &, |, $\ll$ are used for logical and; logical or; and right-shift.

For AES, we provide in Algorithm A1 (resp. Algorithm A3) the original version from `mbedTLS` AES xtimes used in MixColumns (resp. SubBytes). The repaired algorithms, in terms of constant-timing, are in Algorithms A2 and A4.

Similarly, for RSA, the vanilla implementation of mbedTLS (which was inspired from the Handbook of Applied Cryptography [23]), is in Algorithm A5. The repaired version, from a constant-time perspective, is provided in Algorithm A6. Compared to the RSA countermeasure indicted in Table 2, the secret is the exponent E, and the plaintext is the base p. The blinding of the exponent and of the basis is not shown in Algorithms A5 and A6 to avoid overloading them.

Algorithm A1: MbedTLS AES-xtime code: vulnerable.

input : 8-bits word representing an element of GF(256): a
output: xtime(a): $x \times a$

```
1 r ← a ≪ 1;
2 if MSB(a) then
3     r ← r ⊕ 0x1b ;
4 return r
```

Algorithm A2: MbedTLS AES-xtime code: repaired against timing attack.

input : 8-bits word representing an element of GF(256): a
output: xtime(a): $x \times a$

```
1 r ← a ≪ 1;
2 m ← MSB(a) ;
3 r ← r ⊕ (m × 0x1b) ;
4 return r ;
```

Algorithm A3: MbedTLS AES-Sbox code: vulnerable.

input : 8-bits word representing an element of GF(256): a
output: SubBytes(a)

```
1 r ← Sbox[a] ;                                  // Sensitive table access
2 return r ;
```

Algorithm A4: MbedTLS AES-Sbox code: repaired against cache attack.

input : 8-bits word representing an element of GF(256): a
output: SubBytes(a)

```
1 r ← 0;
2 for i ∈ {0,...,255} do                         // Only for the first AES round
3 |  r ← r|Sbox[i] × (i == a) ;
4 return r ;
```

Algorithm A5: Sliding window implementation of modular exponentiation, for a given public constant parameter "window size" *wsize*.

input : p: base, E: exponent, N: modulus
output: $p^E \bmod N$

1 $W[1] \leftarrow p$, $temp \leftarrow p$;
2 **for** $i \in \{1, \ldots, 2^{wsize-1} - 1\}$ **do**
3 $\quad temp \leftarrow temp^2 \bmod N$;
4 $W[2^{wsize-1}] \leftarrow temp$;
5 **for** $i \in \{2^{wsize-1} + 1, \ldots, 2^{wsize} - 1\}$ **do** `// Pre-computation of the windows`
6 $\quad W[i] \leftarrow W[i-1] \times W[1] \bmod N$;
7 $nblimbs \leftarrow sizeInWord(E)$;
8 $bufsize \leftarrow 0$;
9 $nbits \leftarrow 0$;
10 $wbits \leftarrow 0$;
11 $state \leftarrow 0$;
12 **while** *True* **do** `// Main exponentiation loop`
13 $\quad$ **if** $bufsize == 0$ **then**
14 $\quad\quad$ **if** $nblimbs == 0$ **then**
15 $\quad\quad\quad$ **goto step** 35;
16 $\quad\quad nblimbs \leftarrow nblimbs - 1$;
17 $\quad\quad bufsize \leftarrow sizeInByte(machineWord) \ll 3$;
18 $\quad bufsize \leftarrow bufsize - 1$;
19 $\quad ei \leftarrow LSB(getWord(E, nblimbs) \gg bufsize)$;
20 $\quad$ **if** $ei == 0$ & , $state == 0$ **then** `// Skip leading zeros`
21 $\quad\quad$ **goto step** 12;
22 $\quad$ **if** $ei == 0$ & , $state == 1$ **then** `// Only square operations`
23 $\quad\quad X \leftarrow X \times X \bmod N$;
24 $\quad\quad$ **goto step** 12;
25 $\quad state \leftarrow 2$;
26 $\quad nbits \leftarrow nbits + 1$;
27 $\quad wbits \leftarrow wbits | (ei << (wsize - nbits))$;
28 $\quad$ **if** $nbits == wsize$ **then**
29 $\quad\quad$ **for** $i \in \{0, \ldots, wsize - 1\}$ **do**
30 $\quad\quad\quad X \leftarrow X \times X \bmod N$;
31 $\quad\quad X \leftarrow X \times W[wbits] \bmod N$; `// Table access depending on the secret`
32 $\quad\quad state \leftarrow state - 1$;
33 $\quad\quad nbits \leftarrow 0$;
34 $\quad\quad wbits \leftarrow 0$;
35 **for** $i \in \{0, \ldots, nbits\}$ **do** `// process the remaining bits when less than` *wsize*
36 $\quad X \leftarrow X \times X \bmod N$;
37 $\quad wbits \leftarrow wbits \ll 1$;
38 $\quad$ **if** $(wbits \ \& \ (1 \ll nbits))$ **then** `// Conditional branching`
39 $\quad\quad X \leftarrow X \times W[1] \bmod N$;
40 **return** X;

Algorithm A6: Sliding window implementation of modular exponentiation: repaired.

input : p: base, E: exponent, N: modulus
output: $p^E \bmod N$

1 $W[0] \leftarrow 1, W[1] \leftarrow p$;
2 **for** $i \in \{2, \ldots, 2^{wsize} - 1\}$ **do** `// Pre-computation of the windows`
3 $\quad W[i] \leftarrow W[i-1] \times W[1] \bmod N$;
4 $nblimbs \leftarrow sizeInWord(N) + sizeInWord(Blinding)$; `// Fixed exponent size`
5 $bufsize \leftarrow 0$;
6 $nbits \leftarrow 0$;
7 $wbits \leftarrow 0$;
8 $state \leftarrow 0$;
9 **while** *True* **do** `// Main exponentiation loop`
10 $\quad$ **if** $bufsize == 0$ **then**
11 $\quad\quad$ **if** $nblimbs == 0$ **then**
12 $\quad\quad\quad$ **goto step** 31;
13 $\quad\quad nblimbs \leftarrow nblimbs - 1$;
14 $\quad\quad bufsize \leftarrow sizeInByte(machineWord) \ll 3$;
15 $\quad bufsize \leftarrow bufsize - 1$;
16 $\quad ei \leftarrow LSB(getWord(E, nblimbs) \gg bufsize)$;
17 $\quad state \leftarrow 2$;
18 $\quad nbits \leftarrow nbits + 1$;
19 $\quad wbits \leftarrow wbits | (ei << (wsize - nbits))$;
20 $\quad$ **if** $nbits == wsize$ **then**
21 $\quad\quad$ **for** $i \in \{0, \ldots, wsize - 1\}$ **do**
22 $\quad\quad\quad X \leftarrow X \times X \bmod N$;
23 $\quad\quad w \leftarrow 0$;
24 $\quad\quad$ **for** $i \in \{0, \ldots, 2^{wsize}\}$ **do**
25 $\quad\quad\quad sel \leftarrow i == wbits$;
26 $\quad\quad\quad w \leftarrow w | (W[i] \times sel)$;
27 $\quad\quad X \leftarrow X \times w \bmod N$; `// Instead of W[wbits]`
28 $\quad\quad state \leftarrow state - 1$;
29 $\quad\quad nbits \leftarrow 0$;
30 $\quad\quad wbits \leftarrow 0$;
31 **for** $i \in \{0, \ldots, nbits\}$ **do** `// Process of the remaining bits when fewer than` *wsize*
32 $\quad X \leftarrow X \times X \bmod N$;
33 $\quad wbits \leftarrow wbits \ll 1$;
34 $\quad sel = (wbits \ \& \ (1 \ll nbits))$;
35 $\quad w = (W[1] \times sel) \mid (sel \oplus 1) \times one$; `// One as big-integer`
36 $\quad X \leftarrow X \times w \bmod N$; `// Multiply-always algorithm`
37 **return** X;

References

1. ISO/IEC JTC 1/SC 27/WG 3. ISO/IEC 20085-1:2019 (en). Information technology Security Techniques—Test Tool Requirements and Test Tool Calibration Methods for Use in Testing Non-Invasive Attack Mitigation Techniques in Cryptographic Module —Part 1: Test Tools and Techniques. 2019. Available online: https://www.iso.org/standard/70081.html (accessed on 24 September 2020).
2. Wu, M.; Guo, S.; Schaumont, P.; Wang, C. Eliminating timing side-channel leaks using program repair. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Amsterdam, The Netherlands, 16–21 July 2018; pp. 15–26.
3. Walters, A.D.; Kedaigle, E. SLEAK: A Side-Channel Leakage Evaluator and Analysis Kit. November 2014. Technical paper. Available online: https://www.mitre.org/publications/technical-papers/sleak-a-side-channel-leakage-evaluator-and-analysis-kit (accessed on 24 September 2020)
4. Weiser, S.; Zankl, A.; Spreitzer, R.; Miller, K.; Mangard, S.; Sigl, G. DATA–Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 603–620.
5. Whitnall, C.; Oswald, E. A Cautionary Note Regarding the Usage of Leakage Detection Tests in Security Evaluation. Cryptology ePrint Archive, Report 2019/703. 2019. Available online: https://eprint.iacr.org/2019/703 (accessed on 24 September 2020).
6. Mangard, S.; Oswald, E.; Popp, T. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*; Springer: Berlin/Heidelberg, Germany, 2006; p. 338, ISBN 0-387-30857-1. Available online: http://www.dpabook.org/ (accessed on 24 September 2020).
7. Cagli, E.; Dumas, C.; Prouff, E. Convolutional Neural Networks with Data Augmentation Against Jitter-Based Countermeasures. In *Cryptographic Hardware and Embedded Systems–CHES 2017*; Fischer, W., Homma, N., Eds.; Springer: Cham, Switzerland, 2017; pp. 45–68. [CrossRef]
8. Carré, S.; Dyseryn, V.; Facon, A.; Guilley, S.; Perianin, T. End-to-end automated cache-timing attack driven by Machine Learning. *J. Cryptogr. Eng.* **2020**. [CrossRef]
9. Kocher, P.C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, CA, USA, 18–22 August 1996*; Koblitz, N., Ed.; Springer: Berlin/Heidelberg, Germany, 1996; Volume 1109, pp. 104–113. [CrossRef]
10. Rivain, M.; Prouff, E. Provably Secure Higher-Order Masking of AES. In *Lecture Notes in Computer Science, Proceedings of the Cryptographic Hardware and Embedded Systems, CHES 2010, Santa Barbara, CA, USA, 17–20 August 2010*; Mangard, S., Standaert, F.X., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; Volume 6225, pp. 413–427.
11. Coron, J.S. Higher Order Masking of Look-Up Tables. In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology —EUROCRYPT 2014, Copenhagen, Denmark, 11–15 May 2014*; Nguyen, P.Q., Oswald, E., Eds.; Springer: Berlin/Heidelberg, Germany, 2014; Volume 8441, pp. 441–458.
12. Coron, J.S. HTable Countermeasure Against Side-Channel Attacks Available online: https://github.com/coron/htable (accessed on 24 September 2020).
13. Benoît, O.; Peyrin, T. Side-Channel Analysis of Six SHA-3 Candidates. In *Lecture Notes in Computer Science, Proceedings of the Cryptographic Hardware and Embedded Systems, CHES 2010, Santa Barbara, CA, USA, 17–20 August 2010*; Springer: Berlin/Heidelberg, Germany, 2010; Volume 6225, pp. 140–157.
14. Takarabt, S.; Schaub, A.; Facon, A.; Guilley, S.; Sauvage, L.; Souissi, Y.; Mathieu, Y. Cache-Timing Attacks Still Threaten IoT Devices. In *Lecture Notes in Computer Science, Codes, Cryptology and Information Security—Third International Conference (C2SI 2019), Rabat, Morocco, 22–24 April 2019*; Proceedings—In Honor of Said El Hajji; Carlet, C.; Guilley, S.; Nitaj, A.; Souidi, E.M., Eds.; Springer: Cham, Switzerland, 2019; Volume 11445, pp. 13–30. [CrossRef]
15. Bernstein, D.J.; Breitner, J.; Genkin, D.; Bruinderink, L.G.; Heninger, N.; Lange, T.; van Vredendaal, C.; Yarom, Y. Sliding Right into Disaster: Left-to-Right Sliding Windows Leak. In *Cryptographic Hardware and Embedded Systems–CHES 2017*; Fischer, W., Homma, N., Eds.; Springer: Cham, Switzerland, 2017; pp. 555–576. [CrossRef]

16. Schindler, W. A Timing Attack against RSA with the Chinese Remainder Theorem. In *Lecture Notes in Computer Science, Proceedings of the Cryptographic Hardware and Embedded Systems—CHES 2000, Worcester, MA, USA, 17–18 August 2000*; Koç, Ç.K., Paar, C., Eds.; Springer: Berlin/Heidelberg, Germany, 2000; Volume 1965, pp. 109–124.
17. Dugardin, M.; Papachristodoulou, L.; Najm, Z.; Batina, L.; Danger, J.; Guilley, S. Dismantling Real-World ECC with Horizontal and Vertical Template Attacks. In *Lecture Notes in Computer Science, Proceedings of the Constructive Side-Channel Analysis and Secure Design—7th International Workshop (COSADE 2016), Graz, Austria, 14–15 April 2016;* Revised Selected Papers; Standaert, F., Oswald, E., Eds.; Springer: Cham, Switzerland, 2016; Volume 9689, pp. 88–108. [CrossRef]
18. Arnaud, C.; Fouque, P. Timing Attack against Protected RSA-CRT Implementation Used in PolarSSL. In *Lecture Notes in Computer Science, Proceedings of the Topics in Cryptology-CT-RSA 2013—The Cryptographers' Track at the RSA Conference 2013, San Francisco, CA, USA, 25 February–1 March, 2013;* Dawson, E., Ed.; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7779, pp. 18–33. [CrossRef]
19. Whitnall, C.; Oswald, E. A Critical Analysis of ISO 17825 ('Testing Methods for the Mitigation of Non-invasive Attack Classes Against Cryptographic Modules'). In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology-ASIACRYPT 2019-5th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, 8–12 December 2019;* Proceedings, Part III; Galbraith, S.D., Moriai, S., Eds.; Springer: Cham, Switzerland, 2019; Volume 11923, pp. 256–284. [CrossRef]
20. Goguen, J.A.; Meseguer, J. Security Policies and Security Models. In Proceedings of the 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 26–28 April 1982; pp. 11–20. [CrossRef]
21. ISO/IEC JTC 1/SC 27/WG 3. ISO/IEC 17825:2016: Information Technology—Security Techniques—Testing Methods for the Mitigation of Non-Invasive Attack Classes Against Cryptographic Modules. Available online: https://www.iso.org/standard/60612.html (accessed on 24 September 2020).
22. Sawilowsky, S.S. New effect size rules of thumb. *J. Mod. Appl. Stat. Methods* **2009**, *8*, 26. [CrossRef]
23. Menezes, A.J.; van Oorschot, P.C.; Vanstone, S.A. *Handbook of Applied Cryptography*; CRC Press: Boca Raton, FL, USA, 1996; p. 816. Available online: http://www.cacr.math.uwaterloo.ca/hac/ (accessed on 24 September 2020).