When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA

Alejandro Cabrera Aldaya and Billy Bob Brumley

Tampere University, Tampere, Finland
aldaya@gmail.com,billy.brumley@tuni.fi

Abstract. Microarchitecture based side-channel attacks are common threats nowadays. Intel SGX technology provides a strong isolation from an adversarial OS, however, does not guarantee protection against side-channel attacks. In this paper, we analyze the security of the mbedTLS binary GCD algorithm, an implementation that offers interesting challenges when compared for example with OpenSSL, due to the usage of very tight loops in the former. Using practical experiments we demonstrate the mbedTLS binary GCD implementation is vulnerable to side-channel analysis using the `SGX-Step` framework against mbedTLS based SGX enclaves.
We analyze the security of some use cases of this algorithm in this library, resulting in the discovery of a new vulnerability in the ECDSA code path that allows a single-trace attack against this implementation. This vulnerability is three-fold interesting:

1. It resides in the implementation of a countermeasure which makes it more dangerous due to the false state of security the countermeasure currently offers.
2. It reduces mbedTLS ECDSA security to an integer factorization problem.
3. An unexpected GCD call inside the ECDSA code path compromises the countermeasure.

We also cover an orthogonal use case, this time inside the mbedTLS RSA code path during the computation of a CRT parameter when loading a private key. The attack also exploits the binary GCD implementation threat, showing how a single vulnerable primitive leads to multiple vulnerabilities. We demonstrate both security threats with end-to-end attacks using 1000 trials each, showing in both cases single-trace attacks can be achieved with success rates very close to 100%.

Keywords: side-channel analysis · vulnerable countermeasure · ECDSA · RSA · binary GCD · modular inversion · Intel SGX · mbedTLS · CVE-2019-18222

1 Introduction

Side-channel attacks have gained a lot of traction since the pioneering work on timing side-channels by Kocher [Koc96]. The leakage sources differ in nature: time [Koc96, BB05, BT11], power consumption [KJJ99, Cor99, BCO04], microarchitecture states [Per05, AGS07, Ald+19] are just some examples of them. In the microarchitecture domain, several resources can be used as leakage sources, such as cache-timings [YF14], cache-access patterns [OST06], branch-predictors [AGS07], etc. Each microarchitecture attack vector exploits a resource available in microprocessors.

With the increasing demand of security on modern CPUs, Intel has developed some protection features on its processors. One of the most prominent technologies that has received generous attention in the scientific community is Intel SGX (Software Guard Extensions) [Int19, CD16]. This technology aims at offering confidentiality and integrity to software running on some Intel CPU microarchitectures, even considering a compromised

Licensed under Creative Commons License CC-BY 4.0.

OS, hence, the attacker has every OS-level resource at its disposal to bypass Intel SGX security guarantees.

However, Intel SGX does not offer security for side-channel attacks, thus, leaving protections against them to the application developer [CD16]. On this regard, countermeasures against side-channel analysis have already been deployed in many open-source cryptography libraries. One example of this is the mbedTLS library, with a very good set of countermeasures implemented on its elliptic curve cryptography paths. For instance, the scalar multiplication algorithm is based on a protected proposal in [HPB04], also Jacobian projective coordinates randomization is performed before a scalar multiplication takes place [Cor99]. On a high-level design the mbedTLS ECDSA implementation was, to the best of our knowledge, the first to implement a countermeasure for protecting the modular inversion of ECDSA secret nonces using a multiplicative masking.

Modular inversion algorithms, especially those based on the binary GCD algorithm [Ste67], have been targeted by side-channel analysis. For instance, Acıiçmez, Gueron, and Seifert [AGS07] presented a theoretical attack on this algorithm, proposing a *model* that relates algorithm execution flow with its inputs. On the power consumption realm Aravamuthan and Thumparthy [AT07] independently presented the same model as [AGS07] and also proposed a countermeasure to thwart SPA attacks. More recently, Aldaya, Cabrera Sarmiento, and Sánchez-Solano [ACSS17] presented a different model for analyzing the relation of its execution flow with its inputs, showing that the countermeasure proposed in [AT07] is insecure under the new model. On the other hand, Pereida García and Brumley [PGB17] showed that the ECDSA implementation of OpenSSL was vulnerable to a Flush+Reload attack during the modular inversion of the nonce using a variant of the binary GCD algorithm. Independently Weiser, Spreitzer, and Bodner [WSB18] and Cabrera Aldaya et al. [CA+19] demonstrated vulnerabilities during RSA key generation in OpenSSL, specifically during a modular inversion operation. In these two papers, the same vulnerability was attacked using two different microarchitecture components: page-fault attack against an Intel SGX enclave and Flush+Reload combined with performance degradation respectively.

However, all these previous works on attacking binary GCD based algorithms only recover part of their execution flows. In a nutshell, binary GCD execution flow can be summarized using two variables Z_i and X_i (Section 3 expands on this). In these implementations, the recovery of Z_i was doable, while the recovery of X_i was limited. In [ACSS17, PGB17] the authors attacked ECDSA where only a few bits of the nonce are needed to compromise the cryptosystem, whereas in [WSB18, Ald+17, CA+19], the attacked scenario guarantees that the X_i are known beforehand, hence no need to recover them using side-channels. However, there are more use cases in cryptography applications where binary GCD based algorithms are employed, and compromising them requires recovering all input bits, implying that the recovery of X_i is mandatory using a side-channel.

In this paper we developed a side-channel attack against a binary GCD algorithm where we were able to recover both Z_i and X_i with very high reliability. The targeted implementation is part of the mbedTLS library where we developed two end-to-end attacks against a TLS server secured by Intel SGX. Our experiment results are developed using mbedTLS, however, the side-channel methodology to attack the binary GCD algorithm can be generalized to others. In this regard, mbedTLS offers challenges that were not present in other libraries such as OpenSSL, especially during the recovery of Z_i, which is easier in the latter [PGB17, CA+19].

One of these proposed attacks targets a new vulnerability in the countermeasure already deployed in this library to protect the inversion of ECDSA nonces. The fact that the vulnerability resides in the countermeasure implementation highlights its importance because the countermeasure is offering a false state of security. For instance, very recently a Security Advisory was issued by the mbedTLS security team where it is assumed that such countermeasure actually offers protection, while it does not.

The other attack scenario targets an open problem left in a recent paper, very related to the difficulty of recovering X_i using some side-channel. This time the targeted cryptosystem is RSA during the computation of the CRT parameter $q^{-1} \bmod p$.

The main contributions of this paper are the following:

1. New vulnerability in mbedTLS implementation of the countermeasure to protect the inversion of the nonce in ECDSA.
2. Practical attack on RSA-CRT computation of $q^{-1} \bmod p$.
3. Full binary GCD algorithm execution flow recovery.
4. End-to-end attacks on ECDSA and RSA scenarios with bulk simulation results.

The organization of the paper is the following. Section 2 provides a background on Intel SGX, side-channel analysis and the binary GCD algorithm. Section 3 analyzes the security of mbedTLS binary GCD algorithm implementation and the challenges it imposes. Section 4 describes a new vulnerability in the mbedTLS ECDSA implementation, showing how a poorly implemented countermeasure reduces the security of ECDSA to an integer factorization problem. Later, in Section 5 and Section 6 end-to-end attacks are developed against an mbedTLS server targeting ECDSA and RSA cryptosystems respectively. Section 7 discuses mitigation strategies while the conclusions are presented at Section 8.

2 Background

2.1 Side-Channel Attacks on Intel SGX realm

Intel Software Guard Extension (SGX) technology aims at offering confidentiality and integrity to software implementations for Intel CPUs. It provides strong isolation between a *secure world*, named enclave, and the rest of the system even under the presence of very strong adversaries with OS privileges. However, Intel SGX threat model does not include side-channel attacks, thus offering no security guarantees for these attack vectors [Int19, CD16].

This characteristic highlights the importance of side-channel attack protections on software that handle secret data such as cryptography libraries. At the same time, and arguably more important, it opens the door to new attack techniques that fully employ OS-level resources to gathering side-channel signals and reduce their noise.

Microarchitecture side-channels are often noisy, hence the adversary must compensate to extract relevant data. For example, `CacheZoom` [MIE17] and `CacheQuote` [Dal+18] attacks enhance the resolution of a Prime+Probe cache attack controlling some resources that require OS privileges. For instance, the victim process is isolated to a single CPU thus the cache side-channel is not poisoned by other process accesses, hence reducing noise.

In this regard, Xu, Cui, and Peinado [XCP15] introduced the so-called *controlled-channel attacks*. This attack vector exploits the fact that while SGX enclaves enjoy data/code confidentiality and integrity, SGX enclaves defer resource management to the (untrusted) OS, hence to adversaries. In that paper the authors introduced a *page-fault attack* against shielded systems like Intel SGX. The novel idea is based on tracking the sequence of memory pages accessed by an enclave to recover secret information. As enclave memory management is performed by the OS, an adversarial OS can change a page permission (e.g. the No-eXecute flag) that will trigger a page-fault when the targeted page is going to be executed [Xia+17, XCP15]. Applying this procedure to a set of pages, an attacker can obtain a side-channel trace of the sequence of executed pages.

In a non-SGX environment, page-fault metadata contains the address that generates it, however, for security reasons SGX clears the 12 least significant bits of this address, leaving the page-fault tracing attack a 4 KB granularity [CD16]. From an attacker advantage point of view, this *limited* granularity is compensated by the noiseless nature of the obtained signals, a feature that makes this kind of attack a very powerful side-channel source.

However, research in recent years has tackled this granularity issue. The main idea is to force the preemption of an enclave at a high frequency to collect microarchitectural state information (i.e. side-channels) at each preemption window. This kind of attack is known as an *interrupt-driven attack*, and is often achieved by interrupting the enclave at fixed time intervals controlled by the APIC timer on Intel CPUs [MIE17, HCP17, VBPS17]. While previous works achieve different temporal resolutions, the framework `SGX-Step` proposed by Van Bulck, Piessens, and Strackx [VBPS17] increases it to the maximum, allowing to interrupt an enclave such that instruction single-stepping is possible. Therefore an adversary can capture microarchitecture side-channel signals after every executed instruction by the enclave.

The `SGX-Step` framework allows to perform either page-fault or interrupt-driven attacks. While the former is free of noise, the latter can have some noise, but as we show during our experiments in Section 5 and Section 6 it could be handled such that its impact on attack success rate is negligible. While `SGX-Step` has proved useful for carrying out some recent attacks [VB+18, Can+19, Sch+19, Che+19, Isl+19], its application to attacking cryptography algorithm implementations has not been extensively analyzed, in particular the interrupt-driven attack feature. In this regard, in [WSB18] the page-fault feature of `SGX-Step` has been employed to recover an RSA private key during its generation. However, the interrupt-driven attack was not evaluated, thus raising an open question how this feature will perform on attacking cryptography algorithm implementations and how threatening it is. To the best of our knowledge, this paper is the first to address this question, evaluating both page-fault and interrupt-driven attacks on cryptography algorithm implementations using the `SGX-Step` framework.

2.2 Binary GCD algorithm and side-channel analysis

Different models have been proposed to relate the knowledge about the execution flow of binary GCD based algorithms with their input bits [AGS07, ACSS17, PGB17]. Table 1 summarizes them in terms of required knowledge and the amount of bits that can be recovered.

The *All-or-nothing* model, proposed in [AGS07] allows to recover *all* bits of both inputs, but it requires to know the results of *all* conditional branches, hence its name. This represents an issue when the side-channel leakage source contains noise on the results of these condition operations, jeopardizing the attack success rate. At the same time, the attacker also needs to know reliably *both*, algorithm start and its end, adding an extra challenge regarding locating with certainty these moments in a long trace. On the other hand, it does not require knowing one input at all, so, both inputs could be secret, and they can be recovered once the previous conditions are fulfilled.

The *Partial* model was proposed in [ACSS17]. It adds more flexibility in terms of amount of information an attacker needs to recover secret data. In this case, the number of bits that can be recovered depends on the amount of condition operation results known. When partial knowledge about algorithm execution flow is known to the attacker, this model provides an algebraic relation between both input least significant bits. This feature could be interpreted as this model *requires* knowing one algorithm input to recover some bits of the other, especially because until now, it was only used under this scenario. For instance, in [ACSS17] and [Tuv+18] it was used to cryptanalyze modular inversion operations on secret data in DSA-like signature algorithms, where the modulus (one input) is known to the attacker. Also, in [CA+19] it was used to recover RSA private keys during generation: $d = e^{-1} \bmod (p-1)(q-1)$, where the RSA public exponent, e, is also known to the attacker. However, its usage in a both inputs unknown setting is unclear. In this paper we fill this gap showing how this model can be used when both inputs are unknown. At the same time, this model can also be used to recover all bits of one input, even without knowing when the algorithm ends its execution, a practical advantage over the *All-or-nothing* model.

The *Look-up* model was introduced in [PGB17] and is based on the *Partial* model. This model builds a dictionary that relates observed execution flow with input bits. This dictionary is obtained by profiling the algorithm execution flow with a *large* number of random input(s), and annotating which execution flow uniquely represents some input bits. To avoid certainty errors, the number of inputs should be sufficiently large (i.e. increases exponentially with number of bits to recover). This model was presented in an ECDSA nonce inversion context, where two important conditions play nicely with this model: (i) one input is known (inversion modulus), and (ii) the number of bits to recover to compromise ECDSA is small, due to well-known lattice attacks [HS01, NS03]. However, since the dictionary construction method requires at least 2^n samples for trying to recover n bits, its application for recovering a large amount of bits is not practical. On the other hand, it requires one input to be fixed, it could be unknown but must not change between calls to this algorithm as the execution flow highly depends on both inputs bits.

Table 1: Binary GCD side-channel models comparison.

Model	Branches/Bits	Alg. start	Alg. end	Known input
All-or-nothing [AGS07]	All	Yes	Yes	No
Partial [ACSS17]	Variable	Yes	No	?
Look-up [PGB17]	Small	Yes	No	No (Fixed)

In the analyzed applications of this algorithm, the number of bits needed to compromise targeted cryptosystems are not small, thus we discard the *Look-up* model. While both *All-or-nothing* and *Partial* models can be used on these scenarios, we will use the latter mainly for three reasons: (i) Reduced noise influence on processing the whole trace; (ii) Avoid having to identify the trace end position; (iii) Studying the possibility of using the *Partial* model when both inputs are unknown.

3 Vulnerable primitive: mbedTLS binary GCD algorithm

In this section the notation used by the *Partial* model regarding side-channel analysis to binary GCD based algorithms is presented. The objective is to identify which algorithm parts are interesting w.r.t. how they are implemented in mbedTLS.

The algorithm for computing the greatest common divisor (GCD) of two integers in the mbedTLS library follows a variant of the classic binary GCD algorithm [Ste67]. However, there are implementation details that make it interesting from a side-channel perspective due to the challenges they impose.

This algorithm in mbedTLS is implemented in the `mbedtls_mpi_gcd` function. It contains an initialization phase, where input variables, a and b, are assigned to u and v respectively. After this a loop divides u and v by the greatest power of two that divides both. However, without losing generality, we simplify to the case of $\gcd(a, b) = 1$, as it is a requirement in several cryptography use cases of this algorithm. This property implies that at least one input variable is odd, a useful fact for the next phase. The most important phase of this algorithm regarding side-channel analysis due to its input-dependent execution flow is a *main loop* that actually computes $\gcd(u = a, v = b)$.

Figure 1 (Left) shows a flowchart of the mbedTLS implementation of this main loop. It is composed by four condition operations, the first one (from top to bottom), controls the algorithm termination. Knowing its result is not mandatory under the *Partial* model, as it does not require knowing when the algorithm ends (cf. Table 1).

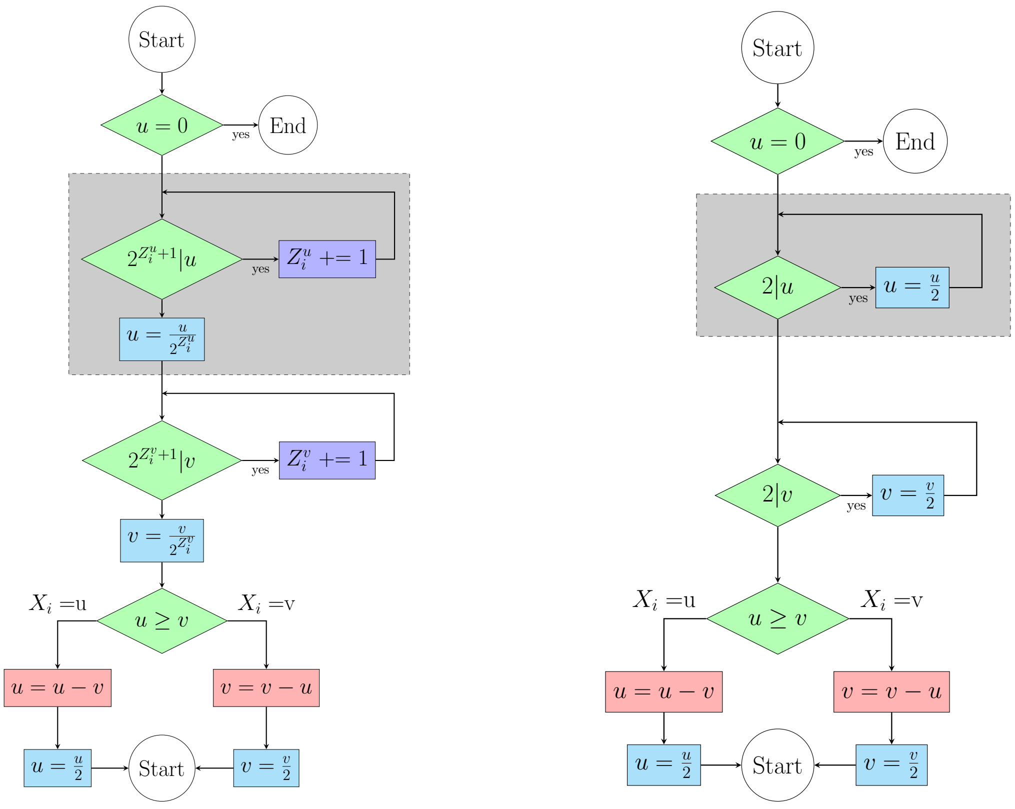

Figure 1: Binary GCD algorithm variant flowcharts. Left: mbedTLS. Right: OpenSSL.

The next two test the evenness of u and v respectively, and control two loops that count the number of trailing bits equal to zero in these variables. As can be seen in this figure, the loops for u and v have the same structure. Therefore we define a variable that will represent how many times these loops are executed at each iteration i as Z_i^x, where x represents the loop variable (u or v). At every iteration start these variables are set to zero by convention. Following the variables' evenness handling in this algorithm and analyzed below, it is easy to check that in all iterations, at least one of Z_i^x will be zero. Therefore $\max(Z_i^u, Z_i^v)$ can be used to count how many times one of these loops is executed at an iteration, regardless of which one was.

Regarding the *Partial* model, a side-channel attacker needs to know how many times a variable is divided by two (right-shifted) at each iteration. In [ACSS17] the variable Z_i is used for this task, thus in the mbedTLS binary GCD algorithm implementation context it can be defined using (1), where the +1 correction when $i > 1$ is explained below.

$$Z_i = \begin{cases} \max(Z_i^u, Z_i^v) & : i = 1 \\ \max(Z_i^u, Z_i^v) + 1 & : i > 1 \end{cases} \tag{1}$$

The fourth conditional expression in Figure 1 (Left) has two very similar branches, where the larger variable is updated by $|u - v|$ then right-shifted one bit. Regarding side-channel analysis, the result of this conditional expression will be stored in a variable called X_i, which takes a binary value that represents the largest variable.

Note that before this fourth conditional expression, both variables will be odd as u and v were previously right-shifted their respective 2-multiplicity times. Therefore, regardless of the value of X_i, the subtraction will result in an even number, and the X_i will define which Z^x_{i+1} could be different from zero. That is why (1) for $i > 1$ has a correction that includes the division by two after the subtraction in the right-shifts count at iteration i. This behavior will be helpful in Section 3.1 for determining some X_i, because if an adversary knows which variable was right-shifted at iteration i then it can infer the previous iteration X_i.

According to the *Partial* model an adversary must know a set of pairs (Z_i, X_i) for $i \in [1 \,..\, t]$, that leads to a linear closed-form expression relating the n-least significant bits of a and b, where $n = \sum_{i=1}^{t} Z_i + 1$. This expression can be obtained by reconstructing the algorithm execution flow starting from the beginning [ACSS17].

Employing symbolic values for the algorithm inputs a and b, it is possible to define $u_i(a, b)$ and $v_i(a, b)$ as functions that represent the values of these variables just before the fourth conditional expression of main loop iteration i. Therefore, as explained before, at every iteration it is know that:

$$|u_i(a,b) - v_i(a,b)| \equiv 0 \mod 2 \tag{2}$$

and at the same time, as was probed in [ACSS17], (2) results in

$$\left|\frac{A_i}{B_i}a - \frac{C_i}{D_i}b\right| \equiv 0 \mod 2 \tag{3}$$

where $\operatorname{lcm}(B_i, D_i) = \operatorname{pow}(2, \sum_{i=1}^{t} Z_i)$ for all i, thus the denominators can be removed by multiplying (3) by $\operatorname{lcm}(B_i, D_i)$, resulting in an expression like (4), that relates a and b modulo 2^n.

$$\left|\frac{\operatorname{lcm}(B_i, D_i)}{B_i}A_i a - \frac{\operatorname{lcm}(B_i, D_i)}{D_i}C_i b\right| \equiv 0 \mod \left(2 \cdot \operatorname{lcm}(B_i, D_i) = 2^{\sum_{i=1}^{t} Z_i + 1}\right) \tag{4}$$

Therefore, the more consecutive pairs (Z_i, X_i) an adversary knows the more bits it can recover [ACSS17]. This model is independent of how the (Z_i, X_i) are obtained: the next section analyzes how it is possible glean them in the mbedTLS implementation of this algorithm.

3.1 Side-channel attack on the mbedTLS binary GCD implementation

Section 5 provides experiment results of an end-to-end attack against an mbedTLS binary GCD algorithm implementation use case with an ECDSA TLS sever secured with Intel SGX. Similarly, Section 6 analyzes another use case of this function on RSA. Both scenarios target the same function, `mbedtls_mpi_gcd`, therefore this section presents a use-case-independent side-channel attack against it.

The vulnerability is demonstrated against a TLS server running inside an SGX enclave, therefore following the Intel SGX threat model it is considered that the OS is adversarial [XCP15, VBPS17]. As part of this evaluation, we employed both the page-fault traceability feature on `SGX-Step` and the interrupt-driven attack.

Threat model and experiment setup. The experiments were performed on Ubuntu 18.04 LTS with kernel 5.0.0-29 running on an Intel i7-7700 (Kaby Lake) processor with SGX support. Compliant with the Intel SGX threat model we disabled TurboBoost and dynamic frequency scaling, also we isolated one CPU core for the victim. We used

`SGX-Step` v1.2.0 (commit 8386858c) paired with Intel SGX SDK v2.2 and Linux SGX driver v2.1. The `mbedTLS-compat-SGX` open-source project[1] was employed to add SGX support to an updated mbedTLS version (v2.16.1) [Sil19]. It is assumed that the adversary knows all page address(es) of every function of interest in the compiled enclave. This can be obtained by static analysis, or in case of an encrypted enclave, the adversary can perform reverse engineering by monitoring all pages used by the enclave and discarding the uninteresting ones by trial and error [XCP15, VBPS17, WSB18].

Following the binary GCD execution flow analysis in the previous section, the attacker is interested in extracting the (Z_i, X_i) pairs. Therefore, it would like to know how many times the trailing zero removal loops execute at each iteration in addition to the result of the comparison step (bottom condition in Figure 1, Left). The procedure for analysis is the following:

1. Identify a page sequence that marks the start of the algorithm (i.e. *trigger*).
2. Select a set of pages that allow to trace every function of interest.
3. Identify trace features to recover (Z_i, X_i) pairs.
4. Capture a page trace, then when the *trigger* sequence occurs enable the interrupt-driven attack.

The first step is about identifying a sequence that marks the start of the `mbedtls_mpi-_gcd` use case of interest. Regarding a generic side-channel analysis of this function, the selection of this sequence is meaningless as it is closely related to the actually attacked use case. For example, the sequence for an ECDSA use case probably will involve a page that is only used in ECDSA and the same for RSA. For this reason, we defer the details of this step to Section 5 and Section 6 when targeting ECDSA and RSA use cases.

The second step involves analyzing the set of pages that could give information about the execution flow of `mbedtls_mpi_gcd`. These could be, for example, the functions called by `mbedtls_mpi_gcd`. Table 2 summarizes the most interesting functions regarding side-channel analysis with their corresponding page offsets and the colors we used to represent them in the Figure 2 trace.

Table 2: `mbedtls_mpi_gcd` pages of interest.

Function	Page offset(s)	Color
`mbedtls_mpi_gcd`	`0x1F000`	Green
`mbedtls_mpi_lsb`	`0x1B000,0x1C000`	Blue
`mbedtls_mpi_cmp_mpi`	`0x1C000`	Orange
`mbedtls_mpi_shift_r`	`0x1C000`	Orange

Page offsets are set at build time and depend on how the linker distribute each function in the binary. In this case we are interested in the page of `mbedtls_mpi_gcd` itself as it will help to identify when an inner operation ends.

The function `mbedtls_mpi_lsb` counts the number of trailing zero bits of an integer, therefore it executes the corresponding loops at Figure 1 (Left). One important aspect of this function is that its execution time depends on how many input trailing bits equal to zero, hence relates to Z_i, a fact that we are going to exploit later (third step in the procedure).

The third function of interest is `mbedtls_mpi_cmp_mpi`, used to test $u \geq v$ with the result determining the X_i. This function also contains several branches that make its

[1]an updated fork of `bl4ck5un/mbedTLS-SGX`

running time input-dependent (third step in the procedure). `mbedtls_mpi_shift_r` is located in the same page as `mbedtls_mpi_cmp_mpi`, thus colored the same in a trace.

One interesting characteristic is that `mbedtls_mpi_lsb` and `mbedtls_mpi_cmp_mpi` share a page, however it does not have a big influence on the attack as it is possible to differentiate each one using the previous executed page. `mbedtls_mpi_cmp_mpi` is called by `mbedtls_mpi_gcd`, so it is expected to see a page access to the latter prior to that of `mbedtls_mpi_cmp_mpi`. On the other hand, `mbedtls_mpi_lsb` starts its execution at `0x1B000` and then continues to `0x1C000`, without access to an `mbedtls_mpi_gcd` page in between, hence the distinction is immediate.

The interrupt-driven attack relies on arming a timer that will interrupt the enclave forcing its preemption. During this pause, the attacker collects side-channel information from microarchitecture resources such as page tables. One interesting side-channel source is the `ACCESSED` bit in a page table entry (PTE) that indicates if a page was accessed or not. Therefore, an adversary at each timer interrupt can clear this bit for a set of monitored pages and in the next interrupt check which one was executed. If the timer is configured such that it interrupts the enclave once per instruction, then using the `ACCESSED` bit it is possible to count the number of executed instructions per page. This scheme was proposed as part of the `SGX-Step` framework [VBPS17], however not deeply evaluated w.r.t. attacking real-world cryptography algorithm implementations.

Therefore, an attacker can combine a page trace attack with an interrupt-driven one to monitor SGX enclave executions with high temporal resolution. The number of executed instructions in a high-end application like a TLS server is large, hence, an interrupt-driven attack starting from the beginning of execution will considerably reduce server performance and increase the chance of detection. Therefore, it is only executed on demand, once a specific page sequence (*trigger*) has been executed. Thus, the attacker uses `SGX-Step` to trace the pages of interest, waiting for the *trigger* condition to happen, then the interrupt-driven attack is started. At this point, the page tracing is disabled to not interfere with the timings, nevertheless, the `ACCESSED` bit traces will also contain the sequence of executed pages.

Figure 2 shows partial traces of an execution of `mbedtls_mpi_gcd`. Each trace belongs to a monitored page (Table 2). They are actually binary values (i.e. `ACCESSED` bit), however for the sake of distinguishability we scaled them using different values on the y-axis.

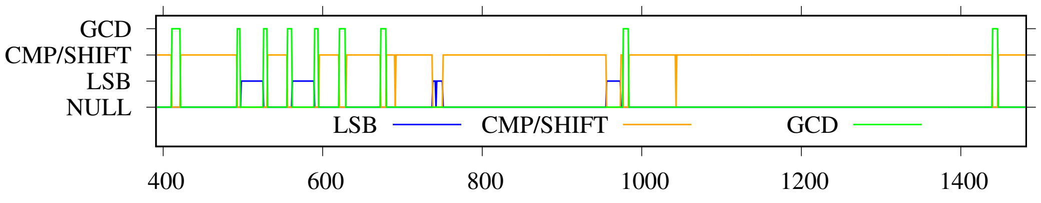

Figure 2: `mbedtls_mpi_gcd` `ACCESSED` bit traces of monitored pages (Table 2).

This figure corresponds to the first iteration of an execution of `mbedtls_mpi_gcd`. The first `mbedtls_mpi_gcd` peak (green) marks the start of this iteration and the last peak the start of the second. Therefore, each iteration is composed of eight `mbedtls_mpi_gcd` peaks. With these traces the attacker has side-channel leakage related to the number of instructions executed at each page and their execution sequence. Following the execution flow of this function, we are interested in the first two `mbedtls_mpi_lsb` executions (blue), because according to Figure 1 (Left) the number of times the `mbedtls_mpi_lsb` page was accessed in these time windows is related to Z_i^u and Z_i^v, thus a potential leak for Z_i using (1).

One significant challenge to attacking this implementation is that these loops are extremely tight, so, a high-temporal resolution side-channel is needed to distinguish them. This represents a new challenge to overcome regarding other binary GCD based algorithms already attacked in the literature [PGB17, WSB18, CA+19]. For instance, compare the OpenSSL version depicted in Figure 1 (Right), where instead of counting the number of trailing zeros in a loop and removing them with a multi-bit shift (mbedTLS), OpenSSL loops include a single-bit right-shift on each iteration, increasing the time window available to track their execution (cf. shaded areas in Figure 1).

Figure 2 gives an idea regarding this timing difference. For instance, the time window between the last two green peaks in this figure, belongs to the execution of a single-bit right-shift of a 1024-bit number (about 455 page accesses), while a single iteration mbedTLS trailing zero count loop takes 10 times less (i.e. 43 page accesses).

One important interrupt-driven attack parameter is the timer interval at which the enclave should be preempted. According to Van Bulck, Piessens, and Strackx [VBPS17] it should guarantee that the timer interrupt arrives just during the execution of the next enclave instruction after it resumes its execution. This parameter is platform specific hence should be determined by trial and error using `SGX-Step` benchmark tools and the targeted implementation. Our tests report that the best trade-off value for this parameter (`SGX_STEP_TIMER_INTERVAL` in `libsgxstep/config.h`) was 25, and other configuration parameters left as defaults [VBPS17] [2].

Using this configuration we captured a set of 1000 traces corresponding to the execution of `mbedtls_mpi_gcd` with known inputs and recorded the number of times the pages of `mbedtls_mpi_lsb` were accessed during the periods of interest in Figure 2 (first two blue valleys). Using this method we determined how well the number of accesses in these valleys are related to Z_i. The results were perfect, for every value of Z_i the number of observed accesses was unique, indicating that the Z_i recovery employing this method is incredibly reliable.

After developing the Z_i recovery procedure, it only remains to craft a corresponding procedure for gathering the X_i. In this implementation X_i leakage comes from two sources, (i) `mbedtls_mpi_cmp_mpi` and (ii) `mbedtls_mpi_shift_r`.

Leakage from `mbedtls_mpi_cmp_mpi`. The comparison $u \geq v$ is executed using the function `mbedtls_mpi_cmp_mpi`. A source code analysis of this function reveals that it has many input-dependent branches with a total of eight exit points. Therefore, the total execution time of this function could be a good leakage source for determining X_i.

Figure 3 (Left) shows the latencies (i.e. number of observed page accesses) of `mbedtls-_mpi_lsb` over iterations of a single call to `mbedtls_mpi_gcd` showing the Z_i latencies are very well clustered resulting in a unique latency per Z_i as described before. Similarly, Figure 3 (Right) shows the latencies of `mbedtls_mpi_cmp_mpi`. These two plots are the result of processing a trace and are the source for recovering (Z_i, X_i) pairs.

Latency behavior in Figure 3 (Right) can be better explained analyzing `mbedtls_mpi-_cmp_mpi` source code. Figure 4 shows a snippet of this function, commenting the most relevant parts.

First, this function determines inputs' number of significant words followed by two early exit points if inputs differ in this magnitude. The last loop in this function is the most frequently executed as part of the binary GCD algorithm, since its behavior tends to maintain equality in the number of bits of X and Y, hence the number of significant words.

Figure 3 (Right) plot can be split in nine very similar *groups*, where each group is formed by two latencies aligned to the grid. For instance, the first group in Figure 3 (Right) is composed by those samples with latencies 74 or 90, roughly between the first

[2]`SGX_STEP_TIMER_INTERVAL=26` performs better for single-stepping using `SGX-Step` benchmark tools, but 25 provides a more reliable Z_i extraction for 1000 targeted binary GCD implementation executions.

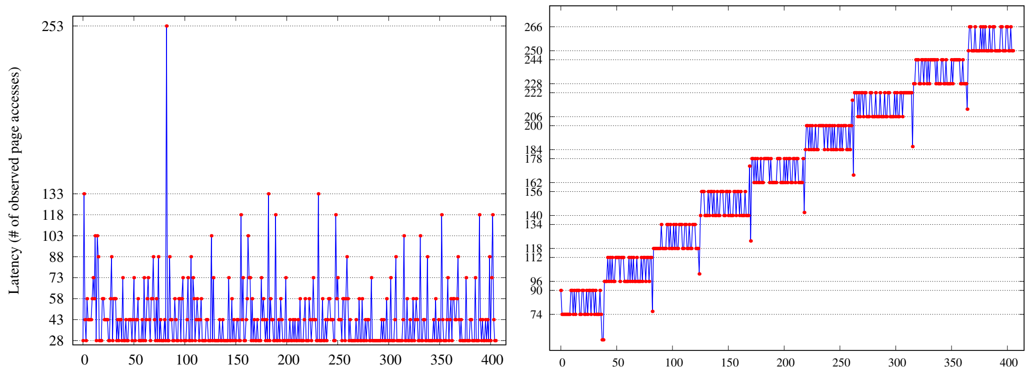

Figure 3: Functions of interest latencies over iterations to recover. Left: `mbedtls_mpi_lsb` (Z_i). Right: `mbedtls_mpi_cmp_mpi` (X_i).

```
1  int mbedtls_mpi_cmp_mpi(const mbedtls_mpi *X, const mbedtls_mpi *Y)
2  {
3      size_t i, j;
4
5      /* Determine number of significant words of X and Y */
6      for( i = X->n; i > 0; i-- )
7          if( X->p[i - 1] != 0 ) break;
8
9      for( j = Y->n; j > 0; j-- )
10         if( Y->p[j - 1] != 0 ) break;
11             ...
12     /* Do inputs have different # of words? */
13     if( i > j ) return(  X->s );
14     if( j > i ) return( -Y->s );
15             ...
16     /* More frequent path */
17     for( ; i > 0; i-- )
18     {
19         if( X->p[i - 1] > Y->p[i - 1] ) return(  X->s );
20         if( X->p[i - 1] < Y->p[i - 1] ) return( -X->s );
21     }
22     return( 0 );
23 }
```

Figure 4: Multiple-precision integer comparison in mbedTLS: `mbedtls_mpi_cmp_mpi`.

40 iterations. We discuss out-of-group latency samples later. Latencies that are part of a group correspond to the execution of the last loop of `mbedtls_mpi_cmp_mpi` (Figure 4) because it is the most common executed path, a behavior that is also observed in Figure 3 (Right). Therefore, the lower latency in a group happens when line 19 of Figure 4 evaluates `true` and the function ends, leaking that $u > v \implies X_i=\text{‘}u\text{’}$. Analogously, if it is `false` and line 20 evaluates `true`, then $X_i=\text{‘}v\text{’}$. One important feature of this group-based X_i distinguisher, is that the latency difference between a group lower and upper values is 16, leaving some space for uncertainties. In this regard, `mbedtls_mpi_cmp_mpi` does not behave so deterministically as `mbedtls_mpi_lsb`, sometimes observed as an error of ± 1. This error does not have any effect on X_i distinction inside a group, but outside it does, as explained below.

The behavior that groups are shifted in the y-axis is due to the binary GCD algorithm reducing the number of bits of u and v progressively, then at some point the number of effective words on these variables will be less than the maximum, therefore, the loops at the start of `mbedtls_mpi_cmp_mpi` (cf. Figure 4) will execute more iterations, hence the shifting.

As can be appreciated, almost every latency sample in Figure 3 (Right) belongs to a group, however, there are a few outliers. These occur during a group transition, and these latencies belong to the early exit points in lines 13 and 14 in Figure 4. The difference between these line latencies is small enough to be inside the ± 1 observed error. Hence, they do not provide an error-free X_i distinguisher. Therefore, we mark these out-of-groups latencies as unknown. For instance, in Figure 3 (Right) there are 11 of them, therefore, as each of them represents an X_i, a binary value, the adversary can exhaustively search the missing X_i. Even if a 2^{11} exhaustive search is not large enough to be impractical, it can be considerably reduced using a stronger probabilistic X_i leakage.

Leakage from `mbedtls_mpi_shift_r`. Just before the comparison $u \geq v$ both variables are odd, then regardless of this condition result, only one of them will be even—that is why there is a division by two just after the subtractions in Figure 1 (Left). Therefore, as the values of u and v can be considered random at the next algorithm iteration, there is a 50% chance that one of them is even at the start of every iteration. Consequently, the one that could be even is determined by the result of $u \geq v$ at the previous iteration (X_i).

Therefore, an X_i leakage could be observed in about half of the iterations by measuring if the right-shifts at iteration $i + 1$, actually right-shift a variable (i.e. $Z_i^x \neq 0$). This represents a strong leakage, because the number of accesses to the `mbedtls_mpi_shift_r` page is considerably more when the number of bits to shift is non-zero.

At each iteration after each `mbedtls_mpi_lsb` call (blue valley) there is a call to `mbedtls_mpi_shift_r` (orange valley). In Figure 2 these orange valleys are quite small (43 page accesses) compared to the last orange valley that belongs to the `mbedtls_mpi_shift_r` call just after a subtraction which has about 10 times more page accesses. Hence distinguishing when `mbedtls_mpi_shift_r` was called with a shift count equal to zero is very reliable due to this big difference in the number of page accesses.

In this manner the adversary has a strong leakage that reveals X_i with a probability of 50%, which can be very useful for recovering the X_i marked as unknown during the `mbedtls_mpi_cmp_mpi` approach. For instance, after applying this leakage source to Figure 3 (Right) trace, the number of unknown X_i dropped from 11 to 7.

At this point an attacker has everything it needs to apply the *Partial* model and start recovering bits. Therefore, we conclude that the mbedTLS binary GCD primitive implementation is vulnerable to side-channel analysis, and this is the first time that this implementation is analyzed in this regard. However, for a practical perspective it is interesting to identify which cryptosystems employ this primitive in a security critical operation. The next two sections analyze ECDSA and RSA protocols in this regard, disclosing new vulnerabilities.

4 Security of an unexpected GCD call in mbedTLS ECDSA

This section presents a new vulnerability in the mbedTLS ECDSA implementation where the vulnerable point resides in a countermeasure deployed in this library for more than five years[3]. The vulnerability resides in a GCD computation; that might sound unexpected because neither the high-level description of ECDSA nor its lower layers nor the countermeasure include this operation at all, but the implementation always has the last word in the field of side-channel attacks. Another interesting feature about this vulnerability is that it resides inside a countermeasure considered to be safe, thus providing a false state of security. For instance, in a recent disclosed vulnerability in the mbedTLS library[4] it is assumed that this countermeasure thwarts side-channel analysis, while it does not.

[3] *Remove potential timing leak in ecdsa_sign()* mbedTLS commit (March 31, 2014)
[4] CVE-2019-16910

The ECDSA algorithm is the elliptic curve variant of the digital signature algorithm standardized by NIST [Fip]. Algorithm 1 shows the pseudocode of the ECDSA signature generation procedure. This algorithm generates a digital signature for a public message (m) employing the secret private key (d), where h corresponds to the application of a hash function to the message m and is also considered public.

Algorithm 1: ECDSA signature generation

```
Input: private key $(d)$, elliptic curve generator $(G)$, hash of $m$ $(h)$, order of $G$ $(p)$
Output: a signature for message $m$ $(r, S)$
1 Select $k$ at random such that $0 < k < p$
2 $(x, y) = k \cdot G$
3 $r = x \mod p$
4 $S = k^{-1}(h + rd) \mod p$
5 if $r = 0$ or $S = 0$ then goto 1
6 return $(r, S)$
```

Each generated signature involves selecting a random *secret* nonce k satisfying $0 < k < p$, performing scalar multiplication of this nonce with the elliptic curve generator point (G), and reducing the resulting value (x) modulo p [Fip]. At line 4, the linear part of the signature generation computes the modular inverse of k and uses it to calculate the public value S.

Regarding side-channel analysis, the scalar multiplication has received a lot of attention since the inception of this field [FV12, Dan+13], but recently vulnerabilities on other operations have emerged, like for example the nonce inversion operation [ACSS17, PGB17] and the multiplication of $rd \bmod p$ [Rya19].

4.1 Vulnerability in nonce blinding countermeasure

The inversion of the nonce in ECDSA is a security critical operation as it is usually implemented using a variant of the binary GCD algorithm for computing modular inverses that is highly dependent on its inputs [AGS07, ACSS17, PGB17]. Therefore, efforts have been made to harden this operation in commonly used cryptography libraries like OpenSSL [Gri+19], whereas mbedTLS was one of the first to add protection to this operation about five years ago.

The countermeasure deployed in mbedTLS masks the nonce before inverting it, thus, any information leakage during its inversion (seemingly) reveals no secret information. The well-known procedure of this countermeasure is the following:

1. $t =$ Draw t at random s.t. $0 < t < p$
2. $b = k \cdot t \bmod p$
3. $f = b^{-1} \bmod p$
4. $k^{-1} \equiv f \cdot t \bmod p$

However, its implementation in mbedTLS does not strictly follow this procedure. Figure 5 (Left) shows a code snippet of this library implementation, where the "masking" operation line is highlighted and the modular inversion takes place at the next line. Our key insight is the mbedTLS implementation lacks a reduction operation after the multiplication takes place, hence this multiplication is performed on $\mathbb{Z}$ instead of $\mathbb{Z}_p^*$. While it is mathematically correct, we show it fails at protection because the product $b = kt$ does indeed reveal information about k.

Figure 5 (Right) shows a code snippet of the mbedTLS modular inverse function `mbedtls_mpi_inv_mod`. This function contains an implementation of the Binary Extended Euclidean Algorithm (BEEA) for computing modular inverses. The highlighted line shows that the input is actually reduced before starting to execute the BEEA code (indicated

```
1   static int ecdsa_sign_restartable( ... )
2   {
3       ...
4       /*
5        * Generate a random value to blind inv_mod in next step,
6        * avoiding a potential timing leak.
7        */
8       mbedtls_ecp_gen_privkey( grp, &t, f_rng_blind,
    ↪   p_rng_blind );
9
10      mbedtls_mpi_mul_mpi( s, pr, d );
11      mbedtls_mpi_add_mpi( &e, &e, s );
12      mbedtls_mpi_mul_mpi( &e, &e, &t );
13      mbedtls_mpi_mul_mpi( pk, pk, &t );
14      mbedtls_mpi_inv_mod( s, pk, &grp->N );
15      mbedtls_mpi_mul_mpi( s, s, &e ) );
16      mbedtls_mpi_mod_mpi( s, s, &grp->N );
17      ...
```

```
1   int mbedtls_mpi_inv_mod( mbedtls_mpi *X, const mbedtls_mpi
    ↪  *A, const mbedtls_mpi *N )
2   {
3       ...
4       mbedtls_mpi_gcd( &G, A, N );
5       ...
6       if( mbedtls_mpi_cmp_int( &G, 1 ) != 0 )
7       {
8           ret = MBEDTLS_ERR_MPI_NOT_ACCEPTABLE;
9           goto cleanup;
10      }
11      ...
12      mbedtls_mpi_mod_mpi( &TA, A, N );
13
14      /* Binary Extended Euclidean Algorithm for computing
        ↪  inverses */
15      ...
```

Figure 5: ECDSA nonce blinding countermeasure in mbedTLS.

with a comment). Therefore, during the BEEA execution the countermeasure would be *complete* and the value to be inverted successfully masked. Yet, `mbedtls_mpi_inv_mod` does the unexpected.

It also computes $\gcd(A = b, N = p)$ to check if an inverse exists[5]. And this happens at line 4, *before* reducing $b = kt$ thus, if the execution flow of `mbedtls_mpi_gcd` can be obtained by some side-channel, as demonstrated in Section 3.1, the countermeasure can be compromised.

For instance, if the attacker knows that the product kt is odd by some side-channel leak, it learns that k is also odd, thus obtaining a 1-bit leak. In theory, this leak could be exploited using Bleichenbacher's approach [TTA18], however no evidence has been published that this attack could be achieved in practice for commonly used ECDSA curves. Therefore, we follow a generic approach, that surprisingly reduces the security of ECDSA to an integer factoring problem.

4.2 When ECDSA security relies on factoring integers

In this section we will first describe the entire attack independent of the side-channel used to obtain it, assuming that an attacker already obtained the product $b = kt$. Then in Section 5 we will demonstrate it against an mbedTLS-backed TLS server secured by Intel SGX, evidencing how an attacker can exploit it in a real-world scenario.

Once an adversary knows b, it also knows that one of its divisors is actually the secret nonce k, therefore, it could do an exhaustive search on every possible divisor of b to see which one satisfies $r = k \cdot G$. Hence, the task is reduced to factoring b. Considering an n-bit ECDSA instance, it means that both k and t are about n-bit numbers, thus, $b = kt$ is roughly a $2n$-bit integer. Therefore, it is interesting to know how many candidates an attacker will have to test in the worst case scenario.

An integer number can be decomposed into its prime factors like (5), where the set of q_i are the different prime factors that divide b and m_i their corresponding multiplicity.

$$b = \prod_{q_i | b} q_i^{m_i} \tag{5}$$

As the attacker must exhaustively search every possible divisor, it is interested in the number of prime factors of b considering multiplicities. Number theory field defines the function $\Omega(\cdot)$ for counting this magnitude, also its distribution has been studied for large integers. According to [Rie94], $\Omega(\cdot)$ follows a normal distribution defined by (6).

$$\Omega(b) \sim \mathcal{N}\left(1.03465 + \ln\ln b,\ \sqrt{\ln\ln b}\right) \tag{6}$$

[5]We hypothesize the developers believe it could save some CPU cycles, yet it most likely does not since in many use cases of this function, coprimality is guaranteed a priori.

Therefore, for 256-bit ECDSA, there is a probability of 99.4% that the number of prime factors of b (i.e. 512-bits) is less than 14. So, with high probability, the worst case number of candidates to test is defined by (7), that following the 256-bit example, means only 2^{13} candidates.

$$\#\text{candidates} = \sum_{i=1}^{\Omega(b)} \binom{\Omega(b)}{i} = 2^{\Omega(b)} \tag{7}$$

Therefore, it is possible to define an attack roadmap for n-bit ECDSA:

1. SCA(`mbedtls_mpi_gcd`) $\implies (Z_i, X_i) : \sum Z_i \geq 2n$
2. (Z_i, X_i) leakage with *Partial* model yields b
3. factor(b) generates $2^{\Omega(b)}$ candidates
4. Test candidates until solution

The first step resumes the side-channel attack part when the adversary gets sufficient (Z_i, X_i) pairs such that applying the partial model (at second step) it could recover the $2n$ least significant bits of b (i.e. all bits of b). Under a perfect leakage the side-channel part is free of errors, so this process only yields a single candidate for b. However, this procedure can handle uncertainty in this step, for instance, maybe the attacker is not sure about the value of some X_i, thus, as this is a binary variable, the attacker can generate all possible combinations, that will yield a set of candidates for b. Section 3.1 shows how the (Z_i, X_i) pairs from an `mbedtls_mpi_gcd` execution can be recovered, whereas Section 2.2 describes the *Partial* model to recover each candidate for b after bruteforcing the missing X_i. After describing this attack, experiment results for 1000 trials are presented in Section 5.

Once a candidate for b has been obtained, it should be factored to enumerate its $2^{\Omega(b)}$ divisors. The factoring phase is the most time-consuming part. Therefore, the attacker would want to reduce the number of candidates for b in the previous step, trading it off with the probability that correct b is in the set.

The last step involves testing which divisor of b is the secret nonce k. This can be done by testing which divisor is the solution to the ECDLP problem $r = k \cdot G$. Therefore, the number of scalar multiplications needed to recover the ECDSA private key would be $(\#b \text{ cands}) \cdot 2^{\Omega(b)}$.

It is worth mentioning, this attack only needs one trace to succeed, however, the attacker can capture a set traces and launch several attack instances in parallel until one yields a solution. This approach could be helpful to overcome the running time of integer factorization.

5 End-to-End Attacks on a SGX-secured mbedTLS server

The next sections present two end-to-end attacks against a TLS server backed by mbedTLS and secured by Intel SGX. For the experiment results we used the `SGX-Step` framework with threat model and setup described in Section 3.1 to attack the mbedTLS binary GCD implementation. The two presented attacks are:

1. Exploit the ECDSA vulnerability described in Section 4.
2. Exploit an RSA vulnerability where both inputs of the binary GCD algorithm are secret (Section 6).

Both attacks exploit the vulnerable binary GCD implementation in the mbedTLS library in two very different scenarios. This result supports the portability of the attack on mbedTLS binary GCD algorithm: an example of how a vulnerable primitive leads to multiple vulnerabilities in the same library.

5.1 Bulk experiments on ECDSA

We performed the attack against the wrongly implemented countermeasure in mbedTLS ECDSA that executes a side-channel vulnerable binary GCD algorithm using NIST curve `secp256r1` [Fip]. We repeated the attack 1000 times to gather sufficient experiment data to evaluate each attack phase, highlighting the following metrics:

1. Number of candidates for b during the SCA of `mbedtls_mpi_gcd`.

2. Statistics about the factoring phase.

To launch the attack against `mbedtls_mpi_gcd` during its vulnerable use case inside mbedTLS ECDSA, we employed the memory page of `ecdsa_sign_restartable` to define the trigger that identifies the targeted `mbedtls_mpi_gcd` start inside ECDSA. For this, we first launched an attack without any defined trigger (only monitoring the pages of interest without an interrupt-driven attack), this phase generates a sequence of accessed page where employing the memory page of `ecdsa_sign_restartable` the identification of a unique trigger sequence was immediate.

Then, the attack is relaunched with the defined trigger that will start the interrupt-driven attack to capture the page `ACCESSED` bit traces. The obtained traces are processed to extract the (Z_i, X_i) pairs as described in Section 3.1. After this step, we bruteforce the missing X_i and apply the *Partial* model for each of them, obtaining a set of b-candidates. At each attack trial the adversary initiates a TLS session with the mbedTLS server and negotiates a ciphersuite with ECDSA as signature algorithm. The client (adversary) collects the signature information for testing, in the last phase of the attack, which divisor of b is the k that solves $r = k \cdot G$.

We repeated the attack 1000 times and computed statistics about the number of b-candidates. From the 1000 traces, two of them were not processed correctly, implying that no (Z_i, X_i) pairs where obtained, hence, the remaining 998 yield a median of four candidates that demonstrates the efficiency of the side-channel phase.

In addition to number of candidates statistics, we computed the success rate of this part of the attack employing the ground truth private key. For each trial, we computed the nonce k, and then checked if one of the b-candidates is divisible by k. This test revealed the side-channel attack phase succeeded in 996 trials of 1000, which demonstrates its very high success rate, with two traces where some X_i were not identified correctly. The next sections will complete the end-to-end attack from an adversary point of view, concluding that the success rate was invariant. In support of Open Science, we released our data and tooling for (part of) the ECDSA end-to-end attack [AB20].

5.2 Factoring

The purpose of the factoring phase is to compute the complete factorization of a given b-candidate. Given that both k and t have no special form other than being drawn uniformly from $\mathbb{Z}_p^*$, i.e. statistically close to 256-bit uniformly random strings, we chose the general purpose “Yet Another Factoring Utility” (YAFU) for this task[6]. The application links against several other libraries for some functionality, e.g. GMP-ECM[7] for the Elliptic Curve Method (ECM) and Msieve[8] and GGNFS[9] for different Number Field Sieve (NFS) stages, yet contains its own implementation of other functionality, such as the Self-Initializing Quadratic Sieve (SIQS). We chose YAFU for its flexibility, parallelization support, and ability to iteratively apply known methods from trial division to NFS, not requiring any

[6] https://sourceforge.net/projects/yafu/
[7] http://ecm.gforge.inria.fr/
[8] https://sourceforge.net/p/msieve/
[9] https://sourceforge.net/projects/ggnfs/

special pre-processing step. We set the SIQS to NFS crossover threshold at 100 decimal digits. We used the latest repository version (as of this writing) of YAFU itself and all the prerequisite software.

Worst case analysis. To upper bound the factoring time, we ran a short experiment to factor an RSA-512 key generated from the OpenSSL command line tool. This represents the rare worst case scenario, where both the nonce and blinding value are 256-bit primes. Academically, Valenta et al. [Val+16] showed how to use the Amazon EC2 infrastructure to factor such a key in under four hours at a cost of 75 USD. As an alternative, to carry out the computation locally we used a 24-core 48-thread Intel Xeon Silver 4116 (Skylake) server clocked at 2.10GHz with 256GB RAM running Ubuntu 16.04.6 LTS. The NFS factorization completed in 53 hours, fully recovering the RSA-512 private key.

Computing environment. Despite the meager upper bound above, our goal is not to demonstrate one successful attack instance, but to understand typical computation requirements over a large number of attack trials. To that end, we carried out the remainder of our results on a computing cluster containing roughly 800 Intel Xeon Gold 6148 (Skylake) cores clocked at 2.40GHz and 2300 Intel Xeon E5-2680v3 cores (Haswell) clocked at 2.50GHz. In the experiments that follow, key enumeration always took place on a single core per task while factoring ranged from a single core to eight parallel cores per task, depending on the factoring complexity.

5.3 Key enumeration

The purpose of the enumeration phase is to calculate the nonce k from a given b-candidate. To enumerate the keys, we wrote a custom application linking against OpenSSL to take advantage of its high-speed P-256 scalar multiplication functionality for AVX architectures. The application takes as input the complete factorization of the b-candidate, and the (public) r-component of the ECDSA signature. It then iterates through the power set of the factors, computes the corresponding k-candidate at each iteration, computes the scalar multiplication $k \cdot G$, and finally checks if this values equals r. If the check passes, this yields the true nonce k for the ECDSA signature, then finally the long-term private key rearranging the (public) S component of the ECDSA signature. There are several simple optimizations to (somewhat) reduce the exponential cost of the power set iteration. As soon as the k-candidate exceeds the group order, that limb can be trimmed. Also, iterating the k-candidates starting from the group order down to zero makes sense statistically, as the number of possible nonces decreases exponentially with the bit length.

5.4 Bulk experiment results

From the 1000 trials, we were left with a maximum 17446 candidates to potentially factor. The median number of candidates per trial was four. We carried out an iterative process to solve for these trials, consisting of limited effort to factorize candidates, followed by enumeration attempting to solve each trial. Denote S_0 these 17446 candidates, and $T_0 = [1 \mathinner{..} 1000]$ the set of trials. Table 3 summarizes the progress of our iterative attack process, with S_i the remaining number of candidates without complete factorization, and T_i the remaining number of unsolved trials at stage i.

On the cluster, we performed an initial ECM factoring pass ($i = 1$) with a per task time limit of 4h. This yielded 11683 complete factorizations (i.e. $|S_1| = 17446 - 11683 = 5763$). Running enumeration, this solved for 639 of the trials (i.e. $|T_1| = 1000 - 639 = 361$). With the remaining partial factorizations from the unsolved trials, we proceeded to more advanced SIQS and NFS factoring techniques ($i = 2$), computing in 8-way parallel per cluster task. The majority of these tasks exceeded the 100 decimal digit SIQS/NFS

Table 3: Factoring effort over all attack trials spanning various stages.

i	$\lvert S_i\rvert$	$\lvert T_i\rvert$	Method
0	17446	1000	—
1	5763	361	ECM
2	4347	44	SIQS/NFS
3	4255	4	NFS

threshold, hence applied NFS methods. After two days of computation, including these new factorizations and re-running enumeration brought the total to 956 solved trials. For the remaining 92 partially factored candidates (for the 44 unsolved trials) ranging from 134 to 154 decimal digits, we made a final NFS factoring pass again in 8-way parallel tasks over five days ($i = 3$). Including these 92 new full factorizations and re-running enumeration brought the total to 996 solved trials, and in total 13191 fully factorized candidates. Of the remaining four unsolved trials, two did not produce any candidates from the post-processing stage, hence were unsolvable from the beginning. Checking the ground truth, the remaining two unsolved trials did not retain the correct candidate. The median factoring time over all fully factorized candidates, i.e. $S_0 \setminus S_3$, was 14 minutes. The median enumeration time was less than a single second with the median number of keys checked through scalar multiplication 129. The median value of $[\Omega(b) : b \in S_0 \setminus S_3]$ was nine.

6 Practical attack on an RSA-CRT computation

A recent paper analyzes an interesting perspective on side-channel attacks where the leakage comes when private keys are loaded [PG+19]. The authors discovered several vulnerable code paths that get triggered when private keys are parsed on popular cryptography libraries such as OpenSSL and mbedTLS. One challenge the authors left as an open problem, especially the recovery of X_i, is attacking the computation of RSA-CRT parameter $q^{-1} \bmod p$ in the mbedTLS library.

This section provides experiment results on this challenge, as well as (for the first time) demonstrating the usefulness of the *Partial* model when both inputs of the binary GCD algorithm are secret, another open problem not covered before in the literature as analyzed in Section 2.2.

The threat model and application scenario of the experiments are very similar to those presented in Section 3. Like the ECDSA case, we consider there is an mbedTLS server secured by Intel SGX where the attacker can launch page-fault and interrupt-driven attacks against it using the `SGX-Step` framework.

Every time an RSA private key is loaded by the mbedTLS library, the Chinese Remainder Theorem (CRT) parameter $q^{-1} \bmod p$ is computed where p and q are the secret prime numbers of that private key, and it is know that $q < p$ and $N = pq$ is a public parameter. This modular inversion is performed employing the same function used to invert the ECDSA nonce (i.e. `mbedtls_mpi_inv_mod`), therefore it has an internal call to `mbedtls_mpi_gcd`.

It is worth noting that in this use case, the modular inversion algorithm that performs this inversion (i.e. BEEA in `mbedtls_mpi_inv_mod`) could also be targeted using a similar approach. However, we chose `mbedtls_mpi_gcd` to demonstrate how the same attack setup employed to compromise mbedTLS ECDSA can be applied to RSA, highlighting `mbedtls_mpi_gcd` attack portability using `SGX-Step` framework.

We developed an attack against this scenario during the loading of 1000 RSA-2048 private keys and estimate its success rate and complexities. In this case we used the memory page of `mbedtls_rsa_deduce_crt` to select a reliable trigger for the interrupt-driven attack similar to the ECDSA case. Then for each trace we recovered the corresponding (Z_i, X_i) pairs that yield the following results.

We configured the trace processing tool to recover sufficient (Z_i, X_i) pairs such that it could be possible to recover 1024 bits of a secret prime using the *Partial* model. Using ground truth values of each private key, we estimated the success rate at 99.1% for 1000 samples. This result shows that the side-channel attack phase performs very similar to the ECDSA case without a success rate reduction when the number of bits to recover doubles. For instance, for ECDSA we targeted to recover 512 bits and now we are targeting 1024 for RSA, achieving in both cases success rates of 99%.

Regarding an end-to-end attack, a *blind* attack description where the *Partial* model is used when both inputs are unknown and involved complexity follows.

6.1 Partial model with two unknown inputs

During the *Partial* model introduction at Section 3, it was stated a set of consecutive (Z_i, X_i) pairs allows an adversary to get an expression like (4). This expression can be simplified to (8), where D_i and E_i are known integer coefficients derived from (4), Z_t is the last known Z_i, and $n = \sum_1^t Z_i + 1$ hence the number of bits that can be recovered.

$$D_i p - E_i q \equiv 2^{Z_t} \mod 2^n \tag{8}$$

In this scenario p and q are the binary GCD algorithm inputs and both are secret. On the other hand, an adversary can employ that $N = pq$ to solve (8). As $N \equiv pq \mod 2^n$, hence solving for q leads to:

$$q \equiv Np^{-1} \mod 2^n \tag{9}$$

where the modular inverse exists as p is odd. Therefore replacing q in (8) with (9), leads to the quadratic modular equation (10)

$$D_i p^2 - 2^{Z_t} p - NE_i \equiv 0 \mod 2^n \tag{10}$$

It can be proved this equation has 16 roots, therefore, each sequence of (Z_i, X_i) pairs derived from a side-channel attack in this scenario will yield 16 candidates for p. For example, if the side-channel attack yields four unknown X_i, then the number of total candidates will be $16 \cdot 2^4 = 256$. This procedure shows how it is possible to adapt the *Partial* model to recover some input bits when both inputs are secret. Naturally, this method is use case dependent, but in our view, the most important part is that this model could also work when both inputs are unknown, therefore it should be considered in these scenarios.

6.2 Bulk experimental results

In this scenario the median of number of candidates after processing 1000 traces and applying the *Partial* model with both inputs unknown was 8192. The cause of this metric increase compared with the ECDSA case is due to three reasons: (i) The number of candidates increases exponentially with the number of unknown X_i; (ii) With the increase in the number of bits to recover (1024 instead of 512), there are more chances that unknown X_i occurs; (iii) The quadratic modular equation (10) yields 16 candidates per missing X_i combination. However, this increase does not have any practical effect on the attack, as 8192 candidates can be tested very quickly. These experiments confirmed the success rate of 99.1% of the attack when trying to recover the 1024 bits of a prime.

7 Mitigation and responsible disclosure

Regarding mitigations against the presented attacks, for the ECDSA case, the straightforward one is *completing* the already deployed nonce inversion countermeasure. This can be achieved by reducing the product $b = kt$ before calling the modular inversion function. This approach was followed by mbedTLS developers immediately after the disclosure.

On the other hand, for the RSA case a constant-time implementation of the binary GCD function and the modular inversion algorithm should be used, for instance following one of the proposals in [Bos14, SK18, BY19].

Another approach for the latter can be implementing the inversion $q^{-1} \bmod p$ using Fermat's Little Theorem (FLT) $q^{p-2} \bmod p$. However, in contrast to FLT usage in ECDSA for protecting the nonce, in this RSA use case the modulus is secret in addition to the exponent, therefore a side-channel secure modular exponentiation algorithm should be used. While this solution could have some performance penalty, it could be more attractive to library developers as it is more likely that they are more aware of (and have already deployed) side-channel secure modular exponentiation than inversion.

Following responsible disclosure procedures, we contacted the mbedTLS security team and shared our findings with them. We stressed the importance of the ECDSA vulnerability as the current status offers a false state of security as evidenced in a recent advisory from the mbedTLS security team. CVE-2019-18222 tracks the ECDSA vulnerability.

8 Conclusion

The most important conclusion of this research is that countermeasure implementations *must* follow their mathematical descriptions rigorously. Even when an alternative implementation is mathematically correct it can introduce or prevent the proper protection offered by the countermeasure, as demonstrated in this paper. In the case of the targeted ECDSA implementation, the protection of the value to be inverted by a multiplicative masking is performed on $\mathbb{Z}$ instead of $\mathbb{Z}_p^*$. This reduces the security of mbedTLS ECDSA implementation to an integer factorization problem.

On the other hand, every day there is more need for execution flow-independent *bignum* implementations (commonly miscalled *constant-time*). Often, only high-level cryptography algorithm implementations are protected with this feature, however, low-level layers are not, leading to execution-flow dependent inputs.

In this paper, we showed how a vulnerable binary GCD implementation leads to (at least) two vulnerabilities in the mbedTLS library, hence the importance of not only protecting high-level implementations, but also the low-level *bignum* ones.

Interrupt-driven attacks against cryptography algorithms are very powerful and provide high temporal resolution. In this research the vulnerable binary GCD primitive has very tight loops, however, not sufficient to stop interrupt-driven attacks based on the `SGX-Step` framework. Applications secured by Intel SGX should pay more care to side-channel threats, as OS-level adversaries have very powerful side-channels at their disposal.

Acknowledgments. We thank Tampere Center for Scientific Computing (TCSC) for generously granting us access to computing cluster resources.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 804476).

References

[AB20] Alejandro Cabrera Aldaya and Billy Bob Brumley. *CVE-2019-18222: research data and tooling.* Zenodo, Jan. 2020. DOI: 10.5281/zenodo.3605804. URL: https://doi.org/10.5281/zenodo.3605804.

[ACSS17] Alejandro Cabrera Aldaya, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. "SPA vulnerabilities of the binary extended Euclidean algorithm". In: *J. Cryptographic Engineering* 7.4 (2017), pp. 273–285. DOI: 10.1007/s13389-016-0135-4. URL: https://doi.org/10.1007/s13389-016-0135-4.

[AGS07] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. "New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures". In: *Cryptography and Coding, 11th IMA International Conference, Cirencester, UK, December 18-20, 2007, Proceedings.* Ed. by Steven D. Galbraith. Vol. 4887. Lecture Notes in Computer Science. Springer, 2007, pp. 185–203. DOI: 10.1007/978-3-540-77272-9_12. URL: https://doi.org/10.1007/978-3-540-77272-9_12.

[Ald+17] Alejandro Cabrera Aldaya, Raudel Cuiman Márquez, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. "Side-channel analysis of the modular inversion step in the RSA key generation algorithm". In: *I. J. Circuit Theory and Applications* 45.2 (2017), pp. 199–213. DOI: 10.1002/cta.2283. URL: https://doi.org/10.1002/cta.2283.

[Ald+19] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. "Port Contention for Fun and Profit". In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019.* IEEE, 2019, pp. 870–887. DOI: 10.1109/SP.2019.00066. URL: https://doi.org/10.1109/SP.2019.00066.

[AT07] Sarang Aravamuthan and Viswanatha Rao Thumparthy. "A Parallelization of ECDSA Resistant to Simple Power Analysis Attacks". In: *Proceedings of the Second International Conference on COMmunication System softWAre and MiddlewaRE (COMSWARE 2007), January 7-12, 2007, Bangalore, India.* Ed. by Sanjoy Paul, Henning Schulzrinne, and G. Venkatesh. IEEE, 2007. DOI: 10.1109/COMSWA.2007.382592. URL: https://doi.org/10.1109/COMSWA.2007.382592.

[BB05] David Brumley and Dan Boneh. "Remote timing attacks are practical". In: *Computer Networks* 48.5 (2005), pp. 701–716. DOI: 10.1016/j.comnet.2005.01.010. URL: https://doi.org/10.1016/j.comnet.2005.01.010.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. "Correlation Power Analysis with a Leakage Model". In: *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings.* Ed. by Marc Joye and Jean-Jacques Quisquater. Vol. 3156. Lecture Notes in Computer Science. Springer, 2004, pp. 16–29. DOI: 10.1007/978-3-540-28632-5_2. URL: https://doi.org/10.1007/978-3-540-28632-5_2.

[Bos14] Joppe W. Bos. "Constant time modular inversion". In: *J. Cryptographic Engineering* 4.4 (2014), pp. 275–281. DOI: 10.1007/s13389-014-0084-8. URL: https://doi.org/10.1007/s13389-014-0084-8.

[BT11] Billy Bob Brumley and Nicola Tuveri. “Remote Timing Attacks Are Still Practical”. In: *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*. Ed. by Vijay Atluri and Claudia Díaz. Vol. 6879. Lecture Notes in Computer Science. Springer, 2011, pp. 355–371. DOI: 10.1007/978-3-642-23822-2_20. URL: https://doi.org/10.1007/978-3-642-23822-2_20.

[BY19] Daniel J. Bernstein and Bo-Yin Yang. “Fast constant-time gcd computation and modular inversion”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.3 (2019), pp. 340–398. DOI: 10.13154/tches.v2019.i3.340-398. URL: https://doi.org/10.13154/tches.v2019.i3.340-398.

[CA+19] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Alvarez Tapia, and Billy Brumley. “Cache-Timing Attacks on RSA Key Generation”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.4 (2019), pp. 213–242. DOI: 10.13154/tches.v2019.i4.213-242. URL: https://tches.iacr.org/index.php/TCHES/article/view/8350.

[Can+19] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. “Fallout: Leaking Data on Meltdown-resistant CPUs”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM, 2019, pp. 769–784. DOI: 10.1145/3319535.3363219. URL: https://doi.org/10.1145/3319535.3363219.

[CD16] Victor Costan and Srinivas Devadas. “Intel SGX Explained”. In: *IACR Cryptology ePrint Archive* 2016.86 (2016). URL: http://eprint.iacr.org/2016/086.

[Che+19] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. “SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution”. In: *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*. IEEE, 2019, pp. 142–157. DOI: 10.1109/EuroSP.2019.00020. URL: https://doi.org/10.1109/EuroSP.2019.00020.

[Cor99] Jean-Sébastien Coron. “Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems”. In: *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES’99, Worcester, MA, USA, August 12-13, 1999, Proceedings*. Ed. by Çetin Kaya Koç and Christof Paar. Vol. 1717. Lecture Notes in Computer Science. Springer, 1999, pp. 292–302. DOI: 10.1007/3-540-48059-5_25. URL: https://doi.org/10.1007/3-540-48059-5_25.

[Dal+18] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. “CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 171–191. DOI: 10.13154/tches.v2018.i2.171-191. URL: https://doi.org/10.13154/tches.v2018.i2.171-191.

[Dan+13] Jean-Luc Danger, Sylvain Guilley, Philippe Hoogvorst, Cédric Murdica, and David Naccache. “A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards”. In: *J. Cryptographic Engineering* 3.4 (2013), pp. 241–265. DOI: 10.1007/s13389-013-0062-6. URL: https://doi.org/10.1007/s13389-013-0062-6.

[Fip] *Digital Signature Standard (DSS).* FIPS PUB 186-4. National Institute of Standards and Technology, 2013. DOI: 10.6028/NIST.FIPS.186-4. URL: https://doi.org/10.6028/NIST.FIPS.186-4.

[FV12] Junfeng Fan and Ingrid Verbauwhede. "An Updated Survey on Secure ECC Implementations: Attacks, Countermeasures and Cost". In: *Cryptography and Security: From Theory to Applications - Essays Dedicated to Jean-Jacques Quisquater on the Occasion of His 65th Birthday.* Ed. by David Naccache. Vol. 6805. Lecture Notes in Computer Science. Springer, 2012, pp. 265–282. DOI: 10.1007/978-3-642-28368-0_18. URL: https://doi.org/10.1007/978-3-642-28368-0_18.

[Gri+19] Iaroslav Gridin, Cesar Pereida García, Nicola Tuveri, and Billy Bob Brumley. "Triggerflow: Regression Testing by Advanced Execution Path Inspection". In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings.* Ed. by Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren. Vol. 11543. Lecture Notes in Computer Science. Springer, 2019, pp. 330–350. DOI: 10.1007/978-3-030-22038-9_16. URL: https://doi.org/10.1007/978-3-030-22038-9_16.

[HCP17] Marcus Hähnel, Weidong Cui, and Marcus Peinado. "High-Resolution Side Channels for Untrusted Operating Systems". In: *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.* Ed. by Dilma Da Silva and Bryan Ford. USENIX Association, 2017, pp. 299–312. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel.

[HPB04] Mustapha Hedabou, Pierre Pinel, and Lucien Bénéteau. "A comb method to render ECC resistant against Side Channel Attacks". In: *IACR Cryptology ePrint Archive* 2004 (2004), p. 342. URL: http://eprint.iacr.org/2004/342.

[HS01] Nick Howgrave-Graham and Nigel P. Smart. "Lattice Attacks on Digital Signature Schemes". In: *Des. Codes Cryptogr.* 23.3 (2001), pp. 283–290.

[Isl+19] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks". In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019.* Ed. by Nadia Heninger and Patrick Traynor. USENIX Association, 2019, pp. 621–637. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/islam.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings.* Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25. URL: https://doi.org/10.1007/3-540-48405-1_25.

[Koc96] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings.* Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Springer, 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9. URL: https://doi.org/10.1007/3-540-68697-5_9.

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. “CacheZoom: How SGX Amplifies the Power of Cache Attacks”. In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 69–90. DOI: 10.1007/978-3-319-66787-4_4. URL: https://doi.org/10.1007/978-3-319-66787-4_4.

[NS03] Phong Q. Nguyen and Igor E. Shparlinski. “The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces”. In: *Des. Codes Cryptogr.* 30.2 (2003), pp. 201–217. DOI: 10.1023/A:1025436905711. URL: https://doi.org/10.1023/A:1025436905711.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: The Case of AES”. In: *Topics in Cryptology - CT-RSA 2006, The Cryptographers’ Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*. Ed. by David Pointcheval. Vol. 3860. Lecture Notes in Computer Science. Springer, 2006, pp. 1–20. DOI: 10.1007/11605805_1. URL: https://doi.org/10.1007/11605805_1.

[Per05] Colin Percival. “Cache Missing for Fun and Profit”. In: *BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proceedings*. 2005. URL: http://www.daemonology.net/papers/cachemissing.pdf.

[PG+19] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. “Certified Side Channels”. In: *CoRR* abs/1909.01785 (2019). arXiv: 1909.01785. URL: http://arxiv.org/abs/1909.01785.

[PGB17] Cesar Pereida García and Billy Bob Brumley. “Constant-Time Callees with Variable-Time Callers”. In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 83–98. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia.

[Rie94] Hans Riesel. *Prime numbers and computer methods for factorization*. 2nd ed. Vol. 126. Progress in mathematics. 1994. ISBN: 978-0-8176-8298-9. DOI: 10.1007/978-0-8176-8298-9. URL: https://doi.org/10.1007/978-0-8176-8298-9.

[Rya19] Keegan Ryan. “Return of the Hidden Number Problem: A Widespread and Novel Key Extraction Attack on ECDSA and DSA”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.1 (2019), pp. 146–168. DOI: 10.13154/tches.v2019.i1.146-168. URL: https://doi.org/10.13154/tches.v2019.i1.146-168.

[Sch+19] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. “ZombieLoad: Cross-Privilege-Boundary Data Sampling”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM, 2019, pp. 753–768. DOI: 10.1145/3319535.3354252. URL: https://doi.org/10.1145/3319535.3354252.

[Sil19] Fábio Silva. *mbedtls-compat-sgx: mbed TLS Intel(r) SGX Compatibility Layer*. https://github.com/ffosilva/mbedtls-compat-sgx/tree/c66e974. 2019.

[SK18] Erkay Savas and Çetin Kaya Koç. “Montgomery inversion”. In: *J. Cryptographic Engineering* 8.3 (2018), pp. 201–210. DOI: 10.1007/s13389-017-0161-x. URL: https://doi.org/10.1007/s13389-017-0161-x.

[Ste67] Josef Stein. “Computational problems associated with Racah algebra”. In: *Journal of Computational Physics* 1.3 (1967), pp. 397–405. ISSN: 00219991. DOI: 10.1016/0021-9991(67)90047-2. URL: https://doi.org/10.1016/0021-9991(67)90047-2.

[TTA18] Akira Takahashi, Mehdi Tibouchi, and Masayuki Abe. “New Bleichenbacher Records: Fault Attacks on qDSA Signatures”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 331–371. DOI: 10.13154/tches.v2018.i3.331-371. URL: https://doi.org/10.13154/tches.v2018.i3.331-371.

[Tuv+18] Nicola Tuveri, Sohaib ul Hassan, Cesar Pereida García, and Billy Bob Brumley. “Side-Channel Analysis of SM2: A Late-Stage Featurization Case Study”. In: *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 147–160. DOI: 10.1145/3274694.3274725. URL: https://doi.org/10.1145/3274694.3274725.

[Val+16] Luke Valenta, Shaanan Cohney, Alex Liao, Joshua Fried, Satya Bodduluri, and Nadia Heninger. “Factoring as a Service”. In: *Financial Cryptography and Data Security - 20th International Conference, FC 2016, Christ Church, Barbados, February 22-26, 2016, Revised Selected Papers*. Ed. by Jens Grossklags and Bart Preneel. Vol. 9603. Lecture Notes in Computer Science. Springer, 2016, pp. 321–338. DOI: 10.1007/978-3-662-54970-4_19. URL: https://doi.org/10.1007/978-3-662-54970-4_19.

[VB+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 991–1008. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[VBPS17] Jo Van Bulck, Frank Piessens, and Raoul Strackx. “SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control”. In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. ACM, 2017, pp. 1–6. DOI: 10.1145/3152701.3152706. URL: https://doi.org/10.1145/3152701.3152706.

[WSB18] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. “Single Trace Attack Against RSA Key Generation in Intel SGX SSL”. In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*. Ed. by Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim. ACM, 2018, pp. 575–586. DOI: 10.1145/3196494.3196524. URL: http://doi.acm.org/10.1145/3196494.3196524.

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems”. In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 640–656. DOI: 10.1109/SP.2015.45. URL: https://doi.org/10.1109/SP.2015.45.

[Xia+17] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. “STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. ACM, 2017, pp. 859–874. DOI: `10.1145/3133956.3134016`. URL: `https://doi.org/10.1145/3133956.3134016`.

[YF14] Yuval Yarom and Katrina Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. USENIX Association, 2014, pp. 719–732. ISBN: 978-1-931971-15-7. URL: `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom`.

[Int19] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual*. 2019.