IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2021, No. 3, pp. 28–59. DOI:10.46586/tches.v2021.i3.28-59

Online Template Attacks: Revisited

Alejandro Cabrera Aldaya and Billy Bob Brumley

Tampere University, Tampere, Finland
alejandro.cabreraaldaya@tuni.fi,billy.brumley@tuni.fi

Abstract. An online template attack (OTA) is a powerful technique previously used to attack elliptic curve scalar multiplication algorithms. This attack has only been analyzed in the realm of power consumption and EM side channels, where the signals leak related to the value being processed. However, microarchitecture signals have no such feature, invalidating some assumptions from previous OTA works.

In this paper, we revisit previous OTA descriptions, proposing a generic framework and evaluation metrics for any side-channel signal. Our analysis reveals OTA features not previously considered, increasing its application scenarios and requiring a fresh countermeasure analysis to prevent it.

In this regard, we demonstrate that OTAs can work in the *backward* direction, allowing to mount an *augmented* projective coordinates attack with respect to the proposal by Naccache, Smart and Stern (Eurocrypt 2004). This demonstrates that randomizing the initial targeted algorithm state does not prevent the attack as believed in previous works.

We analyze three libraries `libgcrypt`, `mbedTLS`, and `wolfSSL` using two microarchitecture side channels. For the `libgcrypt` case, we target its EdDSA implementation using Curve25519 twist curve. We obtain similar results for `mbedTLS` and `wolfSSL` with curve `secp256r1`. For each library, we execute extensive attack instances that are able to recover the complete scalar in all cases using a single trace.

This work demonstrates that microarchitecture online template attacks are also very powerful in this scenario, recovering secret information without knowing a leakage model. This highlights the importance of developing *secure-by-default* implementations, instead of fix-on-demand ones.

Keywords: applied cryptography · public key cryptography · elliptic curve cryptography · side-channel analysis · online template attacks · microarchitecture attacks · libgcrypt · mbedTLS · wolfSSL

1 Introduction

Side-channel attacks are a common threat to computing platforms nowadays. Since the pioneering works by Kocher [Koc96], several kinds of leaky channels have been discovered. For instance, execution time, power consumption, and in the microarchitecture realm cache-timings, sequence of page accesses [Koc96, KJJ99, Per05, YF14, XCP15].

Several techniques have been proposed to exploit said channels. Among them are template attacks, that assume the adversary can profile the targeted implementation side-channel signals [MOP07]. Chari, Rao, and Rohatgi [CRR02] introduced template attacks in the context of power consumption side channels, consisting of three phases: (i) templates building, (ii) target trace capturing, and (iii) template matching. The template building phase is performed on an attacker-controlled implementation very similar to the targeted one. During this phase, the attacker profiles leakage by building leakage templates.

Received: 2020-10-15 Accepted: 2020-12-15 Published: 2021-07-09

Note that this attack description by Chari, Rao, and Rohatgi [CRR02] considered templates built *before* capturing the target trace. Later, Medwed and Oswald [MO08] challenged this order, with template attacks targeting ECDSA scalar multiplication. The authors analyzed several scenarios related to this order, proposing an *on-the-fly* template building [MO08, Sect. 5.3]. That is, creating templates *after* capturing the target trace.

Related to this template attack phases order, Batina et al. [Bat+14] proposed what is known in the literature as Online Template Attacks (OTA), also building the templates *after* capturing the target trace, similar to [MO08]. The main difference between [MO08] and [Bat+14] is how templates are constructed. Medwed and Oswald [MO08] used "vertical" power consumption leakage while Batina et al. [Bat+14] used "horizontal" leakage (see [Cla+10] for definitions). However, despite this difference both approaches agree on the moment when templates are built, and template attacks that follow this approach are labeled as OTAs.

Creating template traces in advance is feasible when the number of possible templates to create is small. For instance, a binary exponentiation algorithm where templates are used to distinguish a single branch result only requires two templates [CRR02]. However, when the number of leaking features to detect is large, e.g. coordinates of an elliptic curve point, the number of different templates could be infeasible to generate in advance. This scenario is where OTAs enter into play by capturing templates on-demand/*online* based on a secret guess [Bat+14].

The original OTA technique was proposed and applied in several works using power consumption/EM side channels. Dugardin et al. [Dug+16] demonstrated a practical OTA on PolarSSL scalar multiplication using EM signals. Regarding power signals, Sandmann [San18] targeted FourQ scalar multiplication and Roelofs [Roe19] instead ECDSA.

Luo [Luo18] used the OTA approach, but generated template traces using a leakage model instead of being captured on a similar device. This approach requires a leakage model rather than a template device, but nevertheless adds attack flexibility wrt the original description [Bat+14].

Bos et al. [Bos+18] analyzed the feasibility of OTAs on the Frodo post-quantum proposal. The authors employed a power consumption trace emulator for modeling both attack and template traces instead of real devices [MOW17]. It would be interesting to study attack feasibility using such an emulator to gather template traces while the target trace belongs to a real device, a gap that this paper fills.

One common property in these works is they use *starting algorithm state* (e.g. initial elliptic curve point coordinates) as attack input. This trend is likely motivated by the fact that the OTA technique was originally presented in this setting, where the starting value of an accumulator is known to the attacker [Bat+14]. However, in this paper we challenge this assumption and show it is not an attack requirement, considerably expanding its applications.

Regarding microarchitecture side-channel attacks, several template attacks have been proposed in the literature [BH09, ABG10, GSM15, WHS12, Che+19, Du+15, Bha+20]. Brumley and Hakala [BH09] developed data cache-timing templates to attack ECDSA scalar multiplication using Prime+Probe, while Acıiçmez, Brumley, and Grabher [ABG10] extended the concept to the L1 instruction cache. Similarly Gruss, Spreitzer, and Mangard [GSM15] showed the feasibility to construct templates from Last Level Cache timings using Flush+Reload to attack AES T-Box implementations. Bhattacharya et al. [Bha+20] constructed templates from performance counters related to the branch prediction unit (BPU) to attack elliptic curve scalar multiplication.

However, regardless of exploited microarchitecture components and applications, all these works follow the original template attack approach by Chari, Rao, and Rohatgi [CRR02], where templates are built *before* capturing the target trace. Based on extensive literature review, it seems microarchitecture-based OTA related works are a research gap.

Therefore, it remains unknown how OTAs apply in the microarchitecture realm, especially considering that the original OTA description was motivated by power consumption side-channel signals that leak differently from microarchitecture ones due to their different natures. The original OTA technique implicitly assumes information on the side-channel signals that might not be present in microarchitecture based ones.

In this paper, we start to fill this gap, not only demonstrating the effectiveness of OTAs on commonly used libraries, but revisiting the original OTA description, demonstrating it is more flexible than previously believed. This leads to new application scenarios regardless of the exploited side channel. Section 2 presents background on elliptic curve scalar multiplication algorithms and microarchitecture side channels. Section 3 revisits the original OTA description, proposing a generic framework for its analysis. Section 3.4 analyzes OTA countermeasures considering its previous description and the proposed one. Later, Section 4 instantiates the proposed OTA framework in the microarchitecture realm, evaluating this attack in three software libraries. Section 5 presents full end-to-end OTA experiments on these libraries, capable of recovering the secret in all instances using a single trace. We present conclusions in Section 6.

Our main contributions are as follows: (i) we revisit the original OTA concept, revealing properties, application scenarios, and evaluation metrics not considered before; (ii) we discover that a countermeasure previously proposed to prevent OTAs could be insufficient; (iii) we propose an *augmented projective coordinates attack* that reduces the number of required traces from thousands to one, wrt the original attack of Naccache, Smart, and Stern [NSS04]; (iv) we present the first microarchitecture OTA analysis; (v) we develop a tool to detect OTA-based leakages in software libraries using different attack vectors; (vi) we demonstrate practical microarchitecture OTAs on three widely used software libraries. The first three contributions revisit the original OTA description and are side-channel independent, whereas the others apply the new methodology to microarchitecture side channels.

2 Background

2.1 Elliptic curve scalar multiplication

Scalar multiplication is one of the main operations in ECC. It computes $P = kG$ for a scalar k and an elliptic curve point G, equivalent to aggregate k times G with itself. Regarding non-quantum elliptic curve cryptosystems, this operation plays a crucial role because inverting it (i.e. recovering k knowing P and G) requires solving the Elliptic Curve Discrete Logarithm Problem (ECDLP), considered hard for well-chosen elliptic curves [HMV04].

On the other hand, scalar multiplication is the most time-consuming operation in these cryptosystems. Among the several approaches to implement it, performance was initially the main goal. But after the groundbreaking work on side-channel analysis by Kocher [Koc96] in 1996, resistance against these attacks is considered a must.

Several approaches exist for computing a scalar multiplication, for instance: double-and-add, Montgomery ladder, window-based methods, etc. [JY02, HPB05, Joy07, HMV04]. Regardless of their differences, all of them share the property that at every iteration a *state* is updated based on secret data. A state can be a single elliptic curve point accumulator (e.g. double-and-add) or a set of them (e.g. Montgomery ladder). This property of scalar multiplication algorithms and the state concept play a crucial role in the OTA analysis presented in Section 3. In this section, we define an abstract scalar multiplication description (Algorithm 1) to represent all of them as it fits better for a generic OTA description. Later during the real-world OTAs in Section 5, we instantiate this algorithm using concrete implementations. Algorithm 1 consists of four generic operations.

Algorithm 1: *Generic scalar multiplication*

Input: Integer k and curve generator G
Output: $P = kG$

```
1 K = Encode(k)
2 S'_0 = Init(G)
3 for K_i in K = {K_1, K_2, .., K_n} do
4 |   S_i = Select(S'_{i-1}, K_i)
5 |   S'_i = Process(S_i)
6 P = Finalize(S'_n)
7 return P
```

`Encode`: This operation encodes the scalar k in a list $K = \{K_1, K_2, .., K_n\}$, where at every algorithm iteration at least one element of K will be processed. For instance, in the *double-and-add* algorithm, K is the binary representation of k. The encoding defines how many possibilities exist for each K_i. The only requirement for this step is that it can be inverted, i.e. it is possible to recover k from K.

`Init`: This operation initializes the first state S'_0 using the point G, as well as performs coordinate conversion and precomputation.

`Select`: This operation defines the state S_i to be processed in the current iteration. This selection depends on K_i and the previous iteration computed state S'_{i-1}. Sometimes this operation is implemented using conditional branches like in the classic *double-and-add* algorithm, making it vulnerable to trivial side-channel attacks. We assume that the implementation of this operation does not leak K_i. This is a common assumption for scalar multiplication algorithms protected against these trivial attacks (e.g. balanced K_i-related branches). Regarding our research, we are more interested in subtle leakages lurking in the ECC hierarchy lower layers, e.g. the finite field implementation.

`Process`: This is the most important operation regarding this research. This step processes the current iteration state S_i, generating current iteration *resulting* state S'_i. This operation is composed by elliptic curve point *doubling* and *addition* executions, according to the curve/coordinates formulae. OTAs aims at identifying which S_i was processed at each iteration, allowing to recover the corresponding K_i. The adversary has freedom to select the `Process` operation. For instance, it can be composed by all point operations in the curve formulae, or only a subset of them. Additionally, the position of `Select` wrt to `Process` could vary between implementations, however adapting the attack to these cases is immediate.

`Finalize`: This operation processes the last computed state S'_n just before returning the output point P. For instance, projective to affine coordinates conversion is usually executed here [HMV04, NSS04].

In addition to the scalar multiplication algorithm, there exist several point coordinate representations that define the formulae employed for computing point *doubling* and *addition* on a given elliptic curve [HMV04]. Regardless of the selected coordinate system, these operations usually require several modular computations (i.e. additions, multiplications, divisions, etc.) performed on multiprecision integers (*bignum*). Therefore an ECC implementation consists of several layers, and eliminating side-channel leakages in all of them is a challenging task.

2.2 Microarchitecture side-channels

Several microarchitecture-based side channels have been discovered, where execution time, cache-timing, and port contention are a few examples [Koc96, Per05, YF14, Ge+18, Ald+19]. Despite technique details, almost all of them aim at exploiting *address-based*

information leakage [AK09, Ge+18, Sze19, Wei+18]. That is, a leak produced by *secret-dependent memory accesses*. When said dependency produces different execution paths, it is labeled as control-flow leakage, whereas a data leak exists if a data-memory access is secret-dependent [Wei+18].

Another kind of leakage that can exist in a computing platform is *value-based leakage*. For instance, some CMOS devices leak the Hamming weight of the processed values through their power consumption [KJJ99, PMO07]. However, microarchitecture side channels that exploit value-based leakages are not common at all [Cop+09, Ge+18, Sze19]. This difference between power and microarchitecture-based side channels challenges the application of OTAs in the microarchitecture realm because the original OTA description inherently assumes that value-based leakage exists in the exploited channel [Bat+14].

The constant-time feature is often used to label a software implementation as side-channel secure. However, regarding address-based leakages, a more accurate term is constant-address implementation [Ge+18]. Nevertheless, we use the term constant-time consistent with the common trend in the literature, but referring to the latter.

OTAs highlight the need to develop constant-time implementations. Side-channel secured scalar multiplication algorithms remove secret dependent branches at its highest level. However, is it the only layer that needs to be constant-time? It is common that the lowest *bignum* arithmetic indeed contains branches, especially when the implementation was inherited from code developed when side-channel leakages were not a concern. For instance, OpenSSL, libgcrypt, mbedTLS, wolfSSL are open-source libraries with this property[1].

Address-based memory leakages often exist at different granularities, and several microarchitecture side channels have been discovered to exploit them. The zoo of available attacks is diverse with different properties like threat model, granularity, targeted information, noise level, etc. The next section presents a generic OTA framework that aims to be applicable to any side channel (e.g. power consumption and microarchitecture signals), and later in Section 4 we use it to analyze three libraries employing two microarchitecture ones.

3 Reexamining the OTA Technique

This section revisits the OTA description, proposing a new framework for its analysis and evaluation, demonstrating some features not considered before. For this analysis, we use the generic scalar multiplication (Algorithm 1) described in Section 2.

Abstractly, the OTA procedure consists of the following steps. We discuss differences regarding the original proposal in the following sections.

1. Capture one side-channel trace I while the targeted implementation processes some secret k.

2. Split the target trace in iterations $I = \{I_1, I_2, .., I_n\}$, where each I_i corresponds to the processing of state S_i according to K_i. Therefore, it is expected I_i corresponds to the `Process` operation in Algorithm 1.

3. *Extend:* Enumerate every possible K_i and—according to the `Select` operation—compute every possible S_i using the previous known state. For each computed S_i, capture a side-channel trace corresponding to the execution of `Process` (S_i). This produces a template trace $T_{i,j}$ for every possible K_i represented by j.

4. *Prune*: Filter out those $T_{i,j}$ that do not *match* I_i.

[1] OpenSSL and wolfSSL are transitioning their *bignum* arithmetic to provide constant-time security at all layers.

5. For all $T_{i,j}$ surviving the pruning phase, repeat for the next iteration (starting from step 3), backtracking if multiple matches are found.

6. *Terminate*: Finish the attack after recovering sufficient K_i.

This algorithm follows an *extend-and-prune* approach, where step 3 *extends* the number of candidates and step 4 *prunes* unlikely ones. The OTA idea is to identify which state S_i was processed at an iteration and derive K_i from it.

Extend. This step can be performed using different approaches related to how much control the adversary has over the template implementation. For instance, which inputs does it accept? Can the adversary modify it?

In this regard, the ideal scenario takes place when the attacker has access to a template implementation where she can obtain traces of the `Process` operation for any state S_i. However, in practice sometimes it is not available due to API limitations. For instance, in embedded systems it is not common that a device exports an API for the `Process` operation alone, but a high level one for the scalar multiplication, or even worse, a protocol one (e.g. ECDSA signature generation/verification). In addition, even if the `Process` API is available, the state representation may differ from the targeted one. For more details on techniques in each scenario, consult [MO08, Bat+14, Pap19, Roe19].

These limited scenarios are more likely on embedded systems, in contrast to the microarchitecture realm. For instance, a common threat model is the adversary and victim share the same computing platform, attacking a shared cryptography library (e.g. cache attacks). In this scenario, the attacker can use the shared library binary, or if it is open source she can construct a fork with a more flexible API. In our practical experiments in Section 5 we explore this path, showing this OTA flexibility in the microarchitecture realm.

Prune. This phase controls the search tree growth based on a *match score*. The signal nature determines the method employed to assess if a template trace matches the targeted one. Pearson's correlation coefficient has been used in power consumption OTAs [Bat+14, Dug+16, Roe19], whereas Ozgen, Papachristodoulou, and Batina [OPB16] explored other classification algorithms. Regardless of the employed approach, this step should minimize the probability of pruning the good candidate, while maximizing the probability of pruning incorrect ones.

In this paper, we only consider algorithms that does not prune the correct solution (i.e. Pr[false negative] = 0). Dealing with "false negatives" requires a highly application-dependent error correction procedure. For instance, if multiple copies of the target trace can be captured, this redundancy can help thwart errors, but not all cryptosystems allow this. We leave the analysis of OTAs combined with error-correction approaches for future work.

Terminate. Algorithm termination depends on the attacked cryptosystem and certainty about the recovered data. Some cryptosystems like ECDSA break when knowing (with certainty) a small number of bits of the scalars used to generate a set of signatures [HS01, NS03]. Therefore, if enough K_i are reliably recovered such that the number of bits of k that they reveal are sufficient to apply said cryptanalysis, there is no need to recover the full scalar [Dug+16].

On the other hand, some cryptosystems like the Edwards curve DSA variant (EdDSA) are designed to prevent such cryptanalysis [Ber+12]. For this scenario, OTAs should recover sufficient bits such that solving the ECDLP is feasible, where naturally recovering the full scalar is also an option.

Partial scalar recovery using OTAs ideally requires a side channel that allows recovering each K_i with absolute certainty. Otherwise, it increases the number of iterations to process, expecting that the pruning removes the incorrect ones [Dug+16]. If a full scalar recovery is desired, the attacker can implement it using depth-first search, considering that the correct solution will survive every pruning step and it is more likely that incorrect ones do not. However, if the pruning phase produces many false positives, the attacker should test each solution produced by the algorithm until finding the correct one.

3.1 Attack input and direction

The OTA concept was presented as a *chosen-input* attack [Bat+14]. However, it is worth highlighting that this requirement is about the template implementation, not the targeted one. This distinction is important because it is considered in the OTA literature that the attacker needs to know the input point to determine the initial state S'_0 and subsequent ones [MO08, Bat+14, Dug+16, MOW17, Luo18, Roe19].

We revisit this claim, discovering that it is not a strict attack requirement. Instead, we propose the following: *The OTA technique applies if the adversary knows that a state processed by the target implementation belongs to a known set of states with feasible enumeration.* Note additionally that it is a sufficient condition, not a necessary one (Section 3.2 expands).

The initial state S'_0 case is covered by previous works. Moreover, the OTA description given above also applies if the adversary knows any S_i and starts the attack there. Such a state could be obtained by a complementary side-channel attack. We have not found an implementation or previous works with such leakage that allows recovering an intermediate state. However, the last state case (S'_n in Algorithm 1) requires more attention.

Knowing the last state might seem harmless regarding previous work on OTAs because it recovers the K_i reproducing the targeted algorithm execution (i.e. *forward* direction). Hence, no state is processed after computing the last state (cf. Algorithm 1). However, we challenge this claim by answering: Could an OTA be executed in the *backward* direction?

Following the OTA description given above, if the adversary knows the last state S_n, she can compute S_{n-1} by inverting the `Process` operation. Depending on the curve formulae, this inversion often involves computing modular roots, possibly obtaining more than one candidate for S_{n-1} [NSS04]. Then the adversary can capture template traces for every computed S_{n-1} and prune those not matching the observed iteration trace I_n. This will allow determining the processed state and eventually the corresponding K_i. Repeating this process for previous iterations could allow recovering all K_i.

This demonstrates OTAs can be applied in the *backward* sense, reversing the target trace iterations order, i.e. $I = \{I_n, I_{n-1}, \ldots, I_1\}$ and using the last computed state S'_n as the attack starting state. This variant could be harder to solve, because each guess for K_i might generate more than one candidate due to the modular roots, thus increasing the number of candidates per iteration and the pruning phase must filter out more candidates. Section 5 demonstrates this attack in two different scenarios that recover the scalar using a single trace, showing feasibility in practice.

Very related to this idea is the *projective coordinates attack* proposed by Naccache, Smart, and Stern [NSS04]. The authors demonstrated the projective representation of the scalar multiplication output point could reveal information about the scalar. The approach is purely algebraic and relies on—when inverting `Process` based on a guess about K_i—no modular roots existing, concluding that said K_i is incorrect. However, due to modular root properties, the search tree explodes very quickly, hence the number of bits that can be recovered is small [NSS04, APGB20].

This attack requires knowing the projective coordinates of the output point (e.g. last state). For instance, this can be obtained using a complementary attack: Maimut et al. [Mai+13] used a fault injection attack while Aldaya, Pereida García, and Brumley

[APGB20] used a microarchitecture side channel. Executing an OTA in the *backward* direction (whereas previous works only considered the *forward* case) could allow recovering all bits of the scalar using a single trace. Therefore, it can be considered as an *augmented projective coordinates attack*.

3.2 Revisited OTA requirements

In this section, we revisit the original OTA requirements, allowing to determine if a scenario could be targeted by an OTA. Evidently, this does not imply the attack will succeed, but allows developers to know if their implementation should take OTAs into account. We define the following OTA requirements:

> *Distinguisher:* A leak of the `Process` implementation can be used as a state distinguisher.
>
> *Reproducible:* The adversary has access to a template implementation that will process the same data as the target implementation for the same input.

The *Distinguisher* requirement has been assumed in previous works due to power consumption side channel properties. Power consumption signals leak about the values being processed, therefore this requirement only depends on the signal-to-noise ratio. However, in the microarchitecture realm, value-based leakages are not common, therefore the attacker should rely on address-based leakage (e.g. non constant-time code). Section 3.3 discusses this requirement and proposes some metrics to evaluate how well a leak of a `Process` implementation can be used as a state distinguisher.

Regarding the *Reproducible* requirement, depending on which input the template implementation accepts, the attack could be either state- or scalar-based.

State-based. Previous works on OTAs assume that an attacker knows the first state. Section 3.1 extends this to *any* state in *both* forward and backward directions. This scenario can be generalized even further to the case no state is known, but the attacker knows a set of states where one of them is the correct one. Intuitively, if said set can be feasibly enumerated, the adversary can perform the attack for every state in it.

Scalar-based. If the template implementation allows executing the same scalar multiplication algorithm as the target one, the adversary can guess the first processed K_i. For instance, when using a binary algorithm where each K_i represents the bit i of k, the attacker can use the template implementation to capture the traces for $[0]G$ and $[1]G$ and then compare with the target one. If only one matches the target trace, the attacker learns a bit of k. The next iteration builds templates using previously learned information on k. In this scenario, the attacker does not require a known state, but expects that one of the states processed in template traces corresponds to the target one, i.e. the adversary can *reproduce* the target implementation execution.

The scalar-based approach only works in the forward direction because the processed states depend on previous ones. During our experiments, we use both approaches to recover the scalar. As mentioned before, the fulfillment of these requirements does not guarantee attack success. The next section proposes some metrics to evaluate an implementation regarding OTAs.

3.3 Evaluation metrics

The original OTA paper claims it could recover a full scalar employing one template trace per key bit [Bat+14]. However, this claim only holds when targeting a binary scalar

multiplication algorithm (K_i is a binary value) and if the distributions for the *matching scores* for correct and incorrect templates are well-separated. Therefore, the performance of an OTA depends on the exploited side channel, its characteristics such as signal-to-noise ratio, error resilience, etc. Previous works on OTAs only consider power-based side channels, making assumptions about the signal that may not hold for other side channels like microarchitecture-based ones.

Following the generic scalar multiplication in Algorithm 1, we represent an implementation of the `Process` operation using (1), where L_i is a side-channel trace resulting from its execution with S_i as input.

$$\texttt{Process}(S_i) \rightsquigarrow L_i \tag{1}$$

Note L_i is equivalent to template traces notation $T_{i,j}$. However, for the sake of notation simplicity, we rename it as L_i as it fits better the following analysis where its K_i relation is meaningless. The objective of each OTA iteration is to detect which template trace matches with the target one. Intuitively, the better L_i represents a state S_i the better the attack will perform.

The ideal attacker scenario happens when there is one and only one L_i for every S_i and vice versa (i.e. bijective sets). This implies leakage determinism, that is, every time S_i is processed the same L_i will be observed. On the other hand, if the set formed by all possible L_i only has one element, it is not possible to distinguish any S_i using L_i. Therefore, an implementation with this feature can be considered OTA-safe.

Regarding power consumption side channels, ideal and safe scenarios are not common due to the channel characteristics. Power consumption signals contain random noise, and the ideal scenario can only be achieved if this noise is removed completely, which is usually not possible in practice [MOP07, Luo18]. At the same time, a safe scenario (as defined before) is challenging to achieve because power signals inherently contain value-related leaks (Section 2), and preventing those requires specific hardware design [MOP07]. Therefore, generally speaking, two different states will inherently generate different signals. Hence, a power side-channel OTA adversary usually handles scenarios that lay between these boundary cases. On the other hand, both ideal and safe scenarios can occur in the microarchitecture realm, taking advantage of the latter's benefits and suffering the curse of the former (see countermeasure analysis in Section 3.4).

We propose some metrics and a procedure allowing a security auditor or an attacker to evaluate if an implementation could be vulnerable to OTAs. Figure 1 shows a flow diagram to guide the evaluation process. The first step is to estimate if the targeted implementation is deterministic, taking into account the considered side channels in the threat model. Determinism can be estimated by capturing a set of traces with identical inputs and comparing them. Ideally this should be done over the entire scalar multiplication algorithm, to detect which algorithm operations have deterministic behavior. For instance, non-determinism during the `Init` operation is good evidence there is a state randomization countermeasure in place [Cor99], while a deterministic `Init` and `Process` could be dangerous.

Non-deterministic case. The determinism test defines which main branch of the evaluation flow in Figure 1 should be followed. If no determinism is observed, the right branch should be taken. Non-determinism implies the *Reproducible* requirement is not perfectly fulfilled. Therefore, the template matching algorithm performance should be considered.

For this task, we propose to estimate the probability that said algorithm produces false negatives ①. As discussed previously, if it is not zero, the attacker must deal with errors in the recovery process. On the other hand, if $\Pr[FN] = 0$ the evaluator knows the solution will remain in the tree. Therefore, in this case the `Process` leakage could be *reproduced* somehow.

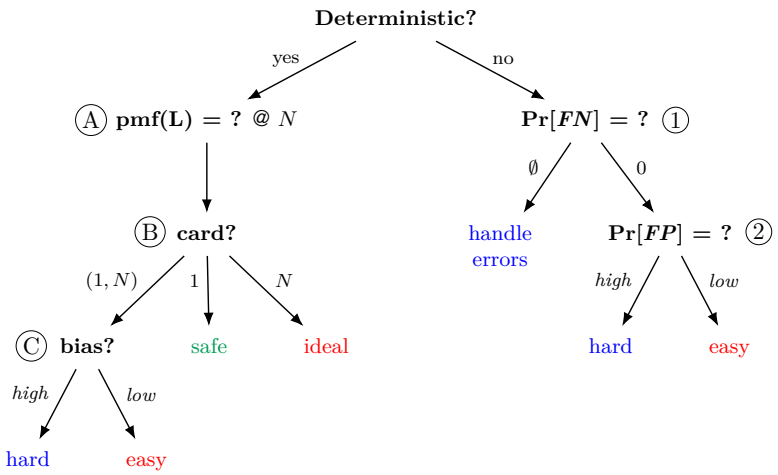

Figure 1: OTA evaluation flow (attacker perspective).

The last metric for the non-deterministic case allows to estimate the fulfillment of the *Distinguisher* requirement. For this purpose, the false positive probability $\Pr[FP]$ can be estimated ②, i.e. probability that the matching algorithm incorrectly classifies a template trace as a match. This probability defines the number of branches in the solution tree, therefore it approaches the ideal attacker scenario as the value decreases, and vice versa. How many false positives the attack can handle depends on the computing resources available to the adversary.

The right branch of Figure 1 is likely to occur in power consumption side channels. Previous OTA works developed attacks with $\Pr[FN] = 0$ and low $\Pr[FP]$ [Bat+14, Dug+16]. In general, this branch fits better for noisy side channels. The left branch of this evaluation flow covers the case where determinism is observed in the targeted implementation using a particular side channel. Therefore, the *Reproducible* requirement is perfectly fulfilled in this scenario.

Deterministic case. In this scenario, it is possible to evaluate how well a side-channel trace of `Process` can be used as a state *Distinguisher*. For this task, we propose to estimate the probability mass function (pmf) of the leakages produced by the `Process` operation (L_i in (1)). We denote this set as $\boldsymbol{L}$ Ⓐ.

The number of possible states is huge, and obtaining an L_i for each of them is infeasible. Therefore, we estimate the $pmf(\boldsymbol{L})$ using several (N) randomly generated S_i, allowing an estimate of the *cardinality* of $\boldsymbol{L}$ (i.e. number of different outcomes), and how *biased* its distribution is.

Figure 1 Ⓑ shows the conclusions drawn with this estimation. If the cardinality is one, it means all processed states produced the same leakage. Therefore, a state cannot be distinguished using the employed side channel, and the implementation can be considered OTA-safe. On the contrary, if the number of observed leakages is equal to the number of states used for the estimation (N), it implies an *ideal* attacker scenario. This means it is very likely an adversary can use the exploited side channel as a perfect state distinguisher.

On the other hand, if the cardinality is between these corner cases, the distribution bias will determine the computing effort in finding the solution Ⓒ. If $pmf(\boldsymbol{L})$ is highly biased towards one outcome (L_i), the number of false positives will increase, and the search tree grows accordingly. On the other hand, if no such high bias exists, then solving the problem is easy. What is considered a high bias depends on attacker computation resources. During our extensive experiments (detailed later) we recover full 256-bit scalars

with bias as high as 62% using a desktop workstation.

What could be considered a state *Distinguisher*? Suppose a state consists of an elliptic curve point, then very deep in the *bignum* implementation of the targeted implementation there is a conditional operation that produces two execution branches based on the evenness of the point coordinate x. Therefore, wlog, assuming a random state, said control-flow leak allows splitting the state space in two equiprobable halves, i.e. such leakage can be used to distinguish if the processed state contains an even x or not. This case will produce an equiprobable $pmf(\boldsymbol{L})$ with two outcomes, yet even this tiny *state distinguisher* is sufficient to succeed using the OTA technique (see Section 5.3 for experiment results).

Note that during an OTA, the adversary is not required to known a model of the exploited leakage, i.e. how a deep *bignum* control-flow leak relates to the processed secret. Instead, the attacker blindly searches for distinguishable features in the side-channel signals that fulfill OTA requirements. This blind approach allows to evaluate state-dependent leakages at any layer of the ECC implementation, no matter how deep they are in the hierarchy.

3.4 Mitigation analysis

To prevent OTAs, it should be sufficient to eliminate one of its requirements in the implementation. For instance, if a call to `Process` produces a random signal, it is not possible to *reproduce* the leakage produced by a given state. Projective coordinates randomization of the starting state can be used for this purpose, as proposed in the original OTA paper [Cor99, Bat+14]. However, according to the analysis presented in Section 3.1, it should also be applied *after* the scalar multiplication to prevent a *backward* OTA (Section 5.2 demonstrates this countermeasure is useless if only executed at the beginning). Furthermore, ideally it should be applied to every input state of `Process`, thus avoiding a potential OTA based on intermediate states.

Another line of defense is based on thwarting the *Distinguisher* requirement. That is, prevent a side-channel leak from being used to distinguish the processed state. Note that this mitigation was not considered in previous works because it is not easy to achieve in the presence of value-based side channels like power consumption. However, in the microarchitecture realm, constant-address code should be sufficient to meet this requirement. Naturally, this countermeasure will only be effective if it is applied to the entire implementation stack.

4 Microarchitecture OTAs

In this section, we instantiate the OTA framework from Section 3 in the microarchitecture realm. In this context, a leak can be divided in three groups based on its nature: (i) executed control-flow, (ii) data accessed, and (iii) value processed. Regarding microarchitecture side channels, the most common leakages are produced by *secret-dependent memory accesses* (first two cases), while *value*-based leakages are less common (see Section 2).

Control-flow based leakages are observed when program execution flow depends on secret data, e.g. due to a conditional instruction result. While different side channels could be used to exploit such leakages, in this research we focus our experiments on two approaches, proved very useful to target Intel SGX enclaves. This scenario is very interesting because SGX technology does not offers protections against side-channel attacks, delegating such defenses to developers [CD16]. Therefore, analyzing *secured* scalar multiplication implementations in open-source libraries regarding OTAs is interesting, to assess their resilience to this attack.

Intel SGX aims at providing confidentiality and integrity of software running in Intel microprocessors even if the OS is under attacker control. Following this threat model, the

controlled-channel attack proposed by Xu, Cui, and Peinado [XCP15] provides access to the sequence of memory pages executed by the victim enclave, a leakage source with 4 KB granularity that can be used to track the enclave execution [Wan+17, Shi+16, VB+17, WSB18]. This attack relies on the fact that SGX leaves control of its memory pages to the untrusted OS. Therefore, an adversarial OS can mark a memory page with SCA relevance as *non-executable* and monitor it. A triggered page fault indicates the execution of the monitored page [XCP15]. Repeating the process for a set of memory pages allows the adversary to track the sequence of executed memory pages, forming an error-free trace that potentially leaks secret data processed by the enclave. For the sake of simplicity, we refer to this page tracking attack as `PageTracer`.

Recently, Moghimi et al. [Mog+20] proposed the `CopyCat` attack that allows an adversary to glean the number of executed instructions in a tracked memory page. While it also works at page granularity, it increases the information provided by `PageTracer`. Both attacks can be carried out using the `SGX-Step` framework proposed by Van Bulck, Piessens, and Strackx [VBPS17].

In this section, we evaluate `mbedTLS`, `libgcrypt`, and `wolfSSL` scalar multiplication implementations using the OTA framework proposed in Section 3 regarding `PageTracer` and `CopyCat` attacks. This evaluation, in addition to highlighting their vulnerability to OTAs, extensively compares both attacks and complements the `CopyCat` research [Mog+20].

Threat model. During the experimental validation of our proposed OTA framework, we employed `PageTracer` and `CopyCat` attacks. They share the same SGX threat model: The adversary has OS privileges and can take advantage of its resources to mount controlled side-channel attacks. This is a typical threat model for targeting SGX enclaves using side channels [XCP15, VBPS17, WSB18, AB20]. In our experiments, we assume an SGX victim application that executes an ECC scalar multiplication with a secret (e.g. EdDSA, ECDSA, and ECDH).

Library selection was not arbitrary. We selected three open-source libraries with multiprecision integer arithmetic not designed to execute with input-oblivious execution flows. This selection is interesting, because while these libraries put significant effort in providing side-channel secure scalar multiplication, usually only the upper layer receives these security improvements, leaving the *bignum* implementation unattended. The rationale behind this trend is related to the *fix-on-demand* development process. At the same time, analyzing how a conditional branch deep in the *bignum* implementation relates to a secret only processed at the highest layer is usually non-trivial, as it requires searching for a leakage model—not part of the library development process. However, OTAs can exploit a deep *bignum* implementation leak without knowing its leakage model. This highlights the need for *secure-by-default* implementations and the analysis of the selected libraries regarding OTAs.

We later analyze these libraries, employing the OTA analysis framework proposed in Section 3. This demonstrates how an implementation can be evaluated regarding OTA security in practice. We next introduce tooling that will assist this evaluation process regarding microarchitecture side channels. Then in Section 5, we employ the results gathered during said OTA evaluation to develop end-to-end attacks on ECC scalar multiplication implementations in these libraries.

4.1 Microarchitecture OTA evaluation tool

For evaluating OTAs on software libraries using microarchitecture side channels, we developed a tool that follows step-by-step the evaluation process shown in Figure 1.

We employed `TracerGrind`[2], a binary dynamic instrumentation tool developed as part of the Side-Channel Marvels project by Bos et al. [Bos+16]. This tool patches `Valgrind`, allowing to record execution traces of a software binary. One of its many features is the ability to track a specific address range, that allows e.g. focusing the analysis on a specific shared-library.

The traces recorded with `TracerGrind` contain the sequence of accessed addresses (both data and code). Therefore, it can be used to model a side-channel trace down to instruction granularity. For instance, `CopyCat` can be emulated using `TracerGrind` by clearing the 12 least significant bits of every executed memory address, then run-length-encoding the resulting trace. This produces a trace that contains the sequence of executed memory pages and the number of instructions executed within them. Similarly, a `PageTracer` trace can be obtained by removing the instruction count information from a `CopyCat` one. Figure 2 represents this `TracerGrind`-based leakage emulation, highlighting the relationship between executed instructions and the emulated side-channel traces.

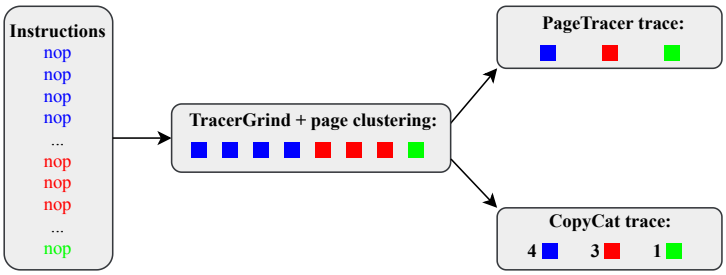

Figure 2: Leakage emulation using `TracerGrind`. Colors represent memory pages.

Section 5 empirically validates the accuracy of `TracerGrind` regarding `PageTracer`, where it is used to capture the *template traces* during real-world attack instances. Regarding this evaluation, `TracerGrind` allows to emulate a side channel for the `Process` operation on each analyzed library, allowing estimation of the metrics proposed in Section 3.3. We released a proof-of-concept code that includes this tool and uses it to emulate a single-trace OTA against `wolfSSL` [AB21].

Naturally, other microarchitecture side-channel signals, like Flush+Reload ones at cache-line granularity, can be modeled using this approach, hence this tool is not side-channel nor SGX-specific. It is sufficient to encode `TracerGrind` output according to the desired leakage. At the same time, noisy side channels might need a template matching algorithm that supports uncertainty. For instance, the edit distance can been used in these scenarios as a template matching metric [Liu+15, Sch+17].

Limitations. Our tooling aims at detecting OTA-exploitable leaks considering a given (or modeled) microarchitecture side channel. Both `PageTracer` and `CopyCat` work at the memory page boundary (i.e. 4kB), therefore the detected leakages depend on the targeted binary layout. This means that (in theory) `PageTracer` and `CopyCat`-vulnerable code might evade detection during evaluation. For instance with `PageTracer`, consider a condition operation with resulting branches executing in the same page. In this case, it is not possible to distinguish the executed branches using `PageTracer`. In the opposite case where the target binary has a different layout such that said branches are in different pages, the tooling indeed detects the leak using `PageTracer`. Evidently, the probability to detect all OTA-exploitable leakages increases with the granularity, i.e. it is more likely that leaks remain undetected using `PageTracer` than `CopyCat`.

[2] https://github.com/SideChannelMarvels/Tracer/tree/master/TracerGrind

We designed this tool with an attacker perspective in mind. Regarding security assessment, it can be used as a necessary condition to declare an implementation OTA-*safe*, but not as a sufficient criteria. For this reason, during disclosure to the development teams of the analyzed libraries, we encouraged generic countermeasures instead of eliminating individual leaks, also taking into account we only tested them using two side-channel signals (see Appendix A for details).

4.2 `libgcrypt` template implementation

Each library follows its own scalar multiplication approach, therefore the definitions of *state* and `Process` vary. Scalar multiplication in `libgcrypt` v1.8.5 follows the *double-and-add-always* approach shown in Algorithm 2. It consists of a main loop that iterates over every bit of k. At every iteration, a pair of *doubling* and *addition* operations are executed regardless of bit i of k. However, even if both operations are executed, R is only updated with the result of the *addition* operation if $k_i = 1$. This is ensured by the conditional assignment at line 5 that, when implemented securely, provides SCA resistance to trivial attacks.

Algorithm 2: *double-and-add always* scalar multiplication

Input: Integer k and elliptic curve point G
Output: $P = kG$

1 $R = \mathcal{O}$
2 **for** $i = \lfloor \log_2 k \rfloor$ **downto** 0 **do**
3 $\quad R = 2R$
4 $\quad T = R + G$
5 $\quad R =$ `cond_assign`(T, R, k_i)
6 **return** R

Regarding our abstract scalar multiplication description, the *state* in Algorithm 2 consists of a single elliptic curve point, R. Similarly, the `Process` operation consists of a point *doubling* and an *addition*. `libgcrypt` exports function wrappers for these point operations on Weierstrass and Edwards curves: `gcry_mpi_ec_dup` and `gcry_mpi_ec_add` respectively. This allows the attacker to build a *template implementation* to execute both operations for every input R, recording its trace with `TracerGrind`. With `libgcrypt`, we focus our research on EdDSA that uses a twisted Edwards curve birationally equivalent to Curve25519 [Ber+12]. The selection of EdDSA is interesting because EdDSA is rarely vulnerable to partial nonce attacks. Compromising EdDSA requires massive leakage from a single trace, and OTAs fit nicely with this requirement.

Following the evaluation flow in Figure 1, we evaluated the *Reproducible* requirement to *estimate* if this implementation is deterministic. For this task, we generated a random curve point and captured 10 independent traces using our template implementation harness. We then repeated the experiment for 1000 random points, observing determinism in both `PageTracer` and `CopyCat` traces in all cases. Therefore, it is likely that the side-channel trace corresponding to the processing of R can be *reproduced* with this implementation. Note that this test demonstrates a fundamental difference between power consumption signals and the employed microarchitecture ones.

According to Figure 1, the next step in a deterministic scenario is to estimate $pmf(\boldsymbol{L})$. For this task, we generated 1000 random points, recording their corresponding traces using `TracerGrind`. This experiment resulted in 1000 unique `CopyCat` traces, implying an *ideal* attacker scenario, and 889 unique `PageTracer` traces, which is close to ideal. Both attack results allow concluding it is very likely a leakage trace of the *doubling* and *addition*

implementations for Edwards curves in `libgcrypt` can be used as a state *distinguisher* for any of these attacks.

Analyzing one of these traces, `libgcrypt` executed code in 28 different memory pages for a single call to the *doubling* and *addition* wrappers for Edwards curves. We repeated the capture using `TracerGrind` but limiting the recording to the `libgcrypt` address space, hence external calls are not recorded (e.g. `libc` ones). The number of executed pages reduces to 17. An adversary can freely choose their configuration, because during an attack they select which memory pages to track. However, in our research we are not only interested in determining if `libgcrypt` is vulnerable to OTAs, but analyzing how the leakage behaves considering every memory page combination. Therefore, as the number of combinations is equal to $2^\ell - 1$ for ℓ pages, we decided to use the limited trace recording to reduce analysis time.

We estimated the $pmf(\boldsymbol{L})$ for every memory page combination, 131071 in total. This allows collecting some statistics about OTA performance, considering Figure 1 metrics. More importantly, it allows pinpointing leakage origins. Table 1 summarizes how many page combinations were classified according to the metrics presented in Section 3.3.

Table 1: `libgcrypt` OTA classification for $2^{17} - 1$ page combinations.

	Combination class			
Attack	**Ideal**	**Easy**	**Hard**	**Safe**
`PageTracer`	0	87%	3%	10%
`CopyCat`	50%	48%	0.8%	0.8%

Expanding on the number of combinations that could be used to perform an OTA, Table 2 shows additional results derived from the previous experiment. The *Insecure* column represents the number of insecure page combinations, i.e. sum of *Ideal* and *Easy* columns in Table 1. This means that an attacker can select any *Insecure* page combination to mount an OTA using these side channels.

Among the interesting evaluation data is the *minimum cardinality* that could lead to a successful OTA and the *maximum bias* observed. For instance, regarding `PageTracer`, at least one insecure page combination exists with only two outcomes in its $pmf(\boldsymbol{L})$ (cardinality = 2). At the same time, the maximum bias observed among all combinations is 50%. This implies there is at least one combination with a two-cardinality $\boldsymbol{L}$ and equiprobable pmf labeled as insecure. Section 5.2 shows the feasibility of attacking this kind of pmf.

The last row of Table 2 provides information on the number of *root* page combinations and their size, i.e. number of pages in them. We define a combination C as root if no smaller combination exists that is a subset of C. For instance, if $C_0 = (P_1)$, $C_1 = (P_1, P_2)$ and $C_2 = (P_1, P_2, P_3)$ are insecure page combinations, then C_0 is a root combination while C_1 and C_2 are not. Similarly, if no single page combination were insecure in this example, then C_1 would become a root one. Root combinations can be used to pinpoint where the leakage comes from, especially when they are single-paged ones like in the `CopyCat` case.

Smaller root combinations imply less addresses to be tracked during the attack. This is not an issue for `PageTracer` and `CopyCat` due to their noise-free feature. On the other hand, noisy attacks like Flush+Reload will definitely benefit from small root

Table 2: `libgcrypt` combination details.

Attack	**Insecure**	**Min card**	**Max bias**	**Root combs/size**
`PageTracer`	87%	2	50%	23/2
`CopyCat`	98%	7	30%	6/1

combinations to decrease noise impact.

`PageTracer` has 23 root combinations, all with two pages in them, while `CopyCat` has six single-page ones. Therefore, it could be possible to perform a successful OTA using only two pages for `PageTracer` and a single one for `CopyCat`, instead of using all 17 pages involved in our `libgcrypt` template implementation. According to the definition of root, all page combinations are composed by mixing the root ones. Therefore, in addition to knowing the smaller combinations that could be used to succeed, it is also interesting to know how many an attacker would need to achieve the maximum cardinality[3].

Figure 3 shows how the cardinality progresses as the number of used memory pages increases from one to four. Insecure combinations are found starting from two pages and reaching the maximum cardinality (889 in our experiments) with four pages. The results for `CopyCat` are even better, achieving an *ideal* attack scenario with only two pages. Therefore, if an adversary wishes to reduce the number of pages to track, e.g. for noise mitigation or error correction, both attacks achieved their maximum cardinality even with a reduced set of pages.

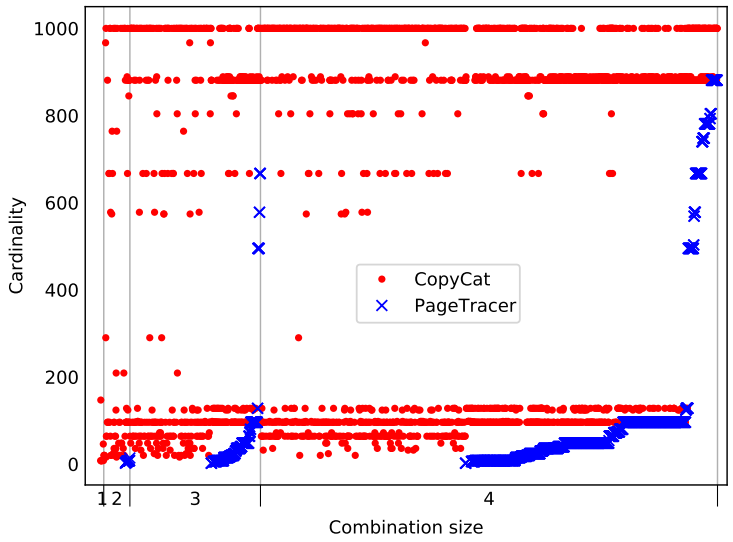

Figure 3: `libgcrypt` cardinality vs. combination size (partial).

4.3 `mbedTLS` template implementation

We analyzed `mbedTLS` v2.16.3 in the context of the elliptic curve `secp256r1` (i.e. NIST P-256). Elliptic curve computations for this curve use Jacobian projective coordinates. Algorithm 3 shows a simplified version of the scalar multiplication algorithm in this library. It follows a *comb* approach based on the proposal in [HPB05]. This algorithm randomizes the starting value of R. However, Section 5 expands on how this affects both OTA variants, surprisingly concluding that it can be ignored for this analysis.

This algorithm encodes the scalar k into a sequence of K_i, where the encoding details are irrelevant because it is invertible. Thus, if an adversary recovers all K_i she immediately obtains k. The second step precomputes an array P which, for the targeted curve, contains 32 multiples of G. At each iteration, one point of this array is employed based on K_i. Hence, identifying which point is *selected* at each iteration will reveal K_i. We employ R as *state* in this implementation, initialized at line 3 to an unknown value based on K_1.

[3]Here cardinality is used as attack performance, considering that all combinations labeled as insecure have an *easy* bias, see Section 4.1.

Algorithm 3: `mbedTLS` *comb* scalar multiplication

Input: Integer k and elliptic curve point G
Output: kG

```
K = Encode(k)
P = Precompute(G)
R = Select(K_1, P)
for K_i ∈ K : i = [2, n] do
    R = 2R
    T = Select(K_i, P)
    R = R + T
return R
```

Still, the attacker knows that $R \in P$ so there are only 32 candidates, and can start by considering all of them.

To demonstrate the flexibility of OTAs, for this implementation we chose a `Process` operation composed of only the point *doubling* operation, ignoring the leakage produced by *addition*. This operation is implemented in function `ecp_double_jac`. In contrast to the `libgcrypt` case, this function is not an exported symbol, therefore building a template implementation to reach it requires additional effort. The first strategy we explored was building our own copy of `mbedTLS` where this symbol is actually exported, hoping that the symbol table does not significantly change wrt an original (attack) build. We analyzed both library binaries and the differences were not significant at 4 KB granularity, therefore we proceeded with this option.

Following the OTA implementation evaluation metrics shown in Figure 1, we used `TracerGrind` to assess the determinism of `ecp_double_jac` using 1000 different points and 10 trials per point. We deduce that this implementation is likely to be deterministic regarding `PageTracer` and `CopyCat`. The execution of `ecp_double_jac` was distributed among 14 memory pages, meaning 16383 page combinations that could be used to mount an attack.

Similar to our `libgcrypt` analysis, we estimated the $pmf(\boldsymbol{L})$ for each of these combinations, and Table 3 summarizes the results. Regarding `CopyCat`, all combinations are insecure, with `PageTracer` close behind. Moreover, the number of total *ideal* attacker combinations is very high for both `PageTracer` and `CopyCat`. The maximum bias found is 62% for `PageTracer`, yet it is still considered insecure based on our estimations.

Table 3: `mbedTLS` combination details.

Attack	Ideal	Insecure	Min card	Max bias	Root combs/size
`PageTracer`	84%	99%	2	62%	63/2
`CopyCat`	99%	100%	9	24%	14/1

The number of root combinations increases significantly in comparison to `libgcrypt`. Centering the analysis on `CopyCat` results, it is worth highlighting that the number of single-sized root combinations is equal to the number of memory pages executed by `ecp_double_jac`. Hence, any page in this set can be used to *distinguish* the processed point.

Figure 4 shows the cardinality progression against combination size. The *ideal* scenario is achieved using `CopyCat` for almost every two-page combination, whereas `PageTracer` requires at least three pages to achieve *ideal*. Both results demonstrate the threat this library faces, especially considering the high number of small size combinations that achieve the ideal scenario.

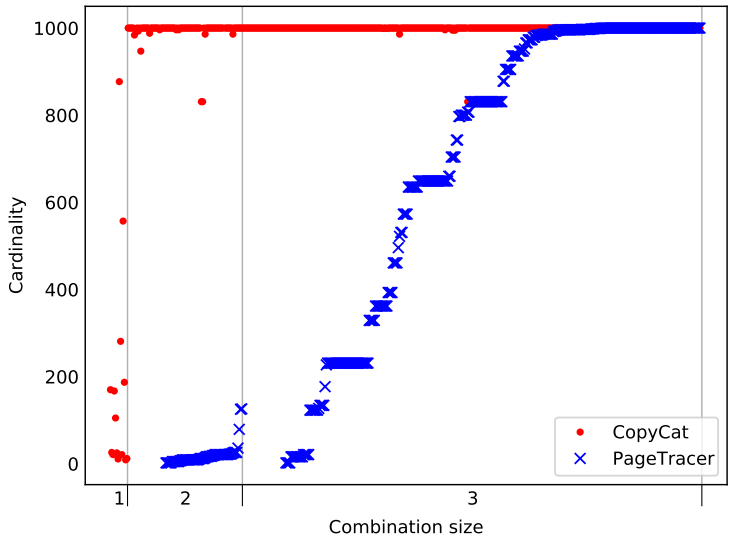

Figure 4: `mbedTLS` cardinality vs. combination size (partial).

4.4 `wolfSSL` template implementation

We analyzed `wolfSSL` v4.4.0 with default build options, including various timing attack countermeasures. Our analysis focuses on `secp256r1`, where the library uses the Montgomery ladder to compute scalar multiplications (Algorithm 4). The state for this implementation consists of two elliptic curve points R and S, initialized to G and $2G$ respectively. The Montgomery ladder aims at providing side-channel resistance against trivial attacks by executing a point *addition* and a *doubling* at each iteration, despite the value for bit i of k. However, the arguments for these operations (i.e. the state) do depend on k_i.

Algorithm 4: *Montgomery ladder* scalar multiplication

Input: Positive integer k and elliptic curve point G
Output: $P = kG$

```
1 R = G, S = 2G
2 for i = ⌊log2(k)⌋ − 1 downto 0 do
3     if k_i = 0 then
4         S = R + S,  R = 2R
5     else
6         R = R + S,  S = 2S
7 return R
```

For this implementation, we selected the *doubling* operation as our targeted `Process`, implemented in function `ecc_projective_dbl_point` that is not exported by default. However, the attacker can build her own version of the library where this symbol is exported, similar to `mbedTLS`. Using `TracerGrind`, we captured some traces for this function and observed only seven memory pages were executed. Hence, the number of page combinations is only 127, a considerable reduction wrt to the thousands of `libgcrypt` and `mbedTLS`.

Similar to the previous cases, we estimated the determinism of this `Process` implementation, concluding that it has deterministic leakage for both `PageTracer` and `CopyCat`.

Following the evaluation flow, we estimated each $pmf(\boldsymbol{L})$ for each page combination. Table 4 shows the results, highlighting that the majority of page combinations are insecure using `PageTracer`, and every page combination is insecure using `CopyCat`. For `PageTracer`, the maximum observed bias was 52% with a minimum cardinality of two. A closer inspection of this leakage revealed it is produced by a modular division by two which executes an addition before dividing if an intermediate value is odd[4].

Table 4: `wolfSSL` combination details.

Attack	Ideal	Insecure	Min card	Max bias	Root combs/size
`PageTracer`	0	69%	2	52%	7/2
`CopyCat`	47%	94%	7	24%	4/1

Figure 5 shows how the cardinality progresses with the combination size for `wolfSSL`. We reach an *ideal* scenario for two-size page combinations with `CopyCat`, while three pages are required to achieve the maximum cardinality with `PageTracer`. In summary, these results show that even when `wolfSSL` employs only seven pages in our targeted `Process`, OTAs are possible for many page combinations.

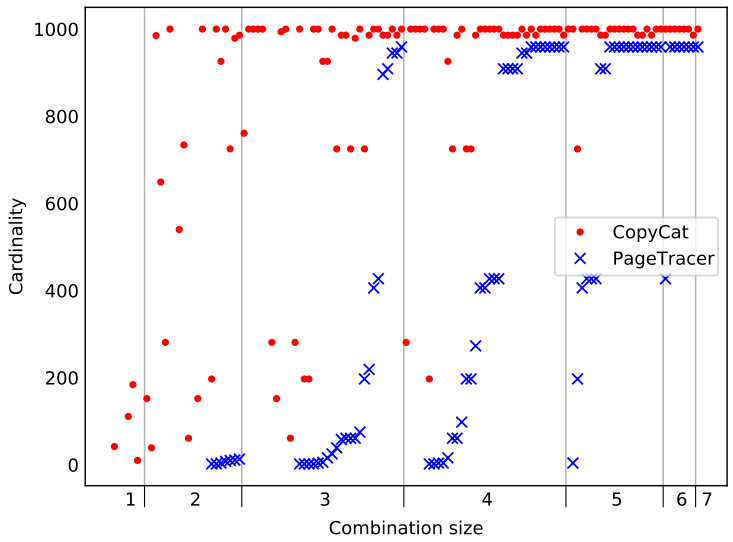

Figure 5: `wolfSSL` cardinality vs. combination size.

5 Real-World Attacks: Evaluating End-to-End OTAs

In this section, we develop end-to-end attacks on `libgcrypt`, `mbedTLS`, and `wolfSSL` scalar multiplication algorithms using the template implementations described in Section 4. All three attacked libraries share the experiment setup and basic OTA approach. We consider the threat model described in Section 4.

Experiments setup. For validating OTAs on the three analyzed libraries, we employed the same environment for capturing the traces. It consists of a desktop workstation running Ubuntu 18.04.1 LTS on an SGX-capable Intel i7-7700 CPU. We used the `Graphene-SGX` framework for a straightforward porting of each targeted library to SGX, requiring no code

[4] https://github.com/wolfSSL/wolfssl/blob/v4.4.0-stable/wolfcrypt/src/ecc.c#L2179

changes [TPV17]. We developed `PageTracer`-based attacks using the `SGX-Step` framework [VBPS17]. Usually `PageTracer` attacks require the adversary to select which pages to track in advance, based on a known leakage model. A huge advantage of the pmf analysis performed for each library in previous sections is that the selection of these pages can be fully automated. That is, just selecting an insecure page combination, without knowledge of what is actually executed in those pages.

For all experiments, we compiled the targeted libraries using the latest versions at the time of writing and the default build options. For each library, we developed an SGX enclave on top of the `Graphene-SGX` framework. These enclaves are our attack targets, which instantiate ECC protocols (`libgcrypt`) and scalar multiplication primitives (`mbedTLS`, `wolfSSL`) from within their respective libraries. In all cases, the exploited scalar multiplication primitives are the default in their libraries for the targeted elliptic curve and protocol: EdDSA in `libgcrypt`, and ECDSA/ECDH in both `mbedTLS` and `wolfSSL`.

Attack implementation. We implemented OTAs as described in Section 2 using depth-first search with an early exit when recovering the targeted scalar. The exit condition varies between library and attack direction. We followed a *state*-based attack, therefore the attacker must compute the processed state based on a K_i guess as explained in Section 3.2. The recovery code is independent of the targeted pmf, therefore we made no optimizations in this regard.

For each library, we selected a pmf and configured `SGX-Step` to track its corresponding page combination. After capturing the trace, the recovery code locates the start of the scalar multiplication execution, then separates its trace in iterations $I = \{I_1, I_2, .., I_n\}$, where each I_i corresponds to the `Process` operation of the implementation. The OTAs proceed from there. We give specific details regarding state computation and scalar multiplication algorithms in the corresponding sections.

Leakage origins and previous works. For each attacked library, we pointed to the leaking source code sections to satisfy reader curiosity. However, we highlight that this information is not needed by an OTA adversary and was not used in our attacks. One of the advantages of OTAs is that they can exploit these weaknesses without knowing those specifics nor the leakage models. During responsible disclosure (see Appendix A), we provided generic leak descriptions and mitigation approaches.

Previous works on microarchitecture side channels do not cover OTAs. This means many attacks follow a leakage model-based approach where authors identify a leaky code path, then find a way to relate it to a secret. A recent example is *LadderLeak*, where the authors located leaky ECC arithmetic code in older versions of OpenSSL and developed a leakage model to recover a single bit from the scalar [Ara+20]. On the other hand, OTAs remove the leakage model requirement from the equation. Using tooling presented in Section 4 and the evaluation metrics we propose, it is possible to detect which implementation features (e.g. memory pages) can be used to exploit these leaks, achieving full scalar recovery in many cases.

The leakage origins mentioned in the next sections are only a subset of those that can be exploited. Enumerating all of them is a time consuming task and irrelevant to this work. For instance, the number of root page combinations gives an approximation of this quantity (e.g. `mbedTLS` has 63 for `PageTracer`, see Table 3). Moreover, even those are not exhaustive because, as previously stated, the tooling was not designed to detect leakages at all granularities. For the inclined, we suggest tools like the *DATA* framework which aims at detecting leakages in software binaries using statistical tools and leakage models [Wei+18, Wei+20]. *DATA* is also interesting for our work because its final step uses known leakage models to assess the severity of a potential leak. However, in many cases it detects a potential leak but omits information concerning impact. Our OTA evaluation

framework can be used to fill this gap in *DATA* and similar tools. Finding a potential leak in the ECC scalar multiplication code path using these leakage detection frameworks, the adversary/evaluator can configure OTA tooling (see Section 4) to estimate the leakage pmf. This helps understand the security impact of these potential leakage points on the implementation.

5.1 End-to-end attack on `libgcrypt`

For an end-to-end attack, we followed the signature generation scenario using EdDSA with the Curve25519 twist curve (i.e. Ed25519). Generating a signature using Ed25519 involves computing a pseudorandom 512-bit nonce r and computing the scalar multiplication rG. This cryptosystem was designed to avoid e.g. lattice cryptanalysis [HS01] where small information disclosures on different r break the scheme. For inner details about this cryptosystem and how r is generated, we refer the reader to [Ber+12]. Regarding our research, we aim at recovering all 512 bits of r that is sufficient to forge signatures, therefore further details are irrelevant.

For this library, we arbitrary selected an insecure (but not ideal) page combination with a pmf of 48 observed outcomes after 1000 samples (see its pmf in Figure 6). This page combination consists of the following offsets: `0xd5000`, `0xd6000`, `0xd7000`, `0xd8000`. Additionally, we used[5] offset `0xa3000` to detect when signature generation starts.

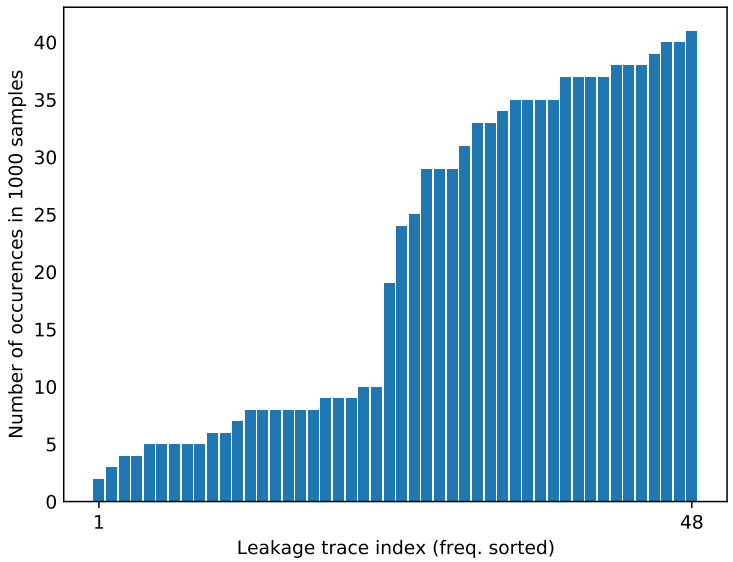

Figure 6: `libgcrypt` attack pmf.

Forward attack. Following the `libgcrypt` scalar multiplication implementation (Algorithm 2), the adversary must detect which point R_i was processed at each iteration i, knowing that it is initialized to $R_1 = \mathcal{O}$. This algorithm scans the binary representation of the scalar (i.e. K_i), starting from the most significant bit that is always set. Therefore, at the start of the second iteration, $R_2 = 2R_1 + G = G$, and this is the input to the *double-and-add*-based `Process`. According to Algorithm 2, R is updated at each iteration using (2), hence each R_i depends on K_{i-1}.

$$R_i = 2R_{i-1} + G \cdot K_{i-1} \tag{2}$$

[5] No code in this page is executed during scalar multiplication, not affecting the pmf.

Thus, $R_3 = 3G$ if K_2 is set and $R_3 = 2G$ otherwise. Here is where OTAs enter into play by generating template traces for each possible value of R_3 (i.e. $T_{3,K_2=1}$ and $T_{3,K_2=0}$). Then eliminating the one that does not match with the target iteration trace (i.e. I_3). For this task, we used the template implementation described in Section 4.2 to generate the required template traces using `TracerGrind`. Table 5 shows example traces, where the memory pages are encoded using the characters { `.`, `a`, `A`, `D` } and the specific assignment is irrelevant. According to these traces, I_3 matches $T_{3,K_2=1}$ and differs from the other: In this example, the adversary deduces $K_2 = 1$. Continuing this process, each I_i and the corresponding template traces can be used to reveal K_{i-1}, eventually disclosing all[6] K_i.

Table 5: Example traces from `libgcrypt`.

Name	Trace
I_3	`aD.DA.DADA.ADA.ADA.ADA.A.AD.DA.DA.A.AD.DA.D.DA.aA.A.A.A.A.A.A.A.A[..].A.A.A.A.A.A.A.A.D`
$T_{3,K_2=1}$	`aD.DA.DADA.ADA.ADA.ADA.A.AD.DA.DA.A.AD.DA.D.DA.aA.A.A.A.A.A.A.A.A[..].A.A.A.A.A.A.A.A.D`
$T_{3,K_2=0}$	`aD.DA.DA.A.DA.ADA.A.ADADA.ADADA.ADA.DA.aA.A.A.A.A.A.A.A.A.A.A.A.A.A.A.A.A.A.A.D`

We captured 100 `libgcrypt` traces corresponding to the generation of Ed25519 signatures. Using our template implementation based on `TracerGrind`, we launched OTAs on them using the page combination described below, recovering the processed Ed22519 nonce for each signature in all trials. The average number of calls to the template implementation was 1038, meaning the attack is very computationally feasible in practice.

Backward attack. This attack assumes the adversary knows the projective coordinates of the last R_i just before converting the point to affine coordinates. This requirement can be fulfilled using a side-channel attack on the modular inversion algorithm in `libgcrypt` [APGB20]. The last value of R will be the initial OTA state, thus the adversary reverses the target trace order, such that the first iteration processed by an OTA will be the last one executed by the algorithm. In this case, the attacker knows the state resulting from the `Process` execution, i.e. R_i in (2). Therefore, the first step is to compute the `Process` input, i.e. R_{i-1} for all possible K_{i-1} using (3).

$$R_{i-1} = (R_i - G \cdot K_{i-1})/2 \tag{3}$$

Indeed (3) implies computing modular roots [NSS04, APGB20]. Therefore, multiple R_{i-1} candidates might be obtained for a single pair (R_i, K_{i-1}). This implies more template traces must be captured wrt the *forward* case.

Using this approach, we recovered the scalars processed in all 100 traces, validating our *augmented projective coordinates attack*. In this case, the average number of calls to the template implementation was 2851. Note that the original projective coordinates attack by Naccache, Smart, and Stern [NSS04] only recovers a few bits of the scalar, hence does not threaten EdDSA. On the other hand, our proposed *augmented* version can be used to break this cryptosystem.

Leakage origin. The leaks exploited in our attacks come from the `libgcrypt` *bignum* implementation. For instance, the function `_gcry_mpi_add` has many conditional branches depending on its inputs[7]. Each of these branches produce different `PageTracer` footprints, hence can be distinguished in a trace. Another leak is produced by the function `ecc_subm`, where the modular reduction after a subtraction is executed using an add loop[8].

[6] The last K_i remains unknown to an OTA adversary, as there is no related I_{i+1}, yet this binary value is recovered through direct inspection of the two cases along with public data.

[7] https://github.com/gpg/libgcrypt/blob/libgcrypt-1.8.5/mpi/mpi-add.c#L88

[8] https://github.com/gpg/libgcrypt/blob/libgcrypt-1.8.5/mpi/ec.c#L290

The advantage of OTAs over ad hoc attacks is that the former can exploit these leaks without knowing a leakage model. That is, the attacker does not need to know how a particular branch in these functions leaks information related to the scalar. Each time these functions are called during an execution of `Process`, they leave leakage footprints related to the data they are processing. Generally speaking, each of these leakages alone may reveal limited information, yet their combination composes stronger leakage.

5.2 End-to-end attack on `mbedTLS`

`mbedTLS` scalar multiplication (Algorithm 3) has an OTA countermeasure in place: It randomizes the starting coordinates of R just after line 3. However, there are at least two scenarios where OTAs can be applied. (i) Said point randomization is only executed *before* the scalar multiplication, therefore it offers protection against a *forward* OTA, but the *backward* approach is still a threat. (ii) Said countermeasure only works if an `mbedTLS` randomization object is passed as an argument to this function. Regarding ECDSA, this randomization takes places as expected. Yet we discovered two cases where it fails: When loading an ECC private key without the public key or in compressed representation, the library computes the public key on the fly without initializing the randomization object. This leaves the door open to a *forward* OTA.

Therefore, the backward case is useful when analyzing protocols like ECDSA or ECDH, and the forward case when the library loads a private key with missing (such keys are valid [PG+20]) or compressed public key. Accordingly, we focus our attention on the scalar multiplication primitive in this library. The attack procedure is very similar to the `libgcrypt` description in Section 5.1. In the curve `secp256r1` case, the scalar has 256 bits and at each iteration out of 52, the adversary must guess 32 possible K_i, due to the windowed feature of Algorithm 3.

We employed the page combination `0x2f000`, `0xf000`, `0x10000`, `0x36000` that has an *ideal* pmf. We used[9] auxiliary page offsets `0x30000` and `0x31000` to detect the start of the scalar multiplication routine and the `Process` operation (i.e. function `ecp_double_jac`).

We captured 100 traces using `PageTracer` against an SGX enclave running scalar multiplication, attempting to recover the scalar using OTAs in both the forward and backward directions. The *forward* attack succeeded for all traces, disclosing all[10] K_i: The number of calls to the template implementation was 1664 per attack (i.e. 32 templates for 52 iterations). This is due to the *ideal* pmf employed acting as a perfect state distinguisher. The *backward* attack instances also succeed for all traces, with an average number of calls to the template implementation of 1959. The number varies between different attacks due to the modular roots involved, generating additional candidates per each guessed K_i (see Section 5.1 for details). Remarkably, our *backward* OTA bypasses the mitigation (projective coordinates randomization of the starting state). A differentiating characteristic compared to the `libgcrypt` case is that the victim executed within an unmodified `mbedTLS` library, while the template implementation used a patched one that exported the `ecp_double_jac` symbol.

The leakage in `mbedTLS` mainly originates from the modular reduction techniques specific to `secp256r1`[11]. This implementation uses NIST fast reduction techniques, involving several[12] branches[13]. Note that every modular reduction executed during the `Process` operation produces leakage. The combination of all these leaks from branch outcomes produces a perfect state distinguisher, hence an ideal pmf.

[9] No code in these pages is executed during `ecp_double_jac`, not affecting the pmf.

[10] The last K_i remains unknown to an OTA adversary, as there is no related `Process` executed after selection, yet this value is recovered through direct inspection of the 32 cases along with public data.

[11] https://github.com/ARMmbed/mbedtls/blob/mbedtls-2.16.3/library/ecp.c#L1000

[12] https://github.com/ARMmbed/mbedtls/blob/mbedtls-2.16.3/library/ecp_curves.c#L1010

[13] https://github.com/ARMmbed/mbedtls/blob/mbedtls-2.16.3/library/ecp_curves.c#L1022

5.3 End-to-end attack on `wolfSSL`

Finally, we demonstrate the feasibility of OTAs using `PageTracer` on `wolfSSL`, where we captured 100 traces using `PageTracer` against an SGX enclave running scalar multiplication. We employed the page combination consisting of the offsets `0x25000`, `0x29000`, `0x2a000`, `0x4b000` that has an *ideal* pmf. We additionally used[14] `0x2c000` to detect the start of the scalar multiplication routine.

We executed only *forward* OTAs against `wolfSSL`, resulting in full key recovery in all trials, requiring 512 calls to the template implementation in all cases. Note that an ideal pmf and a binary scalar multiplication algorithm like the Montgomery ladder allow reducing this number to only 256 (i.e. one call per bit). However, as stated at the beginning of this section, we implemented our recovery algorithm to always be oblivious to the pmf to retain generality.

In addition to the previous pmf, we attacked this implementation using other approaches. (i) We considered a two-cardinality equiprobable pmf, and executed this attack for three traces, recovering the full scalar in all trials. Naturally, the number of calls to the template implementation increases, and also the number of solutions to test. The pairs for these values for the three instances were $(5780, 10)$, $(19454, 76)$, and $(30640, 177)$, all practical attacks. (ii) We considered the *scalar*-based OTA theoretically presented in Section 3.2. Instead of guessing the state, this approach assumes the adversary has access to a template implementation allowing chosen-input scalars. For this scenario, we used 100 traces and the same ideal page combination employed before, indeed recovering the full scalar in all trials. In addition, we released a proof-of-concept tooling that simulates this OTA approach against this library [AB21].

The leaks exploited in our `wolfSSL` OTAs come in different flavors. For instance, the *doubling* function `ecc_projective_dbl_point` contains several branches related to its inputs[15]. The modular division by two is straightforwardly implemented using a branch that first checks if the input is odd[16]. Modular additions[17] and subtractions[18] are handled in similar ways. Additionally, the functions `fp_sub`[19] and `fp_montgomery_reduce`[20] have `PageTracer`-distinguishable branches in their implementations.

As commented at the start of this section, the attacker does not need to know where the leakage originates, rather the page combination that leads to a successful attack. This can be identified during a template implementation evaluation, as explained in Section 4. This way, OTAs abstract the exploited leakage model from the adversary, resulting in very powerful attacks.

6 Conclusion

Previous works related to the OTA technique only considered part of its potential. In this paper, we revisited that description, proposing a framework and evaluation metrics to detect if an implementation is vulnerable to OTAs. Additionally, we demonstrated that OTAs can also work in the *backward* direction, a case not considered before. This shows that randomizing the initial state of the targeted algorithm does not blanketly prevent OTAs, as previously believed. In this regard, an *augmented projective coordinate attack* is one example of a *backward* OTA because it can recover the entire scalar using a single

[14] No code in this page is executed during `ecc_projective_dbl_point`, not affecting the pmf.
[15] https://github.com/wolfSSL/wolfssl/blob/v4.4.0-stable/wolfcrypt/src/ecc.c#L1932
[16] https://github.com/wolfSSL/wolfssl/blob/v4.4.0-stable/wolfcrypt/src/ecc.c#L2179
[17] https://github.com/wolfSSL/wolfssl/blob/v4.4.0-stable/wolfcrypt/src/ecc.c#L2133
[18] https://github.com/wolfSSL/wolfssl/blob/v4.4.0-stable/wolfcrypt/src/ecc.c#L2201
[19] https://github.com/wolfSSL/wolfssl/blob/v4.4.0-stable/wolfcrypt/src/tfm.c#L166
[20] https://github.com/wolfSSL/wolfssl/blob/v4.4.0-stable/wolfcrypt/src/tfm.c#L3035

trace. This is in contrast to the thousands needed by the original projective coordinates attack by Naccache, Smart, and Stern [NSS04].

The three analyzed libraries `libgcrypt`, `mbedTLS`, and `wolfSSL` have many leaky points that can be exploited using OTAs. We demonstrated practical attacks for the three libraries, in all cases recovering the full scalar by employing a single trace using a microarchitecture side channel after extensive experiments. In the microarchitecture realm, it is possible to have an *ideal* attacker scenario as demonstrated for the analyzed libraries. At the same time, it is also possible to achieve *safe* ones if the implementation follows a constant-address approach. These scenarios are not common at all in the power consumption case, where the original OTA technique was proposed.

Our tool proposed to detect OTA vulnerabilities is not exhaustive, therefore there could be additional exploitable paths. At the same time, its idea serves as a starting point to develop a leakage assessment tool for address-based side channels. Such a tool is ideally able to detect any OTA vulnerability in the hierarchy of a cryptosystem implementation, certainly not restricted to ECC.

In conclusion, OTAs can exploit non-trivial input-dependent execution flows without knowing the leakage model, highlighting the need for *secure-by-default* implementations.

Acknowledgments

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).

Supported in part by CSIC's i-LINK+ 2019 "Advancing in cybersecurity technologies" (Ref. LINKA20216).

References

[AB20] Alejandro Cabrera Aldaya and Billy Bob Brumley. "When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.2 (2020), pp. 196–221. URL: https://doi.org/10.13154/tches.v2020.i2.196-221.

[AB21] Alejandro Cabrera Aldaya and Billy Bob Brumley. *Online Template Attacks: Revisited - Proof of Concept*. Zenodo, Apr. 2021. URL: https://doi.org/10.5281/zenodo.4680071.

[ABG10] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. "New Results on Instruction Cache Attacks". In: *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*. Ed. by Stefan Mangard and François-Xavier Standaert. Vol. 6225. Lecture Notes in Computer Science. Springer, 2010, pp. 110–124. URL: https://doi.org/10.1007/978-3-642-15031-9_8.

[AK09] Onur Acıiçmez and Çetin Kaya Koç. "Microarchitectural Attacks and Countermeasures". In: *Cryptographic Engineering*. Boston, MA: Springer US, 2009, pp. 475–504. ISBN: 978-0-387-71817-0. DOI: 10.1007/978-0-387-71817-0_18. URL: https://doi.org/10.1007/978-0-387-71817-0_18.

[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. URL: https://doi.org/10.1109/SP.2019.00066.

[APGB20] Alejandro Cabrera Aldaya, Cesar Pereida García, and Billy Bob Brumley. "From A to Z: Projective coordinates leakage in the wild". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.3 (2020), pp. 428–453. URL: https://doi.org/10.13154/tches.v2020.i3.428-453.

[Ara+20] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. "LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage". In: *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM, 2020, pp. 225–242. URL: https://doi.org/10.1145/3372297.3417268.

[Bat+14] Lejla Batina, Lukasz Chmielewski, Louiza Papachristodoulou, Peter Schwabe, and Michael Tunstall. "Online Template Attacks". In: *Progress in Cryptology - INDOCRYPT 2014 - 15th International Conference on Cryptology in India, New Delhi, India, December 14-17, 2014, Proceedings*. Ed. by Willi Meier and Debdeep Mukhopadhyay. Vol. 8885. Lecture Notes in Computer Science. Springer, 2014, pp. 21–36. URL: https://doi.org/10.1007/978-3-319-13039-2_2.

[Ber+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-speed high-security signatures". In: *J. Cryptographic Engineering* 2.2 (2012), pp. 77–89. URL: https://doi.org/10.1007/s13389-012-0027-1.

[BH09] Billy Bob Brumley and Risto M. Hakala. "Cache-Timing Template Attacks". In: *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*. Ed. by Mitsuru Matsui. Vol. 5912. Lecture Notes in Computer Science. Springer, 2009, pp. 667–684. URL: https://doi.org/10.1007/978-3-642-10366-7_39.

[Bha+20] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. "Branch Prediction Attack on Blinded Scalar Multiplication". In: *IEEE Trans. Computers* 69.5 (2020), pp. 633–648. URL: https://doi.org/10.1109/TC.2019.2958611.

[Bos+16] Joppe W. Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. "Differential Computation Analysis: Hiding Your White-Box Designs is Not Enough". In: *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 215–236. URL: https://doi.org/10.1007/978-3-662-53140-2_11.

[Bos+18] Joppe W. Bos, Simon Friedberger, Marco Martinoli, Elisabeth Oswald, and Martijn Stam. "Assessing the Feasibility of Single Trace Power Analysis of Frodo". In: *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*. Ed. by Carlos Cid and Michael J. Jacobson Jr. Vol. 11349. Lecture Notes in Computer Science. Springer, 2018, pp. 216–234. URL: https://doi.org/10.1007/978-3-030-10970-7_10.

[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] Cai-sen Chen, Yang-xia Xiang, Jia-xing Du, and Zhiwei Cheng. "An Improved Data Cache Timing Attack against RSA Based on Hidden Markov Model". In: *Journal of Computers* 30.1 (2019), pp. 87–95. URL: https://doi.org/10.3966/199115992019023001009.

[Cla+10] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. "Horizontal Correlation Analysis on Exponentiation". In: *Information and Communications Security - 12th International Conference, ICICS 2010, Barcelona, Spain, December 15-17, 2010. Proceedings*. Ed. by Miguel Soriano, Sihan Qing, and Javier López. Vol. 6476. Lecture Notes in Computer Science. Springer, 2010, pp. 46–61. URL: https://doi.org/10.1007/978-3-642-17650-0_5.

[Cop+09] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors". In: *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*. IEEE Computer Society, 2009, pp. 45–60. URL: https://doi.org/10.1109/SP.2009.19.

[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. URL: https://doi.org/10.1007/3-540-48059-5_25.

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. "Template Attacks". In: *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. Ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar. Vol. 2523. Lecture Notes in Computer Science. Springer, 2002, pp. 13–28. URL: https://doi.org/10.1007/3-540-36400-5_3.

[Du+15] Shaoyu Du, Zhenqi Li, Bin Zhang, and Dongdai Lin. "Combined Cache Timing Attacks and Template Attacks on Stream Cipher MUGI". In: *Information Security Practice and Experience - 11th International Conference, ISPEC 2015, Beijing, China, May 5-8, 2015. Proceedings*. Ed. by Javier López and Yongdong Wu. Vol. 9065. Lecture Notes in Computer Science. Springer, 2015, pp. 235–249. URL: https://doi.org/10.1007/978-3-319-17533-1_17.

[Dug+16] Margaux Dugardin, Louiza Papachristodoulou, Zakaria Najm, Lejla Batina, Jean-Luc Danger, and Sylvain Guilley. "Dismantling Real-World ECC with Horizontal and Vertical Template Attacks". In: *Constructive Side-Channel Analysis and Secure Design - 7th International Workshop, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers*. Ed. by François-Xavier Standaert and Elisabeth Oswald. Vol. 9689. Lecture Notes in Computer Science. Springer, 2016, pp. 88–108. URL: https://doi.org/10.1007/978-3-319-43283-0_6.

[Ge+18] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware". In: *J. Cryptographic Engineering* 8.1 (2018), pp. 1–27. URL: https://doi.org/10.1007/s13389-016-0141-6.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches". In: *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. Ed. by Jaeyeon Jung and Thorsten Holz. USENIX Association, 2015, pp. 897–912. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[HMV04] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Professional Computing. Springer-Verlag, New York, 2004, pp. xx+311. ISBN: 0-387-95273-X. URL: https://doi.org/10.1016/s0012-365x(04)00102-5.

[HPB05] Mustapha Hedabou, Pierre Pinel, and Lucien Bénéteau. "Countermeasures for Preventing Comb Method Against SCA Attacks". In: *Information Security Practice and Experience, First International Conference, ISPEC 2005, Singapore, April 11-14, 2005, Proceedings*. Ed. by Robert H. Deng, Feng Bao, HweeHwa Pang, and Jianying Zhou. Vol. 3439. Lecture Notes in Computer Science. Springer, 2005, pp. 85–96. URL: https://doi.org/10.1007/978-3-540-31979-5_8.

[HS01] Nick Howgrave-Graham and Nigel P. Smart. "Lattice Attacks on Digital Signature Schemes". In: *Des. Codes Cryptogr.* 23.3 (2001), pp. 283–290. URL: https://doi.org/10.1023/A:1011214926272.

[Joy07] Marc Joye. "Highly Regular Right-to-Left Algorithms for Scalar Multiplication". In: *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*. Ed. by Pascal Paillier and Ingrid Verbauwhede. Vol. 4727. Lecture Notes in Computer Science. Springer, 2007, pp. 135–147. URL: https://doi.org/10.1007/978-3-540-74735-2_10.

[JY02] Marc Joye and Sung-Ming Yen. "The Montgomery Powering Ladder". In: *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. Ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar. Vol. 2523. Lecture Notes in Computer Science. Springer, 2002, pp. 291–302. URL: https://doi.org/10.1007/3-540-36400-5_22.

[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. 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. URL: https://doi.org/10.1007/3-540-68697-5_9.

[Liu+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 605–622. URL: https://doi.org/10.1109/SP.2015.43.

[Luo18] Chao Luo. "Novel Side-Channel Attacks on Emerging Cryptographic Algorithms and Computing Systems". PhD thesis. Northeastern University, 2018. URL: http://hdl.handle.net/2047/D20316363.

[Mai+13] Diana Maimut, Cédric Murdica, David Naccache, and Mehdi Tibouchi. "Fault Attacks on Projective-to-Affine Coordinates Conversion". In: *Constructive Side-Channel Analysis and Secure Design - 4th International Workshop, COSADE 2013, Paris, France, March 6-8, 2013, Revised Selected Papers*. Ed. by Emmanuel Prouff. Vol. 7864. Lecture Notes in Computer Science. Springer, 2013, pp. 46–61. URL: https://doi.org/10.1007/978-3-642-40026-1_4.

[MO08] Marcel Medwed and Elisabeth Oswald. "Template Attacks on ECDSA". In: *Information Security Applications, 9th International Workshop, WISA 2008, Jeju Island, Korea, September 23-25, 2008, Revised Selected Papers*. Ed. by Kyo-Il Chung, Kiwook Sohn, and Moti Yung. Vol. 5379. Lecture Notes in Computer Science. Springer, 2008, pp. 14–27. URL: https://doi.org/10.1007/978-3-642-00306-6_2.

[Mog+20] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 469–486. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-copycat.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007. ISBN: 978-0-387-30857-9. URL: https://doi.org/10.1007/978-0-387-38162-6.

[MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. "Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages". 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. 199–216. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/mccann.

[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. URL: https://doi.org/10.1023/A:1025436905711.

[NSS04] David Naccache, Nigel P. Smart, and Jacques Stern. "Projective Coordinates Leak". In: *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*. Ed. by Christian Cachin and Jan Camenisch. Vol. 3027. Lecture Notes in Computer Science. Springer, 2004, pp. 257–267. URL: https://doi.org/10.1007/978-3-540-24676-3_16.

[OPB16] Elif Ozgen, Louiza Papachristodoulou, and Lejla Batina. "Template attacks using classification algorithms". In: *2016 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2016, McLean, VA, USA, May 3-5, 2016*. Ed. by William H. Robinson, Swarup Bhunia, and Ryan Kastner. IEEE Computer Society, 2016, pp. 242–247. URL: https://doi.org/10.1109/HST.2016.7495589.

[Pap19] Louiza Papachristodoulou. "Masking Curves: Side-Channel Attacks on Elliptic Curve Cryptography and Countermeasures". PhD thesis. Radboud University, 2019. URL: http://hdl.handle.net/2066/201163.

[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+20] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. "Certified Side Channels". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 2021–2038. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/garcia.

[PMO07] Thomas Popp, Stefan Mangard, and Elisabeth Oswald. "Power Analysis Attacks and Countermeasures". In: *IEEE Design & Test of Computers* 24.6 (2007), pp. 535–543. URL: https://doi.org/10.1109/MDT.2007.200.

[Roe19] N.W.A. Roelofs. "Online Template Attack on ECDSA: Extracting keys via the other side". MA thesis. Radboud University, 2019. URL: https://www.ru.nl/publish/pages/769526/z3_master_thesis_roelofs_v1.pdf.

[San18] Tom Sandmann. "Online Template Attack on a Hardware Implementation of FourQ". MA thesis. Radboud University, 2018. URL: https://www.ru.nl/publish/pages/769526/tom_sandmann-master_thesis_final.pdf.

[Sch+17] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*. Ed. by Michalis Polychronakis and Michael Meier. Vol. 10327. Lecture Notes in Computer Science. Springer, 2017, pp. 3–24. URL: https://doi.org/10.1007/978-3-319-60876-1_1.

[Shi+16] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. "Preventing Page Faults from Telling Your Secrets". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*. Ed. by Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang. ACM, 2016, pp. 317–328. URL: https://doi.org/10.1145/2897845.2897885.

[Sze19] Jakub Szefer. "Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses". In: *J. Hardware and Systems Security* 3.3 (2019), pp. 219–234. URL: https://doi.org/10.1007/s41635-018-0046-1.

[TPV17] Chia-che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". 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. 645–658. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[VB+17] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. "Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution". 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. 1041–1056. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-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. URL: https://doi.org/10.1145/3152701.3152706.

[Wan+17] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX". 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. 2421–2434. URL: https://doi.org/10.1145/3133956.3134038.

[Wei+18] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. "DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries". 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. 603–620. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/weiser.

[Wei+20] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. "Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 1767–1784. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/weiser.

[WHS12] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. "A Cache Timing Attack on AES in Virtualization Environments". In: *Financial Cryptography and Data Security - 16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers*. Ed. by Angelos D. Keromytis. Vol. 7397. Lecture Notes in Computer Science. Springer, 2012, pp. 314–328. URL: https://doi.org/10.1007/978-3-642-32946-3_23.

[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. 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. URL: https://doi.org/10.1109/SP.2015.45.

[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.

A Disclosure and library responses

We contacted the development teams of the analyzed libraries to disclose the leakage sources, proposing countermeasures according to the analysis in Section 3.4. We did not focus the disclosure nor our work on specific vulnerabilities, but instead on a generic countermeasure approach. Therefore, the proposed countermeasures were independent of the exploited code path we used in our practical evaluation (Section 5).

During disclosure, we highlighted the strengths of the projective coordinates randomization countermeasure over constant-time code (i.e. address-independent code):

1. Constant-time code only prevents address-based leakage OTAs (likely sufficient to prevent microarchitecture attacks), but does not thwart value-base leakage OTAs like power consumption.

2. The experiments in this work employed `PageTracer` and `CopyCat` traces, therefore even an implementation with a safe pmf could have leaks under other channels.

3. These libraries have a large inherited codebase developed when side-channel attacks were not a concept, therefore migrating the entire ECC stack to constant-time code, while doable, require significant effort.

Regarding the individual responses from libraries, `libgcrypt` does not implement projective coordinates randomization. However, as part of the disclosure, we were asked to test an upcoming constant-time implementation for the Curve25519 field, hence we repeated the experiments performed in Section 4.2. We found that all page combinations generated a single leakage trace under `PageTracer` and `CopyCat`. Therefore, according to our evaluation flow, said code can be considered OTA-safe regarding these side channels.

The `mbedTLS` team decided to randomize the projective coordinates *before* and *after* the scalar multiplication, preventing both *forward* and *backward* OTAs. This was logical, since their library already featured a randomization primitive implementation. Finally, they fixed the bug preventing randomization in some cases (see Section 5.2 for details).

As a consequence of our work, `wolfSSL` implemented the coordinates randomization countermeasure to prevent both address and value-based OTAs against its ECC implementation. In addition, their library features a very mature constant-time implementation, enabled by setting the non-default flags `WOLFSSL_SP` and `WOLFSSL_HAVE_SP_ECC`. Upon their request, we tested said implementation, repeating the Section 4.4 experiments as part of the disclosure, and found no evidence of leakage under `PageTracer` and `CopyCat` side channels.