High Order Countermeasures for Elliptic-Curve Implementations with Noisy Leakage Security

Abstract. Elliptic-curve implementations protected with state-of-the-art countermeasures against side-channel attacks might still be vulnerable to advanced attacks that recover secret information from a single leakage trace. The effectiveness of these attacks is boosted by the emergence of deep learning techniques for side-channel analysis which relax the control or knowledge an adversary must have on the target implementation. In this paper, we provide generic countermeasures to withstand these attacks for a wide range of regular elliptic-curve implementations. We first introduce a framework to formally model a *regular algebraic program* which consists in a sequence of algebraic operations indexed by key-dependent values. We then introduce a generic countermeasure to protect these types of programs against advanced single-trace side-channel attacks. Our scheme achieves provable security in the noisy leakage model under a formal assumption on the leakage of randomized variables. To demonstrate the applicability of our solution, we provide concrete examples on several widely deployed scalar multiplication algorithms and report some benchmarks for a protected implementation on a smart card.

Keywords: Side-channel countermeasures, elliptic-curve cryptography, masking, noisy leakage model, collision attacks, deep learning-based SCA.

1 Introduction

Elliptic Curve Cryptography (ECC) implementations protected with state-of-the-art countermeasures against side-channel attacks might still be vulnerable to advanced attacks that recover significant information on the secret (randomized) scalar with a single leakage trace. Among these attacks, the so-called (collision-based) horizontal attacks are particularly effective [4,51,50,20,46,2,57]. The principle of these attacks is to defeat the standard randomization techniques without recovering information in the non-randomized variables but by exploiting their collision behavior. This threat is amplified by the recent development of deep learning-based side-channel attacks, which have shown to be especially powerful to generalize and improve advanced side-channel attacks such as template and/or horizontal attacks [11].

In such a context, it is of paramount importance to rely on effective countermeasures achieving quantifiable security. During the last two decades, the cryptographic community has introduced leakage models to build concrete security proofs mainly dedicated to symmetric implementations. Contrary to asymmetric implementations for which first-order countermeasures were enough to resist traditional side-channel attacks, symmetric implementations have long been proven to be vulnerable to more complex higher-order attacks [48,17].

Among the leakage models introduced as basis to reason on the security of these symmetric implementations, the most deployed one is undoubtedly the

probing leakage model introduced by Ishai, Sahai, and Wagner in 2003 [32]. In the d-probing model, a program is secure if any set of d intermediate variables is independent from the secrets. While it easily models the exponential complexity of combining an increasing number of noisy variables, the probing model is often criticized by practitioners for being too far from the reality of embedded devices. A more realistic model, referred to as *noisy leakage model* was formalized by Prouff and Rivain in 2013 [53]. In this model, every atomic operation in a program is assumed to leak a noisy function $f(x)$ of its input x, where the noisy feature is modeled by bounding the statistical distance between the prior and posterior distributions of x (given the observation $f(x)$). A program is secure in this model if the joint distribution of all the noisy intermediate variables does not reveal more than a negligible information of the secrets.

Now that asymmetric implementations have shown to be also vulnerable to higher-order attacks in the shape of the emerging single-trace attacks (*e.g.*, collision and deep learning based side-channel attacks), a similar reasoning can be applied. The same generic leakage models are expected to be used to reason on the concrete security of these implementations which expect the construction of dedicated advanced formal countermeasures achieving quantifiable security.

Our Contributions. This paper presents the first formal model of computation to capture regular implementations of ECC and other exponentiation-based cryptosystems (*e.g.*, RSA, discrete logarithm-based schemes, pairings) in the presence of noisy leakage. In a nutshell, our model of *regular algebraic program* fits any cryptographic algorithm which consists in a sequence of algebraic operations indexed by key-dependent values. We then formally define the noisy leakage model for such programs and introduce an additional assumption on the leakage, the *hideness* property, which aims to capture the effect of randomization operations on the leakage of an algebraic variable. Namely, we assume that the leakage of a randomized variable is hard to distinguish from that of a uniform random value. From this ground, we design a generic countermeasure that can be applied to a wide range of regular ECC implementations (and exponentiation-based cryptosystems)and we provide a formal security proof in the noisy leakage model under the hideness assumption. Finally, we demonstrate its applicability on several widely used scalar multiplication algorithms and provide concrete benchmarks for a protected implementation on a smart card.

Organization. Section 2 gives an overview of the state of affairs regarding side-channel attacks against ECC implementations and the existing countermeasures. Our formal model of computation and leakage is presented in Section 3. We then introduce our generic countermeasure for regular algebraic programs in Section 4 as well as its formal security proof. Applications and implementation results are finally given in Section 5.

2 ECC Implementations and Side-Channel Attacks

Elliptic-curve cryptosystems are a key ingredient of everyday-life applications to provide confidentiality and authentication. They still benefit from an exponential

security compared to the subexponential security of other systems like RSA. A common feature between the deployed elliptic-curve cryptosystems remain their core operation referred to as *scalar multiplication*. It consists in the multiplication of some point $\boldsymbol{P}$ on an elliptic curve by some (usually secret) scalar k:

$$k\boldsymbol{P} = \underbrace{\boldsymbol{P} + \cdots + \boldsymbol{P}}_{k \text{ times}} .$$

Standard scalar multiplication algorithms are similar to efficient (modular) exponentiation algorithms which scan sequentially the bits of the scalar (or exponent) and apply some operation pattern accordingly. It is well known that concrete implementations of these algorithms are prone to side-channel attacks. We summarize the state of affairs regarding these attacks and the existing protections hereafter.

2.1 Regular Scalar Multiplication Algorithms

In a naive binary scalar multiplication algorithm, such as the *double-and-add* algorithm, a loop is processed which scans the bits of the scalar and performs a point doubling followed by a conditional point addition. Whenever the current scalar bit equals 1 the addition is performed, while when it equals 0 the addition is skipped. In the same way, a naive binary exponentiation algorithm (the *square-and-multiply*) performs a square and a conditional multiplication at each iteration depending on the current exponent bit. The conditional operations performed in these algorithms make possible *simple power analysis* (SPA) attacks exploiting a single leakage trace to recover the secret scalar [41]. The reason of such a flaw is that point additions and point doublings (or equivalently square and multiplications) have different operation flows and hence induce different patterns in a leakage trace. A straight observation of these patterns on a single leakage trace might directly reveal the whole secret scalar.

In order to withstand SPA, scalar multiplication algorithms (or exponentiation algorithms) must be rendered regular, namely they should have a constant control flow, independent of the scalar value. A first solution is to make addition and doubling patterns indistinguishable. This can be achieved by using unified formulae for point addition and point doubling [9] or by the mean of side-channel atomicity [13]. In the latter case, the principle is to build point addition and doubling from the same atomic pattern of field operations (so that a leakage trace is made of a sequence of this single atomic pattern). Another possibility is to render the scalar multiplication algorithm regular by nature, independently of the pattern of field operations in each point operation. Namely, the control flow is identical in each iteration of the scalar multiplication whatever the scalar value. This approach was first followed by Coron in [16] who proposed to perform a dummy addition in the binary algorithm loop whenever the scalar bit equals 0. The obtained *double-and-add-always* algorithm performs a point doubling and a point addition at every loop iteration and the scalar bits are no more distinguishable from a leakage trace.

Extensive literature exists on regular scalar multiplication algorithms, see for instance [37,38,56,22,55]. Examples of well deployed regular algorithms which en-

joy attractive features are the Montgomery ladder [44][1] and the fixed-window left-to-right scalar multiplication algorithm, respectively displayed in Alg. 1 and 2. For the latter algorithm, we assume that the scalar in encoded in such a way that its digits belong to a basis $\mathbb{B} \subset \mathbb{Z}$. We note that in order to avoid timing leakage, one should either use a basis such that $0 \notin \mathbb{B}$ (see for instance [39]) or a complete elliptic-curve addition formula (see for instance [6,55]). The former is one of the most popular regular binary algorithm for scalar multiplication on elliptic curves. It relies on two point registers R_0 and R_1 whose difference is constantly equal to the initial point $\boldsymbol{P}$ at the end of each iteration. The latter window technique may be preferred when more memory is available. In this context, the scalar is divided into w-bit elements and each iteration depends on a regular window of w scalar bits rather than on a single one.

Algorithm 1 Montgomery ladder

Input: $\boldsymbol{P}$, $\boldsymbol{k} = (k_0, k_1, \ldots, k_{n-1})_2$
Output: $\boldsymbol{Q} = [\boldsymbol{k}]\boldsymbol{P}$
1. $R_0 \leftarrow \boldsymbol{\mathcal{O}}$
2. $R_1 \leftarrow \boldsymbol{P}$
3. **for** $i = n-1$ **downto** 0 **do**
4. $\quad b \leftarrow k_i$
5. $\quad R_{1-b} \leftarrow R_{1-b} + R_b$
6. $\quad R_b \leftarrow 2 \cdot R_b$
7. **end for**
8. **return** R_0

Algorithm 2 Fixed-window left-to-right scalar multiplication

Input: $\boldsymbol{P}$, $\boldsymbol{k} = (k_0, k_1, \ldots, k_{m-1})_{2^w}$
Output: $\boldsymbol{Q} = [\boldsymbol{k}]\boldsymbol{P}$
1. **forall** $d \in \mathbb{B}$ **do** $T[d] = [d]\boldsymbol{P}$
2. $R \leftarrow T[k_{m-1}]$
3. **for** $i = m-1$ **downto** 0 **do**
4. $\quad R \leftarrow 2^w R$ // w doublings
5. $\quad R \leftarrow R + T[k_i]$
6. **end for**
7. **return** R

We note that whereas such regular algorithms withstand SPA at a first glance, one should also be careful with operations performing data dependent memory accesses. Indeed such algorithms might be vulnerable to cache attacks and other advanced micro-architectural attacks [49,5]. A good practice while implementing such algorithms is to perform operations of the form $R_{1-b} \leftarrow R_{1-b} + R_b$ in a way that avoids data-dependent memory access. Although (correctly implemented) regular algorithms are secure against SPA (and cache attacks), they might still be vulnerable to side-channel attacks, and in particular to so-called *differential power analysis* (DPA).

2.2 Randomization Techniques against DPA

Differential Power Analysis (DPA) are among the most powerful side-channel attacks [41]. The principle of these attacks is to target the leakage on intermediate variables of the computation that mix some known but varying data (typically an input plaintext) together with fixed secret data (typically the target secret key). For some early variables in the computation the number of secret bits involved in the expression is limited and can be exhaustively guessed. The different key guesses can then be (in)validated by correlating the predicted variables with

[1] The double-and-add algorithm proposed by Joye [37] is very close to the Montgomery ladder displayed in Algorithm 1. Namely, Steps 5 and 6 are replaced by the instruction $R_{1-b} \leftarrow 2 \cdot R_{1-b} + R_b$ and the loop is reverted.

the leakage on the actual variables. Different statistical tools can be used to extract such a dependency, see for instance [41,8,43,3], and scalar multiplication algorithms (in which the secret bits are introduced little by little) are typical targets for these attacks.

Scalar multiplication algorithms might also be vulnerable to so-called *address-bit DPA* [33] which consists in targeting the direct manipulation of the sensitive scalar bits (or digits) or any address that depends on those bits (typically the memory addresses of the registers R_b or R_{1-b} in Algorithm 1).

A sound countermeasure to defeat (address-bit) DPA consists in randomizing the intermediate variables, in order to break the dependency between the key-dependent variables and the leakage. This principle, often referred to as *masking*, has been widely applied to different cryptographic algorithms. Dedicated countermeasures have been exhibited by the cryptographic community to secure scalar multiplication algorithms against both kinds of DPA. We briefly recall hereafter some of the most widely used randomization techniques for ECC implementations, which apply to the scalar, the points (projective) coordinates, and the field elements involved in point operations.

Randomization of the scalar. In the context of scalar multiplications on elliptic curves, randomizing the private exponent is the first countermeasure provided by Coron [16] to thwart DPA. In a nutshell, such a scalar multiplication involves a private scalar k and a public elliptic point $\boldsymbol{P}$ and is performed over a curve of order q. Randomizing the private scalar k consists in adding a random multiple of q to k, namely choosing a random number r and defining:

$$k' \leftarrow k + r \cdot q \ .$$

Because $q\boldsymbol{P}$ equals the point at infinity $\boldsymbol{\mathcal{O}}$, the initial multiplication can be performed directly with scalar k' and yield $k'\boldsymbol{P} = (k + r \cdot q)\boldsymbol{P} = k\boldsymbol{P} + r\boldsymbol{\mathcal{O}} = k\boldsymbol{P}$ as output. Variants of this countermeasure consist in dividing the scalar into random shares $k = k_1 + \cdots + k_n \pmod q$, or through an euclidean division $k = \lfloor k/r \rfloor \cdot r + (k \bmod r)$.

Randomization of projective coordinates. A second countermeasure suggested by Coron to thwart DPA [16] consists in randomizing the projective coordinates of the points. Such coordinates are often used for computation on elliptic curves to save costly inversions. In a nutshell, an elliptic point $\boldsymbol{P} = (x, y)$ can be represented through three projective coordinates $(X : Y : Z)$ such that $x = X/Z$ and $y = Y/Z$. The idea of the randomization here is to multiply each of the three coordinates by a random $r \neq 0$, that is defining

$$X' \leftarrow X \cdot r \ ; \quad Y' \leftarrow Y \cdot r \ ; \quad Z' \leftarrow Z \cdot r \ ;$$

(where the above multiplication are performed on the base field). By definition of the projective coordinates, the obtained projective point $(X' : Y' : Z')$ is still a representative of $\boldsymbol{P}$. This principle further generalizes to the widely used Jacobian coordinates, for which $(X : Y : Z)$ represents the point (x, y) with $x = X/Z^2$ and $y = Y/Z^3$, as well as to other inversion-free coordinate systems, *e.g.*, for Edwards curves [30,7].

Randomization of field elements. Finally, randomization techniques might also directly apply to the field elements which compose the coordinates of the points and which are processed in the point operations. In particular, on a prime field elliptic curve, all the intermediate variables of the computation are represented as integer modulo p, where p is the prime characteristic of the base field. A standard randomization technique for such modular integers is to lift them on the ring $\mathbb{Z}_{hp}$, for some factor h, and to randomize them as

$$x' \leftarrow x + r \cdot p \pmod{hp}$$

where r is some random integer. The modular arithmetic is then performed modulo hp, thus enabling possible re-randomization from time to time with the addition of a fresh random multiple of p. At the end of the computation, a reduction modulo p enables to retrieve the correct result, since the relation $x' \bmod p = (x + r \cdot p) \bmod p = x$ is maintained throughout the computation.

Randomization of memory addresses. Another randomization technique usually applied to protect an implementation against address-bit DPA consists in randomizing the memory addresses (as suggested e.g. in [29]) whenever they depend on secret data. Such memory address randomization is especially amenable to a Boolean masking of the scalar, as developed in [34] and later extended in [36][2].

2.3 Advanced Single-Trace Side-Channel Attacks

Standard DPA and address-bit DPA cannot be directly applied in the presence of a good combination of randomization techniques. However, more advanced side-channel attacks might still be possible. When the scalar is randomized, one cannot use correlation techniques (as in standard DPA) or averaging (as in address-bit DPA) over several executions. In such a context, the adversary has no choice but to rely on *single-trace attacks* which exploit a single leakage trace to infer significant information on the (randomized) scalar. Note that partial information on randomized scalars might still be exploitable, either in the fixed scalar case [58,57] or in the nonce scalar case [25], but this partial information must still be significant to retrieve the original scalar (or to break the underlying scheme in the nonce scenario).

Several attack techniques exist which aim at making the most of a single leakage trace to fully recover the (randomized) scalar, or at least a significant part of it. Among these attack techniques, template attacks, (collision-based) horizontal attacks, and deep learning-based attacks are today the main threats of current protected ECC implementations.

The very powerful *template attacks* were first introduced by Chari, Rao and Rohatgi in 2002 [12]. The main idea of such attacks is to estimate the likelihood of a key guess based on a profiling phase where the manipulation of all key guesses were recorded. As all the so-called *profiling attacks*, the adversarial model is

[2] Izumi et al. [36] consider a slightly different leakage model in which the attacker can distinguish whether a register is overwritten by the same data.

quite strong as a copy device is expected to be accessible for the attacker with the ability to choose or at least modify the secret material.

In parallel, *horizontal attacks* [14] exploit algebraic dependencies between several data manipulated at different points in time during a single execution[3]. The detection of such dependencies may strongly reduce the number of key hypotheses. A particular sub-cases of horizontal attacks is known as *collision attacks* [59] which directly exploit the presence of colliding values during a computation whose number of occurrences is correlated to the secret. In particular, it has been shown that collision-based horizontal attacks could defeat ECC implementations [31], even when protected by a combination of scalar randomization and point coordinate randomization [4]. In a nutshell, such attacks consists in detecting collisions within intermediate variables of the algorithms in order to infer information on the scalar bits. For example, on Algorithm 2, one may observe a collision between the point $T[k_i]$ added to R in Step 5 and the point $T[d] = [d]\boldsymbol{P}$ computed at Step 1, for a given $d \in [0, 2^w)$, and hence deduce $k_i = d$. As extensively discussed in [4], this simple principle can be applied to a wide range of ECC implementations and it allows the attacker to defeat standard randomization techniques which does not affect the collision behavior of the algorithm intermediate variables.

A very recent but already large sequence of works suggest to apply *deep learning techniques*, used so far in numerous fields (e.g., document recognition), to side-channel analysis [10,54,40]. Deep learning-based side-channel attacks are often categorized as profiling attacks as they generally are organized in two steps: a profiling phase in which training side-channel traces are learned and an attacking phase in which real side-traces are processed to recover a secret. These attacks happen to be very powerful. Indeed, they can capture (collision-based) horizontal attacks with the advantages of the template attacks but without manually isolating interesting points in time. By detecting horizontal and vertical dependencies in power traces, deep learning-based side-channel attacks appear as one of the main current threat against cryptographic implementations. In particular, it has been demonstrated in [11] that deep learning-based attacks could defeat implementation of RSA protected by classical (first-order) randomization countermeasures. Weissbart, Picek, and Batina further demonstrated that convolutional neural networks could be used to perform profiling attacks on EdDSA using Curve25519 [62]. Together with Chmielewski, the authors then extended their work to target protected even implementations (*e.g.*, with projective coordinate re-randomization and scalar randomization) [61]. In the same vein, Mukhtar et al. [45] get use of convolutional neural networks to successfully attack elliptic-curve scalar multiplications (based on Montgomery Power Ladder algorithm) protected with Coron's countermeasures.

2.4 Related Works

Boolean masking of the scalar was first proposed by Itoh et al. [34] as a memory address randomization in order to thwart address-bit DPA. Unlike previous coun-

[3] *Doubling attacks*, introduced by Fouque and Valette in 2003 [23] are close variants but detect dependencies on carefully chosen iterations of two related executions.

termeasures in the same vein, *i.e.*, the exponent splitting countermeasure [15] and the randomized window method [35], which required at least twice the initial execution time, Itoh et al's proposal only suffers a small overhead. In a nutshell, since the security threat comes from the dependency between the addresses of the registers and the secret bit values, the authors followed the idea of first-order Boolean masking to randomize the latter. Namely, they proposed to xor a one-time random value to the secret key to perfectly randomize the register addresses. Other countermeasures are then still necessary to further provide resistance against classical (*i.e.* data-bit) DPA.

This first proposal was later extended by Izumi et al. [36] to additionally circumvent DPA with overwrites. While Itoh et al. focused on side-channel attackers observing intermediate variables or addresses separately, embedded devices may actually leak information on two variables that are consecutively stored in the same register at the same time. In order to ensure protection against the latter attacks, Izumi et al. modified Itoh et al.'s countermeasure to make sure that the address of the source register was always different from the address of the destination register. Despite this additional constraint, the authors also managed to achieve better performances in terms of data and time complexity.

In 2018, Tunstall, Papachristodoulou, and Papagiannopoulos [60] went a step further with a countermeasure which was not limited to thwart address-bit DPA. They reuse the idea of Boolean masking but also protect the exponent shares against classical (data-bit) DPA. Concretely, the authors are able to achieve security against a broader range of first-order side-channel attacks (exploiting the length of the exponent and other intermediate values) as long as the intermediate states of some registers do not leak. From this assumption, the authors demonstrate that an attacker would need a combination of leakage (in the sense of higher-order side-channel attacks) to recover the exponent.

2.5 Our Approach

Our work aims to go one step further by designing and proving a generic countermeasure achieving quantifiable security levels for any masking order while considering the leakage of all the intermediate states. Still relying on Boolean masking to protect asymmetric implementations, it goes beyond the previous sequence of works on several aspects:

- *Masking order:* Our countermeasure is designed and is proven secure for *any* masking order while the countermeasures from [34], [36], and [60] are limited to thwart first-order side-channel attacks. Hence the security level can be adapted depending on the actual leakage through the masking order which acts as a security parameter. In particular, attacks exploiting several collisions are covered by our security model.
- *Leakage on algebraic variables:* Several countermeasures are limited to masking the exponent and do not consider any leakage on the algebraic variables (*e.g.*, [34,36,60]). However, these variables arguably generate most of the leakage in an exponentiation or a scalar multiplication through group operations (in particular multiplications).

– *Formal proof:* While many countermeasures only come with light/informal statements (*e.g.*, [60]) or experiments on some specific devices, such methods are not enough for proving higher-order security. In this paper, we provide the first formal security proof under a up-to-date formal leakage model with well defined assumptions.

Finally, our work puts forward a novel approach to prove the security of elliptic-curve / public-key implementations in the presence of leakage, which is currently missing. We believe our formal framework (with the proper definition of regular algebraic program and the introduction of the hideness assumption) offers a fertile ground for further research and formal results on leakage-resistant elliptic-curve / public-key implementations.

3 Computation and Leakage Models

3.1 Mathematical Background and Notations

All along the paper, the interval $[i, j]$ or $[i, j)$ with $i, j \in \mathbb{Z}$ are understood as integer interval, *i.e.* $\mathbb{Z} \cap [i, j]$ and $\mathbb{Z} \cap [i, j)$.

Let $p_X : x \mapsto \Pr(X = x)$ denote the probability mass function (pmf) of a discrete random variable X. Let us further denote by $\overrightarrow{p_X}$ the $|\mathcal{X}|$-dimensional vector made of the pmf outputs: $\overrightarrow{p_X} = (p_X(x))_{x \in \mathcal{X}}$. The statistical distance between two random variables X and Y of domain $\mathcal{X}$ is defined as $\Delta(X; Y) := \|\overrightarrow{p_X} - \overrightarrow{p_Y}\|$ with respect to some norm $\|\cdot\|$. Unless otherwise specified, we will consider the statistical distance based on the L_1 norm weighted by $\frac{1}{2}$, as traditionally used in cryptography, which is

$$\Delta_1(X; Y) := \frac{1}{2} \sum_{x \in \mathcal{X}} \mid p_X(x) - p_Y(x) \mid \ . \tag{1}$$

We shall further make use of the notion of *statistical closeness* inherited from the above distance. Namely, two random variables X and Y will be said to be *ε-close* for some $\varepsilon \geq 0$, denoted $X \approx_\varepsilon Y$, if they verify $\Delta_1(X; Y) \leq \varepsilon$. We shall denote by $X \leftarrow \mathcal{A}(in)$ the action of evaluating a (probabilistic) algorithm on input in and defining the variable X as the output. We shall further denote $X \leftarrow \mathcal{X}$ to define X as a uniform random element from a finite set $\mathcal{X}$.

3.2 Computation Model

We introduce hereafter a formal model for what we call a *regular algebraic program* (RAP). Such a program is composed of algebraic operations whose operands and results are read from and written to an *algebraic* memory (*i.e.* a memory storing algebraic variables) where the latter memory is addressed through a second type of variables (with their own memory): the *indexes*. This way, we can capture any cryptographic algorithm which consists in a sequence of algebraic operations indexed by key-dependent values, such as classical regular algorithms for elliptic curve cryptography (or other exponentiation-based cryptosystems).

We consider an algebraic structure $\mathbb{A}$ equipped with a set of 2-ary operators. For instance, $\mathbb{A}$ could be a finite field $\mathbb{F}_p$ or a ring $\mathbb{Z}_q$, in both cases equipped with addition and multiplication operators, or $\mathbb{A}$ could be an additive group, such as the set of points of an elliptic curve, equipped with an addition operator.

A *regular algebraic program* on $\mathbb{A}$ is a program that takes as input a tuple $\boldsymbol{X} \in \mathbb{A}^{\ell_X}$ and a tuple $\boldsymbol{k} \in \mathbb{Z}^{\ell_k}$, and computes as output a tuple $\boldsymbol{Y} \in \mathbb{A}^{\ell_Y}$ satisfying $\boldsymbol{Y} = f_{\boldsymbol{k}}(\boldsymbol{X})$ for an algebraic function $f_{\boldsymbol{k}}$ that belongs to some class $\{f_{\boldsymbol{k}}; \boldsymbol{k} \in \mathbb{Z}^{\ell_k}\}$. For this purpose a regular algebraic program operates with two memories, an algebraic memory and an index memory. The algebraic memory, denoted $(X_1, X_2, \ldots, X_m)$, is composed of m cells (aka algebraic variables), each one taking a value in $\mathbb{A}$. The index memory, denoted $(ind_1, ind_2, \ldots, ind_n)$, is composed of n cells (aka index variables), each one taking a value in $\mathbb{Z}$. At the beginning of the computation, the input tuples $\boldsymbol{X}$ and $\boldsymbol{k}$ are written at the beginning of the algebraic and index memories respectively; at the end of the computation, the output tuple $\boldsymbol{Y}$ is read from the beginning of the algebraic memory. The program consists of a sequence of instructions which can be of two natures: index instructions and algebraic instructions. An index instruction performs an operation (integer or Boolean operation) on the index variables. Specifically, an index instruction is of the form

$$ind_{i_1} \leftarrow \mathsf{op}(ind_{i_2}, ind_{i_3}) \quad \text{for } i_1, i_2, i_3 \in [n] \ ,$$

$$\text{or} \qquad ind_{i_1} \leftarrow ind_{i_2} \quad \text{or} \quad ind_{i_1} \leftarrow cst_{\mathbb{Z}} \quad \text{for } i_1, i_2 \in [n] \ ,$$

where op denotes some operation which is either the addition (or subtraction) on $\mathbb{Z}$, the multiplication on $\mathbb{Z}$, the bitwise addition, the bitwise multiplication, and where $cst_{\mathbb{Z}}$ denotes some constant value in $\mathbb{Z}$. An algebraic instruction performs an algebraic operation on algebraic variables, possibly indexed by index variables. Specifically, an algebraic instruction is of the form

$$X_{i_1} \leftarrow \mathsf{Op}(X_{i_2}, X_{i_3}) \quad \text{for } i_1, i_2, i_3 \in [m] \cup \{ind_1, ind_2, \ldots, ind_n\}$$

where Op denotes an operation of $\mathbb{A}$, or of the form

$$X_{i_1} \leftarrow X_{i_2} \quad \text{or} \quad X_{i_1} \leftarrow cst_{\mathbb{A}} \quad \text{for } i_1, i_2 \in [m] \cup \{ind_1, ind_2, \ldots, ind_n\}$$

where $cst_{\mathbb{A}}$ denotes some constant from $\mathbb{A}$.

In order to capture randomization techniques in this computation model, we will further consider regular algebraic programs augmented with an additional randomization operation:

$$X_{i_1} \leftarrow \mathsf{R}(X_{i_2}) \quad \text{for } i_1, i_2 \in [m] \cup \{ind_1, ind_2, \ldots, ind_n\} \ .$$

From an abstract computation level, this instruction has no effect: it simply copies the algebraic variable X_{i_2} into X_{i_1}. On the other hand, for a given implementation, the representation of elements of $\mathbb{A}$ might be randomizable. This is a case for instance when elements of $\mathbb{F}_p$ are represented by elements of the ring $\mathbb{Z}_{hp}$ (for some h) which can be (re-)randomized as $x \leftarrow x + R \cdot p \bmod hp$ for some random R. A reduction modulo p finally enables to retrieve the good

representative. Another example is the group of points of an elliptic curve in Jacobian (or other projective) coordinates. Such a point $(X : Y : Z)$ can be randomized as $(X : Y : Z) \leftarrow (XR^2 : YR^3 : ZR)$ for some random R. These are two classical examples but one could consider many other randomization techniques for different algebraic structures.

A regular algebraic program augmented with such a randomization operation is called a *randomized regular algebraic program* hereafter.

Remark 1. We note that index variables could equally be defined over some other (finite) ring, like $\mathbb{Z}_{2^w}$ to capture a w-bit architectures, but this difference has no impact on our result.

3.3 Noisy Leakage of Regular Algebraic Programs

Informally, the δ-noisy leakage model states that during the evaluation of a program P, each atomic operation leaks a noisy function $f(x)$ of its input x. The noisiness of the leakage is captured by assuming that the statistical distance between the distribution of x and the distribution of x given an observation $f(x)$ is bounded by δ. In other words, an observation $f(x)$ only implies a bounded bias on the distribution of x.

In order to get a formal definition of such a noisy function, one must consider an a priori distribution for x. In the original definition [53] and subsequent generalizations [19,52], this distribution is naturally taken to be the uniform distribution over the definition set of x. The statistical distance between X and $(X \mid f(X) = y)$ is further averaged over the observation $f(X) = y$. This gives the following definition:

Definition 1 (Noisy function [53]). *Let $\mathcal{X}$ be a finite set and let $\delta \in \mathbb{R}$. A δ-noisy function f on $\mathcal{X}$ is a function of domain $\mathcal{X} \times \{0,1\}^{\ell_R}$ for some $\ell_R \in \mathbb{N}$ such that*

$$\sum_{y \in \mathrm{Im}(f)} \Pr(Y = y) \cdot \Delta(X; (X \mid Y = y)) \leq \delta \ , \tag{2}$$

where X is a uniform random variable over $\mathcal{X}$ and where $Y = f(X, R)$ for a uniform random variable R over $\{0,1\}^{\ell_R}$.

Note that the above definition depends on the notion of statistical distance Δ. In the original definition [53], the authors suggest to use a statistical distance based on the Euclidean norm (or L_2 norm), while [19] argues that taking the L_1 norm is a more natural choice. It was further recently suggested in [52] to use a statistical distance notion based on the *relative error*, specifically:

$$\Delta_{\mathrm{RE}}((X \mid Y = y); X) = \max_{x \in \mathcal{X}} \left| \frac{\Pr(X = x \mid Y = y)}{\Pr(X = x)} - 1 \right| \ . \tag{3}$$

We note that the above notion of distance is not symmetric and the order of the argument matters: $\Delta_{\mathrm{RE}}((X \mid Y = y); X) \neq \Delta_{\mathrm{RE}}(X; (X \mid Y = y))$. Noisy functions based on this distance are referred to as *average relative error* (ARE) noisy functions in [52] since the relative error in (3) is averaged over the distribution of

the leakage Y according to Definition 1. The authors of [52] argue that using the above notion does not imply a stronger assumption on the leakage in practice, compared to the definitions based on L_1 and L_2 norms, whereas it has some advantages for security proofs based on reductions to the *random probing model*. For this reason, we shall use this notion in our security proof (see Section 4.2) although our result simply generalizes to the other notions.

Remark 2. Formally, a noisy leakage function has two arguments: an input X which corresponds to the variable (or tuple of variables) that leaks, and an input R which corresponds to the *random tape* of the function, *i.e.*, the randomness from which the noise is generated. This random tape input is freshly sampled from $\{0,1\}^{\ell_R}$ for each realization of a leaking variable. For the sake of simplicity, we shall skip the random tape argument from the exposition in the rest of the paper. The noisy leakage of a value $x \in \mathcal{X}$ (or variable) will hence be denoted $f(x)$, to mean "$f(x, R)$ for a fresh random tape $R \leftarrow \{0,1\}^{\ell_R}$", and the resulting $f(x)$ will be considered as a random variable.

Application to regular algebraic programs. We formally define the noisy leakage model hereafter in the context of our computation model. Namely, we consider a (randomized) regular algebraic program P taking as input a pair of tuples $(\boldsymbol{X}, \boldsymbol{k}) \in \mathbb{A}^{\ell_X} \times \mathbb{Z}^{\ell_k}$. We shall assume that every index instruction $ind_{i_1} \leftarrow \mathsf{op}(ind_{i_2}, ind_{i_3})$ leaks a noisy function of the input pair of indexes (ind_{i_2}, ind_{i_3}). On the other hand, an algebraic instruction $X_{i_1} \leftarrow \mathsf{Op}(X_{i_2}, X_{i_3})$ shall leak a noisy function of the input pair of algebraic values (X_{i_2}, X_{i_3}) together with the triplet of indexes (i_1, i_2, i_3). Finally, a randomization operation $X_{i_1} \leftarrow \mathsf{R}(X_{i_2})$ is assumed to leak a noisy function of $(X_{i_1}, X_{i_2}, i_1, i_2)$[4].

Remark 3. We could alternatively assume that the algebraic values and the indexes leak independently (through two noisy leakage functions) but our assumption is more general. Indeed a noisy function of the form $f(X_{i_2}, X_{i_3}, i_1, i_2, i_3) = (f_1(X_{i_2}, X_{i_3}), f_2(i_1, i_2, i_3))$ would be a particular case of our leakage model.

Remark 4. Note that the aforementioned leakage model does not directly capture physical defaults (*e.g.* glitches or transitions between variables successively stored in the same register) which are strongly dependent on both the implementation (*e.g.* choice of registers and operations) and the underlying devices. Nevertheless, this model and our subsequent proofs can be easily adapted to such specific scenarios by extending the leakage functions to a group of operations that are jointly leaking.

In order to ease the exposition, we introduce a deterministic algorithm which from a (randomized) regular algebraic program P and an input pair $(\boldsymbol{X}, \boldsymbol{k})$, produces a *computation trace* of P on input $(\boldsymbol{X}, \boldsymbol{k})$:

$$\mathcal{T} \leftarrow \mathsf{ComputTrace}(P, \boldsymbol{X}, \boldsymbol{k}) \ .$$

[4] Not that we do not need to give the attacker the (independent) internal random values that are generated for the randomization operation as the knowledge of a noisy leakage function of both the input and the ouput is actually stronger.

This computation trace contains the leaking tuples of all the instructions executed by P on input $(\boldsymbol{X}, \boldsymbol{k})$. Concretely, $\mathsf{ComputTrace}$ initializes $\mathcal{T}$ to the empty list. Then, it sequentially evaluates each instruction in P and adds its leaking tuple to $\mathcal{T}$. Specifically,

- for an index instruction $ind_{i_1} \leftarrow \mathsf{op}(ind_{i_2}, ind_{i_3})$, $\mathsf{ComputTrace}$ adds the pair (ind_{i_2}, ind_{i_3}) to $\mathcal{T}$;
- for an algebraic instruction $X_{i_1} \leftarrow \mathsf{Op}(X_{i_2}, X_{i_3})$, $\mathsf{ComputTrace}$ adds the tuple $(X_{i_2}, X_{i_3}, i_1, i_2, i_3)$ to $\mathcal{T}$;
- for a randomization operation $X_{i_1} \leftarrow \mathsf{R}(X_{i_2})$ $\mathsf{ComputTrace}$ adds the tuple $(X_{i_1}, X_{i_2}, i_1, i_2)$ to $\mathcal{T}$.

Then, we consider a probabilistic algorithm $\mathsf{LeakageSampler}$ which on input a computation trace $\mathcal{T}$ and a δ-noisy function family $\mathcal{F} = \{f_i\}_i$ outputs a list of leakage values:

$$\mathcal{L} \leftarrow \mathsf{LeakageSampler}(\mathcal{T}, \mathcal{F}) \ ,$$

such that $\mathcal{L} = \{f_i(d_i)\}$ for $\mathcal{T} = \{d_i\}_i$. Specifically, for every element d_i in the computation trace $\mathcal{T}$, the $\mathsf{LeakageSampler}$ algorithm samples a random tape r_i uniformly at random from $\{0,1\}^{\ell_R}$ and adds the evaluation $f_i(d_i, r_i)$ to the leakage trace $\mathcal{L}$.

We can now formally define the noisy leakage of a program.

Definition 2 (Noisy leakage). *Let $\mathcal{F}$ be a family of δ-noisy functions and let P be a (randomized) regular algebraic program. The noisy leakage of P on input $(\boldsymbol{X}, \boldsymbol{k})$ is the distribution $\mathcal{L}(P, (\boldsymbol{X}, \boldsymbol{k}))$ obtained by composing the computation-traces and assign-leakage samplers as*

$$\mathcal{L}_{\mathcal{F}}(P, (\boldsymbol{X}, \boldsymbol{k})) = \mathsf{LeakageSampler}(\mathsf{ComputTrace}(P, \boldsymbol{X}, \boldsymbol{k}), \mathcal{F}) \ .$$

Remark 5. A classical noisy function which fairly well fits the reality of embedded devices could be defined as the sum of the input variable's Hamming weight and a random Gaussian noise. But note that as it is defined, the noisy leakage also captures the well known *random probing leakage* [32,19] which states that during the evaluation of a program P, each intermediate variable leaks its value with some probability (and leaks nothing otherwise). This specific model would require the noisy function to be defined as the identity of its input variable with some probability (computed from its second input) and to an empty leakage otherwise.

Remark 6. We stress that for the considered randomization operations, the random bits sampled in an execution $X_{i_1} \leftarrow \mathsf{R}(X_{i_2})$ can be expressed as a deterministic function of the input-output pair (X_{i_1}, X_{i_2}). Therefore, modelling the leakage as a noisy function of (X_{i_1}, X_{i_2}) is without loss of generality compared to modeling the leakage as a noisy function of the input and the randomness. Our convention is simply more convenient since we do not need to make the randomness explicit.

We can now define the notion of *leakage resilience* for a randomized regular arithmetic program. Let Enc be a probabilistic encoding algorithm that maps an

index variable $\boldsymbol{k} \in \mathbb{Z}^{\ell_k}$ to an encoded input $\widehat{\boldsymbol{k}} \leftarrow \mathsf{Enc}(\boldsymbol{k}) \in \mathbb{Z}^{\ell'_k}$, with $\ell'_k \geq \ell_k$. The notion of leakage resilience is defined with respect to a class of leakage functions $\mathcal{F}$ (noisy leakage functions in our context) and an encoding Enc as follows.

Definition 3 (Leakage resilience). *A randomized regular arithmetic program Π is said* ε-leakage resilient *against a class of leakage functions $\mathcal{F}$ with respect to encoding* Enc *if there exists a simulator* Sim *such that for every $(\boldsymbol{X}, \boldsymbol{k})$:*

$$\mathsf{Sim}(P, \boldsymbol{X}) \approx_\varepsilon \mathcal{L}_{\mathcal{F}}(\Pi, (\boldsymbol{X}, \mathsf{Enc}(\boldsymbol{k}))).$$

3.4 Leakage of Randomized Variables

Following this conclusion, we introduce an additional assumption to capture the effect of the randomization operation R on the leakage of an algebraic variable. In a nutshell, we require that the leakage of a randomized variable $\mathsf{R}(X)$ is hard to distinguish from the leakage of a random value R uniformly sampled from $\mathcal{A}$. This is formalized hereafter by the notion of *ε-hideness*.

Simple definition and examples. In order to simplify the exposition, we first consider a single noisy function f with a single input variable from $\mathbb{A}$, and generalize the notion later.

Definition 4 (Hideness, simple version). *Let f be a noisy function of domain $\mathcal{A}$ and let R be a randomization operation. The pair (f, R) is said to be* ε-hiding *if for every $x \in \mathbb{A}$, the distribution of $f(\mathsf{R}(x))$ is ε-close to the distribution of $f(U)$ where U is a uniform random variable over $\mathbb{A}$.*

In the above definition, the notion of randomization operation should be formally understood as a probabilistic algorithm and $\mathsf{R}(x)$ as a random variable. We illustrate the hideness notion with the two following:

Example 1. Let us further consider the randomization operation which maps an element of $\mathbb{Z}_p$ to a randomized representation in $\mathbb{Z}_{hp}$, namely

$$\mathsf{R}(x) = x + R \cdot p \quad \text{with} \quad R \leftarrow [0, h) \ .$$

We can say that the pair (f, R) is ε-hiding if the statistical distance between the two distributions

$$f(\mathsf{R}(x)) = f(x + R_1 \cdot p) \quad \text{and} \quad f(R_2) \quad \text{with} \quad \begin{cases} R_1 \leftarrow [0, h) \\ R_2 \leftarrow [0, hp) \end{cases}$$

is upper bounded by ε. Or equivalently, the best distinguishing algorithm of the two above distributions (without limit of time and memory) has a maximal success probability of $\frac{1}{2} + \varepsilon$.

Example 2. A second example is the randomization operation with maps a point defined by its Jacobian coordinates on an elliptic curve of base field $\mathbb{F}_p$ to a randomized representation, namely

$$\mathsf{R}(\boldsymbol{P}) = \mathsf{R}((X : Y : Z)) = (X\lambda^2 : Y\lambda^3 : Z\lambda) \quad \text{with} \quad \lambda \leftarrow \mathbb{F}_p^* \ .$$

We consider a leakage function f applied on points of this elliptic curve. Similarly, we can say that the pair (f, R) is ε-hiding if the statistical distance between the two distributions

$$f(\mathsf{R}(\boldsymbol{P})) = f((X\lambda^2 : Y\lambda^3 : Z\lambda)) \quad \text{and} \quad f(\boldsymbol{U})$$

with $\lambda \leftarrow \mathbb{F}_p^*$ and $\boldsymbol{U} \leftarrow (\mathbb{F}_p)^3$, is upper bounded by ε.

In Appendix A, we provide experimental results giving some practical evidence regarding the hideness assumption in the Hamming weight with Gaussian noise leakage model. We notably show that the statistical distance between the two distributions exponentially decreases with the randomness length $|h|$ or $|\lambda|$. More generally, it would be interesting to investigate the hideness assumption in practice for different class of leakage functions. These questions are related to the general open issue of defeating this type of randomization techniques from typical side-channel leakage beyond the fact of observing the presence or absence of collisions.

This issue has not been widely investigated which is presumably due to the hardness of exploiting such leakage. To the best of our knowledge, only one recent work studies how to defeat the randomization of projective (and Jacobian) coordinates [1]. The authors investigate strategies to recover the random value λ by exploiting the leakage of the randomization operation $\mathsf{R}(\boldsymbol{P})$ assuming a known point $\boldsymbol{P}$ (typically the initial point randomization). Among many contributions, this work illustrates the difficulty to efficiently exploit the leakage on point randomization even in the favorable context of 8-bit devices.

Complete definition. We now introduce the complete definition of ε-hideness for a randomized regular algebraic program. In such a program, after randomization, an algebraic variable may be input of several subsequent operations and hence leak several time. Moreover, we should also capture the leakage of variables defined as functions of one or several randomized variables. In a nutshell, we generalize the ε-hideness definition to the following cases:

– A leakage function f associated to a multi-input operation, *i.e.* applying to several algebraic variables and index variables. For such a function f, the ε-hideness means

$$f(\mathsf{R}(X), \boldsymbol{W}, \boldsymbol{i}) \approx_\varepsilon f(R, \boldsymbol{W}, \boldsymbol{i}) \ \text{ with } \ R \leftarrow \mathbb{A}$$

where $\boldsymbol{W}$ denotes a tuple of algebraic variables and $\boldsymbol{i}$ denotes a tuple of index variables.

– An algebraic variable which expression involves a previously randomized variable. In such a case, the ε-hideness of f generalizes to

$$f(g(\mathsf{R}(X), \boldsymbol{Z}), \boldsymbol{W}, \boldsymbol{i}) \approx_\varepsilon f(g(R, \boldsymbol{Z}), \boldsymbol{W}, \boldsymbol{i}) \text{ with } R \leftarrow \mathbb{A}$$

where $\boldsymbol{Y}$ and $\boldsymbol{Z}$ denote tuples of algebraic variables, $\boldsymbol{i}$ denotes a tuple of index variables, and g denotes an algebraic function $\mathbb{A}^{\ell_g} \to \mathbb{A}$.
– Several leakage functions $f_1, \ldots, f_t$ associated to several (multi-input) operations involving the same randomized variable (either as direct input or involved in the expression of an input variable). In such a case, the ε-hideness $f_1, \ldots, f_t$ is defined as:

$$\begin{aligned}\big(f_1(g_1(Y,\ldots),\ldots),\ldots,f_t(g_t(Y,\ldots),\ldots)\big) &\text{ with } Y \leftarrow \mathsf{R}(X)\\ \approx_{t\varepsilon} \big(f_1(g_1(R,\ldots),\ldots),\ldots,f_t(g_t(R,\ldots),\ldots)\big) &\text{ with } R \leftarrow \mathbb{A}\ .\end{aligned}$$

Formally, we have the following definition:

Definition 5 (Hideness). *Let $\mathcal{F}$ be a family of noisy functions and let R be a randomization operation. Let $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_t$, $\boldsymbol{Z}_1, \ldots, \boldsymbol{Z}_t$ be tuples of algebraic variables, let $\boldsymbol{i}_1, \ldots, \boldsymbol{i}_t$ be tuples of index variables, and let $g_1, \ldots, g_t$ be algebraic functions. Let $\mathcal{D}(Y)$ be the distribution defined w.r.t. $f_1, \ldots, f_t \in \mathcal{F}$ and $Y \in \mathbb{A}$ as:*

$$\mathcal{D}(Y) = \big(\begin{array}{r} f_1(g_1(Y, \boldsymbol{Z}_1), \boldsymbol{W}_1, \boldsymbol{i}_1)\\ f_2(g_2(Y, \boldsymbol{Z}_2), \boldsymbol{W}_2, \boldsymbol{i}_2)\\ \vdots\\ f_t(g_t(Y, \boldsymbol{Z}_t), \boldsymbol{W}_t, \boldsymbol{i}_t)\big)\end{array}$$

The pair $(\mathcal{F}, \mathsf{R})$ is said to be ε-hiding *if for every $f_1, \ldots, f_t \in \mathcal{F}$ and every $x \in \mathbb{A}$, the distribution $\mathcal{D}(\mathsf{R}(x))$ is $(t\varepsilon)$-close to the distribution $\mathcal{D}(R)$ where R is a uniform random variable over $\mathbb{A}$.*

Although the above assumption might seem strong, it soundly captures the class of collision-based horizontal attacks whenever setting $\varepsilon = 0$. Indeed, the attacks do not defeat the randomization and do not recover nor exploit any information on the unrandomized variables. They only observe whether two variables in the control flow of the algorithm collide or not. This collision behavior is not affected by assuming that the target program achieves ε-hideness even while setting ε to 0.

Moreover, we would like to stress that our assumption is not ideal in the sense that there always exists an ε for which a set of leakage functions (together with a randomization method) satisfies the hideness definition. We leave the assessment of such an ε in various contexts (depending on the leakage model and on the randomization technique) as an open research question. Such a question is related to the design of side-channel attacks that exploit the leakage of randomized variables to recover information on the unrandomized variables (beyond the

Algorithm 3 Generic scalar multiplication (or exponentiation) algorithm

Input: $\boldsymbol{X}$, $\boldsymbol{k} = (k_0, k_1, \ldots, k_{n-1})$
Output: $\boldsymbol{Y} = f_{\boldsymbol{k}}(\boldsymbol{X})$
1. $(X_1^0, X_2^0, \ldots, X_m^0) \leftarrow \mathsf{preprocess}(\boldsymbol{X})$
2. **for** $i = 0$ **to** $n - 1$ **do**
3. $\quad((X_1^0, X_1^1), (X_2^0, X_2^1), \ldots, (X_m^0, X_m^1)) \leftarrow \mathsf{sequence}(X_1, X_2, \ldots, X_m)$
4. $\quad(b_1, b_2, \ldots, b_m) \leftarrow \mathsf{indexes}(k_i)$
5. $\quad(X_1^0, X_2^0, \ldots, X_m^0) \leftarrow (X_1^{b_1}, X_2^{b_2}, \ldots, X_m^{b_m})$
6. **end for**
7. $\boldsymbol{Y} \leftarrow \mathsf{postprocess}(X_1^0, X_2^0, \ldots, X_m^0)$
8. **return** $\boldsymbol{Y}$

collision behavior) which to the best of our knowledge has not been addressed so far.

4 A Generic and Provably-Secure Countermeasure

4.1 The Generic Countermeasure

We consider a particular form of regular algebraic program which encompasses a large class of algorithms including elliptic-curve scalar multiplications, ring or field exponentiations, or pairing evaluations (Miller's algorithm). This generic algorithm (see Alg. 3) takes as input a tuple of algebraic variables $\boldsymbol{X}$ and a secret index variable $\boldsymbol{k} = (k_0, \ldots, k_{n-1})$ for some $n \in \mathbb{N}$. We assume that each coordinate of the input secret variable $\boldsymbol{k}$ is a w-bit integer, that is $k_i \in [0, 2^w)$ for every $i \in [0, n)$. The algorithm is composed of an initialization procedure preprocess that initializes the algebraic variables $X_1^0, X_2^0, \ldots, X_m^0$ from the input $\boldsymbol{X}$, a main loop that applies a sequence of operations on the algebraic variables, and a final procedure postprocess to generate the output. In the main loop, the procedure sequence implements a sequence of operations between algebraic variables of constant indexes, and the procedure indexes derives a set of indexes from the current index variable k_i from which the algebraic variables entering the next iteration are selected. In the following, we shall assume that this function is linear with respect to the bitwise addition. Specifically, in Step 4, each b_j is defined as a linear combination of the w bits of k_i.

For the sake of clarity, we do not strictly stick to the notations introduced in Section 3 for the algebraic and index memories. However, it should be clear from the algorithm description that

- the X notations are used for algebraic variables whereas the k and b notations are used for index variables,
- the procedures preprocess, sequence and postprocess only perform algebraic operations with constant indexes,
- the procedure indexes only perform index operations,
- Step 5 performs a sequence of m copies of algebraic variables indexed by index variables.

In the following, we shall denote N_{pre}, N_{post}, N_{seq} the number of algebraic operations performed by the preprocess, postprocess and sequence procedures respectively. We will further consider that the indexes procedure is composed of m index operations. The total number N_P of operations in Alg. 3 is hence given by:

$$N_P = N_{pre} + N_{post} + n \cdot (N_{seq} + 2m) \ . \tag{4}$$

Remark 7. In the description of Alg. 3, we implicitly assume the same sequence procedure for all the iterations but we could generalize to different $\mathsf{sequence}_i$ procedures without effect on our result. Couples of variables are used in Step 3 to record both scenarios for each input, but notice that in practice a single extra register can be enough to treat both cases for each of the m input variables.

We show in Section 5 that this general algorithm can be applied to the most widely used elliptic-curve scalar multiplication algorithms, such as the Montgomery ladder [44], the Joye ladder [37], the full signed binary algorithm [56], the fixed window double-and-add algorithm [28].

Encoding of the key. Our generic countermeasure is based on a Boolean sharing of $\boldsymbol{k}$ of *masking order* $d \in \mathbb{N}$ (which acts as security parameter of the countermeasure). Specifically, $\boldsymbol{k}$ is encoded as

$$\widehat{\boldsymbol{k}} =\leftarrow \mathsf{Enc}(\boldsymbol{k}) := ((k_0^j)_{0\le j\le d}, (k_1^j)_{0\le j\le d}, \ldots, (k_{n-1}^j)_{0\le j\le d}) \tag{5}$$

where the encoding algorithm Enc randomly samples

$$k_i^j \leftarrow [0, 2^w) \quad \text{for every } \ 0 \le i \le n-1 \ \text{ and } \ 1 \le j \le d$$

and define

$$k_i^0 \leftarrow k_i \oplus k_i^1 \oplus \cdots k_i^d \quad \text{for every } \ 0 \le i \le n-1 \ .$$

Protected algorithm. Our generic countermeasure against horizontal side-channel attacks is described in Alg. 4. The function R applies the randomization operation formalized in our randomized regular algebraic program model. The function swap performs a conditional swapping of the two input variables X^0 and X^1 depending on the third input b, so that it returns $(X^b, X^{b\oplus 1})$ (the variables are swapped iff $b = 1$). In the RAP model, this function can be implemented by the following sequence of instructions with the use of two extra algebraic variable R_0 and R_1:

1. $(R_0, R_1) \leftarrow (X^0, X^1)$
2. $X^0 \leftarrow R_b$
3. $X^1 \leftarrow R_{b\oplus 1}$

In a nutshell, Alg. 4 performs the same computation as Alg. 3 but the selection of the variables in Step 5 is protected by a Boolean masking of order d. By linearity of the indexes procedure, we have

$$b_\ell = b_\ell^0 \oplus b_\ell^1 \oplus \cdots b_\ell^d \ , \tag{6}$$

which implies that

$$\mathsf{swap}(X_\ell^0, X_\ell^1, b_\ell) = \mathsf{swap}(\ldots \mathsf{swap}(\mathsf{swap}(X_\ell^0, X_\ell^1, b_\ell^0), b_\ell^1), \ldots, b_\ell^d) \ . \tag{7}$$

for every $\ell \in [1, m]$. At the end of the inner loop, we hence correctly get (a randomized version of) the variable $X_\ell^{b_\ell}$ copied in X_ℓ^0 for every ℓ.

Algorithm 4 Generic countermeasure

Input: $\boldsymbol{X}$, $\widehat{\boldsymbol{k}} = ((k_0^j)_{0\le j\le d}, (k_1^j)_{0\le j\le d}, \ldots, (k_{n-1}^j)_{0\le j\le d})$
Output: $\boldsymbol{Y} = f_{\boldsymbol{k}}(\boldsymbol{X})$

$(X_1^0, X_2^0, \ldots, X_m^0) \leftarrow \mathsf{preprocess}(\boldsymbol{X})$
for $i = 0$ **to** $n-1$ **do**
  $((X_1^0, X_1^1), (X_2^0, X_2^1), \ldots, (X_m^0, X_m^1)) \leftarrow \mathsf{sequence}(X_1^0, X_2^0, \ldots, X_m^0)$
  for $j = 0$ **to** d **do**
    $(b_1^j, b_2^j, \ldots, b_m^j) \leftarrow \mathsf{indexes}(k_i^j)$
    for $\ell = 1$ **to** m **do**
      $X_\ell^0 \leftarrow \mathsf{R}(X_\ell^0)$
      $X_\ell^1 \leftarrow \mathsf{R}(X_\ell^1)$
      $((X_\ell^0, X_\ell^1)) \leftarrow \mathsf{swap}(X_\ell^0, X_\ell^1, b_\ell^j)$
    end for
  end for
end for
$Y \leftarrow \mathsf{postprocess}(X_1^0, X_2^0, \ldots, X_m^0)$
return Y

We shall consider that the swap procedure is composed of 2 algebraic operations (copies of algebraic variables) although the above description makes appear 4 copies. Indeed, in practice one could avoid having the same variables in input and output of the function and simply implements it as $(Y^0, Y^1) \leftarrow (X^{0\oplus b}, X^{1\oplus b})$. The total number N_Π of operations in the protected algorithm is hence given by:

$$N_\Pi = N_{pre} + N_{post} + n \cdot (N_{seq} + 5m(d+1)) \ . \tag{8}$$

Practical security. Our algorithm defeats advanced (higher-order) side-channel attacks. In particular, it is secure against template attacks of order d which target (masked) scalar manipulations (and related addresses), as well as against collision-based horizontal attacks of order d. The former attacks consist in targeting the direct manipulation of the sensitive bits (or digits) k_i or any address (or index in our formalism) that depends on k_i. In our countermeasure, all these sensitive bits and associated addresses are masked at order d and one must at least target a $(d+1)$-tuple of such index variables to recover sensitive information (this is formally proved hereafter). The latter attacks consists in finding collision between processed algebraic variables in order to infer information of the sensitive indexes. By our re-randomization and swap strategy, it can be checked

that one must at least target $d+1$ algebraic variables to recover sensitive information. For instance, observing a collision between the variable X_ℓ^0 output of the randomization at step 7 and the variable X_ℓ^0 output from the swap at Step 9 would allow one to learn that $b_\ell^j = 0$, while a non-collision would mean $b_\ell^j = 1$. A collision-based horizontal attacker should observe $d+1$ such (non-)collision in order to learn the sensitive bit b_ℓ. Illustration of such *higher-order* collision attack on Alg. 4 is given in Fig. 1 where the attacker observes successive executions of step 7 to recover indexes b_1^j. Security against this kind of attacks is also captured by our formal proof.

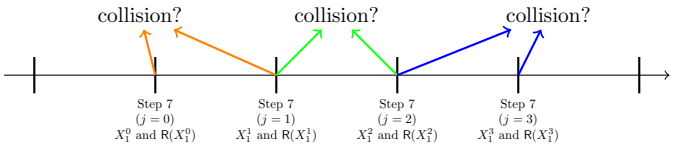

Fig. 1: Illustration of fourth-order collision attack on Alg. 4.

Practical efficiency. From a high level viewpoint, our countermeasure to Alg. 3 switches the complexity from $\mathcal{O}(nm)$ to $\mathcal{O}(nmd)$, which means a linear slowdown with respect to the masking order d. However, we stress the practical efficiency of our generic countermeasure can be boosted in the case *lighweight* re-randomization technique can be applied. Indeed, our countermeasure does not add any algebraic operation to the original program besides copies and re-randomizations. In practice, the performances of ECC and other exponentiation-based cryptosystems is mainly impacted by (field or ring) multiplications (or point doublings/additions at the group level), namely the operations that appear in the sequence procedure in Alg. 3. If copy and re-randomizations can be made negligible compared to costly multiplications then our countermeasure would be virtually free. Such a paradigm would be unrealistic for large values of the masking order d but the trade-off between costly algebraic operations and light copies and re-randomizations could lead to a reasonable impact on the performances.

We also note that our countermeasure is greedy in terms of randomness generation with the frequent use of re-randomization. But this is also true for other classical countermeasures achieving high level of provable security in the noisy leakage model [19,24,21]. Moreover, this might be mitigated in practice by reducing the number of random bits used in a re-randomization. One could for instance argue that for the randomization in Example 1 the fresh random value R could be sampled on a smaller space than $[0, h)$ (*i.e.* with less than $\log_2(h)$ bits), which would constitute a trade-off between the hideness property and the efficiency. Let us stress once again that to the best of our knowledge and besides [1] (which shows the difficulty of defeating elliptic-curve point randomization), no attack has been published that defeats these types of randomization besides looking at the collision behavior of randomized variables (which is fully captured by our hideness notion with $\varepsilon = 0$).

4.2 Security Proof

We state our formal security result in the following theorem.

Theorem 1. *The program Π defined in Alg. 4 is γ-leakage resilient against ε-hiding δ-noisy leakage functions with*

$$\gamma \leq n(5m\delta)^{d+1} + N_{\Pi} \cdot \varepsilon \ . \tag{9}$$

Theorem 1 shows that if ε is negligible and if $\delta < \frac{1}{5m}$, then our countermeasure achieves exponential security in the masking order d, at the cost of a linear slowdown for the protected algorithm.

Remark 8. In the above theorem, we implicitly assume that δ-noisy leakage functions are defined with respect to the relative error distance (see (3)), as suggested in [52]. We stress that a different choice would not affect our proof except for the tightness of the bound (9). For instance, taking the L_1-norm statistical distance considered in [19] would imply a factor $|\mathcal{X}|$ to δ in (9) (where $\mathcal{X}$ is the domain of the leakage function). But as discussed in [52], this factor seems to be a proof artifact which does not apply while considering practically relevant leakage models.

We demonstrate Theorem 1 through a sequence of three games. Each game is composed of two experiments: a *real* experiment that outputs the real leakage distribution following the considered family of δ-noisy leakage functions $\mathcal{F}$, and a *simulation* experiment that output the simulated leakage distribution. The γ-leakage resilience is achieved if and only if the output distributions of the two experiments are γ-close.

Game 0 (illustrated in Figure 4, Appendix B) exactly fits the definition of leakage resilience (Definition 3) where $\mathcal{F}$ is a family of ε-hiding δ-noisy leakage functions. We first show that this game reduces to Game 1 in which the $\mathsf{ComputTrace}$ algorithm is replaced by a different $\mathsf{ComputTrace}'$ algorithm. Then we show that Game 1 reduces to Game 2 in which the functions from $\mathcal{F}$ are replaced by corresponding δ-*random probing* functions. We finally exhibit a simulator which achieves γ_2-closeness of the experiment outputs in Game 2 for some γ_2. According to the two previous game transitions, we obtain γ_1-closeness and γ_0-closeness for the experiment outputs in Game 1 and Game 0 for some γ_1 and γ_0 that we shall exhibit throughout the proof.

Our first technical lemma (transition from Game 0 to Game 1) is given hereafter (the proof is in Appendix B):

Lemma 1. *Assume that there exists a simulator* Sim *such that* $\mathsf{ExpReal}_1(\mathcal{A}, \Pi, \mathcal{F})$ *and* $\mathsf{ExpSim}_1(\mathcal{A}, \Pi, \mathcal{F}, \mathsf{Sim})$ *outputs γ_1-close distributions. Then, for the same simulator,* $\mathsf{ExpReal}_0(\mathcal{A}, \Pi, \mathcal{F})$ *and* $\mathsf{ExpSim}_0(\mathcal{A}, \Pi, \mathcal{F}, \mathsf{Sim})$ *outputs γ_0-close distributions with*

$$\gamma_0 \leq \gamma_1 + N_{\Pi} \cdot \varepsilon \tag{10}$$

Before introducing our second technical lemma, we shall recall the definition of random probing function.

Definition 6 (Random probing function). *A δ-random probing function φ on a set $\mathcal{X}$ is a noisy function which satisfies*

$$\varphi(x) = \begin{cases} x & \text{with probability } \delta \\ \perp & \text{with probability } 1-\delta \end{cases}$$

for every $x \in \mathcal{X}$, where $\perp$ denotes a void output and $Im(\varphi) = \mathcal{X} \cup \{\perp\}$.

In the above definition, we skip the randomness argument from the definition of the function φ as discussed in Remark 2. In the context of a random probing function, this randomness determines whether the event $\varphi(\cdot) = \perp$ occurs (with probability δ) or not. We can now formally state our second technical lemma (transition from Game 1 to Game 2) whose proof in given in Appendix B:

Lemma 2. *For every family $\mathcal{F}$ of δ-noisy functions, there exists a family $\mathcal{F}'$ of δ-random probing functions, for which the following holds.*

- *if there exists a simulator Sim' such that the experiments $\mathsf{ExpReal}_2(\mathcal{A}, \Pi, \mathcal{F}')$ and $\mathsf{ExpSim}_2(\mathcal{A}, \Pi, \mathcal{F}', \mathsf{Sim}')$ outputs γ_2-close distributions,*
- *then there exists a simulator Sim such that the experiments $\mathsf{ExpReal}_1(\mathcal{A}, \Pi, \mathcal{F})$ and $\mathsf{ExpSim}_1(\mathcal{A}, \Pi, \mathcal{F}, \mathsf{Sim})$ outputs γ_1-close distributions with $\gamma_1 \leq \gamma_2$.*

We finally give our last technical lemma on the security of Game 2 (the proof is in Appendix B)):

Lemma 3. *Let Π be the randomized RAP described in Alg. 4 for masking order d and let $\mathcal{F}'$ be a family of δ-random probing functions. There exists a simulator Sim such that $\mathsf{ExpReal}_2(\mathcal{A}, P, \mathcal{F}')$ and $\mathsf{ExpSim}_2(\mathcal{A}, P, \mathcal{F}', \mathsf{Sim})$ output γ_2-close distributions, with*

$$\gamma_2 \leq n(5m\delta)^{d+1} \ . \tag{11}$$

5 Applications

We now provide concrete examples of regular algorithms – including examples from Section 2 – and show that they perfectly fit the generic shape of Algorithm 3. We first show how to apply our generic countermeasure to various binary scalar multiplication algorithms, and then to the classic left-to-right fixed-window algorithm (see Algorithm 2).

5.1 Application to Binary Ladders

Montgomery ladder. To rewrite the Montgomery ladder (see Alg. 1) in our framework, observe that a loop iteration can be rewritten as:

$$\begin{cases} b \leftarrow k_i \\ (R_1, R_0) \leftarrow (R_{1-b}, R_b) \\ R_1 \leftarrow R_1 + R_0 \\ R_0 \leftarrow 2 \cdot R_0 \\ (R_1, R_0) \leftarrow (R_{1-b}, R_b) \end{cases} \iff \begin{cases} b \leftarrow k_{i+1} \oplus k_i \\ (R_1, R_0) \leftarrow (R_{1-b}, R_b) \\ R_1 \leftarrow R_1 + R_0 \\ R_0 \leftarrow 2 \cdot R_0 \end{cases}$$

where the above equivalence holds by defining $k_n := 0$ and returning R_{k_0} instead of R_0 in the latter case. We thus obtain Alg. 5 which rewrites the Montgomery ladder in the framework of Alg. 3, where at the end of each iteration (X_1^0, X_2^0) matches (R_1, R_0), while (X_1^1, X_2^1) matches (R_0, R_1).

Algorithm 5 Montgomery ladder at the point level

Input: $\boldsymbol{P}$, $\boldsymbol{k} = (k_n, k_{n-1}, \ldots, k_1, k_0)$ with $k_n := 0$
Output: $[k]\boldsymbol{P}$

1. $(X_1^0, X_2^0) \leftarrow (\boldsymbol{\mathcal{O}}, \boldsymbol{P})$ // preprocess($\boldsymbol{P}$)
2. **for** $i = 0$ **to** $n - 1$ **do**
3. $\quad b \leftarrow k_{i+1} \oplus k_i$ // indexes(k_i)
4. $\quad (X_1^0, X_2^0) \leftarrow (X_1^b, X_2^b)$
5. $\quad (X_1^0, X_2^0) \leftarrow (X_1^0 + X_2^0, 2X_2^0)$ // sequence(X_1^0, X_2^0)
6. $\quad (X_1^1, X_2^1) \leftarrow (X_2^0, X_1^0)$
7. **end for**
8. $b \leftarrow k_0$ // indexes(k_i)
9. $(X_1^0, X_2^0) \leftarrow (X_1^b, X_2^b)$
10. **return** X_2^0

Montgomery ladder at coordinate level. Our approach can also be applied at the coordinate level *i.e.* when internal registers store field element rather than points. We illustrate this on the Montgomery ladder recently exhibited by Hamburg [27] and which is today the most efficient binary ladder for standard Weierstrass curves. Alg. 7 (Appendix C) rewrites this algorithm in our framework.

We note that the Montgomery ladder for Curve25519 and Curve448 described in RFC 7748 [42] at the coordinate level is already expressed in a similar paradigm as ours (in particular making use of conditional swap instructions). Our generic countermeasure therefore directly applies to this algorithm.

Joye ladder. Similarly, Alg. 8 (Appendix C) rewrites the double-and-add algorithm of Joye [37] so that it perfectly matches our generic regular program displayed in Alg. 3.

Signed binary ladder. Another example is the signed binary algorithm from [56]. It is particularly efficient when the points are represented using (X, Y)-only co-Z coordinates [26,56]. In this algorithm, the input scalar is assumed to satisfy $k_0 = k_{n-1} = 1$ and it is encoded under its (unique) full signed binary expansion in which each digit belongs to $\{-1, 1\}$. Applying the standard left-to-right algorithm to this signed expansion yields a regular signed binary ladder which can also be rewritten in our framework as shown in Alg. 9 (Appendix C).

From the rewrites of the binary ladders, we can directly apply our generic countermeasure and obtain a leakage resilient program. In a nutshell, for some masking order d, the steps in Alg. 5, 8, and 9 that correspond to the sequence procedure will remain unchanged, while registers (X_1^0, X_1^1) and (X_2^0, X_2^1) will be

refreshed $d+1$ times and pairwisely swapped in place the scalar-dependent selection step. The obtain leakage resilient program achieves the provable security result of Theorem 1 with $m = 2$.

5.2 Application to Fixed-Window Scalar Multiplication

The fixed-window scalar multiplication described in Alg. 2 can also be rewritten in the generic shape of Alg. 3, at the cost of a few differences (see Alg. 6). As explained in Section 2, this algorithm requires the pre-computation of a table T to store the multiples $d\boldsymbol{P}$ for digits d belonging to some basis $\mathbb{B} \subset \mathbb{Z}$. Different choices are possible for such a basis depending on the context. Our description of Alg. 6 implicitly assumes $|\mathbb{B}| = 2^w$. For any digit $d \in \mathbb{B}$, we denote $\tilde{d} \in [0, 2^w - 1]$ its index in the basis $\mathbb{B}$, and assume that each digit k_i of the scalar is represented by its w-bit index $\tilde{k}_i$ (the sharing being applied to those index values).

Contrary to binary algorithms, Alg. 6 loops on w-bit windows and it requires 2^w temporary registers $(X^i)_{0 \leq i \leq 2^w - 1}$ instead of the two registers per input introduced in Alg. 3. Finally, to ease the exposition we consider two $\mathsf{sequence}$ procedures, at the begining and at the end of the loop iterations. Note that the $\mathsf{sequence}$ procedure at the end of an iteration could be considered to be part of the next iteration to fit to the original model (and for the last iteration, it can be seen as a sub-step of the post processing). In both cases, this step only counts in the number of operations of each of these generic functions, but does not alter the security proof.

Algorithm 6 Fixed window

Input: $\boldsymbol{P}$, $\boldsymbol{k} = (k_{n-1}, \ldots, k_1, k_0)_{2^w}$
Output: $[k]\boldsymbol{P}$

1. **forall** $d \in \mathbb{B}$ **do** $T[\tilde{d}] = [d]\boldsymbol{P}$ // $\mathsf{preprocess}(\boldsymbol{P})$
2. **for** $i = 0$ **to** $n - 1$ **do**
3. $\quad (X^0, X^1, \ldots, X^{2^w-1}) \leftarrow T$ // $\mathsf{sequence}_1(X)$
4. $\quad c \leftarrow \tilde{k}_i$ // $\mathsf{indexes}(k_i)$
5. $\quad X^0 \leftarrow X^c$
6. $\quad X \leftarrow X^0 + [2^w]X$ // $\mathsf{sequence}_2(X)$
7. **end for**
8. $\boldsymbol{Y} \leftarrow X$ // $\mathsf{postprocess}(X^0)$
9. **return** $\boldsymbol{Y}$

The main difference while applying our countermeasure is in the type of the index c at Step 6 which is not a bit anymore but a w-bit coordinate of $\boldsymbol{k}$. The swapping procedure must hence be generalized to work with w-bit Boolean shares, which can be done by defining it as:

$$\mathsf{swap}(X^0, X^1, \ldots, X^{2^w-1}, c) = (X^{c \oplus 0}, X^{c \oplus 1}, \ldots, X^{c \oplus (2^w-1)}) \ . \tag{12}$$

Under such a definition, we still have the desired property

$$\mathsf{swap}(X^0, \ldots, X^{2^w-1}, c) = \mathsf{swap}(\ldots \mathsf{swap}(\mathsf{swap}(X^0, \ldots, X^{2^w-1}, c^0), c^1), \ldots, c^d)$$

which is satisfied for every Boolean sharing $c^0 \oplus c^1 \oplus \cdots \oplus c^d = c$. While applying our countermeasure, Step 6 can thus be replaced by:

for $j = 0$ **to** d **do**
  $c^j \leftarrow \tilde{k}_i^j$
  $(X^0, X^1, \ldots, X^{2^w-1}) \leftarrow (\mathsf{R}(X^0), \mathsf{R}(X^1), \ldots, \mathsf{R}(X^{2^w-1}))$
  $(X^0, X^1, \ldots, X^{2^w-1}) \leftarrow \mathsf{swap}(X^0, X^1, \ldots, X^{2^w-1}, c)$
end for

Our security proof works the same way on the obtained program which hence satisfy the leakage-resilience result of Theorem 1.

6 Performances

6.1 Performance Analysis

A high-level complexity analysis of our countermeasure was provided in Section 4. Namely, we reach a complexity in $\mathcal{O}(nmd)$ with n the scalar bit length, m the number of cells of the algebraic memory (*i.e.*, the number of algebraic variables), and d the masking order. Going a little deeper into the details, denoting $C_{\mathtt{op}}$ the complexity of operation `op`, the overall complexity of Alg. 4 is exactly equal to

$$C_{\mathtt{preprocess}} + C_{\mathtt{postprocess}} + n \cdot \big(C_{\mathtt{sequence}} + (d+1) \cdot (C_{\mathtt{indexes}} + m \cdot (2 \cdot C_{\mathsf{R}} + C_{\mathtt{swap}}))\big).$$

The complexity of the `indexes` operation $C_{\mathtt{indexes}}$ being reduced to m index operations and the complexity of the `swap` operation to two algebraic copies, the prominent operations remain the multiplications in `sequence` and the randomization of algebraic variables R. We estimate the randomization complexity C_{R} when implemented with one of the two examples displayed in Section 3, and also with both. We shall further denote $C_{\mathtt{mult}_\alpha}$ the complexity of a multiplication in $\mathbb{Z}_q$ between operands of bit sizes α and approximate its cost with $\mathtt{c_m} \cdot \alpha^2$ for some constant $\mathtt{c_m}$.

In Example 1, we consider the randomization operation which maps an element of $\mathbb{Z}_p$ to a randomized representation in $\mathbb{Z}_{hp}$, namely $\mathsf{R}(x) = x + R \cdot p$ with $R \leftarrow [0, h)$. It requires a multiplication in $\mathbb{Z}_{hp}$ of complexity

$$C_{\mathsf{R}_1} = \mathtt{c_m} \cdot |p| \cdot |h|$$

and an addition in $\mathbb{Z}_{hp}$ that we shall consider to be negligible compared to the multiplication. Using this randomization in Alg. 4 would require the `sequence` operation and all the subsequent randomizations to be performed in $\mathbb{Z}_{hp}$. Denoting by γ the number of multiplications in `sequence`, the overall complexity would remain equivalent to

$$n \cdot \big(\gamma \cdot C_{\mathtt{mult}_{|p|+|h|}} + 2 \cdot (d+1) \cdot m \cdot C_{\mathsf{R}_1}\big)$$

In Example 2, the randomization operation maps a point defined by its Jacobian coordinates on an elliptic curve of base field $\mathbb{F}_p$ to a randomized representation, namely $\mathsf{R}(\boldsymbol{P}) = \mathsf{R}((X : Y : Z)) = (X\lambda^2 : Y\lambda^3 : Z\lambda)$ with $\lambda \leftarrow \mathbb{F}_p^*$.

This randomization mainly requires 5 multiplications between elements in $\mathbb{F}_p$ or $\mathbb{F}_p^*$ (assuming that the squaring operation λ^2 is performed by a multiplication). We thus express its complexity as follows:

$$C_{\mathtt{R}_2} = 5 \cdot C_{\mathtt{mult}_{|p|}} = 5 \cdot \mathtt{c_m} \cdot |p|^2.$$

This complexity could be reduced in our countermeasure when considering the manipulation of two points in co-Z coordinates in the main loop. The double randomization of complexity $2 \cdot C_{\mathtt{R}_2}$ could then be performed with only 6 multiplications. This would result in an overall complexity equivalent to

$$n \cdot \left(\gamma \cdot C_{\mathtt{mult}_{|p|}} + 6 \cdot (d+1) \cdot m \cdot C_{\mathtt{mult}_{|p|}}\right)$$

One step further, we could gather both randomization techniques to improve the security level (*i.e.*, lower parameter ε) of our countermeasure. We would randomize the Jacobian coordinates, previously augmented with multiples of the modulus p. With co-Z coordinates, the overall complexity would thus be equivalent to

$$n \cdot \left(\gamma \cdot C_{\mathtt{mult}_{|p|+|h|}} + 10 \cdot (d+1) \cdot m \cdot C_{\mathtt{mult}_{|p|+|h|}}\right).$$

Table 2 in Appendix D illustrates the overhead (the multiplicative factor) between protected implementations and the regular program with different levels of countermeasures.

6.2 Implementation Benchmarks

We provide performance benchmarks for a concrete protected scalar multiplication implementation on a 0.13um 32-bit Contact Smartcard IC. The IC was EAL4+ certified in Asia and features an ARM SecureCore SC100 processor with 18KB of RAM, 8KB of ROM, and 548KB of FLASH[5].

Our implementation is based on the signed binary ladder using (X, Y)-only co-Z coordinates [56, Algorithm 8]. In the basic setting (without applying our countermeasure), the scalar is randomized by the addition of a 288-bit multiple of the curve order and the Jacobian coordinates of the points are randomized before starting the main loop. Benchmarks for this implementation with the NIST elliptic curve P-256 [47] are provided in the column *Regular program* of Figure 7 (in Appendix D, averaged over 100 executions).

Then, we modified our implementation to follow the description of Alg. 9 and applied the generic countermeasure introduced in Section 4 for a customizable masking order. The randomization step follows the described randomization of Jacobian coordinates simultaneously applied on two points in (X, Y)-only coordinates as well as on additional inverted Y-coordinate. This makes a total of 2 multiplications to derive λ^2 and λ^3 from λ plus 5 multiplications with the point coordinates. The additional cost of this step is mainly due to these seven extra multiplications. We provide the corresponding benchmarks at orders 1, 2, 4, and 8. From these results, we can interpolate a running time of $400 + 266(d + 1)$

[5] We refer the interested reader to [11] for further details on this device.

ms for a masking order d, as displayed in Table 1 and in Figure 7. The performances of the first row are computed when the randomness is generated on the smart card. Using a constant instead (to simulate the cost of the randomness generation), we obtain the results displayed on the second row. The randomness cost in the regular program is very light (*i.e.*, the difference is low between the complexity when using card and free randomness) since, without our countermeasure, only a little randomness is actually used. Note that the gap between the regular program and the first-order masked implementation comes from the randomization which is applied directly on both shares in the latter case. As for the memory complexity, our regular program requires 1220 bytes of RAM while the countermeasure add 33 bytes per masking order.

Table 1: Benchmarks of Alg. 9 and its countermeasure at several orders

	Regular program	Our countermeasure			
		order 1	order 2	order 4	order 8
card randomness	416,5 ms	933,1 ms	1200,8 ms	1734,0 ms	2804,9 ms
free randomness	415,2 ms	612,4 ms	718,5 ms	930,4 ms	1355,9 ms
RAM complexity	1,220 kB	1,253 kB	1,286 kB	1,352 kB	1,484 kB

References

1. M. Azouaoui, F. Durvaux, R. Poussier, F.-X. Standaert, K. Papagiannopoulos, and V. Verneuil. On the worst-case side-channel security of ECC point randomization in embedded devices. In K. Bhargavan, E. Oswald, and M. Prabhakaran, editors, *INDOCRYPT 2020*, volume 12578 of *LNCS*, 2020. Springer, Heidelberg.
2. L. Batina, L. Chmielewski, L. Papachristodoulou, P. Schwabe, and M. Tunstall. Online template attacks. *Journal of Cryptographic Engineering*, 2019.
3. L. Batina, B. Gierlichs, E. Prouff, M. Rivain, F.-X. Standaert, and N. Veyrat-Charvillon. Mutual information analysis: a comprehensive study. *Journal of Cryptology*, 2011.
4. A. Bauer, É. Jaulmes, E. Prouff, and J. Wild. Horizontal collision correlation attack on elliptic curves. In T. Lange, K. Lauter, and P. Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, 2014. Springer, Heidelberg.
5. N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom. "ooh aah... just a little bit": A small amount of side channel can go a long way. In L. Batina and M. Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, 2014. Springer, Heidelberg.
6. D. J. Bernstein and T. Lange. Faster addition and doubling on elliptic curves. In K. Kurosawa, editor, *ASIACRYPT 2007*, volume 4833 of *LNCS*, 2007. Springer, Heidelberg.
7. D. J. Bernstein and T. Lange. Inverted edwards coordinates. In S. Boztas and H. Lu, editors, *AAECC-17*, volume 4851 of *LNCS*, 2007. Springer.
8. E. Brier, C. Clavier, and F. Olivier. Correlation power analysis with a leakage model. In M. Joye and J.-J. Quisquater, editors, *CHES 2004*, volume 3156 of *LNCS*, 2004. Springer, Heidelberg.
9. E. Brier and M. Joye. Weierstraß elliptic curves and side-channel attacks. In D. Naccache and P. Paillier, editors, *PKC 2002*, volume 2274 of *LNCS*, 2002. Springer, Heidelberg.

10. E. Cagli, C. Dumas, and E. Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures - profiling attacks without preprocessing. In W. Fischer and N. Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, 2017. Springer, Heidelberg.
11. M. Carbone, V. Conin, M.-A. Cornélie, F. Dassance, G. Dufresne, C. Dumas, E. Prouff, and A. Venelli. Deep learning to evaluate secure RSA implementations. *IACR TCHES*, 2019. `https://tches.iacr.org/index.php/TCHES/article/view/7388`.
12. S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. In B. S. Kaliski Jr., Çetin Kaya. Koç, and C. Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, 2003. Springer, Heidelberg.
13. B. Chevallier-Mames, M. Ciet, and M. Joye. Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity. *IEEE Trans. Computers*, 2004.
14. C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. Horizontal correlation analysis on exponentiation. In M. Soriano, S. Qing, and J. López, editors, *ICICS 10*, volume 6476 of *LNCS*, 2010. Springer, Heidelberg.
15. C. Clavier and M. Joye. Universal exponentiation algorithm. In Çetin Kaya. Koç, D. Naccache, and C. Paar, editors, *CHES 2001*, volume 2162 of *LNCS*, 2001. Springer, Heidelberg.
16. J.-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In Çetin Kaya. Koç and C. Paar, editors, *CHES'99*, volume 1717 of *LNCS*, 1999. Springer, Heidelberg.
17. J.-S. Coron, E. Prouff, M. Rivain, and T. Roche. Higher-order side channel security and mask refreshing. In S. Moriai, editor, *FSE 2013*, volume 8424 of *LNCS*, 2014. Springer, Heidelberg.
18. T. M. Cover and J. A. Thomas. *Elements of information theory, Second Edition.* 2006.
19. A. Duc, S. Dziembowski, and S. Faust. Unifying leakage models: From probing attacks to noisy leakage. In P. Q. Nguyen and E. Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, 2014. Springer, Heidelberg.
20. M. Dugardin, L. Papachristodoulou, Z. Najm, L. Batina, J.-L. Danger, and S. Guilley. Dismantling real-world ECC with horizontal and vertical template attacks. In F.-X. Standaert and E. Oswald, editors, *COSADE 2016*, volume 9689 of *LNCS*, 2016. Springer, Heidelberg.
21. S. Dziembowski, S. Faust, and K. Zebrowski. Simple refreshing in the noisy leakage model. In S. D. Galbraith and S. Moriai, editors, *ASIACRYPT 2019, Part III*, volume 11923 of *LNCS*, 2019. Springer, Heidelberg.
22. A. Faz-Hernández, P. Longa, and A. H. Sanchez. Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves. In J. Benaloh, editor, *CT-RSA 2014*, volume 8366 of *LNCS*, 2014. Springer, Heidelberg.
23. P.-A. Fouque and F. Valette. The doubling attack - why upwards is better than downwards. In C. D. Walter, Çetin Kaya. Koç, and C. Paar, editors, *CHES 2003*, volume 2779 of *LNCS*, 2003. Springer, Heidelberg.
24. D. Goudarzi, A. Joux, and M. Rivain. How to securely compute with noisy leakage in quasilinear complexity. In T. Peyrin and S. Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, 2018. Springer, Heidelberg.
25. D. Goudarzi, M. Rivain, and D. Vergnaud. Lattice attacks against elliptic-curve signatures with blinded scalar multiplication. In R. Avanzi and H. M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, 2016. Springer, Heidelberg.

26. R. R. Goundar, M. Joye, A. Miyaji, M. Rivain, and A. Venelli. Scalar multiplication on Weierstraß elliptic curves from co-Z arithmetic. *Journal of Cryptographic Engineering*, 2011.
27. M. Hamburg. Faster montgomery and double-add ladders for short Weierstrass curves. *IACR TCHES*, 2020. `https://tches.iacr.org/index.php/TCHES/article/view/8681`.
28. D. Hankerson, A. Menezes, and S. Vanstone. 2004.
29. J. Heyszl, A. Ibing, S. Mangard, F. D. Santis, and G. Sigl. Clustering algorithms for non-profiled single-execution attacks on exponentiations. In A. Francillon and P. Rohatgi, editors, *CARDIS 2013*, volume 8419 of *LNCS*, 2013. Springer.
30. H. Hisil, K. K.-H. Wong, G. Carter, and E. Dawson. Twisted Edwards curves revisited. In J. Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, 2008. Springer, Heidelberg.
31. N. Homma, A. Miyamoto, T. Aoki, A. Satoh, and A. Shamir. Collision-based power analysis of modular exponentiation using chosen-message pairs. In E. Oswald and P. Rohatgi, editors, *CHES 2008*, volume 5154 of *LNCS*, 2008. Springer, Heidelberg.
32. Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In D. Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, 2003. Springer, Heidelberg.
33. K. Itoh, T. Izu, and M. Takenaka. Address-bit differential power analysis of cryptographic schemes OK-ECDH and OK-ECDSA. In B. S. Kaliski Jr., Çetin Kaya. Koç, and C. Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, 2003. Springer, Heidelberg.
34. K. Itoh, T. Izu, and M. Takenaka. A practical countermeasure against address-bit differential power analysis. In C. D. Walter, Çetin Kaya. Koç, and C. Paar, editors, *CHES 2003*, volume 2779 of *LNCS*, 2003. Springer, Heidelberg.
35. K. Itoh, J. Yajima, M. Takenaka, and N. Torii. DPA countermeasures by improving the window method. In B. S. Kaliski Jr., Çetin Kaya. Koç, and C. Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, 2003. Springer, Heidelberg.
36. M. Izumi, J. Ikegami, K. Sakiyama, and K. Ohta. Improved countermeasure against address-bit DPA for ECC scalar multiplication. In G. D. Micheli, B. M. Al-Hashimi, W. Müller, and E. Macii, editors, *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010*, 2010. IEEE Computer Society.
37. M. Joye. Highly regular right-to-left algorithms for scalar multiplication. In P. Paillier and I. Verbauwhede, editors, *CHES 2007*, volume 4727 of *LNCS*, 2007. Springer, Heidelberg.
38. M. Joye. Highly regular m-ary powering ladders. In M. J. Jacobson Jr., V. Rijmen, and R. Safavi-Naini, editors, *SAC 2009*, volume 5867 of *LNCS*, 2009. Springer, Heidelberg.
39. M. Joye and M. Tunstall. Exponent recoding and regular exponentiation algorithms. In B. Preneel, editor, *AFRICACRYPT 09*, volume 5580 of *LNCS*, 2009. Springer, Heidelberg.
40. J. Kim, S. Picek, A. Heuser, S. Bhasin, and A. Hanjalic. Make some noise: Unleashing the power of convolutional neural networks for profiled side-channel analysis. *IACR TCHES*, 2019. `https://tches.iacr.org/index.php/TCHES/article/view/8292`.
41. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, 1999. Springer, Heidelberg.
42. A. Langley, M. Hamburg, and S. Turner. Elliptic Curves for Security. 2016.
43. S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks - revealing the secrets of smart cards*. 2007.
44. P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 1987.

45. N. Mukhtar, A. P. Fournaris, T. M. Khan, C. Dimopoulos, and Y. Kong. Improved hybrid approach for side-channel analysis using efficient convolutional neural network and dimensionality reduction. *IEEE Access*, 2020.
46. E. Nascimento, L. Chmielewski, D. Oswald, and P. Schwabe. Attacking embedded ECC implementations through cmov side channels. In R. Avanzi and H. M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, 2016. Springer, Heidelberg.
47. NIST. 2013.
48. E. Oswald, S. Mangard, C. Herbst, and S. Tillich. Practical second-order DPA attacks for masked smart card implementations of block ciphers. In D. Pointcheval, editor, *CT-RSA 2006*, volume 3860 of *LNCS*, 2006. Springer, Heidelberg.
49. D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. 2002. `https://eprint.iacr.org/2002/169`.
50. G. Perin and L. Chmielewski. A semi-parametric approach for side-channel attacks on protected RSA implementations. In N. Homma and M. Medwed, editors, *CARDIS 2015*, volume 9514 of *LNCS*, 2015. Springer.
51. G. Perin, L. Imbert, L. Torres, and P. Maurine. Attacking randomized exponentiations using unsupervised learning. In E. Prouff, editor, *COSADE 2014*, volume 8622 of *LNCS*, 2014. Springer, Heidelberg.
52. T. Prest, D. Goudarzi, A. Martinelli, and A. Passelègue. Unifying leakage models on a Rényi day. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, 2019. Springer, Heidelberg.
53. E. Prouff and M. Rivain. Masking against side-channel attacks: A formal security proof. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, 2013. Springer, Heidelberg.
54. E. Prouff, R. Strullu, R. Benadjila, E. Cagli, and C. Dumas. Study of deep learning techniques for side-channel analysis and introduction to ASCAD database. 2018. `https://eprint.iacr.org/2018/053`.
55. J. Renes, C. Costello, and L. Batina. Complete addition formulas for prime order elliptic curves. In M. Fischlin and J.-S. Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, 2016. Springer, Heidelberg.
56. M. Rivain. Fast and regular algorithms for scalar multiplication over elliptic curves. 2011. `https://eprint.iacr.org/2011/338`.
57. T. Roche, L. Imbert, and V. Lomné. Side-channel attacks on blinded scalar multiplications revisited. In S. Belaïd and T. Güneysu, editors, *CARDIS 2019*, volume 11833 of *LNCS*, 2019. Springer.
58. W. Schindler and K. Itoh. Exponent blinding does not always lift (partial) spa resistance to higher-level security. In J. Lopez and G. Tsudik, editors, *ACNS 11*, volume 6715 of *LNCS*, 2011. Springer, Heidelberg.
59. K. Schramm, T. J. Wollinger, and C. Paar. A new class of collision attacks and its application to DES. In T. Johansson, editor, *FSE 2003*, volume 2887 of *LNCS*, 2003. Springer, Heidelberg.
60. M. Tunstall, L. Papachristodoulou, and K. Papagiannopoulos. Boolean exponent splitting. 2018. `https://eprint.iacr.org/2018/1226`.
61. L. Weissbart, L. Chmielewski, S. Picek, and L. Batina. Systematic side-channel analysis of curve25519 with machine learning. *J. Hardw. Syst. Secur.*, 2020.
62. L. Weissbart, S. Picek, and L. Batina. One trace is all it takes: Machine learning-based side-channel attack on eddsa. In S. Bhasin, A. Mendelson, and M. Nandi, editors, *Security, Privacy, and Applied Cryptography Engineering - 9th International Conference, SPACE 2019, Gandhinagar, India, December 3-7, 2019, Proceedings*, volume 11947 of *Lecture Notes in Computer Science*, 2019. Springer.

Supplementary Material

A Experimental Results for the Hideness Assumption

We present hereafter some experimental results to motivate the hideness assumption for classic randomization techniques and leakage models. Specifically, we consider the naive, yet practically relevant, leakage model of Hamming weight with additive Gaussian noise:

$$f(x) = \mathrm{Hw}(x) + N \quad \text{with} \quad N \leftarrow \mathcal{N}(0, \sigma_N) \ , \tag{13}$$

for some standard deviation parameter σ_N.

On the other hand, we consider two classic randomization operations:

- *Modulus randomization*: The randomization operation which maps an element of $\mathbb{Z}_p$ to a randomized representation in $\mathbb{Z}_{hp}$, namely

$$\mathsf{R}(x) = x + R \cdot p \quad \text{with} \quad R \leftarrow [0, h) \ .$$

- *Point randomization*: The randomization operation which maps a point defined by its Jacobian coordinates on an elliptic curve of base field $\mathbb{F}_p$ to a randomized representation, namely

$$\mathsf{R}(\boldsymbol{P}) = \mathsf{R}((X : Y : Z)) = (X\lambda^2 : Y\lambda^3 : Z\lambda) \quad \text{with} \quad \lambda \leftarrow \mathbb{F}_p^* \ .$$

Hideness for modulus randomization . For the modulus randomization, we can say that the pair (f, R) is ε-hiding if the statistical distance between the two distributions

$$\mathcal{D}_1 := f(\mathsf{R}(x)) = f(x + R_1 \cdot p) \quad \text{and} \quad \mathcal{D}_2 := f(R_2) \quad \text{with} \quad \begin{cases} R_1 \leftarrow [0, h) \\ R_2 \leftarrow [0, hp) \end{cases}$$

is upper bounded by ε.

Hideness for point randomization . For the point randomization, we consider that the leakage function f is applied to each coordinate of the point. Similarly, we can say that the pair (f, R) is ε-hiding if the statistical distance between the two distributions

$$\mathcal{D}_1 := f(\mathsf{R}(\boldsymbol{P})) = f((X\lambda^2 : Y\lambda^3 : Z\lambda)) \quad \text{and} \quad \mathcal{D}_2 := f(\boldsymbol{U})$$

with $\lambda \leftarrow \mathbb{F}_p^*$ and $\boldsymbol{U} \leftarrow (\mathbb{F}_p)^3$, is upper bounded by ε.

Gaussian approximation and Kullback-Leibler divergence. In order to illustrate the hideness assumption on those two specific cases, we estimate the distance between the distributions $\mathcal{D}_1$ and $\mathcal{D}_2$. Since the Hamming weight of a random variable can be fairly approximated by a Gaussian distribution (in particular in the presence of an additive Gaussian noise), we estimate the means μ_1, μ_2 and standard deviations σ_1, σ_2 of the two distributions and approximate

$\mathcal{D}_i \sim \mathcal{N}(\mu_i, \sigma_i)$ for $i \in \{1, 2\}$. For the point randomization, the distributions are three-dimensional; we hence get multivariate Gaussian distributions $\mathcal{D}_i \sim \mathcal{N}(\mu_i, \Sigma_i)$ with mean vectors $\mu_i \in \mathbb{R}^3$ and covariance matrices $\Sigma_i \in \mathbb{R}^{3\times 3}$.

For each estimated set of parameters for $\mathcal{D}_1$ and $\mathcal{D}_2$, we compute the Kullback-Leibler divergence $D_{\mathrm{KL}}(\mathcal{D}_1 \parallel \mathcal{D}_2)$ between $\mathcal{D}_1$ and $\mathcal{D}_2$. The KL divergence (a.k.a. relative entropy) is a classic measure of distance between two distributions. For continuous distributions, it is defined as

$$D_{\mathrm{KL}}(\mathcal{D}_1 \parallel \mathcal{D}_2) := \int_{-\infty}^{\infty} p_1(x) \log\left(\frac{p_1(x)}{q_2(x)}\right) dx$$

where p_1 and p_2 respectively denote the pdfs of $\mathcal{D}_1$ and $\mathcal{D}_2$. We made the choice of the KL divergence for convenience. Indeed, the KL divergence between two Gaussian distributions $\mathcal{D}_i \sim \mathcal{N}(\mu_i, \Sigma_i)$ can be simply evaluated as:

$$D_{\mathrm{KL}}(\mathcal{D}_1 \parallel \mathcal{D}_2) = \tfrac{1}{2}\left(\log \tfrac{|\Sigma_2|}{|\Sigma_1|} - d + \mathrm{tr}(\Sigma_2^{-1}\Sigma_1) + (\mu_2 - \mu_1)^{\mathsf{T}}\Sigma_2^{-1}(\mu_2 - \mu_1)\right) \quad (14)$$

where d denotes the dimension ($d \in \{1, 3\}$ in our context) and $\mathrm{tr}(\cdot)$ the trace function. Moreover, $D_{\mathrm{KL}}(\mathcal{D}_1 \parallel \mathcal{D}_2)$ gives an upper bound of the statistical distance between $\mathcal{D}_1$ and $\mathcal{D}_2$ [18]:

$$\Delta_1(\mathcal{D}_1; \mathcal{D}_2) \leq \left((\ln 2/2) \cdot D_{\mathrm{KL}}(\mathcal{D}_1 \parallel \mathcal{D}_2)\right)^{\frac{1}{2}} . \quad (15)$$

Experimental setup. For both studied randomization techniques, we have estimated the distribution parameters to derive an approximation of the KL divergence with respect to

- an increasing randomness bit-length, $|h| \in [4, 64)$ for modulus randomization, $|\lambda| \in [4, 32)$ for point randomization, without noise (*i.e.* $\sigma_N = 0$),
- an increasing noise $\sigma_N \in [0, 100]$ for a fixed (small) randomness bit-length $|h| = |\lambda| = 8$.

Our results are depicted in Figure 2 for the modulus randomization and in Figure 3 for the point randomization. The Gaussian means and standard deviations (resp. covariance matrices) and corresponding KL divergences were estimated based on 10^7 random samples for each tested set of parameters.

Observations. From the left figures, we observe an exponential decrease of the KL divergence –and hence of the statistical distance– with respect to the randomness bit-length. In both cases, the decrease stops beyond the threshold $D_{\mathrm{KL}}(\mathcal{D}_1 \parallel \mathcal{D}_2) \leq 10^{-6}$, which is due to the estimation error. We validated this intuition by observing higher thresholds while experiencing with less samples for our estimations. We stress that our experiments using 10^7 samples in the different configurations took several days using SageMath on a 3,7 GHz Intel i5 CPU. This exponential decrease with respect to the randomness length suggests that for sufficiently large randomness, and for the considered leakage function

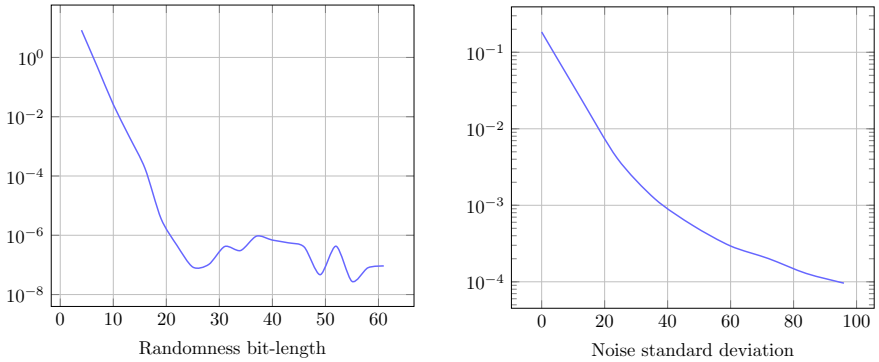

Fig. 2: Modulus randomization. Left: KL divergence wrt randomness bit-length (no noise). Right: KL divergence wrt noise standard deviation (8-bit randomness).

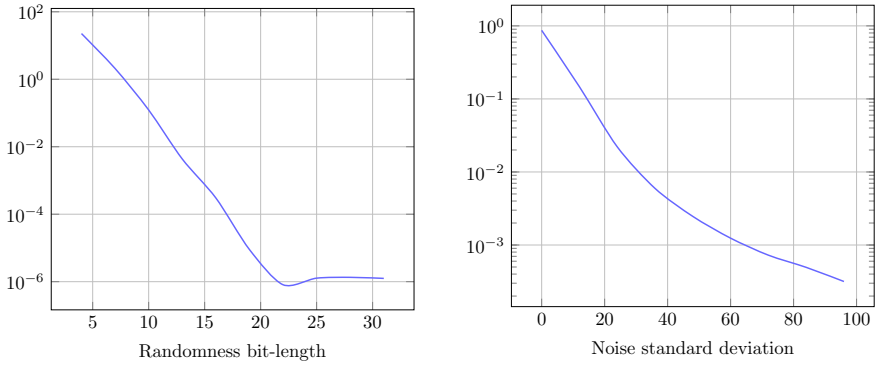

Fig. 3: Point randomization. Left: KL divergence wrt randomness bit-length (no noise). Right: KL divergence wrt noise standard deviation (8-bit randomness).

f (Hamming weight model), the two distributions are indistinguishable, *i.e.* the hideness assumption holds for a negligible ε.

From the right figures, we further observe that the noise standard deviation seems to have a sub-exponential impact on the KL divergence. This is the expected behavior since it can be checked that the contribution of the additive Gaussian noise N has a $\Theta\big(1/\sigma_N^2\big)$ impact to the formula (14) –and hence a $\Theta(1/\sigma_N)$ impact to the statistical distance.

Potential bias *vs.* modulus and point coordinates. For the modulus randomization, we used the ANSSI elliptic curve prime modulus

$p_{\text{ANSSI}} =$ `0xF1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C03`

We deliberately avoided to use a sparse prime such as the NIST P-256 prime

$$p_{\text{NIST}} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \ .$$

Indeed, for such a prime, the effect of the randomness does not propagate to all the bits whenever $|h| < 96$ which implies a direct bias in the distribution $\mathcal{D}_1 = f(x + R_1 \cdot p)$ compared to $\mathcal{D}_2 = f(R_2)$. To avoid this bias, we have to use a non-sparse prime p or a randomness bit-length which is greater than the longest chain of 0's or 1's in the prime binary expansion (*i.e.* greater than 96 for p_{NIST}). For the distribution $\mathcal{D}_1 = f(x + R_1 \cdot p)$ we arbitrarily chose $x = 0$, but we observed similar results for other values of x.

For the point randomization, we used the NIST elliptic curve base point, defined by the coordinates:

$$x_{\boldsymbol{P}} = \texttt{0x6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296}$$
$$y_{\boldsymbol{P}} = \texttt{0x4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5}$$

Since the randomization works on Jacobian coordinates $(X : Y : Z)$ we further had to define a Z-coordinate. To avoid a direct bias in the distribution $\mathcal{D}_1 = f((X\lambda^2 : Y\lambda^3 : Z\lambda))$ compared to $\mathcal{D}_2 = f(\boldsymbol{U})$, the Jacobian coordinates should not be sparse or particularly small compared to p. The obvious choice of $(X, Y, Z) = (x_{\boldsymbol{P}}, y_{\boldsymbol{P}}, 1)$ implies such a direct (and strong) bias. For this reason, we picked a random Z in our experiments. Specifically Z is randomly drawn from $\mathbb{F}_p$ at the beginning of the experiment and we then fix the coordinates $(X, Y, Z) = (x_{\boldsymbol{P}} \cdot Z^2, y_{\boldsymbol{P}} \cdot Z^3, Z)$ for all the generated samples.

We stress that for each randomization technique, the direct bias exhibited above can be avoided by taking a large enough randomness –typically $|h| = |p|$ for the modulus randomization and $|\lambda| = |p|$ for point randomization– but as illustrated in our experiments, shorter randomness seems to be sufficient in the absence of such pathological cases.

B Games and Lemmas' Proofs

B.1 Games

Our three games are presented in Figures 4, 5, and 6.

$\underline{\mathsf{ExpReal}_0(\mathcal{A}, \Pi, \mathcal{F})}$:

1. $(\boldsymbol{X}, \boldsymbol{k}) \leftarrow \mathcal{A}()$
2. $\widehat{\boldsymbol{k}} \leftarrow \mathsf{Enc}(\boldsymbol{k})$
3. $\mathcal{T} \leftarrow \mathsf{ComputTrace}(\Pi, \boldsymbol{X}, \widehat{\boldsymbol{k}})$
4. $(\ell_1, \ldots, \ell_z) \leftarrow \mathsf{LeakageSampler}(\mathcal{T}, \mathcal{F})$
5. Return $(\ell_1, \ldots, \ell_z)$

$\underline{\mathsf{ExpSim}_0(\mathcal{A}, \Pi, \mathcal{F}, \mathsf{Sim})}$:

1. $(\boldsymbol{X}, \boldsymbol{k}) \leftarrow \mathcal{A}()$
2. $(\ell_1, \ldots, \ell_z) \leftarrow \mathsf{Sim}(\Pi, \boldsymbol{X}, \mathcal{F})$
3. Return $(\ell_1, \ldots, \ell_z)$

Fig. 4: Game 0 (initial security game). The choice of the input $(\boldsymbol{X}, \boldsymbol{k})$ is modeled by a probabilistic algorithm $\mathcal{A}()$ (the adversary). The secret index variable $\boldsymbol{k}$ is then encoded into $\widehat{\boldsymbol{k}}$ using the Boolean sharing described in (5). Then the real experiment generates the computation trace $\mathcal{T}$ for the program Π on input $(\boldsymbol{X}, \widehat{\boldsymbol{k}})$ from which the leakage sampler produces the output $(\ell_1, \ldots, \ell_z)$ by applying the leakage functions from $\mathcal{F}$. On the other hand, the simulation experiment calls the simulator Sim on input Π, $\boldsymbol{X}$ and $\mathcal{F}$ which produces a simulated leakage output $(\ell_1, \ldots, \ell_z)$.

$\underline{\mathsf{ExpReal}_1(\mathcal{A}, \Pi, \mathcal{F})}$:

1. $(\boldsymbol{X}, \boldsymbol{k}) \leftarrow \mathcal{A}()$
2. $\widehat{\boldsymbol{k}} \leftarrow \mathsf{Enc}(\boldsymbol{k})$
3. $\mathcal{T} \leftarrow \mathsf{ComputTrace}'(\Pi, \boldsymbol{X}, \widehat{\boldsymbol{k}})$
4. $(\ell_1, \ldots, \ell_z) \leftarrow \mathsf{LeakageSampler}(\mathcal{T}, \mathcal{F})$
5. Return $(\ell_1, \ldots, \ell_z)$

$\underline{\mathsf{ExpSim}_1(\mathcal{A}, \Pi, \mathcal{F}, \mathsf{Sim})}$:

1. $(\boldsymbol{X}, \boldsymbol{k}) \leftarrow \mathcal{A}()$
2. $(\ell_1, \ldots, \ell_z) \leftarrow \mathsf{Sim}(\Pi, \boldsymbol{X}, \mathcal{F})$
3. Return $(\ell_1, \ldots, \ell_z)$

Fig. 5: Game 1. This game is similar to Game 0 but the (deterministic) $\mathsf{ComputTrace}$ algorithm is raplaced by a (probabilistic) $\mathsf{ComputTrace}'$ algorithm. In the latter, the computation trace is produced $\mathcal{T}$ is produced in a similar way as with the former algorithm, that is by evaluating the program on the inputs $\boldsymbol{X}$ and $\widehat{\boldsymbol{k}}$. But this time, the output of each randomization operation R is randomly sampled from $\mathcal{A}$ (according to the hideness property of the leakage).

$\underline{\mathsf{ExpReal}_2(\mathcal{A}, \Pi, \mathcal{F}')}$:

1. $(\boldsymbol{X}, \boldsymbol{k}) \leftarrow \mathcal{A}()$
2. $\widehat{\boldsymbol{k}} \leftarrow \mathsf{Enc}(\boldsymbol{k})$
3. $\mathcal{T} \leftarrow \mathsf{ComputTrace}'(\Pi, \boldsymbol{X}, \widehat{\boldsymbol{k}})$
4. $(\ell_1, \ldots, \ell_z) \leftarrow \mathsf{LeakageSampler}(\mathcal{T}, \mathcal{F}')$
5. Return $(\ell_1, \ldots, \ell_z)$

$\underline{\mathsf{ExpSim}_2(\mathcal{A}, \Pi, \mathcal{F}', \mathsf{Sim})}$:

1. $(\boldsymbol{X}, \boldsymbol{k}) \leftarrow \mathcal{A}()$
2. $(\ell_1, \ldots, \ell_{z'}) \leftarrow \mathsf{Sim}(\Pi, \boldsymbol{X}, \mathcal{F}')$
3. Return $(\ell_1, \ldots, \ell_z)$

Fig. 6: Game 2. This game is similar to Game 1 but the leakage function family $\mathcal{F}$ is replaced by a family $\mathcal{F}'$ of δ-random probing functions (formally defined hereafter in Definition 6). Specifically, for each δ-noisy function $f_i \in \mathcal{F}$, there exists a δ-random probing function f'_i such that $f_i(\cdot) = g_i \circ f'_i(\cdot)$ for some noisy function g_i (see details in the proof of Lemma 2). The functions f_i in $\mathcal{F}$ are replaced by those f'_i functions to build $\mathcal{F}'$.

B.2 Proof of Lemma 1

Proof. By definition of the ε-hideness property and since Π has a total number of N_Π operations, the leakage distribution output by $\mathsf{ExpReal}_1$ experiment is $(N_\Pi \cdot \varepsilon)$-close to the distribution output by the $\mathsf{ExpReal}_0$ experiment. The distribution output by the ExpSim_0 is identical as the distribution output by the ExpSim_1 experiment, which is γ_1-close to that of $\mathsf{ExpReal}_1$ and $(\gamma_1 + N_\Pi \cdot \varepsilon)$-close to that of $\mathsf{ExpReal}_0$. We hence get (10).

B.3 Proof of Lemma 2

Proof. Lemma 2 holds from the standard reduction from noisy-leakage security to random-probing security put forward in [19]. We specifically use the tighter variant of this result based on the relative error distance in the definition of noisy functions [52, Lemma 3]. According to this lemma, for any δ-noisy function f, there exists a δ-random probing function f' such that $f(x) = g \circ f'(x)$ for every x and for some noisy function g. The noisy leakage functions $f_i \in \mathcal{F}$ applied in $\mathsf{ExpReal}_1(\mathcal{A}, \Pi, \mathcal{F})$ are replaced by the corresponding random probing functions $f'_i \in \mathcal{F}'$ in $\mathsf{ExpReal}_2(\mathcal{A}, \Pi, \mathcal{F}')$. In order to construct the simulator Sim, we simply run $\mathsf{Sim}'(\Pi, \boldsymbol{X}, \mathcal{F}')$ which outputs a distribution which is γ_2-close to the random probing leakage output of $\mathsf{ExpReal}_2$. Then Sim applies the g_i functions to this simulated leakage. Let us denote by $(\ell_1^{\mathrm{R}}, \ldots, \ell_z^{\mathrm{R}})$ the output distribution of $\mathsf{ExpReal}_2$ and by $(\ell_1^{\mathrm{S}}, \ldots, \ell_z^{\mathrm{S}})$ the output distribution of ExpSim_2, which satisfies

$$(\ell_1^{\mathrm{R}}, \ldots, \ell_z^{\mathrm{R}}) \approx_{\gamma_2} (\ell_1^{\mathrm{S}}, \ldots, \ell_z^{\mathrm{S}}) \ . \tag{16}$$

According to the aforementioned lemma [52, Lemma 3], the output distribution of $\mathsf{ExpReal}_1$ is identically distributed to $(g_1(\ell_1^{\mathrm{R}}), \ldots, g_z(\ell_z^{\mathrm{R}}))$. On the other hand, by construction of Sim the output of ExpSim_1 is defined as $(g_1(\ell_1^{\mathrm{S}}), \ldots, g_z(\ell_z^{\mathrm{S}}))$. Finally, (16) clearly implies

$$(g_1(\ell_1^{\mathrm{R}}), \ldots, g_z(\ell_z^{\mathrm{R}})) \approx_{\gamma_1} (g_1(\ell_1^{\mathrm{S}}), \ldots, g_z(\ell_z^{\mathrm{S}})) \ .$$

for some $\gamma_1 \leq \gamma_2$ which concludes the proof.

B.4 Proof of Lemma 3

Proof. We demonstrate hereafter that there exists a simulator Sim such that for every input $\boldsymbol{X}$, $\boldsymbol{k} = (k_0, k_1, \ldots, k_n)$, and with γ_2 satisfying (11), we have

$$\mathsf{Sim}(\Pi, \boldsymbol{X}) \approx_{\gamma_2} \mathsf{LeakageSampler}(\mathsf{ComputTrace}'(\Pi, \boldsymbol{X}, \boldsymbol{k}), \mathcal{F}').$$

Our demonstration proceeds in two steps: we first describe a simulator Sim^* that achieves a perfect simulation assuming that the leakage sampler reveals the entire computation trace of the program (*i.e.* the full inputs of all the operations) except for one iteration in the inner loop (from Step 4 to Step 10), *i.e.* for one value of the loop index j, where this non-leaking iteration might be different in each iteration of the outer loop, *i.e.* for each i. We then show that we can use Sim^* to construct our simulator Sim which outputs a perfect simulation of

$$\mathsf{LeakageSampler}(\mathsf{ComputTrace}'(\Pi, \boldsymbol{X}, \boldsymbol{k}), \mathcal{F}')$$

with probability γ_2 and fails otherwise (which implies the lemma statement).

For each i between 0 and $n-1$, let us denote by j_i^* the loop index value corresponding to the non-leaking inner loop iteration. Namely, for the ith iteration of the main loop, we assume that the operations in Steps 5, 7, 8 and 9 do not leak when $j = j_i^*$. All the other operations, either inside the inner loop for $j \neq j_i^*$, or outside the inner loop, completely leak their input values. We explain hereafter how to complete a perfect simulation of such a leakage from Π (the program description) and $\boldsymbol{X}$ the input tuple of algebraic variables. The corresponding simulator is denoted $\mathsf{Sim}^*(\Pi, \boldsymbol{X}, \mathcal{J})$ where $\mathcal{J} = \{j_0^*, \ldots, j_{n-1}^*\}$ is the set of non-leaking indexes.

All the operations in Step 1 (the $\mathsf{preprocess}$ procedure) have inputs and outputs derived from $\boldsymbol{X}$ which makes all the leakage in this step perfectly simulable from $\boldsymbol{X}$ (by simply evaluating $\mathsf{preprocess}$ on $\boldsymbol{X}$). Then, for the first iteration $i = 0$ of the main loop, the leakage of Step 3 can also be perfectly simulated from $\boldsymbol{X}$. The simulation Sim^* then generates k_0^j uniformly at random from $[0, 2^w)$ for $j \in [0, d] \backslash \{j_0^*\}$. Steps 5, 7, 8 and 9 can thus be perfectly simulated for $j \in [0, j_0^* - 1]$. Note that for the randomization operations of the form $X_{out} \leftarrow \mathsf{R}(X_{in})$, the computation trace output of the $\mathsf{ComputTrace}'$ sampler contains the pair of input output (X_{in}, X_{out}) of the instruction, where the variable X_{out} is substituted for a fresh random value. This can be perfectly simulated by Sim^* by sampling fresh random values for all the outputs X_{out} (and keeping track of these sampled values for the forthcoming operations). For iteration $j = j_0^*$, no leakage must be simulated. For $j = j_0^* + 1$, all the input variables in this iteration are also output variables of a randomization operation R. They can therefore be perfectly simulated with uniform values. Following iterations $j > j_0^* + 1$ can be perfectly simulated from these randomly sampled values and by picking k_0^j uniformly at random from $[0, 2^w)$.

Remark 9. We stress that one would not be able to additionally simulate the leakage in iteration j_0^* without knowledge of k_0. Indeed, by definition the share

$k_0^{j_0^*}$ satisfies

$$k_0^{j_0^*} = k_0 \oplus \bigoplus_{j \neq j_0^*} k_0^j \ .$$

Any leakage in the $\mathsf{indexes}$ procedure for $j = j_0^*$ would (partially) depend on $k_0^{j_0^*}$ and hence (together with the other k_0^j) of k_0. In the same way, a leakage in the swap procedure would depend on the bit $b_\ell^{j_0^*}$ which is derived from $k_0^{j_0^*}$. Furthermore, the leakage of any randomization operation in the loop iteration j_0^* would also depend on the bit $b_\ell^{j_0^*}$ together with the leakage of other iterations. For instance, if the randomization of X_ℓ^1 would leak, then one would observe which random value is sampled as output of $\mathsf{R}(X_\ell^1)$ and consequently deduce whether a swap has occurred or not from the input variables of the next iteration (hence recovering $b_\ell^{j_0^*}$).

For any following iteration $i \geq 1$, we consider two cases depending on whether $j_{i-1}^* = d$ or not. Whenever $j_{i-1}^* = d$, the algebraic variables in input of iteration i are the output of randomization operations that do not leak. They can therefore be perfectly simulated with fresh random values independently of the previous leakage. Whenever $j_{i-1}^* < d$, the input algebraic variables of iteration i are taken from the simulation of randomization operation in the inner loop iteration $j = d$ of the last main loop iteration $i-1$. Now that we have explained how to perfectly simulate the input variables of iteration i we can proceed with a simulation of this iteration in the exact same way as for the first iteration $i = 0$. In particular, Sim^* generates k_i^j uniformly at random from $[0, 2^w)$ for $j \in [0, d] \backslash \{j_i^*\}$ and fresh random values for all the output of the randomization operations from which a perfect simulation is achieved as described above.

Our simulator $\mathsf{Sim}(\Pi, \boldsymbol{X})$ works as follows: it first draws some coin with probability δ' for each operation in order to determine whether the operation leaks or not (*i.e.* whether the random probing function reveals its input or $\perp$). For each iteration i of the main loop, we denote by Fail_i the event that at least one operation leaks in each iteration j of the inner loop. We further denote by Fail the union of the failure events for all the iterations, that is

$$\mathsf{Fail} = \mathsf{Fail}_0 \cup \mathsf{Fail}_1 \cup \ldots \cup \mathsf{Fail}_{n-1} \ .$$

Whenever Fail occurs, the simulator Sim stops and the simulation fails. Whenever Fail does not occur, there exists at least one index j_i^* per iteration i for which the iteration $j = j_i^*$ in the inner loop does not leak. The simulator Sim then runs $\mathsf{Sim}^*(\Pi, \boldsymbol{X}, \mathcal{J})$ with $\mathcal{J} = \{j_0^*, \ldots, j_{n-1}^*\}$ from which a perfect simulation is obtained for all the operations except the non-leaking iterations. From this perfect simulation Sim can then extract the values to be simulated according to the drawn random probing coins.

We thus get a simulator $\mathsf{Sim}(\Pi, \boldsymbol{X})$ which outputs a perfect distribution or fails with probability

$$\gamma_2 = \Pr(\mathsf{Fail}) = 1 - \prod_{i=0}^{n-1} (1 - \Pr(\mathsf{Fail}_i)) \ .$$

Let us denote by Fail_i^j the failure event that at least one operation leaks in the inner iteration j of the iteration i of the main loop. The failure event Fail_i occurs at iteration i if all the iterations of the inner loop leak at least for one operation (in Step among 5, 7, 8, and 9), that is

$$\Pr(\mathsf{Fail}_i) = \Pr(\mathsf{Fail}_i^0 \cap \mathsf{Fail}_i^1 \cap \ldots \cap \mathsf{Fail}_i^d) = \prod_{j=0}^{d} \Pr(\mathsf{Fail}_i^j) \ ,$$

where the product of probabilities holds from the mutual independence of the Fail_i^j's. Each inner loop iteration is composed of m index operations in the procedure indexes, $2m$ randomization operations, and $2m$ copies in the swap procedure which makes a total of $5m$ operations per iteration. For every pair (i, j), the probability that none of these operations leak hence equals

$$1 - \Pr(\mathsf{Fail}_i^j) = (1-\delta)^{5m} \ .$$

From this we, get $\Pr(\mathsf{Fail}_i) = \left(1 - (1-\delta)^{5m}\right)^{d+1} \leq (5m\delta)^{d+1}$, which finally yields

$$\Pr(\mathsf{Fail}) = 1 - \prod_{i=0}^{n-1} (1 - \Pr(\mathsf{Fail}_i)) = 1 - \left(1 - \left(1 - (1-\delta)^{5m}\right)^{d+1}\right)^n \leq n(5m\delta)^{d+1} \ .$$

C Binary Ladder in our Framework

C.1 Montgomery Ladder at the coordinate level

The interested reader is referred to the original paper [27] for the descriptions of `setup-ladder` and `finalize-ladder` functions). For more efficiency, the computations from lines 7 to 11 have been deported after the selection of the algebraic variables (line 6). As for our countermeasure, they can be seen as belonging to the `sequence` operation of the next iteration. Also, we consider here that the scalar has been rewritten so that the indexes can be selected from each bit directly.

Algorithm 7 Montgomery ladder at the coordinate level from [27]

Input: $\boldsymbol{P}$, $\boldsymbol{k} = (k_{n-1}, \ldots, k_1, k_0)$
Output: $[k]\boldsymbol{P}$

1. $(X_{\boldsymbol{QP}}, X_{\boldsymbol{RP}}, M, Y_{\boldsymbol{P}}) \leftarrow \texttt{setup} - \texttt{ladder}(\boldsymbol{P})$
2. $(X_1^0, X_2^0, X_3^0, X_4^0) \leftarrow (X_{\boldsymbol{QP}}, X_{\boldsymbol{RP}}, M, Y_{\boldsymbol{P}})$ // preprocess($\boldsymbol{P}$)
3. **for** $i = 0$ **to** $n - 1$ **do**
4. $\quad (X_1^0, X_1^1), (X_2^0, X_2^1) \leftarrow (X_2^0, X_1^0), (X_1^0 - X_2^0, X_2^0 - X_1^0)$ // sequence(X_1^0, X_2^0)
5. $\quad (b_1, b_2) \leftarrow (k_i, k_i)$ // indexes(k_i)
6. $\quad (X_1^0, X_2^0) \leftarrow (X_1^{b_1}, X_2^{b_2})$
7. $\quad T \leftarrow Y_{\boldsymbol{P}} + 2 \cdot M \cdot X_1^0$
8. $\quad X_1^0 \leftarrow T^2 \cdot (T^2 + 2 \cdot M \cdot T \cdot X_2^0)$
9. $\quad X_2^0 \leftarrow (X_1^0 \cdot (X_2^0)^2)^2 + Y_{\boldsymbol{P}} \cdot T \cdot (X_2^0)^3$
10. $\quad X_3^0 \leftarrow X_1^0 (X_2^0)^2 - T^2 - M \cdot T \cdot X_2^0$
11. $\quad X_4^0 \leftarrow Y_{\boldsymbol{P}} \cdot T^3 \cdot (X_2^0)^3$
12. **end for**
13. $Y \leftarrow \texttt{finalize-ladder}(X_1^0, X_2^0, X_3^0, X_4^0)$ // postprocess($X_1^0, X_2^0, X_3^0, X_4^0$)
14. **return** Y

C.2 Joye Ladder

Algorithm 8 Joye ladder

Input: $\boldsymbol{P}$, $\boldsymbol{k} = (k_0, k_1, \ldots, k_{n-1})$
Output: $\boldsymbol{Y} = f_{\boldsymbol{k}}(\boldsymbol{P})$

1. $(X_1^0, X_2^0) \leftarrow (\boldsymbol{\mathcal{O}}, \boldsymbol{P})$ // preprocess($\boldsymbol{P}$)
2. **for** $i = 0$ **to** $n - 1$ **do**
3. $\quad T \leftarrow 2X_1^0 + X_2^0$ // sequence(X_1^0, X_2^0)
4. $\quad (X_1^0, X_1^1), (X_2^0, X_2^1) \leftarrow (T, X_1^0), (T, X_2^0)$
5. $\quad (b_1, b_2) \leftarrow (1 - k_i, k_i)$ // indexes(k_i)
6. $\quad (X_1^0, X_2^0) \leftarrow (X_1^{b_1}, X_2^{b_2})$
7. **end for**
8. **return** X_1^0

C.3 Signed binary ladder

Algorithm 9 Signed binary ladder [56, Alg. 8]

Input: $\boldsymbol{P}$, $\boldsymbol{k}' = (k_{n-1} \oplus k_{n-2}, \ldots, k_2 \oplus k_1, k_1 \oplus 1)_2$
Output: $[k]\boldsymbol{P}$

1. $(X_1^0, X_2^0) \leftarrow (\boldsymbol{P}, 3\boldsymbol{P})$ // preprocess($\boldsymbol{P}$)
2. **for** $i = 0$ **to** $n - 2$ **do**
3. $\quad T \leftarrow 2X_1^0 + X_2^0$ // sequence(X_1^0, X_2^0)
4. $\quad (X_1^0, X_1^1), (X_2^0, X_2^1) \leftarrow (T, T), (-X_2^0, X_2^0)$
5. $\quad b \leftarrow 1 - k_i'$ // indexes(k_i)
6. $\quad (X_1^0, X_2^0) \leftarrow (X_1^b, X_2^b)$
7. **end for**
8. $Y \leftarrow X_1^0$ // postprocess(X_1^0, X_2^0)
9. **return** Y

D Complexity Overheads

D.1 Performance Analysis

Table 2 illustrates our countermeasure's complexity by giving the multiplicative factor between a regular program and a program with one randomization or the other at different masking orders. Neglecting the linear operations, we evaluate the overall complexity as an equivalent of

$$n \cdot \big(C_{\texttt{sequence}} + (d+1) \cdot (C_{\texttt{indexes}} + m \cdot (2 \cdot C_{\mathsf{R}}))\big)$$

for protected operations and

$$n \cdot C_{\texttt{sequence}}$$

for regular programs. Following the example of the Montgomery ladder in Algorithm 5, we assume one point addition and one point doubling in the `sequence` step, *i.e.* around 12 multiplications. We also assume that we have 2 algebraic variables and h can vary in $\{32, 64, 128\}$. λ for the second randomization is generated in the field ($\mathbb{Z}_p$ or $\mathbb{Z}_{hp}$). We also take p of size 256 bits.

The last row of the table estimates the complexity of a very naive countermeasure in the probing model using ISW (for its inventor Ishai, Sahai, and Wagner) multiplications [32]. The latter would require $(d+1)^2$ multiplications, hence the resulting overheads. Note that they are not enough to protect the scheme in the probing model but we obtain a lower bound of the expected overhead in complexity.

Table 2: Complexity Approximation

	Our countermeasure (overhead)			
	order 1	order 2	order 4	order 8
R_1 - $h = 32$	1,35	1,39	1,47	1,64
R_1 - $h = 64$	1,73	1,81	1,98	2,31
R_1 - $h = 128$	2,58	2,75	3,08	3,75
R_2	3	4	6	10
R_1 & R_2 - $h = 32$	5,48	7,59	11,81	20,25
R_1 & R_2 - $h = 64$	6,77	9,38	14,58	25
R_1 & R_2 - $h = 128$	9,75	13,5	21	36
naive ISW	4	9	25	81

D.2 Implementation Benchmarks

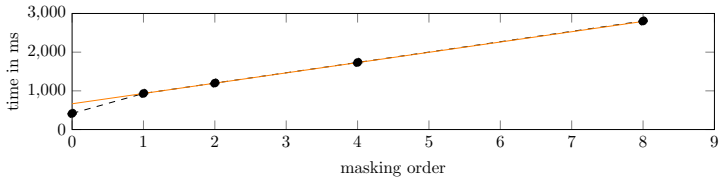

Fig. 7: Benchmarks from Table 1 where masking order 0 corresponds to a regular algorithm. The orange line represents the function $x \mapsto 400 + 266(x + 1)$.