Protecting the Most Significant Bits in Scalar Multiplication Algorithms

Estuardo Alpirez Bock[1(✉)], Lukasz Chmielewski[2,3], and Konstantina Miteloudi[3]

[1] Aalto University, Espoo, Finland
estuardo.alpirezbock@aalto.fi
[2] Masaryk University, Brno, Czechia
chmiel@fi.muni.cz
[3] Radboud University, Nijmegen, The Netherlands
konstantina.miteloudi@ru.nl

Abstract. The Montgomery Ladder is widely used for implementing the scalar multiplication in elliptic curve cryptographic designs. This algorithm is efficient and provides a natural robustness against (simple) side-channel attacks. Previous works however showed that implementations of the Montgomery Ladder using Lopez-Dahab projective coordinates easily leak the value of the most significant bits of the secret scalar, which led to a full key recovery in an attack known as *LadderLeak* [3]. In light of such leakage, we analyse further popular methods for implementing the Montgomery Ladder. We first consider open source *software* implementations of the X25519 protocol which implement the Montgomery Ladder based on the ladderstep algorithm from Düll et al. [15]. We confirm via power measurements that these implementations also easily leak the most significant scalar bits, even when implementing Z-coordinate randomisations. We thus propose simple modifications of the algorithm and its handling of the most significant bits and show the effectiveness of our modifications via experimental results. Particularly, our re-designs of the algorithm do not incurring significant efficiency penalties. As a second case study, we consider open source *hardware* implementations of the Montgomery Ladder based on the complete addition formulas for prime order elliptic curves, where we observe the exact same leakage. As we explain, the most significant bits in implementations of the complete addition formulas can be protected in an analogous way as we do for Curve25519 in our first case study.

Keywords: ECC · Montgomery Ladder · Curve25519 · Complete addition formulas · Side-channel analysis

1 Introduction

Elliptic curve and isogeny based cryptographic implementations commonly make use of the Montgomery Ladder for performing the scalar point multiplication

E. Alpirez Bock—Work partially done while at Radboud University.
L. Chmielewski—This work was partially supported by the Technology Innovation Institute (https://www.tii.ae/) and by Ai-SecTools (VJ02010010) project.

L. Batina et al. (Eds.): SPACE 2022, LNCS 13783, pp. 118–137, 2022.
https://doi.org/10.1007/978-3-031-22829-2_7

[13,21]. The preference for the Montgomery Ladder comes from its efficiency and also from its natural robustness against simple side channel attacks, such as timing, simple power analysis (SPA), and simple electromagnetic analysis. Its robustness comes from the fact that for each loop iteration, we always perform a point addition followed by a point doubling, independent of the bit value we are processing for the scalar (see Algorithm 1). Nevertheless, previous works have shown that implementations of the Montgomery Ladder based on Lopez-Dahab projective coordinates [22] easily leak at least one bit of the scalar via simple side channel observations [3,9]. Lopez-Dahab projective coordinates represent the points on the curve only by means of their x-coordinate in the form $x = \frac{X}{Z}$ and allow for fast computation of the Montgomery Ladder since no divisions need to be performed during the main loop. The leakage in the implementations is caused by the initialisation phase of the algorithm, where the projective representation of the input point is defined as $x = \frac{x}{1}$ and thus one set of input variables of the algorithm is initialised as $X_1 = x$ and $Z_1 = 1$. The other input variables are initialised with the values $X_2 = x^4 + b$ and $Z_2 = x^2$. As we then enter the main loop of the algorithm, we perform some multiplications with $Z_1 = 1$ as operand. However the number of such multiplications varies depending on the value of the key bit we are processing: if the first loop iteration is performed for a key bit with value 1, we perform only one multiplication with $Z_1 = 1$ as operand. On the other hand if the first loop iteration is performed for a key bit with value 0, then we perform three multiplications with $Z_1 = 1$ as operand. Multiplications with operands with value 1 usually consume a notably smaller amount of power than multiplications between two larger values.[1] Thus, if we observe the region of a power trace corresponding to the first loop iteration of the Montgomery Ladder using Lopez-Dahab projective coordinates, we can easily tell the value of the key bit being processed. This value corresponds to the second most significant bit of the key, assuming that the most significant bit is always 1.

Although we are only talking about one bit, the LadderLeak attack [3] took advantage of this leakage to fully break the ECDSA protocol implemented in recent OpenSSL versions. They showed how this small leakage can be exploited together with advanced approaches for solving the *hidden number problem* [11], leading thus to a complete recovery of the secret scalar. In an earlier work, the authors of [9] also identified the same leakage on a hardware implementa-

Algorithm 1. Montgomery Ladder

Inputs: $k = (k_{n-1}, ..., k_0)$,
$G = (G_x, G_y, G_z)$
Output: $R_0 \leftarrow k \cdot G$

```
R0 ← O,   R1 ← G
for i from n − 1 downto 0 do
    if k_i = 1 then
        R0 ← R0 + R1,   R1 ← 2R1
    else
        R1 ← R0 + R1,   R0 ← 2R0
    end if
end for
```

[1] In this paper we will use the term *balanced value* to refer to large values or bitstrings containing similar amounts of 0s and 1s. While we expect operations on such values to consume a notably larger amount of power than operations on small values like zero or one, this may not always be clearly visible due, e.g. to software optimisations.

tion of the Montgomery Ladder. They showed the leakage via simulated power traces and proposed a simple countermeasure: a re-design of the initialisation phase of the algorithm as well as a special treatment of the first loop iteration. The idea is that no registers are initialised with a value equal 1 and when the loop is entered for the first time; we do not need to perform multiplications with operands equal to 1 since the results of such operations are already known. The authors showed that implementing the Montgomery Ladder in this alternative way barely implies any additional costs in terms of execution time, area and power consumption.

In this paper we consider further case studies of the Montgomery Ladder when implemented with other projective coordinates or other point addition and doubling algorithms. We consider both, software and hardware ECC implementations over prime fields and explore whether the aforementioned leakage is present. We confirm its presence, even when Z-coordinate randomisation is employed, and propose corresponding countermeasures to mitigate the easy extraction of the most significant scalar bit(s). Our work is motivated by the results of the LadderLeak attack, which showed how such leakage could be exploited, but also by the fact that previous works showed that mitigating such leakage could be done in a simple and efficient way. Below we elaborate on the implementations we study and modify in this paper.

1.1 Software Implementations of Curve25519

We begin our studies with a software implementation of Curve25519 based on the ladderstep algorithm introduced in [15] (see Algorithm 3 below). This algorithm is a popular choice for implementing the X25519 key-exchange protocol in software [7] (see [25] for alternative implementations of this algorithm and [14] for a tutorial on implementations of Curve25519 on ARM Cortex-M0). Concretely, we consider a recent open source implementation of X25519 from [4][2]. This implementation performs the scalar multiplications via the Montgomery Ladder as described in Algorithm 2 and the ladderstep function is implemented according to Algorithm 3.[3] Note that Algorithm 2 does not assume that the most significant bit of the scalar has a value equal to 1. Instead, the algorithm initialises the registers X_1, Z_1, X_2, Z_2 with values corresponding to the point at infinity and the input point x_P respectively, and then the algorithm executes the ladderstep function for the most significant bit of k, independently of its value. Clearly, if the first loop iteration(s) is (are) executed for scalar bits with value 0, we know that the resulting outputs of the loops are basically equal to their input values, and thus, such loop iterations are not really necessary. However,

[2] The source code from [4] is located in the following repository: https://github.com/sca-secure-library-sca25519/sca25519.

[3] Note that in the X25519 protocol, the most significant (254th) bit of the secret scalars is always set to 1; this is done by anding the most significant scalar byte with `0x7F|0x40` in [4]. However, since we consider the ECDSA protocol then the most significant scalar bits can be 0 and we need to consider fully random scalars.

the algorithm is implemented in this way with the scope of protecting the length of the key with respect to timing and side channel attacks. That is, if we only start executing the main loop of the algorithm once we've reached the first key bit with value 1, we would obtain power traces of different sizes, depending on where this first 1 is located. Implementing the Montgomery Ladder according to Algorithm 2 also relaxes the assumption that the most significant bit of the scalar is always 1, and thus we always talk about a key space of size $2^{|k|}$ and not of size $2^{|k|-1}$.

However as we show in this paper, this approach does not really protect the values of the most significant bits (MSBs) of the scalar when considering SPA. Namely, loop iterations for MSBs with value 0 can be easily distinguished from the rest of the loop iterations, and thus it is easy for an adversary to extract all MSBs of the scalar up to (and including) the first 1. This happens because the loop iterations for MSBs with value 0 have a notably different power consumption than the rest, given that many operations performed within those loops use operands with the value 0 or 1. Such operands are only overwritten with larger, balanced values once we finally iterate a loop for a key bit with value 1.

Leakage on DPA-Protected Implementations. We also verify the presence of this leakage on the second implementation from [4] (see their Algorithm 2), which is an SCA-protected implementation of ephemeral X25519 that randomises the projective representation of the input value. However, the leakage is still present since the input coordinates representing the point at infinity are initialised with the values of 0 and 1. We thus show that projective coordinate randomisation does not protect the MSBs of Curve25519 implementations.

Countermeasure. We modify Algorithm 2 to remove the aforementioned leakage in a simple, but effective way (see Algorithm 6). Our approach relies on always executing the ladderstep function using balanced operands as inputs. This way, the corresponding measurements always have similarly looking patterns and it is not easy to determine when the first loop iteration for a key bit with value 1 is executed. Our approach is implemented as follows. We initialise all input variables X_1, Z_1, X_2, Z_2 with randomly chosen, balanced values. Additionally, we use two new variables W_1 and W_2 initialised with values needed for the first ladderstep execution for a scalar bit with value 1. These values are the result of additions and subtractions with operands with value 1, and we pre-calculate them to avoid performing such operations during ladderstep. Now, if the most significant scalar bit is 0, we execute the normal ladderstep from Algorithm 3, but with balanced input variables. Note also that the outputs of these loops are irrelevant for the actual calculation of kP. We repeat this process for all key bits until we reach the first scalar bit with value 1. For this scalar bit, we execute a special version of ladderstep (see Algorithm 7), where we use the pre-calculated W_1 and W_2. After this loop iteration, we finally have all variables X_1, Z_1, X_2, Z_2 overwritten with correct, balanced values. Thus from this point on, we can simply continue with the regular execution of the Montgomery Ladder using the

standard ladderstep from Algorithm 3. We avoid potential operation leakage by ensuring that the special and regular ladderstep both consist of the same operation sequence, and by using only constant-time operations.

For implementing the countermeasure described above, we consider two alternative software techniques and compare the costs of each. First we make use of arithmetic constant-time "conditional swap" operations (referred to as cswap operations in this paper) for alternating between the two versions of the ladderstep function as described above. The cswap(X, Y, c) routine simply swaps the first two inputs if and only if $c = 1$. This is achieved without traditional conditional statements in the following way. First $c \in \{0, 1\}$ is converted to the form $c' = -c$ (now $c' = 0$ if $c = 0$ and $c' = 0xFF...$, otherwise). Then, the conditional swap on the first argument (and similarly on the second one) is performed arithmetically: X `^=` c' `&` $(X$`^`$Y)$. Thus, the value of X remains the same for $c' = 0$ and is overwritten with Y, otherwise.

The resulting re-design using cswap, while secure, incurs a notable performance penalty due to the extra arithmetic operations. However, our second re-design alternative is based on secret-memory access and incurs a much smaller performance penalty. Here, instead of doing a swap depending on c, we put X and Y into an array and we access them through memory access depending on the value of c. Note that in our implementation, c depends directly on the secret scalar, hence the name *secret-memory* access. We note that the security of our second re-design may be dependent on the architecture used for running the code. Namely if memory access is not always constant-time (as is the case for architectures equipped with memory caching, for example), some small key dependencies may be visible on power consumption traces.

We would like to underline that the above countermeasures aim to efficiently protect against SPA, but not against more sophisticated single-trace attacks, like [26], for which extra costly countermeasures are required.

1.2 Hardware Implementations of the Complete Addition Formulas

Our second case study is performed analogously to our first one, but we consider *hardware* implementations based on the complete addition formulas from Renes, Costello and Batina [28]. These formulas gained popularity since they allow addition of any two points on Weierstrass curves and avoid thus exceptions during the computations. Moreover, these formulas can be used for implementing both the point addition and point doubling operations within the main loop of the Montgomery Ladder. We consider an open-source implementation presented in [27] which is based on Algorithm 7 in [28]. This implementation does not assume that the MSB of the scalar is equal to 1 and for each loop iteration, it executes Algorithm 7 of [28] twice: once for a point addition and once for a point doubling. As for our first case study on Curve25519, we show via power consumption measurements that the MSBs of the scalar can be (very) easily extracted from this hardware implementation. We believe that a countermeasure in the same style as for Curve25519 can be proposed and we leave that for future work.

Algorithm 2. Montgomery Ladder for x-coordinate-based scalar multiplication on $E : y^2 = x^3 + 486662x^2 + x$ [4]

Inputs: $k \in \{0, ..., 2^{255} - 1\}, x_P$

$X_1 \leftarrow 1; \quad Z_1 \leftarrow 0; \quad X_2 \leftarrow x_P; \quad Z_2 \leftarrow 1; \quad p \leftarrow 0$
for $i \leftarrow 254$ **downto** 0 **do**
  $c \leftarrow k[i] \oplus p; \quad p \leftarrow k[i]$ ▷ $k[i]$ denotes bit i of k
  $(X_1, Z_1, X_2, Z_2) \leftarrow \mathsf{cswap}(X_1, Z_1, X_2, Z_2, c)$
  $(X_1, Z_1, X_2, Z_2) \leftarrow \mathsf{ladderstep}(x_p, X_1, Z_1, X_2, Z_2)$
end for
return (X_1, Z_1)

Exploiting the Leakage. The leakage discussed int this paper may be particularly useful for preparing template [5,23] and single-trace horizontal attacks [17,20], since it easily reveals the length of loop executions. It also may reveal time interval cycles when specific operations such as multiplications take place, which is useful information for performing fault injection [10]. On implementations of the complete addition formulas, the leakage might let an adversary distinguish a point addition from a doubling (see Sect. 5). Finally, this leakage can be used for *zero value attacks* [1,16], which require the knowledge of some initial scalar bits.

2 Background and Experimental Setup

In [24] Montgomery introduced efficient x-coordinate-only formulas for computing addition and doubling operations between points in elliptic curves. These formulas would later be simply referred to as the *Montgomery Ladder*, described in Algorithm 2. The ladderstep process corresponds to a point addition and a point doubling operation. There exist different formulas for implementing the ladderstep process using projective coordinates (e.g. [15,22,28]), and the choice of the formulae is usually determined by the type of implementation we are considering (software vs hardware, type of curve used, etc.). In [15] the authors proposed Algorithm 3 for efficient software implementations of Curve25519. Implementations based on this algorithm only need to make use of two extra variables (T_1 and T_2) for the ladderstep processes. Additionally, each ladderstep process consists of only 6 multiplications and 4 squarings, plus a few addition and subtraction operations. This method of implementing the ladderstep has also been embraced on a recent work [4], where the authors implement the X25519 key exchange protocol in combination of a large amount of side-channel countermeasures.

Curve25519. The X25519 key-exchange protocol is based on Curve25519 defined over $\mathbb{F}_{2^{255}-19}$ as: $E : y^2 = x^3 + 486662x^2 + x$. The protocol uses 255 bit long public and secret keys, where the secret key is a randomly generated scalar k and the public key corresponds to a little-endian encoding of the x-coordinate

Algorithm 3. Single Montgomery ladder step on Curve25519 from [15]

Inputs: x_P, X_1, Z_1, X_2, Z_2

1: $T_1 \leftarrow X_2 + Z_2$
2: $X_2 \leftarrow X_2 - Z_2$
3: $Z_2 \leftarrow X_1 + Z_1$
4: $X_1 \leftarrow X_1 - Z_1$
5: $T_1 \leftarrow T_1 \cdot X_1$
6: $X_2 \leftarrow X_2 \cdot Z_2$
7: $Z_2 \leftarrow Z_2 \cdot Z_2$
8: $X_1 \leftarrow X_1 \cdot X_1$
9: $T_2 \leftarrow Z_2 - X_1$
10: $Z_1 \leftarrow T_2 \cdot a24$
11: $Z_1 \leftarrow Z_1 + X_1$
12: $Z_1 \leftarrow T_2 \cdot Z_1$
13: $X_1 \leftarrow Z_2 \cdot X_1$
14: $Z_2 \leftarrow T_1 - X_2$
15: $Z_2 \leftarrow Z_2 \cdot Z_2$
16: $Z_2 \leftarrow Z_2 \cdot x_P$
17: $X_2 \leftarrow T_1 + X_2$
18: $X_2 \leftarrow X_2 \cdot X_2$

return (X_1, Z_1, X_2, Z_2)

of a point P on the curve. In the protocol, the shared secret corresponds to the resulting point on the curve from the scalar multiplication of kP. For calculating kP, we use the Montgomery Ladder algorithm. Note that in X25519, the most significant scalar bit is always set to 1, and if the scalar needs to be a multiple of word length (that is 256 bits for many architectures) then it is extended with zeroes. However, note that Curve25519 is also used in EdDSA protocols, which are ECC-based signature schemes [8].[4] Here, the scalar is the resulting hash of the message to be signed together with an auxiliary parameter b. Thus in this case, the resulting scalar does not have a fixed value for its most significant bits.

Complete Addition Formulas. Renes et al. [28] introduced the *complete addition formulas for prime order elliptic curves*, which are optimisations on formulas presented earlier by Bosma and Lenstra in [12]. These formulas are said to be *complete* on prime order Weirstrass curves of the form $y^2 = x^3 + ax + b$ since they can compute the sum of any two points on these curves. Moreover, these addition formulas can be used for implementing both, the point addition and point doubling operations within an implementation of the Montgomery Ladder. It is believed that using the same addition formula for implementing both operations may provide additional robustness in light of SCA attacks, since it becomes more difficult to distinguish a point addition from a point doubling operation, and behavioural effects of branching can be easily mitigated.

These formulas also use projective representation of the input points, in the form $P = (X, Y, Z)$. Thus, we additionally use the y-coordinate of the input points for these formulas. The authors present one general addition formula and further optimisations for special families of curves, in cases where the constant a has the values -3 or 0. In this paper, we will focus on an open source implementation of Algorithm 7 of [28]. This flavour of the formula is applicable for short Weierstrass curves which set the constant $a = 0$. As the authors explain, such a curve has appeared in Certicom's SEC-2 standard [29] which specifies the curve secp256k1, used in the Bitcoin protocol.

[4] We refer to EdDSA with the parameters of Curve25519 as Ed25519 [7].

2.1 Experimental Setup and Side-Channel Evaluation

We perform our experiments and verifications with respect to three open source implementations. The first two correspond to implementations from [4], which are designs of X25519 using Montgomery Ladder for performing scalar multiplications. For simplicity and for obtaining a general result, we first focus on the plain and unprotected design from the repository, which is an implementation of the Montgomery Ladder according to Algorithm 2, and implements the ladderstep according to Algorithm 3. We will then show that the second design from the repository, which implements some SCA countermeasures, also leaks the most significant bits of the scalar. We run these designs on a Cortex-M4 on an STM32F407IGT6 board clocked at 168 MHz. For side-channel evaluation of the designs, we measure current using the Riscure Current Probe [30] and we collect the traces using the PicoScope 3406D oscilloscope with the sampling frequency of 10^9 samples per second. Finally for side-channel analysis we use the Inspector software by Riscure [31]. Subsequently, we re-design Algorithms 2 and 3 and propose countermeasures. We test our new designs by performing experiments on the same experimental setup as described above.

We conduct our second case study on the hardware accelerator implementing the Montgomery Ladder using the complete addition formulas [27]. We run the design on an FPGA SAKURA-G board [19] and measure its power consumption via a Teledyne Lecroy Waverunner 8404M oscilloscope with the sampling of frequency 10^8 samples per second.

3 Leakage on Curve25519

In this section we analyse the Montgomery Ladder implemented according to Algorithms 2 and 3 and explain how the most significant scalar bits can be extracted via SPA. We confirm our intuitions via experimental results by measuring the power consumption of the algorithm when running on a microcontroller.

3.1 Initial Loop Iterations

We now focus on Algorithm 2, whereby the ladderstep process is defined in Algorithm 3. As we see, the input variables are initialised as $X_1 \leftarrow 1$, $Z_1 \leftarrow 0, X_2 \leftarrow x_P$, and $Z_2 \leftarrow 1$. If the most significant scalar bit equals 0 then all variables will hold these values when we enter the ladderstep process. Otherwise, the cswap operation will be executed and the variables will have the values $X_1 = x_P, Z_1 = 1, X_2 = 1, Z_2 = 0$. We first focus on the former case. In the following, we refer to $k[0]$ as the most significant bit of the scalar. Observe that in Algorithm 4 (and in all algorithms in this paper) "=" denotes only the equality relation and "←" is used for assignment.

Case $k[0] = 0$. Algorithm 4 shows the explicit operand values that will be used during the first execution of ladderstep if the most significant bit of the scalar

Algorithm 4. Ladder step when entered just after the initialisation for $k[0] = 0$

Inputs: $x_P, X_1 = 1, Z_1 = 0, X_2 = x_p, Z_2 = 1$

1: $T_1 \leftarrow x_P + 1$
2: $X_2 \leftarrow x_P - 1$
3: $Z_2 \leftarrow 1 + 0$
4: $X_1 \leftarrow 1 - 0$
5: $T_1 \leftarrow T_1 \cdot 1$
6: $X_2 \leftarrow X_2 \cdot 1$
7: $Z_2 \leftarrow 1 \cdot 1$
8: $X_1 \leftarrow 1 \cdot 1$
9: $T_2 \leftarrow 1 - 1 = 0$
10: $Z_1 \leftarrow 0 \cdot a24$
11: $Z_1 \leftarrow 0 + 1$
12: $Z_1 \leftarrow 0 \cdot 1$
13: $X_1 \leftarrow 1 \cdot 1$
14: $Z_2 \leftarrow (x_P + 1) - (x_P - 1) = 2$
15: $Z_2 \leftarrow 2 \cdot 2 = 4$
16: $Z_2 \leftarrow 4 \cdot x_P$
17: $X_2 \leftarrow (x_P + 1) + (x_P - 1) = 2x_P$
18: $X_2 \leftarrow 2x_P \cdot 2x_P = 4x_P^2$
return $(X_1 = 1, Z_1 = 0)$;
$(X_2 = 4x_P^2, Z_2 = 4x_P)$

has a value of 0. We highlight in gray all operations that will be performed with a variable with value 1 or 0 as operand, or operations where the variables are not overwritten with any new values. Particularly interesting are the multiplications performed in steps 5 through 8 and in step 13, which are all multiplications with at least one operand with value 1. In steps 10 and 12, we perform multiplications with operands with value 0. We can expect to see very small power consumption peaks in the power trace regions corresponding to the execution of these steps.

Note that by the end of the process, i.e. by the end of this first loop iteration, the variables X_1 and Z_1 preserve their values of 1 and 0 respectively. Moreover, for X_2 and Z_2, note that $\frac{X_2}{Z_2} = \frac{4x^2}{4x} = \frac{x}{1}$, i.e. these projective coordinates preserve their original value as well. Thus, the input values for the next loop iteration (for the second bit of the scalar) are equivalent to the input values for the first iteration. If the second loop iteration is executed again for a key bit with value 0, i.e. if $k[1] = 0$, we have basically the same situation as the one described in Algorithm 4. Namely, although variable Z_2 enters the loop with a value different from 1, Z_2 is quickly overwritten in step 3 with a value equal to 1. This holds for all following loop iterations until we finally process a key bit with value 1.

Case $k[0] = 1$. We now analyse the first ladderstep execution for a key bit with value equal to 1. Recall that when executing the ladderstep for a key bit with value 1, we first swap the content of the variables via the cswap operation. Thus the variables enter the loop with values $X_1 = x_P, Z_1 = 1, X_2 = 1, Z_2 = 0$. Recall that the values of X_2 and Z_2 may vary from x_P and 1 respectively if the ladderstep function was previously executed for a key bit with value 0. However, the variables will retain the relation $\frac{X_1}{Z_1} = \frac{x}{1}$. For simplicity, we assume here that $Z_1 = 1$. Algorithm 5 shows the explicit operand values that will be used during this loop execution. In this algorithm, the variables w_i denote some operand value larger than 1 (usually, some balanced operand value). As we can see, only steps 5 and 6 consist of multiplications with operands with value 1.

We can expect higher power consumption peaks on the power trace region corresponding to this execution of the ladderstep, in comparison to the regions corresponding to the $k[0] = 0$ case. Moreover, note that by the end of the loop,

Algorithm 5. Ladder step executed for the first scalar bit equal to 1 ($k[i] = 1$)

Inputs: $x_P, X_1 = x_P, Z_1 = 1, X_2 = 1, Z_2 = 0$

1: $T_1 \leftarrow 1 + 0$
2: $X_2 \leftarrow 1 - 0$
3: $Z_2 \leftarrow x_P + 1$
4: $X_1 \leftarrow x_P - 1$
5: $T_1 \leftarrow 1 \cdot (x_P + 1)$
6: $X_2 \leftarrow 1 \cdot (x_P - 1)$
7: $Z_2 \leftarrow (x_P + 1) \cdot (x_P + 1) = (x_P + 1)^2$
8: $X_1 \leftarrow (x_P - 1) \cdot (x_P - 1) = (x_P - 1)^2$
9: $T_2 \leftarrow (x_P + 1)^2 - (x_P - 1)^2 = w_1$
10: $Z_1 \leftarrow w_1 \cdot a24 = w_2$
11: $Z_1 \leftarrow w_2 + (x_P - 1)^2 = w_3$
12: $Z_1 \leftarrow w_1 \cdot w_3 = w_4$
13: $X_1 \leftarrow (x_P + 1)^2 \cdot (x_P - 1)^2$
14: $Z_2 \leftarrow (x_P - 1) - (x_P + 1) = w_5$
15: $Z_2 \leftarrow w_5 \cdot w_5 = w_6$
16: $Z_2 \leftarrow w_6 \cdot x_P$
17: $X_2 \leftarrow (x_P - 1) + (x_P + 1) = w_7$
18: $X_2 \leftarrow w_7 \cdot w_7 = w_8$
return
$(X_1 = (x_P + 1)^2 \cdot (x_P - 1)^2, Z_1 = w_4)$;
$(X_2 = w_8, Z_2 = w_6 x_P)$

all variables have been overwritten with some more balanced values. Thus in all following executions of ladderstep, we can expect to see high power consumption peaks. Next, we verify our assumptions via power consumption measurements.

3.2 Experimental Verification

We run the implementation of the Montgomery Ladder with selected scalar values, and record its power consumption as described in Sect. 2.1. We consider cases where the most significant bit(s) of the scalar are 0s and cases where the most significant bit of the scalar is 1. More concretely, we consider two keys with the following values for their first bits: $k_1[0..7] = \texttt{0x04} = 00000100$ and $k_2[0..7] = \texttt{0x7F} = 01111111$.[5] When comparing the power traces generated for each key, we expect notably different power consumption profiles for the regions corresponding to the processing of the first 5 bits of the scalars. After that, we expect to see very similar power consumption profiles.

Figure 1 shows two power traces overlapped. The blue coloured trace corresponds to the execution of the algorithm on k_1 and a fixed input point P.[6] The yellow coloured trace corresponds to the execution with k_2 and the same input P. We mark in the red box the regions corresponding to the most significant bits of the scalars. As we can observe, these regions differ notably from each other and they do not align. The remaining regions of the power traces align very well with each other, since they correspond to executions of the ladderstep where we always use operands with balanced values. We repeated the experiment in a similar setting but with random input points and the result was very similar.

In Fig. 2 we include power traces measured on the second implementation from [4], which is also an implementation of Algorithm 2, but additionally protected with projective Z-coordinate randomisation. We can observe a very

[5] Note that 256th bit of the scalar is always set to 0 since $p = 2^{255} - 19$.

[6] $P_x = \texttt{0x67C5590EF5591AEEE312308D155579DC042E497FEC764BB3CAF3DE88597B8C24}$.

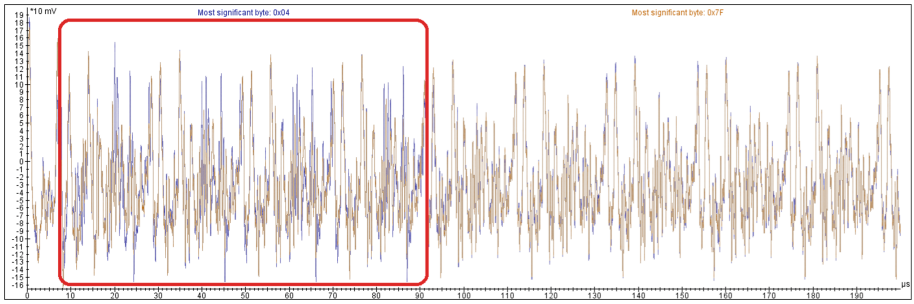

Fig. 1. Comparison of power profiles with scalars starting with `0x04` and `0x7F`.

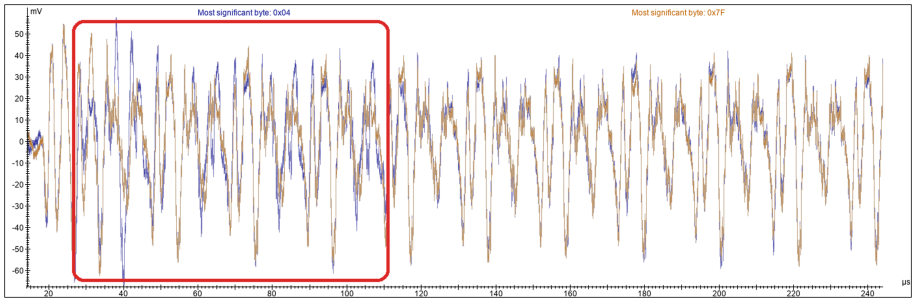

Fig. 2. Comparison of power profiles with scalars starting with `0x04` and `0x7F` when the Z-coordinate is randomized.

similar leakage as in Fig. 1.[7] As explained in the introduction, the projective randomisation is applied only to the variables corresponding to the input point P, but not to the variables corresponding to the point at infinity and thus, many operations with operands equal to 1 or 0 are performed during the initial loops.

4 Protecting the Most Significant Bits in Curve25519

We now present our proposed modification of the Montgomery Ladder, which protects the most significant bits of the scalar.[8] As mentioned before, our idea consists on always using balanced operands during all executions of the ladder-step process. For this, we initialise the input variables with dummy, balanced values and use these values for all executions of the main loop until we reach the first 1 of the scalar. For the loop iteration corresponding to the first 1 of the

[7] We acknowledge that the traces in Fig. 2 look different than the ones collected from the first implementation. This is caused not only by differences in implementations, but also due to the fact that these new traces were collected later on with a new physical setup (although probes and oscilloscopes were equivalent models).

[8] We will provide a link to the code repository in the final version of the paper.

Algorithm 6. Modified x-coordinate-based Montgomery Ladder

Inputs: $k \in \{0, ..., 2^{255} - 1\}, x_P$

$X_1 \leftarrow\$ \mathbb{F}_p$; $Z_1 \leftarrow\$ \mathbb{F}_p$; $X_2 \leftarrow\$ \mathbb{F}_p$; $Z_2 \leftarrow\$ \mathbb{F}_p$;
$W_1 \leftarrow x_P + 1$; $W_2 \leftarrow x_P - 1$; $p, s \leftarrow 0$;
for $i \leftarrow 254$ **downto** 0 **do**
  $t \leftarrow (k[i] \vee s) \oplus s$ ▷ $k[i]$ denotes bit i of k
  $c \leftarrow k[i] \oplus p$; $p \leftarrow k[i]$
  $(X_1, Z_1, X_2, Z_2) \leftarrow \mathsf{cswap}(X_1, Z_1, X_2, Z_2, c)$
  $(X_1, Z_1, X_2, Z_2) \leftarrow \mathsf{ladderstep}(x_p, X_1, Z_1, X_2, Z_2, W_1, W_2, t)$
  $s \leftarrow s \vee t$
end for
return (X_1, Z_1)

scalar, we execute a special version of the ladderstep, where we finally use the operand values necessary for a correct calculation of kP.

Algorithm 6 describes our proposed modification of the Montgomery Ladder. All input variables X_1, Z_1, X_2, and Z_2 are initialised with randomly chosen balanced values. These are 32-byte values which represent elements of $\mathbb{F}_p$. Note that these values can either be chosen at random for each execution, they can be derived from the input point P or they can also just be hardcoded in the implementation. Additionally, we also use two new variables W_1 and W_2 that we initialise as follows: $W_1 \leftarrow x_P + 1, W_2 \leftarrow x_p - 1$. Both W_1 and W_2 contain values needed when we execute ladderstep for the first 1 in the scalar. Note that in Algorithm 5 these two values ($x_P + 1$ and $x_P - 1$) are obtained from an addition and subtraction with value 1, performed in steps 3 and 4. These are the first operations in the loop which actually depend on the input point value x_P. Moreover, these two values are used as operands in steps 7, 8, 14 and 17 in Algorithm 5.

We now describe the operation flow of Algorithm 6 when processing the scalar bits. If the most significant bit is 0 then we execute the normal ladderstep as described in Algorithm 3. Note that in this case, the inputs to the ladderstep process will be (dummy) balanced variables, set in the second line of the algorithm. We repeat this process until we reach the first scalar bit with the value 1. When we reach the first 1 of the scalar, we execute a special variation of the ladderstep function (Algorithm 7), where we make use of the pre-calculated values W_1 and W_2. After this iteration, we have all variables X_1, Z_1, X_2, Z_2 overwritten with correct values and we simply continue the regular execution of the Montgomery Ladder using the standard ladderstep function (Algorithm 3).

4.1 Implementing Our Proposed Modification

For implementing Algorithm 6, we need to take special care of the two following aspects. First, we need to determine *when* we encounter the first non-zero scalar bit so we can execute the modified ladderstep (for $t = 1$). Second, the modified ladderstep loop should be executed only once. Naturally, all operations

Algorithm 7. Single Montgomery ladder step for the case $t = 1$

Inputs: $x_P, X_1, Z_1, X_2, Z_2, W_1, W_2$

1: $T_1 \leftarrow X_2 + Z_2$
2: $X_2 \leftarrow X_2 - Z_2$
3: $Z_2 \leftarrow X_1 + Z_1$
4: $X_1 \leftarrow X_1 - Z_1$
5: $T_1 \leftarrow T_1 \cdot X_1$
6: $X_2 \leftarrow X_2 \cdot Z_2$
7: $Z_2 \leftarrow W_1 \cdot W_1$
8: $X_1 \leftarrow W_2 \cdot W_2$
9: $T_2 \leftarrow Z_2 - X_1$
10: $Z_1 \leftarrow T_2 \cdot a24$
11: $Z_1 \leftarrow Z_1 + X_1$
12: $Z_1 \leftarrow T_2 \cdot Z_1$
13: $X_1 \leftarrow Z_2 \cdot X_1$
14: $Z_2 \leftarrow W_2 - W_1$
15: $Z_2 \leftarrow Z_2 \cdot Z_2$
16: $Z_2 \leftarrow Z_2 \cdot x_P$
17: $X_2 \leftarrow W_2 + W_1$
18: $X_2 \leftarrow X_2 \cdot X_2$

return (X_1, Z_1, X_2, Z_2)

need to be implemented in constant-time, else we might observe small scalar-dependent operation leakages in the power traces. In the following we explain how we identify the first non-zero bit of the scalar and how we ensure that the modified ladderstep algorithm (Algorithm 7) is executed only once and is hard to distinguish from a regular ladderstep.

Note that the variable s is initialised to 0. Then, if $k[0] = 0$ (at the beginning of the scalar multiplication), variable t is set to 0. This follows for all subsequent scalar bits that equal to 0 because at the end of the loop s retains the value 0 ($s = 0 \vee 0$). When $k[i] = 1$ for the first time, t is set to 1 right at the beginning of the loop. Namely, $s = 0$ and thus we calculate $t \leftarrow (1 \vee 0) \oplus 0$. Now we will execute the special case for ladderstep since $t = 1$. Note that at the end of this loop, right after executing the special ladderstep, s will be set to 1: $s \leftarrow 0 \vee 1$. In all subsequent loops t will be set to 0 regardless the value of $k[i]$ because $(k[i] \vee 1) \oplus 1$ always equals 0. By ensuring that $t = 0$ for all subsequent loop iterations, we ensure that we execute the standard ladderstep process from Algorithm 3.

Note that the original ladder step (for $t = 0$) executes the exact same instructions as the modified one (for $t = 1$). Their only difference is the use of some registers as operands as we explain in the next subsection. Thus, there may *still* be data leakage present in the above operations, but there is no operation leakage. Moreover, as we show later in Sect. 4.3 via side-channel evaluation, the present data leakage is small and not visible by SPA means.

4.2 Implementations of the Ladder Step

Our proposed Montgomery Ladder described in Algorithm 6 needs to switch seamlessly between both Algorithm 3 and Algorithm 7. Note that the algorithms execute the same operations but differ only on how the following steps are implemented: 7, 8, 14, and 17—see Table 1 for details. As we see from the table, we need to seamlessly alternate between parameters Z_2 and W_1 in step 7, X_1 and W_2 in step 8, and T_1, X_2 and W_2, W_1 in steps 14 and 17 respectively.

We now describe how we implement the ladderstep from Algorithm 6 alternating smoothly between both versions of the ladderstep. Essentially, we want to ensure that the same sequence of operations is always performed, regardless of the used ladderstep version in order to stop SPA. To achieve this, we combine Algorithms 3 and 7 into one software implementation since both algorithms use

Table 1. Different Steps between Algorithm 3 and Algorithm 7.

Step	Algorithm 3	Algorithm 7
7	$Z_2 \leftarrow Z_2 \cdot Z_2$	$Z_2 \leftarrow W_1 \cdot W_1$
8	$X_1 \leftarrow X_1 \cdot X_1$	$X_1 \leftarrow W_2 \cdot W_2$
14	$Z_2 \leftarrow T_1 - X_2$	$Z_2 \leftarrow W_2 - W_1$
17	$X_2 \leftarrow T_1 + X_2$	$X_2 \leftarrow W_2 + W_1$

the same sequence of operations and only their operands differ. When the most significant scalar bit is 1, we choose operands as in Algorithm 7. When the bit is 0, we choose operands as in Algorithm 3. We propose two alternatives for implementing this operand switch:

1. a cswap-based implementation and
2. an implementation based on secret-memory accesses.

These methods differ in provided security guarantees and performance impact as we explain bellow.

cswap Based Implementation. Our first design is based on conditional swap (cswap). The cswap(X, Y, c) routine swaps the content of the inputs X and Y if and only if $c = 1$. For the sake of simplicity let us consider 32-bit values. In this case cswap can be implemented as follows:

```
c' = - c; //now c'=0xFFFFFFFF if c=1 and 0 otherwise
TMP = X;
X ^= c' & (X ^ Y);
Y ^= c' & (TMP ^ Y);
```

Since in our implementation the operands are 255-bit values, the last 3 lines need to be repeated multiple times to swap all words of the operands.

While this implementation is not very fast, it has the following advantage: the sequence of addresses accessed by the algorithm does not depend on the secret scalar. Therefore, the implementation is constant-time even on a target equipped with data caching and we obtain a design which is constantly robust against SPA, independently of the platform we are running it on. For implementing, we use the same extra memory as for our secret memory access-based method (described below), but instead of accessing the memory directly we perform cswaps (depending on the t value from Algorithm 6) twice: just before and just after the operations from Table 1.

Secret-Memory Accesses. Our second proposed re-design uses secret scalar-dependent access to memory locations. The memory locations correspond to the operands we are using within the loop. The bit t from Algorithm 6 indicates which operands we use. We access the memory corresponding to the operands from Algorithm 7 if $t = 1$, and according to Algorithm 3 otherwise.

Table 2. Performance Evaluation.

Implementation	Time (milliseconds):	Extra Memory (bytes):
Unprotected Imp.:	5.62	-
Cswap-based Imp.:	7.6 (+35.2%)	$8 * 32 = 256$
Secret-Memory Access Imp.:	5.81 (+3.4%)	$8 * 32 = 256$

This implementation is fast but it is constant-time only if the access to the memory by the microcontroller is constant-time. Thus, the robustness of this countermeasure depends on the used platform. Since, our target, a Cortex-M4 on an STM32F407IGT6 board, does not have data caches, the memory access is expected to be constant-time as long as the same SRAM region is accessed;[9] as shown in [2] this target has 2 different regions with different characteristics. To increase the probability that the memory accesses are to the same region, we declare the alternating operands as global variables next to each other. In particular, we keep pairs of the values in an array with two elements and access either the original value for $t = 0$ or W_1 and W_2 for $t = 1$. There are 4 values in total for which we need to keep the corresponding pre-computed values. Additionally, there are 4 balanced values that we pre-calculate and which are hard-coded in our implementation. Thus, we increase the memory usage by 8 coordinates of 32-bytes each.

4.3 Evaluation of Our Countermeasures

We now present our benchmark results comparing our two proposed re-designs with the original one from [4]. We later perform a side-channel evaluation and confirm the effectiveness of our proposed countermeasures. All experiments presented in this section are performed as described in Sect. 2.1.

Performance Evaluation. The performance evaluation results are presented in Table 2. We have checked that all implementations are constant-time. Sometimes, minimal jitter takes place due to instruction caching.[10] As expected, the SCA countermeasure against SPA comes at a cost in our re-designs since we are performing additional arithmetic operations on each loop execution when setting the values of the variables s and t. For our implementation using cswap operations, the overhead is of 35.2%. However for our re-design using secret memory access, the overhead is only of about 3.4%. The memory overheard is small: for both implementations it consists of 256 extra bytes, which come from 8 extra global variables in the finite field $\mathbb{F}_p$, where $p = 2^{255} - 19$.

[9] The target has however an instruction cache. This caching mechanism is randomized, but since the sequence of instructions is always the same in our algorithms, this potential timing difference is independent from the scalar.

[10] The sequence of instructions performed by our algorithms is always the same but the instruction caching of our target seems to be random.

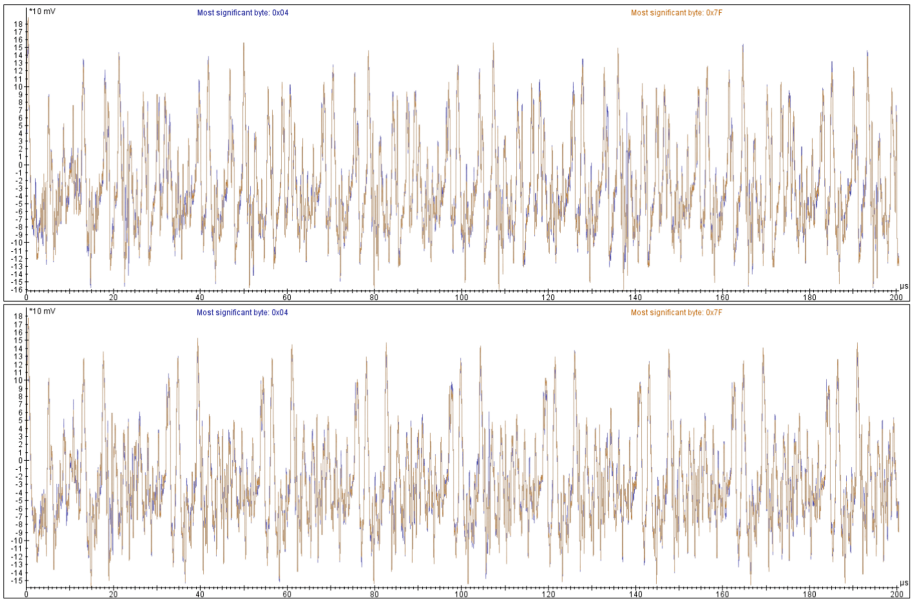

Fig. 3. Comparison of power profiles scalars starting with `0x04` and `0x7F` for the cswap-based implementation (top) and the secret-memory access implementation (bottom).

Side-Channel Evaluation. We run both of our modified implementations using the same inputs as in Sect. 3.2. Namely, we consider scalars k_1 and k_2 starting with `0x04` and `0x7F`, and the same fixed point for each case. Figure 3 shows the resulting power traces for both implementations. We conduct the experiment a total of 20 times for verification, obtaining always the same result for both cases. The top plot in Fig. 3 shows that indeed the cswap-based implementation is protected against SPA. The bottom plot confirms that different memory-access does not generate an SPA-detectable leakage on our evaluation target. We also repeated the experiment in a similar setting but with random input points and the result was very similar for both implementations. Therefore, we can confirm the effectiveness of our designs.

Note that as expected (given our performance evaluation), the secret-memory access implementation is visibly faster than the cswap-based one. This is visible from the repeating pattern in the traces, which corresponds to a loop iteration in the Montgomery Ladder. This pattern is notably longer in the traces corresponding to the cswap-based implementation.

Evaluation Limitations. Since we only consider SPA, we use visual means to determine the leakage. We are aware that even for our protected implementations, automated leakage detection like TVLA [6,18] would indicate leakage exploitable by more advanced side-channel attacks. This is expected since we do not use randomisations and we do not consider such attacks in this work.

5 Leakage on the Complete Addition Formulas

We now analyse *hardware* implementations of the Montgomery Ladder based on the complete addition formulas from [28]. We provide a more compact analysis as for our previous case study, since the reasons for the leakage and its possible mitigation can be explained and proposed analogous. We will focus on an open source implementation from [27]. This design implements the point addition and doubling operations based on Algorithm 7 in [28] (for space reasons we do not include a description of the algorithm and refer the reader to the original paper). For point addition, the inputs to the algorithm correspond to the points we want to add (i.e. R_0 and R_1). For point doubling we provide the same point twice as input and perform thus $R_0 + R_0$ or $R_1 + R_1$. This implementation also relaxes the assumption that the most significant bit of the secret scalar is 1, and performs a loop iteration for each MSB of the scalar, even if it has the value 0. To this scope, the first register R_0 is initialised with coordinates corresponding to the point at infinity, i.e. $R_0 = (0, 1, 0)$, and the second register R_1 is initialised with coordinates corresponding to the input point P, i.e. $R_1 = (x_P, y_P, 1)$, (see Section IV D in [27]). However as we explain next, this way of executing the algorithm leads to the exact same side-channel vulnerability discussed so far in this paper.

We now describe what happens if the MSB of the scalar has a value of 0, with focus on the point doubling step, since this is the part of the algorithm where the most leakage will be visible. If the MSB of the scalar is equal to 0, we will perform a point doubling with values corresponding to the point at infinity. That is, both inputs to the algorithm will have the values $R_0 = (0, 1, 0)$. Consequently, the first 5 operations of the algorithm are executed as follows: $t_0 \leftarrow 0 \cdot 0; t_1 \leftarrow 1 \cdot 1; t_2 \leftarrow 0 \cdot 0; t_3 \leftarrow 0 + 1; t_4 \leftarrow 0 + 1$. We are thus performing multiplications and additions exclusively with operands equal to 0 and 1 in the beginning of the algorithm. Moreover, the registers t_0 and t_2 are overwritten with 0 and registers t_1, t_3 and t_4 are overwritten with a 1 which leads to a very large amount of leaky additions, subtractions and multiplications throughout the rest of the algorithm execution. Note that equally as for our previous case studies, such a leakage will be visible in the next loop iteration if the next bit of the scalar also has a value of 0, and the leakage will only be gone once we execute a loop iteration for a key bit with a value of 1. Figure 4 shows power traces from this implementation running first on a scalar whose initial bits are `0xFF` and then with a scalar whose initial bits are `0x08`, both times with the same input point. These power trace measurements confirm the presence of the leakage. We repeated the experiment in a similar setting but with random input points and also with enabled Z-coordinate randomisation. For all cases, the leakage was still very present.

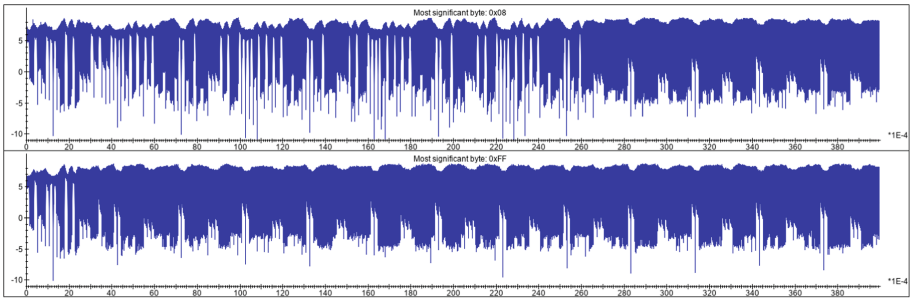

Fig. 4. Power profiles of the hardware implementation using complete addition formulas with scalars starting with `0x08` (above) and `0xFF` (below).

6 Conclusions and Future Work

In this paper we studied and verified an SCA leakage commonly found in ECC implementations. We studied the leakage on implementations of Curve25519 and proposed re-designs of its scalar multiplication algorithm with the goal of removing this leakage. We verified the effectiveness of our re-designs via experimental results in Sect. 4.

It remains to propose a complete re-design for the implementations using the complete addition formulas, which we studied in Sect. 5. To remove the leakage of these implementations, we can outline a similar re-design as the one presented for Curve25519. Namely, we can initialise all input registers with random values, and execute "dummy" loops for all MSBs with value equal 0. Once we reach the first key bit with value 1, we can perform a special variation of the loop iteration, where we plug-in pre-calculated values depending on the input point P. We can alternate between the two possible loop iterations by means of a final state machine, as usually done for VHDL designs. We leave a complete description of a re-design for our second case study (and its evaluation) as future work.

Acknowledgments. The work of Estuardo Alpirez Bock was in part supported by MATINE, Ministry of Defence of Finland.

References

1. Akishita, T., Takagi, T.: Zero-value point attacks on elliptic curve cryptosystem. In: Boyd, C., Mao, W. (eds.) ISC 2003. LNCS, vol. 2851, pp. 218–233. Springer, Heidelberg (2003). https://doi.org/10.1007/10958513_17
2. Andrikos, C., et al.: Location, location, location: revisiting modeling and exploitation for location-based side channel leakages. In: Galbraith, S.D., Moriai, S. (eds.) ASIACRYPT 2019. LNCS, vol. 11923, pp. 285–314. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34618-8_10
3. Aranha, D.F., Novaes, F.R., Takahashi, A., Tibouchi, M., Yarom, Y.: LadderLeak: breaking ECDSA with Less than One Bit of Nonce Leakage, pp. 225–242. Association for Computing Machinery, New York (2020)

4. Batina, L., Chmielewski, L., Haase, B., Samwel, N., Schwabe, P.: Sca-secure ECC in software - mission impossible? IACR Cryptol. ePrint Arch., p. 1003 (2021)
5. Batina, L., Chmielewski, L., Papachristodoulou, L., Schwabe, P., Tunstall, M.: Online template attacks. In: Meier, W., Mukhopadhyay, D. (eds.) INDOCRYPT 2014. LNCS, vol. 8885, pp. 21–36. Springer, Heidelberg (2014)
6. Becker, G.T., et al.: Test vector leakage assessment (TVLA) methodology in practice. In: International Cryptographic Module Conference (2013)
7. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Heidelberg (2006). https://doi.org/10.1007/11745853_14
8. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.-Y.: High-speed high-security signatures. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 124–142. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23951-9_9
9. Alpirez Bock, E., Dyka, Z., Langendoerfer, P.: Increasing the robustness of the montgomery kP-algorithm against SCA by modifying its initialization. In: Bica, I., Reyhanitabar, R. (eds.) SECITC 2016. LNCS, vol. 10006, pp. 167–178. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47238-6_12
10. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-69053-0_4
11. Boneh, D., Venkatesan, R.: Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In: Koblitz, N. (ed.) CRYPTO'96. LNCS, vol. 1109, pp. 129–142. Springer, Heidelberg (1996)
12. Bosma, W., Lenstra, H.W.: Complete system of two addition laws for elliptic curves. J. Number Theory (1995)
13. Castryck, W., Lange, T., Martindale, C., Panny, L., Renes, J.: CSIDH: an efficient post-quantum commutative group Action. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018. LNCS, vol. 11274, pp. 395–427. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03332-3_15
14. CryptoJedi. Micro salt: μnacl - the networking and cryptography library for microcontrollers. http://munacl.cryptojedi.org/curve25519-cortexm0.shtml
15. Düll, M., et al.: High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. Des. Codes Cryptogr. **77**(2–3), 493–514 (2015)
16. De Feo, L., et al.: Sike channels. Cryptology ePrint Archive, Paper 2022/054 (2022). https://eprint.iacr.org/2022/054
17. Genêt, A., Kaluđerović, N.: Single-trace clustering power analysis of the point-swapping procedure in the three point ladder of cortex-M4 SIKE. In: Balasch, J., O'Flynn, C. (eds.) COSADE 2022. LNCS, vol. 13211, pp. 164–192. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99766-3_8
18. Goodwill, G., Jun, B., Jaffe, J., Rohatgi, P.: A testing methodology for side-channel resistance validation, niat. Workshop record of the NIST Non-Invasive Attack Testing Workshop (2011). csrc.nist.gov/CSRC/media/Events/Non-Invasive-Attack-Testing-Workshop/documents/08Goodwill.pdf
19. Guntur, H., Ishii, J., Satoh, A.: Side-channel attack user reference architecture board sakura-g. In: 2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE), pp. 271–274 (2014)

20. Heyszl, J., Ibing, A., Mangard, S., De Santis, F., Sigl, G.: Clustering algorithms for non-profiled single-execution attacks on exponentiations. In: Francillon, A., Rohatgi, P. (eds.) CARDIS 2013. LNCS, vol. 8419, pp. 79–93. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08302-5_6
21. Jao, D., De Feo, L.: Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In: Yang, B.-Y. (ed.) PQCrypto 2011. LNCS, vol. 7071, pp. 19–34. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25405-5_2
22. López, J., Dahab, R.: Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 316–327. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48059-5_27
23. Medwed, M., Oswald, E.: Template attacks on ECDSA. In: Chung, K.-I., Sohn, K., Yung, M. (eds.) WISA 2008. LNCS, vol. 5379, pp. 14–27. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00306-6_2
24. Montgomery, P.L.: Speeding the pollard and elliptic curve methods of factorization. Math. Comput. **48**(177), 243–264 (1987)
25. Nascimento, E., Chmielewski, Ł: Applying horizontal clustering side-channel attacks on embedded ECC implementations. In: Eisenbarth, T., Teglia, Y. (eds.) CARDIS 2017. LNCS, vol. 10728, pp. 213–231. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75208-2_13
26. Nascimento, E., Chmielewski, L., Oswald, D.F., Schwabe, P.: Attacking embedded ECC implementations through cmov side channels. In: Selected Areas in Cryptography - SAC 2016–23rd International Conference, St. John's, NL, Canada, 10–12 August, 2016, Revised Selected Papers, pp. 99–119 (2016)
27. Pirotte, N., Vliegen, J., Batina, L., Mentens, N.: Design of a fully balanced ASIC coprocessor implementing complete addition formulas on weierstrass elliptic curves. In: 2018 21st Euromicro Conference on Digital System Design (DSD), pp. 545–552 (2018)
28. Renes, J., Costello, C., Batina, L.: Complete addition formulas for prime order elliptic curves. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9665, pp. 403–428. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49890-3_16
29. Certicom Research. Sec 2: Recommended elliptic curve domain parameters, version 2.0. www.secg.org/sec2-v2.pdf
30. Riscure. Current probe. security test tool for embedded devices (2018). www.riscure.com/product/current-probe/. Accessed 05 May 2021
31. Riscure. Side channel analysis security tools (2021). www.riscure.com/security-tools/inspector-sca/