Attacking embedded ECC implementations through cmov side channels

Erick Nascimento[1], Łukasz Chmielewski[2],
David Oswald[3], and Peter Schwabe[4] ⋆

[1] Institute of Computing, University of Campinas, Campinas, Brazil
enascimento.pub@gmail.com
[2] Riscure BV, Delft, The Netherlands
Chmielewski@riscure.com
[3] School of Computer Science, University of Birmingham, Birmingham, UK
d.f.oswald@cs.bham.ac.uk
[4] Digital Security Group, Radboud University, Nijmegen, The Netherlands
peter@cryptojedi.org

Abstract. Side-channel attacks against implementations of elliptic-curve cryptography have been extensively studied in the literature and a large tool-set of countermeasures is available to thwart different attacks in different contexts. The current state of the art in attacks and countermeasures is nicely summarized in multiple survey papers, the most recent one by Danger et al [21]. However, any combination of those countermeasures is ineffective against attacks that require only *a single trace* and directly target a conditional move (cmov) – an operation that is at the very foundation of all scalar-multiplication algorithms. This operation can either be implemented through arithmetic operations on registers or through various different approaches that all boil down to loading from or storing to a secret address. In this paper we demonstrate that such an attack is indeed possible for ECC software running on AVR ATmega microcontrollers, using a protected version of the popular μNaCl library as an example. For the targeted implementations, we are able to recover 99.6% of the key bits for the arithmetic approach and 95.3% of the key bits for the approach based on secret addresses, with confidence levels 76.1% and 78.8%, respectively. All publicly available ECC software for the AVR that we are aware of uses one of the two approaches and is thus in principle vulnerable to our attack.
Keywords: ECC, Montgomery ladder, power analysis, AVR, conditional move.

1 Introduction

For many years, efficient software implementations of cryptographic algorithms for constrained embedded processors were mainly restricted to symmetric ci-

⋆ This work was done while Erick Nascimento was visiting Radboud University. This work was supported by the Netherlands Organisation for Scientific Research (NWO) through Veni 2013 project 13114. Permanent ID of this document: bb3c834d7cc8ffbe7e7520f1c21bd408. Date: July 18, 2016

phers. However, in recent years, various libraries for elliptic curve cryptography (ECC) have been published that offer acceptable runtime and code size also on microcontrollers with limited computational resources, e.g., the 8-bit AVR ATmega series of processor. Notable examples for these ECC implementations are summarized in Table 1.

Name	Description	SCA countermeasures
micro-ecc [44]	8/32/64-bit C impl. for NIST curves	not documented; apparently randomized projective coordinates
nano-ecc [34]	Derivate of micro-ecc	same as micro-ecc
μNaCl [33,24,49]	Curve25519 for 8/16/32-bit processors	constant-time
AVR-Crypto-Lib [53]	ECDSA with NIST P-192	none
FLECC_IN_C [59]	8/16/32/64-bit C impl. for various curves	constant time, randomized projective coordinates
RELIC [2]	Various curves and fields supported	constant-time
WM-ECC [58]	Impl. for sensor networks	none
TinyECC [43]	Impl. for sensor networks	none
MIRACL [13]	Lib. supporting multiple curves	none
WolfSSL [60]	Support for AVR unclear	none
Wiselib [1]	Lib. for distributed systems	none
CRS ECC [56]	Commercial, closed source	none

Table 1: Overview of ECC implementations for embedded processors.

Due to the fact that an adversary often has physical access to an embedded device performing ECC operations, implementation attacks and in particular side-channel analysis (SCA) are severe threats in this scenario. Consequently, several libraries comprise countermeasures against SCA, for example, by performing computations in constant-time, or by using randomized projective coordinates. The protected implementations are further detailed in Table 1.

Many common SCA countermeasures assume that the adversary needs access to multiple traces (with identical scalar) to recover the secret key, which inherently protects protocols with ephemeral scalars. In this paper, we challenge this assumption and target fundamental building blocks of any ECC implementation, namely *conditional moves* and loads/stores from/to secret memory addresses. We show that template attacks allow to recover most of the secret scalar with a single trace of elliptic-curve scalar multiplication (ECSM) in both cases, which in turn renders all currently published ECC implementations for the AVR (and likely other, similar architectures) insecure.

Note that although this paper focuses on implementations of ECC, our attacks also apply to exponentiation algorithms as used in, e.g., RSA, classical

Diffie-Hellman, DSA, or ElGamal. We actually expect the attacks to work even better there, because group elements are larger and thus require more loads (or conditional moves). We leave this investigation for future work.

Related work. Carefully combining countermeasures like uniformity of modular operations, (re-)randomization of the projective representation of points, scalar blinding, point blinding, and random field (or curve) isomorphisms prevent classical side-channel attacks like timing [40], SPA [20], DPA [41], CPA [11] or collision attacks [26,32]. These attacks require a fixed scalar for multiple measured power or electromagnetic traces. The main protection relies on the full randomization of intermediate data, including input point, scalar and group, during the execution of an ECSM [19,4,25]. In this work we consider implementations based on the Montgomery ladder algorithm, protected by scalar randomization (SR) and projective-coordinate randomization[5].

To overcome the aforementioned countermeasures two kinds of attacks have emerged: template and horizontal attacks. Although in general template attacks [14] can be used to attack multiple traces that share the same scalar, we need to attack ECSM traces independently, because of the SR. Template attacks combine statistical modeling and power-analysis, and consist of two phases. In the first phase, called *profiling*, the attacker builds templates by executing a sequence of instructions using a fixed scalar (with SR turned off). The second phase is called *matching*, in which the attacker matches the templates to attacked single traces (with SR turned on). The assumption is that the attacker possesses a *profiling* device, in order to build templates, that behaves the same as the target device, and runs the same implementation.

Template attacks on ECC trace back to an attack on ECDSA demonstrated by Medwed and Oswald [45]. However, this attack requires an offline DPA on the ECSM during profiling, in order to select the points of interest. Moreover, since the attack exploits data-dependent leakage it requires profiling with multiple templates (i.e., 33) while for our attacks two templates are enough. Furthermore, the attack only needs to recover a few bits of the multiple ephemeral scalars and can then employ ECDSA-specific lattice techniques to recover the long-term secret key [10]. This is not possible in the context of our work, since we do not target ECDSA: an attacker has only a single trace to recover sufficiently many bits of the randomized scalar using SCA to be able to compute the remaining bits.

Another template attack on ECC is presented in [31]. This attack follows a similar approach to our attack, but instead of exploiting address-dependent leakage, it exploits register location based leakage using a high-resolution inductive EM probe. As a result the attack is considerably expensive to execute. A template attack on a wNAF ECC algorithm is presented in [61]. However, this attack is applied to an implementation that is not protected with either,

[5] The implementations actually attacked apply only projective coordinates randomization, however, our attack also works on an implementation with SR enabled, because we do not make any assumption about the secret scalar, i.e., it may be different from one execution to another.

scalar randomization or base-point randomization. Another approach to attack ECC are the so called online template attacks [5,23]. These attacks work if SR is enabled, but not when point randomization is enabled.

The template attack from [16] targets load instructions. However, multiple traces are required in the attack phase. Therefore, this attack does not work against implementations protected by SR. The template attack from [29] aims to extract a random multiplicative mask (base-blinding) out of a single measurement exploiting data leakage; then it is possible to unmask all intermediate values and run DPA.

Horizontal attacks on RSA [57,18,17,8,6,30,55,15,9,54] and ECC [7,28] are emerging forms of side-channel attacks on exponentiation-based or scalar-multiplication-based algorithms. Their methodology allows recovering the exponent bits through the analysis of individual traces. Therefore, these attacks are efficient against SR even when combined with point and group randomization. The attacks employ different common distinguishers: SPA, horizontal correlation analysis [18], Euclidean distance [57], horizontal collision-correlation [17,8,6,7], horizontal cross-correlation [28], or clustering [30,55].

An interesting horizontal address-based DPA attack on Montgomery multiplications is presented in [15]. The approach is similar to ours, but this attack exploits Hamming weight leakage of addresses. Furthermore, the analysis in [15] lacks the results for a full modular exponentiation (only a few iterations are attacked) and success rates.

The main issue of horizontal attacks is that extracting leakage from a single unlabeled trace is usually heavily limited by noise. Therefore, we have decided to attack our state-of-the art implementations, that contains scalar and point randomizations, using a more powerful attack paradigm, from the point of view of the attacker setting, namely, template attacks.

Contributions. The main contributions of this paper are threefold:

1. First, by the example of a protected version of μNaCl, we show that the single-trace leakage of conditional moves within the Montgomery ladder can be exploited to recover the scalar.
2. Second, we show that a similar attack applies to loads and stores from/to secret-dependent addresses. In doing so, we show that even implementations on embedded devices *without* cache cannot tolerate secret-dependent memory accesses.
3. Finally, we generalize the method from [27] to tolerate a certain number of incorrectly recovered scalar bits without relying on normal or side-channel-enhanced exhaustive search. Furthermore, we present experimental results for our algorithm.

Organization of the paper. The remainder of this paper is structured as follows: in Section 2, we review the use of conditional moves in scalar multiplication algorithms, together with possible countermeasures against side-channel analysis. Then, in Section 3, we describe the measurement setup and target

implementation used for our attacks presented subsequently: while Section 4 deals with template attacks on the (arithmetic) conditional swap within the Montgomery ladder, Section 5 applies similar methods to recover the scalar by exploiting the leakage of secret load addresses. Section 6 discusses how to tolerate a certain number of incorrectly recovered scalar bits more efficiently than by simple exhaustive search. Finally, we conclude in Section 7 with directions for future work, in particular regarding countermeasures.

2 Scalar multiplication and conditional moves

The most basic scalar-multiplication algorithm is the double-and-add algorithm, which scans through the bits of the scalar and performs a double operation for each zero bit and a double-and-add operation for each one bit. This algorithm is well known to be vulnerable to all kind of side-channel attacks, including power analysis and timing attacks.

The first step to side-channel protection is to always perform the same sequence of finite-field operations, independent of the scalar. The most common approaches to achieve such a structure are either to use (fixed-window) double-and-add-always scalar multiplication or ladder-based approaches (typically the Montgomery ladder [46] or, for general Weierstrass curves, the Brier-Joye ladder [12]). Another layer of side-channel protection then adds randomization of the scalar (through one of various blinding methods), and the internal representation of points (for example through projective randomization, field isomorphisms, or curve isomorphisms). By re-randomizing before or after each ECSM loop iteration, most horizontal collision or cross-correlation attacks are thwarted.

Interestingly, even with all those countermeasures in place, scalar-multiplication algorithms contain operations that *choose one out of two (or more) curve points* depending on bit(s) of the scalar. An attacker who learns all of these choices from side-channel information from just one trace, learns all of the scalar bits used in this scalar multiplication and thus obtains the secret key. On microcontrollers with restricted register space, there are essentially two different ways to implement this *conditional move* (cmov): either by loading from (or storing to) addresses that depend on the secret scalar, or by using arithmetic operations to perform a conditional register-to-register move. The latter approach is very common on large processors with cache, where the former approach leaks through cache-timing information. Essentially, the idea is to replace a computation of the form $R \leftarrow P[s]$, where s is a secret scalar bit, by a computation of the form $R \leftarrow sP[1] + (1-s)P[0]$. Note that this approach does not require actual multiplications; it is much easier to expand s to a bit mask of all ones or all zeros and use bit-logical instructions.

Most implementations of ECSM contain considerably more than just one secretly-indexed load, store, or conditional move. Sometimes this is a choice made by the implementors to improve performance (by avoiding otherwise unnecessary loads and stores); sometimes it is an inherent property of the ECSM algorithm. For example, the Montgomery ladder needs a conditional swap (cswap) of two

points instead of a conditional move, which requires significantly more operations that involve the secret scalar bit than a simple cmov (for details, see Section 4).

The side-channel attacks described in the remainder of this paper attack both implementations that make use of secretly indexed memory accesses (in Section 5) and implementations that use the arithmetic cmov operation (or more specifically, the cswap operation) in Section 4. The idea of attacking loads from secret positions through side-channel information is not new: it is not only used in various cache-timing attacks (that do not apply to simple architectures such as the AVR), but it is also the underlying principle of address-bit-DPA [35]. What is novel is the fact that we need only a single trace. This renders countermeasures such as scalar blinding and address randomization [36,38] ineffective.

3 Attack setup

In this section, we describe the targeted implementations, the utilized microcontroller, our measurement setup. The trace pre-processing, frequency filtering and alignement, are described in Appendix B.

3.1 Target implementations

We target two protected ECSM implementations based on [49]. Both employ the Montgomery ladder, with the pseudocode given in Algorithm 1. The main difference between the two variants is the realization of the cmov (i.e., the function CSWAP__COORDS): The first implementation, described in more detail in Section 4.1, consists of applying an arithmetic conditional swap of the respective coordinates values of the working points $P_1 = (X_1 : Z_1)$ and $P_2 = (X_2 : Z_2)$. The second, described in Section 5.1, replaces the arithmetic conditional swap by a conditional swap of pointers to the coordinate values. Both implementations utilize projective-coordinate re-randomization as the main side-channel countermeasure. A randomly generated $\lambda \in \mathbb{F}_p$ is multiplied with the coordinates of $P_1 = (X_1 : Z_1)$ and $P_2 = (X_2 : Z_2)$ at the beginning of every ECSM iteration. We make publicly available the source code for both implementations [48].

3.2 Target device and measurement setup

We carried out our experiments with an ATmega328P 8-bit microcontroller placed on the target board of the ChipWhisperer [51] side-channel evaluation platform. While the ChipWhisperer also provides the possibility to capture analog signals (e.g., power consumption or electro-magnetic emanation), we used a separate oscilloscope (Picoscope 5203) due to the limited bandwidth, memory, and sample rate of the ChipWhisperer.

The targeted ATmega328P has a 32 KB of Flash, 2 KB of SRAM, and 1 KB of EEPROM. The register file contains 32 registers (R0–R31), among which 6 serve as pointers for indirect 16-bit addressing and have the following aliases: X (R27:R26), Y (R29:R28) and Z (R31:R30). Arithmetic instructions take 1 cycle,

Algorithm 1 Montgomery ladder with arithmetic cswap and randomized projective coordinates.

```
// ... initialization omitted ..
bprev ← 0
for i = 254...0 do
    RE__RANDOMIZE__COORDS(work)
    b ← bit i of scalar
    s ← b ⊕ bprev
    bprev ← b
    CSWAP__COORDS(work, s)
    LADDERSTEP(work)
end for
```

with the exception of multiplication instructions, which take 2 cycles. Loads and stores from/to SRAM take 2 cycles. Loads from Flash take 3 cycles. More technical details about the target device are given in Appendix A.

4 Attacking arithmetic cswaps

In this section, we describe a template attack on conditional swaps (cswaps) in the Montgomery ladder step. In our case, the cswap is implemented using Boolean and arithmetic operations in constant time.

4.1 Target implementation

In the Montgomery ladder (Algorithm 1), the function CSWAP__COORDS implements the cswap (based on input bit s) by first creating a mask m, which is either 0x00 or 0xFF for $s = 0$ and $s = 1$, respectively, by setting $m = -s$ (assuming m, s are 8-bit values). Then, a (conditional) XOR swap is executed as follows:

Listing 1.1: Conditional XOR swap.

```
1 ld xx, X    ; X register points to first value
2 ld yy, Z    ; Z register points to second value
3 mov tt, xx
4 eor tt, yy
5 and tt, m   ; tt = (xx XOR yy) AND m
6 eor xx, tt  ; xx = xx XOR tt
7 eor yy, tt  ; yy = yy XOR tt
8 st X+, xx   ; Store first value
9 st Z+, yy   ; Store second value
```

In other words, if $m = 0x00$ ($s = 0$), $tt = 0$ and the XORs xx = xx ⊕ tt and yy = yy ⊕ tt leave the values unchanged. Otherwise, if $m = 0xFF$ ($s = 1$), we have a standard XOR swap, i.e., xx = xx ⊕ xx ⊕ yy = yy (equivalent for yy).

4.2 Template generation and matching

We generated templates for the `and` instruction (line 5 of Listing 1.1), grouping the traces in the profiling set into two sets V_0 and V_1. Traces in V_0 represent those where $m = 0$ (i.e., an `AND` with 0x00), while V_1 are traces where $m =$ 0xFF. Note that the traces were cut to only contain the clock cycle for the targeted `and` instruction, i.e., each trace is $64 \cdot 67 = 4288$ samples long (cf. Appendix B). For V_i, $i = 0, 1$, we subsequently computed templates consisting of the pointwise mean vector $\boldsymbol{\mu}^{(i)}$ and the covariance matrix $\boldsymbol{\Sigma}^{(i)}$ [14]. Note that the two possible leakages 0x00 (all bits zero) and 0xFF (all bits one) can be expected to be maximally (or at least to a large degree) different, which should facilitate template attacks in this particular case.

We matched the templates to the traces in the test set with the standard approach, i.e., computing the respective probabilities using the multivariate normal distribution pdf and identifying the template with the highest probability to recover the respective bit of the scalar. The respective success rates wrt the size of the profiling set are given in Section 4.3.

Classification. For each template we computed the Euclidean distance between the sample vector and the template mean vector. The template (T_0 or T_1) that results in the smallest distance is considered the best match for the sample vector. In this attack, the index of the closest template (0 or 1) corresponds to the `swap` bit.

Confidence score and confidence level. For the first classification method we derived a simple confidence score on the recovered bit value based on the distances (d_0 and d_1) to each template. It varies linearly for a particular $d_0 + d_1$ value, ranging from 0 (no confidence) and 1 (full confidence):

$$\text{conf\_score} = 2 \cdot \left| 0.5 - \frac{min(d_0, d_1)}{d_0 + d_1} \right| \tag{1}$$

We furthermore define the *confidence level* of a given trace (in the test set) as follows: Let us call a recovered bit *suspicious* if its confidence level is less than the greatest confidence score of any falsely identified bit (whereas this threshold is determined experimentally in the profiling phase). Then, the confidence level is the percentage of bits that are not suspicious, i.e., that can be unambiguously recovered. Note that the average confidence level (over all number of traces in the test set) is always less than or equal to the average success rate, since an incorrectly recovered bit is always suspicious.

4.3 Attack results

Figure 1 shows the average and best case success rates (computed over all 255 scalar bits), together with the respective confidence levels over the number of traces used for template generation and matching. Note that each full trace comprises 255 ECSM iterations, which were all used for generating the

templates – in other words, each full trace contributes 255 "effective" traces to the profiling set.

The traces used for template generation and matching were taken from different trace sets (coming from different capture sessions). The same number of traces was used for profiling and testing, i.e., a given value on the horizontal axis of Figure 1 is the same for profiling and testing.

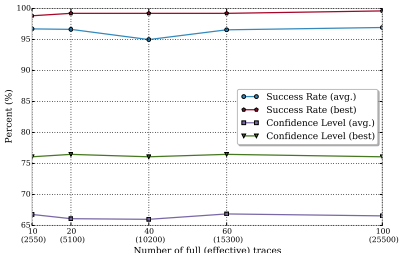

Fig. 1: Success rates for the template attack on cswap for different number of full traces.

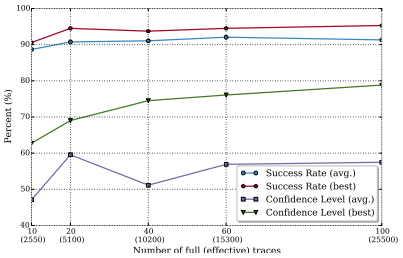

Fig. 2: Results for the template attack on loads/stores for different number of full traces.

As evident in Figure 1, already for 10 full traces (i.e., about 2,550 effective traces), the average success rate reaches 96.71%, i.e., we can recover most of the bits of the scalar. Furthermore, the best success rate reaches 99.6% with the confidence level 76.1%. By increasing the number of traces, both success rate and confidence level change only minimally; due to the strong leakage of the targeted device, most information can be already extracted with a low trace count.

5 Attacking secret-dependent memory accesses

In general, ECC (and in particular NaCl-derived) implementations avoid loads from secret-dependent addresses altogether due to the possibility of cache-timing attacks. However, for embedded implementations without caches, secret load addresses are sometimes deemed acceptable. In this section, we show that template attacks can be employed to exploit this leakage.

5.1 Target implementation

The targeted implementation replaces the cswap of the $(X_1 : Z_1)$ and $(X_2 : Z_2)$ coordinates values used in the targeted implementation in Algorithm 1 by working with pointers to those coordinates, and conditionally swapping these pointers. Besides being slightly faster, this implementation also potentially exhibits less leakage, because it uses the secret-dependent mask m in an `AND` operation

only twice for each pointer cswap[6], rather than 32 times as in the ECSM implementation based on arithmetic cswap (cf. Section 4.1).

However, in implementations of finite-field operations both input and output operands are pointers. The values of these pointers are addresses to the memory holding the actual field element value, and those addresses directly depend on whether the swap occurred or not, which in turn depends on the value of the secret mask bit.

AVR memory access instructions internals. Memory access instructions (loads and stores) on an AVR take 2 clock cycles to execute. According to the ATmega328 datasheet [3], the effective address for such instructions is computed in the first cycle, while during the second cycle, the data word is read (load) or written (store) if the effective address is valid. Our proposed attack focuses on the address leakage of memory access instructions, and thus any data-dependency may negatively impact the attack success rate if not detected and mitigated. Therefore, we take advantage of this architectural feature by using only the samples from the first clock period of such instructions.

Targeted loads and stores. During each iteration of the Montgomery ladder, the actual field arithmetic occurs in the so-called LADDERSTEP function (cf. Algorithm 1). We target the loads and stores addresses in the first three field operations in LADDERSTEP, i.e., addition, subtraction, and addition. Each of these operations has two $\mathbb{F}_p$ inputs (a and b) and one output r.

Finite-field addition and subtraction are implemented with reduction modulo $2^{256} - 38$. The reduction step also execute loads and stores, of which the samples are also used for template creation and matching. Listing 1.2 shows a small segment of the execution trace containing the loads of the first operands bytes and the store of the first byte of the result (before reduction):

Listing 1.2: Segment of the execution trace for a field addition.

```
1 0x171a: fp_add+0x5   LD R20, X+          ; first byte of a
2 0x171a: fp_add+0x5   CPU-waitstate
3 0x171c: fp_add+0x6   LD R21, Y+          ; first byte of b
4 0x171c: fp_add+0x6   CPU-waitstate
5 0x171e: fp_add+0x7   ADD R20, R21
6 0x1720: fp_add+0x8   ST Z+, R20          ; first byte of r
7 0x1720: fp_add+0x8   CPU-waitstate
```

Our oscilloscope's memory is divided into 255 segments, each of which is 65 kSample in length. A memory segment holds the samples captured from a single ECSM iteration. Due to the 65 kSample limit for each ECSM iteration, we were able to capture the samples from all the loads and stores from the first field addition and the first field subtraction, but only half of the loads and stores from the arithmetic part of the second field addition. Note that the memory limitation

[6] For the AVR architecture, pointers are 16 bit wide and one `AND` with the secret-dependent bit is required to cswap a byte. Thus a pointer cswap requires two `AND`s.

is due to the relatively low-cost oscilloscope we used—high-end equipment would further facilitate the presented attack.

Table 2 shows the number of executed instructions of each type that are used in the attack. We used a total of 372 instructions, which are concatenated into a single sample vector. After trace preprocessing, 67 power samples are available per clock cycle, and as only the first clock period of a memory access instruction is used, the sample vector per ECSM iteration has $n_v = 24,924$ samples.

Table 2: Number of executed instructions of each type that are used in the attack.

Type	1st `fp_add`	`fp_sub`	2nd `fp_add`	Total
`LD R20, X+`	32	32	16	80
`LD R21, Y+`	32	32	16	80
`LD R20, Z+0`	33	33	0	66
`ST Z+, R20`	65	65	16	146

5.2 Template generation

Each load or store instruction accesses at most two possible addresses. If it always accesses the same address, then it does not provide useful leakage relevant for the attack. Considering only those loads and stores that may access two addresses, during any execution of the LADDERSTEP, only two distinct sequences of addresses can be accessed: A_{noswap}, containing the addresses accessed before the first pointers swap has taken place[7], i.e., an even state (`noswap` state); and A_{swap} containing the addresses accessed in an odd state (`swap` state).

First, we grouped the sample vectors into two sets. The first set, V_0, consists of the load/store sample vectors for addresses in the set A_{noswap}, while the second set, V_1, contains those originating from addresses in set A_{swap}. Then, we computed various statistics for each sample index of V_i, $i = 0, 1$: mean $\boldsymbol{\mu}^{(i)}$, standard deviation $\boldsymbol{\sigma}^{(i)}$, median $\boldsymbol{md}^{(i)}$, as well as lower $\boldsymbol{l}^{(i)}$ and upper $\boldsymbol{u}^{(i)}$ percentiles (the actual percentiles used are discussed in 5.3). The collection of these statistics for V_0 and V_1, called T_0 and T_1, are the two possible templates.

5.3 Point-of-interest selection

The POI selection consists of using the lower and upper percentile vectors $\boldsymbol{l}^{(i)}$ and $\boldsymbol{u}^{(i)}$ (i=0,1) to compute the intersection of the pair of intervals $[l_j^{(0)}, u_j^{(0)}]$ and $[l_j^{(1)}, u_j^{(1)}]$ for each sample index $j = 1, \ldots, n_v$. The sample indices where the intersection is empty are the considered POIs.

[7] These addresses are the same as those accessed after the 2nd but before the 3rd swap, or after the 4th but before the 5th swap, and so on.

Intuitively, the sample indices with an empty intersection are those that are good distinguishers for the two templates, because in these points the samples tend to be clustered around the median (and also typically around the mean) of one template, rather than being scattered.

Different values for the lower and upper percentiles may give a different number of POIs, and that directly affects the success rate and confidence level of the attack. Thus, we tested the attack for different pairs of values for these parameters, ranging from wider and more selective percentiles $(12.5, 87.5)$[8] to narrow, less selective $(40, 60)$.We emphasize that the POI selection is completely based on the samples of the traces used for the generation—it does not depend on the samples of the trace being attacked (i.e., the sample vector to classify). In fact, the POIs are represented as a Boolean vector used during template matching to select the samples from the target trace vector to be classified.

POI selection refinements. To improve the confidence level of the attack, we tested two POI selection refinements, as explained above. First, we noticed that when using more selective percentile parameters, the current selection method returned sample indices that were clustered in a few instructions, while most of the remaining instructions were not covered by any sample, although they should in theory contribute some leakage. To make the POIs more evenly distributed and exploit leakage from all useful instructions, we forced a minimum of one sample index per instruction to be included in the POI vector. If there was no sample index for a given instruction in the current POI vector, one was randomly selected. Second, also due to the clustering of the POIs in a few instructions, we limit the number of samples per instruction to one. In the case that sample indices had to be removed, we selected those randomly as well.

5.4 Template matching

At first, without using any POI selection, we tried to use the standard multivariate Gaussian model, taking advantage of both the mean vector and covariance matrix computed from V_0 and V_1 (also known as *complete templates*) similar to the approach of Section 4. However, in contrast to Section 4, the sample vectors to be classified and the mean template vectors are relatively long $(24,924$ samples) and relatively similar to each other (i.e., their Euclidean distance is very small), numerical instability issues due to almost singular matrices arose during the computation of the probability density function. For those reasons, we decided to use *reduced templates* instead, which uses only the mean vectors.

After applying POI selection, the matched sample vectors are much smaller, and thus full templates could then in principle be applied, as the covariance matrices would not lead to numerical instability. However, due to the high success rates achieved using the reduced templates, we decided to not use full templates to avoid increasing storage and computational requirements.

We also evaluated the effect on the attack success rate and confidence level of compressing the sample vector using normal and absolute sum for different

[8] I.e., the lower is the 12.5-percentile and the upper is the 87.5-percentile.

window lengths. In addition, we applied a straightforward outlier detection to remove samples that have likely been subject to larger distortions: In the matching phase, we discarded all samples that have a distance of more than a multiple of standard deviations to the mean trace at the respective point in time. Using reduced templates, template matching boils down to computing the (squared) Euclidean distance between the sample vector to match and the template mean vectors. The lower that distance is, the stronger is the match. In this case, other distinguishers can be used in a straightforward way, and thus we also tested the attack using the Pearson correlation coefficient.

Classification methods and confidence score. As a first classification method to test, we selected the template closer to the sample vector (cf. Section 4.2). We also tested majority voting classification, where each sample is individually classified, also based on its distance to the corresponding element in the templates mean vectors, and the majority vote wins. In both cases, as each template directly corresponds to a scalar bit value, the classification output is the recovered bit value. The confidence score was computed in the same way as in Section 4.2.

5.5 Attack results

Figure 2 depicts average and best case success rates for the template attack on secret-dependent memory accesses for the best and average cases. Again, as in Section 4.3, the trace sets used for template generation and matching were recorded in different capture sessions, and the same number of traces was used for each set. Again, only a limited number of profiling traces was sufficient to reach success rates exceeding 90%; the best success rate reaches 95.3% (there are only 12 errors) with the confidence level 78.8% (the 12 errors are included in the 54 suspicious bits). To investigate the effect of various pre-processing steps and attack parameters, using 10 traces we investigated the average success rate and confidence level depending on various attack parameters. In particular, we investigated various signal frequency filtering options, POI selection methods, classification and compression methods, outlier filtering, and distinguishers; the result of the investigation are described in Appendix B. The best parameters that we discovered, were used to perform the main attack described in this section.

6 Error detection and correction

Due to noise, data leakage (note that we are aiming at exploiting the address leakage only), and other aspects that interfere with the side-channel analysis (misalignment, clock jitter, etc), the derivation of the final scalar for a single trace likely contains errors. If the amount of wrong bits is sufficiently small, then a brute-force attack may still be feasible. However, first the attacker needs a metric to indicate the location of the possible wrong bits in the recovered scalar. The notion of suspicious bits (cf. Section 4.2) can be used as a reference for the scalar bits selection with respect to a brute-force attack.

Let us consider the trace with smallest amount of suspicious bits from the experiment from Section 5; for this trace there are 54 suspicious bits that comprise all falsely identified bits. Unfortunately, to recover a full randomized scalar, even in this case, the attacker needs $O(2^{54})$ operations, which is generally impractical. Note, that we consider only the worst-case complexity and not the average case.

To improve the brute-force complexity, there are two options. The first approach is to try to exploit the distribution of suspicious bits for incorrectly (red) and correctly (blue) recovered bits (Figure 3). While there is a clear trend for incorrect bits to have lower confidence score, the intersection between correct and incorrect bits is large. Still, it may possible to exploit the trend with an informed brute force attack [42], prioritizing bits with the lowest confidence score. Unfortunately this attack works well if the bits containing errors are adjacent and that is not the case in our setting.

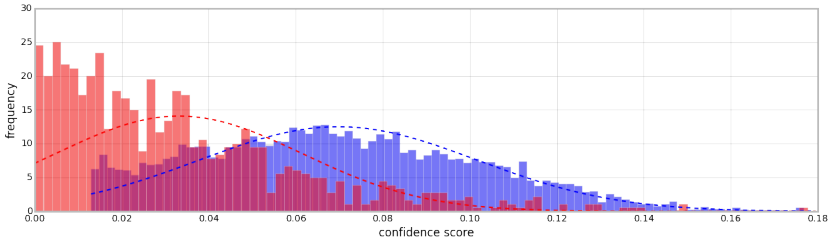

Fig. 3: Distribution of confidence scores over all traces for suspicious bits. Red: incorrectly recovered bits, blue: correctly recovered but suspicious bits.

Alternatively (or combined with the informed brute-force search), we apply the second algorithm from [27], which is originally designed for *square-and-multiply chains*, to the Montgomery ladder. We describe how the algorithm works using the aforementioned example trace, which contains $s = 54$ suspicious bits, as an example. Let us represent the indices of these bits as a list sorted in descending order: $i_s, \ldots i_1$, where each $i_j \in \{0, \ldots 254\}$ and $s \geq j \geq 1$; note that there are 255 bits in total. Let x denote the bit index $i_{\lfloor \frac{s}{2}+1 \rfloor}$ (namely, i_{28} for the example trace). Let a be the number represented by the bit string corresponding to the left part of the scalar from x (including i_x) and let b be the number corresponding to the bit string of the (least significant) right part. Furthermore, we know that $R = [k]P$, where R is the resulting point, k the scalar to be recovered, and P the input point. Then, clearly $R = [k]P = [a \cdot 2^{i_x} + b]P = [a]([2^{i_x}]P) + [b]P$. If we denote $[2^{i_x}]P$ by H, then the above equation reduces to

$$R - [b]P = [a]H \tag{2}$$

We can use Equation 2 to check correctness of our guess. Now, following [27], we use a time-memory trade-off technique to speed up an exhaustive search: Consider all different possible guesses for a. For each guess, we compute $[a]H$

and store all pairs $(a, [a]H)$. We then sort all pairs based on the value of $[a]H$ and store them in an ordered table.

Next, we make a guess for b and compute $z = R - [b]P$. If our guess for b is correct, then z is present in the second column of some row in the table we built—the first column is the corresponding a. Finding such a pair can be done using binary search, as the table is sorted as per the second column. If z is present, we are done since we have determined the scalar. Otherwise, we make a new, different guess for b and continue. Since there are approximately $2^{\frac{s}{2}}$ guesses for a and b, the time complexity is $O(2^{\frac{s}{2}})$ operations. As there are $2^{\frac{s}{2}}$ guesses for a, the table has that many entries and the space complexity is $O(2^{\frac{s}{2}})$ points. This way, we limit the time complexity to $O(2^{\frac{s}{2}})$ (cf. [27] for a detailed complexity analysis), which is $O(2^{27})$ for the example trace.

We do not know which trace contains the smallest number of suspicious bits since we do not know the maximum confidence score of a falsely identified bit. However, to use the above algorithm we assume that we know the number of suspicious bits to be bruteforced to recover the correct scalar. This can be determined by using templates to attack some traces, for which we know the randomized key. Furthermore, note that if the attack fails, we can extend the execution to the second most likely suspicious bit and reuse the previously obtained data. Based on our experiments, we determined that the number 54 of suspicious bits should cover all falsely identified bits for at least one trace. Our complete attack works as follows: in parallel, we run the above algorithm for each of the n traces. We stop the attack as soon as the time-memory trade-off technique succeeds for one trace.

Since we are running the attack n times in parallel the complexity of the complete attack is multiplied by n. It totals to $O(n \cdot 2^{\frac{s}{2}})$ operations and $O(n \cdot 2^{\frac{s}{2}})$ points in memory. For the attack from the previous section, this corresponds to $O\left(100 \cdot 2^{27}\right) = O\left(2^{32}\right)$ operations. Therefore, We conclude that we can efficiently recover the scalar successfully even in the presence of multiple errors and uncertain bits (for experimental results see section 6.1). Furthermore, we believe that the above technique may be of independent interest since it can be applied to a commonly used ECSM algorithm, i.e., Montgomery ladder, even if errors are spread independently in the scalar.

6.1 Algorithm implementation and experimental results

The first challenge we faced is how to compute the point subtraction in Equation 2. Curve25519 is a curve in the Montgomery form, and as such, there is an efficient formula for differential point addition using XZ coordinates, but no efficient formula to compute a standard point addition, as far as we know. For that reason, we decided to do the point addition in affine coordinates, which costs a field inversion and a few multiplications. However, to use them we need to know the y-coordinates $y(R)$ and $y([b]P)$. The attack assumes that $x(R)$ (the ECSM output) is known, but $y(R)$ is not, and thus has to be computed. To do so, we use the curve formula directly to compute the two possible values for $y(R)$, at the cost of a field square root, an expensive operation, but it has to be

done only once for each value of R. In the case of $y([b]P)$, an efficient algorithm by Okeya and Sakurai [52] costs one field inversion.

To generate the table of precomputed points $A = [a]H$ and to compute $B = [b]P$ in eq. (2), the naive approach is to compute a full ECSM for each value of a and b. A more efficient method is to apply Gray coding to the suspicious bits in scalars a and b. One property of such a code is that consecutive code words differ in just a single bit, which means that, in our context, we can generate $[k']P$ from $[k]P$ using a single point addition (if the bit changed from 0 to 1) or point subtraction (if the change is from 1 to 0), where k and k' are scalars whose unknown bits are represented as Gray code words, and the code word in k' is the successor of the respective code word in k. To compute the sequence of points $[k_i]P$ $(i = 0, 1...)$, we first construct the scalar k_0, by setting the unknown bits to zero and the (assumed correct) recovered bits from the output of the SCA attack to their respective values. Then, we apply the full ECSM algorithm to compute $[k_0]P$, and from there we use the aforementioned method to generate the sequence of points $[k_1]P, [k_2]P \ldots$, which costs essentially a point addition per each computed point.

We implemented the key recovery algorithm with the aforementioned arithmetic-level optimizations as a single-threaded program. We tested our implementation in a smaller scale, to recover 40 suspicious bits of a scalar on a PC with 8GB of RAM total, but only 5GB available for the program, a i7-3740QM CPU, running at 2.7GHz. It took 1h23 to recover the correct scalar, where about 1.5ms is spent to add a single entry to the table and about 3ms to test a possible value of b. By using these time values as a reference, we estimate that the time for the recovery of a scalar with 60 suspicious bits using the current implementation is around 18 days. The source code of the key recovery implementation is publicly available [47].

7 Conclusions and Possible Countermeasures

In this paper we show that the single-trace leakage of conditional moves can be exploited to recover the scalar using a template attack. We also show that a similar attack applies to loads and stores from/to secret-dependent addresses. Furthermore, we generalize the method from [27] to tolerate a certain number of incorrectly recovered scalar bits without relying on normal exhaustive search.

Now we discuss possible countermeasures against our attack. We consider evaluating or improving our attack to work against these countermeasures as future work. First of all, note that any countermeasure based on modifying the base point before or during the scalar multiplication does not protect against our attacks, since they aim at exploiting address-dependent and the cswap leakage. Similarly, scalar blinding or splitting does not affect the attack, since we require only one trace and could hence recover the blinded or split scalar. The knowledge of the randomized scalar (or the split scalars) is sufficient to either recover the original scalar or to compute the correct scalar multiplication result. A potential countermeasure against our attack is presented in [50], perform-

ing online data randomization during the exponentiation to prevent horizontal collision-correlation attacks. The main idea is to the split scalar to two parts and to randomly interleave two scalar multiplications. However, we believe that our attack might still be mounted if four templates are used recognize which bit is processed and during which ECSM.

The idea behind a another memory-address countermeasure [35] is to store sensitive variables at a memory address with the same Hamming weight for the two different addresses. We believe that although this would cause our attack to be less effective, the addresses leakage may still be identified by template matching. Randomization of memory addresses of the coordinates used in the Montgomery ladder before the ECSM might lead to our attack being less effective, since the templates are prepared assuming fixed addresses. The above countermeasure can be improved by randomizing not only the addresses but also the memory accesses [37,39,38].

The countermeasure of [31] protects against localized EM template attacks on the ECC Montgomery ladder. The main idea is to randomly swap the ladder registers at the end of a ladder iteration; the addressing of the registers within the loop is inverted according to whether the registers have been swapped. The countermeasure is uniform in its operation sequence, and hence, our template attacks would be infeasible in principle. In addition, several randomized techniques protecting the Montgomery ladder are presented in [22]. Similarly to the countermeasure of [31], these techniques generate operation sequences independent from the scalar. Thus we assume that our attack would be less effective or ineffective against these countermeasures. We therefore regard as future work evaluating and improving our attacks with respect to the three latter countermeasures.

References

1. D. Amaxilatis. A generic algorithms library for heterogeneous, distributed, embedded systems. https://github.com/ibr-alg/wiselib. 2
2. D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic. 2
3. Atmel. Atmega328P datasheet. http://www.atmel.com/devices/atmega328p.aspx, 2016. 10
4. J.-C. Bajard, L. Imbert, P.-Y. Liardet, and Y. Teglia. Leak resistant arithmetic. *Cryptographic Hardware and Embedded Systems – CHES 2004*, vol. 3156 of *LNCS*, 62–75. Springer, 2004. 3
5. L. Batina, Ł. Chmielewski, L. Papachristodoulou, P. Schwabe, and M. Tunstall. Online template attacks. *Progress in Cryptology – INDOCRYPT 2014*, vol. 8885 of *LNCS*, 21–36. Springer, 2014. 4
6. A. Bauer and É. Jaulmes. Correlation analysis against protected SFM implementations of RSA. *Progress in Cryptology – INDOCRYPT 2013*, vol. 8250 of *LNCS*, 98–115. Springer, 2013. 4
7. A. Bauer, É. Jaulmes, E. Prouff, J. Reinhard, and J. Wild. Horizontal collision correlation attack on elliptic curves – extended version –. *Cryptography and Communications*, 7:91–119, 2015. 4

8. A. Bauer, E. Jaulmes, E. Prouff, and J. Wild. Horizontal and vertical side-channel attacks against secure rsa implementations. *Topics in Cryptology – CT-RSA 2013*, vol. 7779 of *LNCS*, 1–17. Springer, 2013. 4
9. S. Bauer. Attacking exponent blinding in RSA without CRT. *Constructive Side-Channel Analysis and Secure Design*, vol. 7275 of *LNCS*, 82–88. Springer, 2012. 4
10. N. Bengeri, J. Pol, N. P. Smart, and Y. Yarom. "ooh aah... just a little bit" : A small amount of side channel can go a long way. *Cryptographic Hardware and Embedded Systems – CHES 2014*, vol. 8731 of *LNCS*, 75–92. Springer, 2014. 3
11. E. Brier, C. Clavier, and F. Olivier. Correlation power analysis with a leakage model. *Cryptographic Hardware and Embedded Systems – CHES 2004*, vol. 3156 of *LNCS*, 16–29. Springer, 2004. 3
12. E. Brier and M. Joye. Weierstraß elliptic curves and side-channel attacks. *Public Key Cryptography*, vol. 2274 of *LNCS*, 335–345. Springer, 2002. 5
13. CertiVox. MIRACL Cryptographic SDK. https://github.com/CertiVox/MIRACL. 2
14. S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. *Cryptographic Hardware and Embedded Systems – CHES 2002*, vol. 2523 of *LNCS*, 13–28. Springer, 2003. 3, 8
15. C.-N. Chen. Memory address side-channel analysis on exponentiation. *Information Security and Cryptology – ICISC 2014*, vol. 8949 of *LNCS*, 421–432. Springer, 2015. 4
16. O. Choudary and M. G. Kuhn. Efficient template attacks. *Smart Card Research and Advanced Applications*, vol. 8419 of *LNCS*, 253–270. Springer, 2014. 4
17. C. Clavier, B. Feix, G. Gagnerot, C. Giraud, M. Roussellet, and V. Verneuil. ROSETTA for single trace analysis. *Progress in Cryptology – INDOCRYPT 2012*, vol. 7668 of *LNCS*, 140–155. Springer, 2012. 4
18. C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. Horizontal correlation analysis on exponentiation. *Information and Communications Security*, vol. 6476 of *LNCS*, 46–61. Springer, 2010. 4
19. J.-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. *Cryptographic Hardware and Embedded Systems*, vol. 1717 of *LNCS*, 292–302. Springer, 1999. 3
20. J. Courrège, B. Feix, and M. Roussellet. Simple power analysis on exponentiation revisited. *Smart Card Research and Advanced Application*, vol. 6035 of *LNCS*, 65–79. Springer, 2010. 3
21. J.-L. Danger, S. Guilley, P. Hoogvorst, C. Murdica, and D. Naccache. A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards. *Journal of Cryptographic Engineering*, 3(4):1–25, 2013. 1
22. C. H. T. Duc-Phong Le and M. Tunstall. Randomizing the montgomery powering ladder. Cryptology ePrint Archive, Report 2015/657, 2015. 17
23. M. Dugardin, L. Papachristodoulou, Z. Najm, L. Batina, J. Danger, S. Guilley, J. Courrège, and C. Therond. Dismantling real-world ECC with horizontal and vertical template attacks. Cryptology ePrint Archive, Report 2015/1001, 2015. 4
24. M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe. High-speed curve25519 on 8-bit, 16-bit and 32-bit microcontrollers. *Design, Codes and Cryptography*, 77(2), 2015. 2
25. V. Dupaquis and A. Venelli. Redundant modular reduction algorithms. *Smart Card Research and Advanced Applications*, vol. 7079 of *LNCS*, 102–114. Springer, 2011. 3

26. P.-A. Fouque and F. Valette. The doubling attack – why upwards is better than downwards. *Cryptographic Hardware and Embedded Systems – CHES 2003*, vol. 2779 of *LNCS*, 269–280. Springer, 2003. 3
27. K. Gopalakrishnan, N. Thériault, and C. Z. Yao. Solving discrete logarithms from partial knowledge of the key. *Progress in Cryptology – INDOCRYPT 2007*, vol. 4859 of *LNCS*, 224–237. Springer, 2007. 4, 14, 15, 16
28. N. Hanley, H. Kim, and M. Tunstall. Exploiting collisions in addition chain-based exponentiation algorithms using a single trace. *Topics in Cryptology – CT-RSA 2015*, vol. 9048 of *LNCS*, 431–448. Springer, 2015. 4
29. C. Herbst and M. Medwed. Using templates to attack masked montgomery ladder implementations of modular exponentiation. *Information Security Applications*, vol. 5379 of *LNCS*, 1–13. Springer, 2009. 4
30. J. Heyszl, A. Ibing, S. Mangard, F. D. Santis, and G. Sigl. Clustering algorithms for non-profiled single-execution attacks on exponentiations. *Smart Card Research and Advanced Applications*, vol. 8419 of *LNCS*, 79–93. Springer, 2013. 4
31. J. Heyszl, S. Mangard, B. Heinz, F. Stumpf, and G. Sigl. Localized electromagnetic analysis of cryptographic implementations. *Topics in Cryptology – CT-RSA 2012*, vol. 7178 of *LNCS*, 231–244. Springer, 2012. 3, 17
32. N. Homma, A. Miyamoto, T. Aoki, A. Satoh, and A. Shamir. Comparative power analysis of modular exponentiation algorithms. *IEEE Trans. Computers*, 59(6):795–807, 2010. 3
33. M. Hutter and P. Schwabe. NaCl on 8-bit AVR microcontrollers. *Progress in Cryptology – AFRICACRYPT 2013*, vol. 7918 of *LNCS*, 156–172. Springer, 2013. 2
34. iSec Partners. nano-ecc – a very small ECC implementation for 8-bit microcontrollers. https://github.com/iSECPartners/nano-ecc, 2016. 2
35. K. Itoh, T. Izu, and M. Takenaka. Address-bit differential power analysis of cryptographic schemes OK-ECDH and OK-ECDSA. *Cryptographic Hardware and Embedded Systems – CHES 2002*, vol. 2523 of *LNCS*, 129–143. Springer, 2002. 6, 17
36. K. Itoh, T. Izu, and M. Takenaka. A practical countermeasure against address-bit differential power analysis. *Cryptographic Hardware and Embedded Systems – CHES 2003*, vol. 2779 of *LNCS*, 382–396. Springer, 2003. 6
37. K. Itoh, T. Izu, and M. Takenaka. A practical countermeasure against address-bit differential power analysis. *Cryptographic Hardware and Embedded Systems – CHES 2003*, vol. 2779 of *LNCS*, 382–396. Springer, 2003. 17
38. M. Izumi, J. Ikegami, K. Sakiyama, and K. Ohta. Improved countermeasure against address-bit DPA for ECC scalar multiplication. *2010 Design, Automation & Test in Europe Conference and Exhibition (DATE 2010)*, 981–984. IEEE, 2010. 6, 17
39. M. Izumi, K. Sakiyama, and K. Ohta. A new approach for implementing the MPL method toward higher SPA resistance. *International Conference on Availability, Reliability and Security, 2009 . ARES '09*, 181–186. IEEE, 2009. 17
40. P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. *Advances in Cryptology – CRYPTO'96*, vol. 1109 of *LNCS*, 104–113. Springer, 1996. 3
41. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. *Advances in Cryptology – CRYPTO '99*, vol. 1666 of *LNCS*, 388–397. Springer, 1999. 3
42. T. Lange, C. van Vredendaal, and M. Wakker. Kangaroos in side-channel attacks. *Smart Card Research and Advanced Applications*, vol. 8968 of *LNCS*, 104–121. Springer, 2015. 14

43. A. Liu and P. Ning. TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks (Version 1.0). http://discovery.csc.ncsu.edu/software/TinyECC/ver1.0/index.html. 2
44. K. Mackay. micro-ecc – ECDH and ECDSA for 8-bit, 32-bit, and 64-bit processors. https://github.com/kmackay/micro-ecc, 2016. 2
45. M. Medwed and E. Oswald. Template attacks on ECDSA. *Information Security Applications*, vol. 5379 of *LNCS*, 14–27. Springer, 2008. 3
46. P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987. 5
47. E. Nascimento. SAC 2016 - Implementation of algorithm for ECDLP with errors based on a time-memory tradeoff. https://github.com/enascimento/SCA-ECC-keyrecovery, 2016. 16
48. E. Nascimento. SAC 2016 - Targeted Curve25519 implementations for AVR. https://github.com/enascimento/sac2016-avr-target-impls, 2016. 6
49. E. Nascimento, J. López, and R. Dahab. Efficient and secure elliptic curve cryptography for 8-bit AVR microcontrollers. *Security, Privacy, and Applied Cryptography Engineering*, vol. 9354 of *LNCS*, 289–309. Springer, 2015. 2, 6
50. C. Negre and G. Perin. Trade-off approaches for leak resistant modular arithmetic in RNS. *Information Security and Privacy*, vol. 9144 of *LNCS*, 107–124. Springer, 2015. 16
51. C. O'Flynn and Z. D. Chen. ChipWhisperer: An open-source platform for hardware embedded security research. *Constructive Side-Channel Analysis and Secure Design*, vol. 8622 of *LNCS*, 243–260. Springer, 2014. 6
52. K. Okeya and K. Sakurai. Efficient elliptic curve cryptosystems from a scalar multiplication algorithm with recovery of the y-coordinate on a montgomery-form elliptic curve. *Cryptographic Hardware and Embedded Systems – CHES 2001*, vol. 2162 of *LNCS*, 126–141. Springer, 2001. 15
53. D. Otte. Avr-crypto-lib. https://git.cryptolib.org/avr-crypto-lib.git, 2016. 2
54. G. Perin and Ł. Chmielewski. A semi-parametric approach for side-channel attacks on protected rsa implementations. *Smart Card Research and Advanced Application*, vol. 9514 of *LNCS*, 102–114. Springer, 2015. 4
55. G. Perin, L. Imbert, L. Torres, and P. Maurine. Attacking randomized exponentiations using unsupervised learning. *Constructive Side-Channel Analysis and Secure Design*, vol. 8622 of *LNCS*, 144–160. Springer, 2014. 4
56. Sigma. ECDSA and ECDH cryptographic algorithms for 8-bit AVR microcontrollers. http://www.cmmsigma.eu/products/crypto/crs_avr010x.en.html. 2
57. C. D. Walter. Sliding windows succumbs to Big Mac attack. *Cryptographic Hardware and Embedded Systems – CHES 2001*, vol. 2162 of *LNCS*, 286–299. Springer, 2001. 4
58. H. Wang. WM-ECC is an Elliptic Curve Cryptography (ECC) primitive suite developed exclusively for wireless sensor motes. http://cis.csuohio.edu/~hwang/WMECC.html. 2
59. E. Wenger, T. Unterluggauer, and M. Werner. 8/16/32 shades of elliptic curve cryptography on embedded processors. *Progress in Cryptology – INDOCRYPT 2013*, vol. 8250 of *LNCS*, 244–261. Springer, 2013. 2
60. wolfSSL. Embedded Web Server for AVR. https://www.wolfssl.com/wolfSSL/Blog/Entries/2010/11/16_Embedded_Web_Server_for_AVR.html. 2
61. Z. Zhang, L. Wu, Z. Mu, and X. Zhang. A novel template attack on wnaf algorithm of ECC. *2014 Tenth International Conference on Computational Intelligence and Security (CIS)*, 671–675. IEEE, 2014. 3

A Target Details

The microcontroller was clocked at $f_{dev} = 7.3728\,\text{MHz}$ in our setup, i.e., the duration of one cycle is 135.63 ns. We placed a 49.9 Ohm resistor into the ground path of the microcontroller to measure the current consumption of the device using the Picoscope 5203 at a sample rate $f_{sample} = 500\,\text{MHz}$. Note that due to limitations on the size of the scope's memory, it is not possible to capture a single trace of a full scalar multiplication, which has a duration of approximately 2 s. Hence, we chose to capture only the interesting parts of each ECSM iteration, by using the segmented memory feature of the oscilloscope, partitioning the memory into small segments and triggering at each ECSM iteration. Note that in a real application, the adversary would likely not have a trigger signal for each ECSM iteration, but could easily overcome this problem with a pattern-based trigger generator or using an oscilloscope with larger memory.

B Trace Pre-processing

Since the sample rate f_{sample} is not a multiple of the device clock frequency f_{dev}, we first re-sampled the recorded traces to $f_{resample} = 493.978\,\text{MHz}$ (i.e., one cycle is composed of 67 sample points) to facilitate subsequent processing steps, in particular the cutting into single clock cycles, cf. Section Appendix B. For re-sampling, we used `libsamplerate`[9].

Filtering We digitally bandpass-filtered the traces using a Butterworth filter with a lower cutoff frequency $f_l = 300\,\text{kHz}$ and an upper cutoff frequency of $f_u = 2 \cdot f_{dev} = 14.75\,\text{MHz}$. These frequencies were determined experimentally, by testing various choices and selecting the parameters that yield the highest success rate (cf. Section 4.3 and Section 5.5).

Alignment and cutting To align the recorded traces, we employed a standard pattern-based approach: we selected a part of the first trace as the reference, and computed the euclidean distance or correlation for each offset within a chosen range for each following trace. We then shifted each trace by the respective offset that minimized the distance measure.

Finally, the filtered and aligned traces were cut into parts based on a cycle-accurate execution trace of the test implementation generated using an AVR simulator. This enabled us to generate templates for a specific instruction or an instruction sequence with cycle accuracy.

C Attack Parameters Investigation

To investigate the effect of various pre-processing steps and attack parameters, Table 3 gives the average success rate and confidence level for a range of choices. We selected the parameter and method combination that yields the highest overall confidence level to produce Figure 2.

[9] http://www.mega-nerd.com/SRC/

Table 3: Results from the Load attack with templates generated from a set of 10 traces and tested on a set of 10 traces from a different capture session. Only the first 15 ECSM iterations were targeted. Success Rate and Confidence Level values are averaged.

Class	Method / Param. Name	Param. Value	SR (%)	CL (%)
Filtering	No filtering	-	57.3	-
	Upper cutoff freq.	2.5 * f_{dev}	92.9	-
	"	2.3 *f_{dev}	93.6	-
	"	2.1 * f_{dev}	93.6	-
	"	2.0 * f_{dev}	94.3	-
	"	1.7 * f_{dev}	92.9	-
	"	1.5 * f_{dev}	90.7	-
	"	1.3 * f_{dev}	90.0	-
	Upper cutoff freq.	f_{sample}/1.9	94.3	-
POI Selection	(pLow, pHigh); nPOI	(12.5, 87.5); 23	58.5	32.4
	"	(25, 75); 71	76.4	33.9
	"	(35, 65); 324	94.3	36.8
	"	(37.5, 62.5); 686	69.8	33.4
	(pLow, pHigh); nPOI	(40, 60); 1500	64.1	31.6
	Force $\geq$ 1 sp per instr.	(35, 65); 669	92.1	68.6
	Force $\geq$ 1 sp per instr.	(40, 60); 1724	90.0	71.1
	Limit 1 sp per instr.	(35, 65); 134	85.7	8.6
	Limit 1 sp per instr.	(40, 60); 723	78.6	28.6
Classification	Sum of distances + POI	(35, 65); 324	94.3	33.9
	Majority voting + POI	(35, 65); 324	57.0	9.8
Win. compression	Normal sum + POI	1; (35, 65)	94.3	38.6
	"	10; (35, 65)	92.8	36.4
	Normal sum + POI	67; (35, 65)	79.3	20.7
	Absolute sum + POI	1; (35, 65)	94.3	23.1
	"	10; (35, 65)	92.1	27.6
	Absolute sum + POI	67; (35, 65)	77.1	18.3
Outlier removal	Multiple of stdev	2.0	92.1	40.7
	"	1.7	90.0	40.7
	Multiple of stdev	1.4	88.6	36.4
Distinguisher	Euclidean Distance	-	92.1	57.1
	Pearson Correlation	-	93.6	61.4
Combinations	EuclDst. + $\geq$ 1 sp per instr.	(35, 65); 669	92.1	79.3
	Corr. + $\geq$ 1 sp per instr.	(35, 65); 669	93.6	65.0