Evaluating the Security of Open-Source Cryptographic Libraries Against Physical Attacks: An End-to-End Study on Botan

Damien Jauvart[1][0009−0005−1028−9228], Aurélien Vasselle[1][0000−0002−9320−2685]

eShard, Pessac, France damien.jauvart@eshard.com

Abstract. This paper presents an end-to-end analysis of side-channel vulnerabilities in the Elliptic Curve Cryptography (ECC) implementation of the Botan open-source library, which incorporates countermeasures against physical attacks. Focusing on scalar multiplication with a 4-bit fixed window, we investigate leakages under both emulated (register-level) and real-device conditions (NXP *i.MX* 8 System-on-Chip).
We performed a thorough leakage assessment and identified three distinct vulnerabilities leading to key recovery: a timing leak from improper zero-window handling, a table lookup leakage, and collision attacks on precomputed tables. The first is exploitable via unsupervised clustering on real devices (99% success), while the second enables full scalar recovery through supervised deep learning (98.85% accuracy). The third approach, though effective in emulation (84% success), faces practical limitations on real hardware.
Our work demonstrates that even well-established countermeasures like scalar blinding and point randomization can leave exploitable vulnerabilities through simple and advanced side-channel attacks.

Keywords: Side-Channel Analysis · Elliptic Curve Cryptography · Deep Learning · Unsupervised Clustering · Collision Attacks.

1 Introduction

In this study, we focus on analyzing the open-source Botan library's ECC through both emulated and real-device experiments. Our methodology combines code analysis and statistical leakage assessment to detect vulnerabilities. By identifying critical leakage points and evaluating their exploitability, we propose mitigation to improve the security of this cryptographic implementation.

The state-of-the-art reveals a gap in practical end-to-end side-channel analyses of windowed ECC implementations. Existing work focuses predominantly on algorithmic countermeasures or isolated attack demonstrations. We aim to fill this gap by providing a complete analysis, from code-level observations to real-device exploitation, applied to Botan's ECC implementation.

Our work makes three main contributions. First, we present a practical end-to-end methodology for side-channel evaluation that combines source code analysis, statistical leakage assessment on emulated traces, and real-device validation

(NXP *i.MX* 8), applied to a widely used open-source cryptographic library. Second, we introduce a deep-learning-based collision oracle that classifies whether the same EC point is manipulated during precomputation and scalar multiplication, even if the operations are different by nature. Third, we demonstrate that Botan's countermeasures (scalar blinding, point randomization, and claimed constant-time execution) can all be circumvented through implementation-level subtleties, including a microarchitectural timing leak and a table lookup leakage exploitable via supervised deep learning. In total, we identify three distinct attack paths: a non-constant-time failure in zero-window handling (Section 5), high leakages in a table lookup (Section 6), and a deep learning-based collision attack on precomputed tables (Section 7).

1.1 Side-Channel Attacks against ECC

An elliptic curve over $\mathbb{F}_p$ is defined by $E : y^2 = x^3 + ax + b$. The set of solutions (x, y) forms an Abelian group under point addition. Arithmetic operations rely on the field operations, particularly modular inversion, which is expensive compared to multiplication or squaring. To reduce the computational cost, Jacobian coordinates are often used. In this system, points are represented as $(X : Y : Z)$ where the affine coordinates are $(X/Z^2, Y/Z^3)$. Then, doubling and addition operations can be performed without inversions.

Side-channel attacks, first brought to prominence by [14], can take advantage of a variety of leakages, such as power consumption, timing, electromagnetic radiation, and even acoustic signals. In the context of ECDSA and ECDH, the primary target of such attacks is the scalar multiplication, where a secret scalar k multiplies a point P on the curve.

Standard countermeasures against side-channel key recovery attacks include scalar and point randomization [5], complete addition formulas [4], Montgomery ladder [13,17], and scalar recoding methods such as Non-Adjacent Form [16], w-NAF [18], and randomized m-ary recoding [3,15]. Surveys provide overviews of SCA countermeasures [2] and formal evaluation guidelines [1]. In particular, windowed implementations have been shown exploitable via single-trace correlation [11] and via clustering techniques [12]. Unsupervised deep learning has also been combined with horizontal attacks to recover key bits after blinding [19]. Despite this work, windowed implementations remain comparatively understudied in side-channel literature, while being the most popular choice for high-performance devices and software implementations.

1.2 Evaluation Methodology

Our evaluation follows the BSI AIS 36 guidelines [1]: we map each side-channel attack against Botan's countermeasures, assess remaining attacks for applicability, and attempt to exploit those in scope. These guidelines defines a list of recommended attacks, organized by threat profile, and specifies the requirements each attack scenario must satisfy to demonstrate resistance. We enumerate these recommendations as RM:Category-Reference (e.g., RM:ScalarMult-2 is the first

one at page 45), mapping each identified leakage to the corresponding entry. The methodology proceeds in three stages: (1) source code analysis, focusing on secret-key operations and precomputed table lookups; (2) statistical leakage assessment using t-test and Correlation Power Analysis (CPA) on emulated traces (using eShard esFirmware); and (3) exploitation via clustering and supervised learning, tested on both emulated and real traces. Sections 5, 6 and 7 detail these three assessed points.

For an open-source library, we assume a profile-level threat model: the adversary has access to identical hardware and can compile and execute the same cryptographic library with the same configuration, enabling profiling of the device's leakage behavior. Our analysis focuses on passive side-channel leakages targeting the core scalar multiplication primitive, using a variable input point for precise control.

We did not consider the attacks on the transfer and loading of the secret scalar (ATT::L:SPA:2 of the RM:ScalarGen-2) and those targeting the scalar blinding mechanism itself (ATT::L:DPA:6 of the RM:ScalarMult-2, [9]). We also did not consider the direct clustering on the scalar window value (ATT::L:CLU:1 of RM:ScalarMult-5) due to the high remaining entropy after successful clustering (this point is analyzed and solved in [12] in the case of perfect clustering).

2 Botan's Implementation Structure

Botan is one of the main C++ libraries designed for cryptographic applications. The analyzed implementation uses a fixed-window approach, and precomputed tables store intermediate elliptic curve points. The algorithm begins with table precomputation, and coordinates conversion to Jacobian representation; then randomizes the scalar and the input point. For each subsequent window, it performs doublings (in projective coordinates), reads the scalar window value and the corresponding point from the table, and adds it back to the accumulator result using mixed coordinates.

Moreover, the code analysis of the Botan 3.7.1 scalar multiplication implementation reveals the following countermeasures:

- **Input point randomization.** The accumulator point Q in the scalar multiplication is represented in Jacobian coordinates. A point re-randomization happens for the **first four** windows processed.
- **Scalar blinding.** The 256-bit input scalar k is randomized with the additive blinding technique. A 64-bit mask r is randomly drawn, and its high and least significant bits are set to one. The blinded scalar k' is $k + r \cdot n$.
- **Constant time operation.** The library claims that the ECC algorithms are written to avoid timing and cache-based side channel attacks.
- **Table access protection.** The table of precomputed multiples is accessed using a masked lookup, which should not leak side-channel information.

The code structure of the windowed scalar multiplication is summarized by the pseudo-code given in Algorithm 1. The parameter $window_{size}$ is set to 4 in

this study implementation. The 4-bit window method is the default for scalar multiplication in the context of ECDH in Botan version 3.7.1. The chosen elliptic curve for this work is the `secp256k1`, recommended in the Standards for Efficient Cryptography (https://www.secg.org/sec2-v2.pdf). We compile the library, in its version 3.7.1 (https://github.com/randombit/botan/tree/3.7.1), thanks to the provided `configure.py` file and with the parameters in Appendix A.

Algorithm 1: Windowed scalar multiplication in Botan

Input: Scalar k, base point P, and $window_{size}$
Output: $[k]P$

`/* Scalar blinding */`
$k' \leftarrow k + n \cdot r$, with a random 64-bit r ; `// msb & lsb of` r `are fixed to 1`
`/* Precomputation */`
$T \leftarrow \left[P, [2]P, [3]P, \ldots, [2^{window_{size}} - 1]P\right]$; `// construct table of points`
Convert coordinates of points in T into affine coordinates;
`/* Main loop over each bit windows of` k' `*/`
$w_0 \leftarrow$ the first window value;
$Q \leftarrow T[w_0 - 1]$; `// set the accumulator`
Projective randomization of point Q;
for $j = 1$ **to** *number of windows - 1* **do**
  for $i = 0$ **to** $window_{size} - 1$ **do**
    $Q \leftarrow [2]Q$; `// point doubling in projective coordinates`
  end
  $w_j \leftarrow$ the window value;
  $Q \leftarrow Q + T[w_j - 1]$; `// point addition in mixed coordinates`
  If $j < 4$ then randomize Q;
end

3 Generating and Capturing Datasets

3.1 Emulation

Our primary instrument for this investigation is eShard's esFirmware tool, which enables the execution of the target firmware within a virtual environment that traces the system at instruction layer abstraction. Specifically, the tool records the program counter at each instruction, allowing us to precisely locate and isolate the scalar multiplication routine. The tool further provides access to the state of all CPU registers at each operation. This granularity enables us to capture the precise values manipulated during the computation and to model them as Hamming weight. Each sample in the trace therefore contains the weight of input and output registers of a single executed instruction. This approach yields deterministic, noise-free traces that serve as a baseline for validating our

attack scenarios and statistical tests against sensitive assets. The emulated traces are visualized in Figure 1.

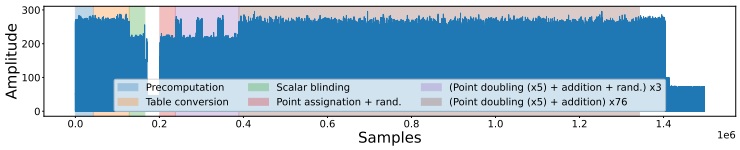


Fig. 1. Emulation: Scalar multiplication with fixed window. The blank frame corresponds to the blinded scalar being written into a file (non-targeted area).

3.2 Measurement on NXP *i.MX* 8

One of our goals was to validate the soundness of security testing on emulated traces, for that we compared with the same library running on real hardware. The *i.MX* 8 series by NXP Semiconductors is a family of ARM-based application processors designed for advanced embedded systems. We selected this board because its ARM Cortex-A architecture accurately represents modern System on Chip (SoCs) and is compatible with a complete Linux environment. While the device runs a full Android OS, it allows for the straightforward deployment of our ELF binary directly at the underlying Linux level, where it executes exactly like a native system service or library. The firmware executed on this device is identical to the one used in emulation.

For electromagnetic trace capture, the device was placed under a Langer RF-B 0.3-3 H-field probe (30 MHz to 3 GHz). The captured signal was amplified using a Langer PA 303 BNC preamplifier (100 kHz to 3 GHz) and acquired with a Lecroy 404HD oscilloscope at sampling rate 20 GS/s. All traces were synchronized using a hardware trigger aligned with the start of the scalar multiplication routine, Figure 2.

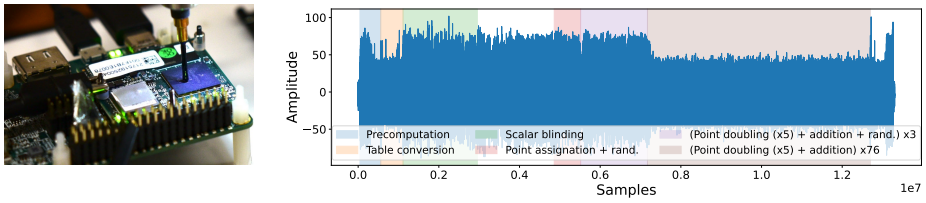


Fig. 2. Real device: Board and scalar multiplication trace.

Observable timing differences exist because the emulator abstracts complex microarchitectural behaviors, such as pipelining, branch prediction, and data

fetch latency. While a memory access might require only a single cycle in simulation, a physical cache miss resulting in a DRAM read can stall the real device for approximately 100 ns (nearly 200 clock cycles). The emulator does not capture the inherent hardware latency of physical random number generation.

Both traces exhibit the same structural decomposition in steps: precomputation, table conversion, scalar blinding, point assignment with randomization, and the main scalar multiplication loop ($3 + 76 = 79$ iterations of doubling and addition). However, notable differences exist. The emulated trace, generated from register-level Hamming weights, exhibits a noise-free signal with sharply defined step boundaries. In contrast, the real Electromagnetic (EM) trace shows a lower signal-to-noise ratio, making individual operations less distinct. The transitions between steps in the EM trace are also less sharp. Despite these differences, the overall structure is preserved, validating the emulation as a useful baseline for attack design while underscoring the additional challenges posed by physical noise on real hardware.

4 Global Leakage Assessment

4.1 t-test on the Scalar

We emulate two datasets: one where the scalar k remains fixed while the point P varies across different random instances, and another where both k and the points P are the same as before. By applying the t-test to these datasets, we identify statistical differences that indicate potential leaks on the scalar.

The significance threshold is determined using the Šidák correction on $\alpha = 0.05$, and the critical value is derived from the standardized normal distribution (approximately ± 5.52). In our analysis, significant crossings above the threshold occur during accumulator randomization and scalar multiplication step.

The scalar blinding countermeasure does not fully mask the blinded scalar. Indeed, as explained in [8], some bits at certain positions of the blinded scalar are actual bits from the original scalar that remained unprotected[1]. Consequently, the side-channel traces are still correlated to the scalar, and the t-test statistic exceeds the threshold during the manipulation of those bits. As shown in Figure 3, the t-test has a high value (around ± 100) in the frame 0.56– 0.76 MSamples.

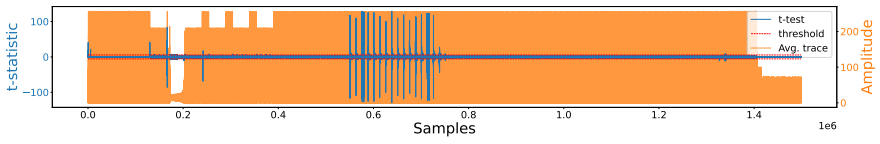


Fig. 3. Emulation: t-test on scalar.

[1] This inherent leakage depend on the curve parameters and the blinding random size.

Figure 3 also reveals three other leakages between $0.13 - 0.25$ MSamples. The first two are from the scalar blinding procedure. The blinding procedure loads the original scalar k, draws a random mask r, and computes $k' = k + r \cdot n$. The t-test shows leakage at the loading of k, and at the computation of the blinded scalar. These leakages, which correspond to BSI requirements RM:ScalarMult-2 and RM:ScalarGen-2, are not further studied in this paper to prioritize analysis on the higher-impact leakages described in the subsequent attack sections. The third leakage (≈ 0.24 MSamples) appears during the first doubling operation in the scalar multiplication, this remains unexplained.

4.2 t-test on the Input Point

The t-test is also applied to the input point (Figure 4). We generate two datasets: one with fixed input points (in affine coordinates) and another with random input points, using the same random scalar for both.

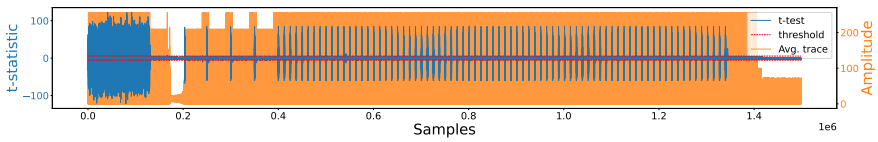


Fig. 4. Emulation: t-test on point.

The t-test reveals significant leakages tied to the input point. At the beginning of the trace, the table precomputation and point coordinates conversion constitute a massive leakage source, as the affine coordinates of the input point P are repeatedly manipulated. Furthermore, the t-test shows a distinct peak at each scalar multiplication loop iteration. This corresponds to adding one point to the table. These leakages map to BSI requirement RM:ScalarMult-5 (attack ATT::L:SPA:13) and form the basis for the third attack (Section 7).

A key observation from this assessment is that not all detected leakages are systematically exploitable for recovering sensitive information. The most striking example is the leakage on the most significant bit of the blinded scalar (revealed by the t-test zoom above). This leakage could have a critical effect on the nonce scalar in the ECDSA context through lattice attacks (see [6]), while in ECDH it does not lead to recovering sensitive assets.

Conversely, some vulnerabilities might elude detection by the t-test, such as non-constant timing in handling specific window values (c.f. attack in Section 5). This highlights the importance of combining statistical analysis with deeper security evaluations.

5 First Attack: Non-Constant Time Failure

The Botan library states that the computation is done in constant time. We investigate this claim for the scalar multiplication processing of each window. Indeed, the fixed window is 4 bits; cases where the processed window equals 0 appear with non-negligible probability: $\frac{1}{2^4} = 6.25\%$. In that case, Botan enforces constant-time execution through a dummy addition operation using point at infinity $\mathcal{O}$. A distinctive behavior emerges during this special EC point addition, as explained in the code analysis.

5.1 Code Analysis

The code analysis aims to demonstrate the use of the point at infinity as a standard point in the addition, leading to the non-constant time vulnerability. The main loop performing the scalar multiplication appears in the file `src/lib/math/pcurves/pcurves_impl/pcurves_impl.h` at the line 1740. In an effort to achieve constant-time processing, no special treatment happen for windows equal to 0.

The `AffinePoint::ct_select` function (file `pcurves_impl.h`, line 772) returns the point from the precomputed table `m_table`. It returns the identity if the requested index is 0 (*i.e.,* window value zero). The identity is set at $(0, 0)$.

Afterwards, the operator `+=` performs the point addition of `accum` in Jacobian coordinates with the table point in affine coordinates. This operation $P + Q$ is handled by the `add_mixed` function. Even when one of the operands is $\mathcal{O}$, the point ADD computation proceeds without distinction. Yet, the computed outcome is meaningless since the ADD formula does not apply, thus the final result is replaced by the other operand at the end. As a consequence, coordinates with value 0 are involved in field multiplication computation. These operations should produce remarkable behavior in the side-channel traces.

5.2 Emulated Trace Analysis

Figure 5 shows three consecutive scalar multiplication loop iterations. It reveals a dense region of peaks followed by a drop. This pattern appears if and only if the processed window equals zero. We can see very noticeable differences in activity, but the number of instructions remains the same.

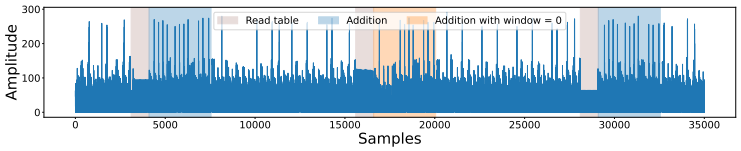


Fig. 5. Emulation: trace for window equal to 0.

5.3 Attack Validation on Real Device

However, on real device traces, it appeared that the countermeasure failed to enforce constant-time. This failure occurs only on the real device because it stems from microarchitectural behavior. Therefore, tools such as *ctgrind* are unable to detect it. Our first analysis verifies the impact of processing a window equal to 0. As a preliminary step, the real traces are stacked: the 79 point additions per trace are arranged vertically and synchronized using a comb (corresponding to the `ct_select` operation). Then, we calculate the average of the traces where the window equals 0 and where it differs from 0. As shown in Figure 6, a visual difference appears.

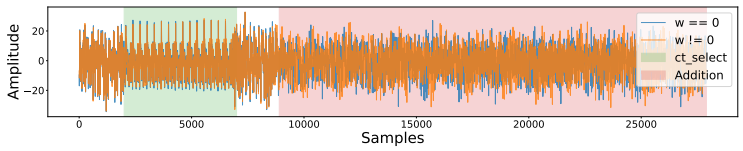


Fig. 6. Real device: average traces for window equal to 0 and non-0. The blue trace is ahead of the orange one when the EC point addition starts.

More precisely, the difference translates to traces duration where windows equal to 0 execute faster than others, which is unexpected since the Botan library claims constant-time computation. Our emulation did not reveal this, as microarchitectural behavior is not simulated.

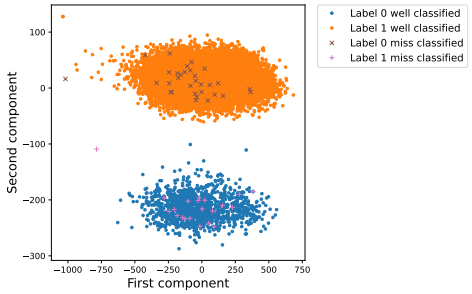
We implemented a clustering classification to automatically detect differences between traces. All synchronized traces at the location of the EC point addition are fed into a `KMeans` classifier. The algorithm is parameterized to create 2 clusters; we expect a distribution of $\frac{1}{16}$, $\frac{15}{16}$ ($w = 4$, then $2^w = 16$) with the labels representing traces corresponding to window value 0.

The chosen frame targets the beginning of the visual differences. The chosen normalization scales using the maximum absolute value (`MaxAbsScaler`).Then, `KMeans(n_clusters=2, n_init=50)` is applied to these inputs. The unsupervised classification results are visualized by projecting the traces into two dimensions with Principal Component Analysis (PCA) in Figure 7.

This achieves a 99% success rate. An unsupervised attack could recover all window processes with value 0 with high probability, recovering some bits of the blinded scalar used in the scalar multiplication.

5.4 Implication: Partial Nonce Recovery via Unsupervised Profiling

Although initially identified during the analysis of scalar multiplication in an ECDH environment, we explicitly verified that this vulnerability is inherently present in ECDSA due to the execution of the exact same leaking point addition



		Prediction	
		0	Non-0
Real	0	1232	31
	Non-0	21	19440

Fig. 7. Results for window 0: clustering visualization (left); confusion matrix (right).

function (`add_mixed`). Lattice attack exploiting such leakages against ECDH are discussed in [20].

By employing this unsupervised profiled attack scheme, an adversary can recover partial information (several bits) of the masked ephemeral nonce. Here, this attack reveals multiple nonce bits equal to 0. This partial leakage can be exploited against the ECDSA private key using lattice-based cryptanalysis, modeled as a Hidden Number Problem (HNP), to deduce the long-term private signature key, even within a scalar blinding countermeasure [22,7]. This is even more critical given the partial unmasking in the middle of the blinded scalar identified in Section 4.1.

6 Second Attack: Point Copy during Table Selection

In windowed implementation with precomputed tables, the secret scalar bits are used to read an EC point from the table, which becomes a critical asset.

6.1 Code Analysis

The constant-time table reading process is handled by the `ct_select` method (file `pcurves_impl.h`, line 772). To avoid memory-dependent timings, a loop reading the entire table is executed; When the loop index matches the requested point, a Boolean is set to true, causing a copy of the table point to the output buffer through `conditional_assign` (file `pcurves_impl.h`, line 310).

Before the iteration where $\texttt{idx} - 1 = i$, the resulting point remains $\mathcal{O}$. At the matching iteration, it receives the expected table point $T[i]$, and retains that value for all subsequent iterations. This repeated assignment is the root of the observable leakage.

6.2 Emulated Trace Analysis

A change in the register Hamming weight is clearly identified at the processed iteration matching the window value (*i.e.*, $\texttt{idx} - 1 = i$) as shown in Figure 8. As

a result, the window value is directly readable on the traces through SPA. This leads to the recovery of the blinded scalar, which is critical for the security of both ECDH and ECDSA.

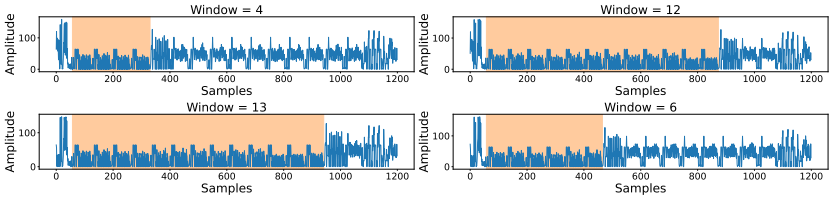


Fig. 8. Emulation: trace of table EC point reading. The side-channel clearly reveals the window value at the iteration where the index matches.

6.3 Validation on Real Device

Leakage analyses. From a security perspective, the `ct_select` operation remains constant-time, but shall be assessed against side-channel leakages. Due to much lower signal-to-noise ratio, it is not possible to directly retrieve the secret value visually. On one hand, a correlation power analysis on the point coordinates $[i]P$, in the Hamming weight model reveals high leakages. Indeed, the 64-bit point coordinates are manipulated during these operations and provoke leakages (see Figure 14 in Appendix B).

On the other hand, the selection index i is in itself a secret that is manipulated. A leakage assessment within the ANOVA (ANalysis Of VAriance) distinguisher locates areas such manipulations, shown in Figure 9.

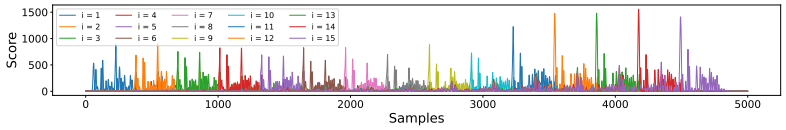


Fig. 9. Real device: leakages of `ct_select` window values. A window value leaks at a precise location where the loop processes the matched index.

Deep learning classification. Unlike with emulated traces, directly identifying the window value visually is difficult on real device traces. However, these leakages are exploitable with a deep learning model. A Multi-Layer Perceptron (MLP) was trained to classify the traces at the `ct_select` location according to their processed window value.

This is a 4-layer MLP with a funnel-like architecture with decreasing neuron counts per layer. The precise structure and training parameters are provided in Appendix C. The training is performed over 20 epochs on 59300 traces, and validated on 6600 traces. Figure 10 demonstrates the high learning efficiency. The confusion matrix in Figure 11 on the validation dataset confirms excellent results across all possible window values. The window value recovery accuracy is approximately 98.85%, allowing single-trace extraction of secret bits.

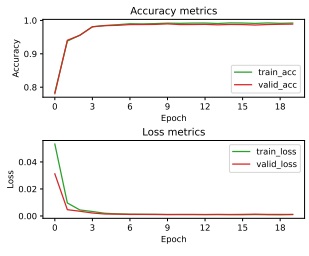


Fig. 10. Real device: deep learning metrics on window values during `ct_select`.

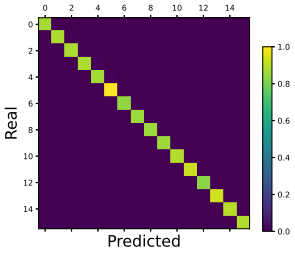


Fig. 11. Real device: confusion matrix for DL window value classification.

6.4 Implication: Full Scalar Recovery via Supervised Profiling

This vulnerability is critical, applicable to both ECDH and ECDSA protocols and enabling the direct, complete recovery of the scalar used during the multiplication step. It relies on a supervised profiled attack methodology, feasible with an open-source library. An adversary must possess an identical, fully controllable "clone" device for an extensive training phase. Once successfully profiled, it allows for the immediate extraction, in a single trace, of the static private key in ECDH, or the ephemeral nonce in ECDSA, leading to full key recovery.

7 Third Attack: Collision with Precomputed Table

This attack detects collisions (manipulation of the same elements) between table points during their precomputation and their usage during scalar multiplication. With a fixed 4-bit window, the table contains $15 = 2^4 - 1$ possible multiples of the input point P, noted by $T = \left[P, [2]P, [3]P, \ldots, [15]P\right]$ (*i.e.* $T[i] = [i]P$).

During the scalar multiplication loop, these EC points are used. The goal is to identify which one is used at each iteration. We proceed with a training phase and then an attack phase on a **single trace**.

7.1 Code Analysis

Addition in the windowed scalar multiplication. The point addition $Q + T[i]$ is invoked in `accum += AffinePoint::ct_select(m_table, w_i);`, of the file

`pcurves_impl.h`. The `+=` operator calls `Self::add_mixed(a, b)`. This adds the accumulator Q in Jacobian coordinates with the table point $T[i]$ in affine coordinates. We identify where the coordinates of $T[i]$ are manipulated in this function, see Equation 1 for the first operations. The affine coordinates of the table point are used as the left operand of modular multiplications. This is the key leakage point.

$$\begin{aligned} r_0 &\leftarrow Z_Q^2, \\ r_1 &\leftarrow \boldsymbol{x_{T[i]}} \cdot Z_Q^2, \\ r_2 &\leftarrow \boldsymbol{y_{T[i]}} \cdot Z_Q \cdot Z_Q^2, \quad \ldots \end{aligned} \tag{1}$$

Precomputation table and point coordinate conversion. The precomputed table is built in two stages (file `pcurves_impl.h`, line 1711). First, 15 projective points are generated iteratively: odd index $T[i]$ as a doubling of $T[i/2]$, even index as $T[i-1]+P$. The second stage converts all points to affine coordinates via batch inversion using the Montgomery trick [10], algorithm 2.26. This is the first and only step where every table point's affine coordinates are manipulated. Then, a side-channel observer sees a concentrated manipulation of all $x_{T[i]}$ and $y_{T[i]}$ values together.

7.2 Emulated Trace Analysis

Before building an attack on those manipulated point coordinates, a leakage assessment ensures that the side-channel traces contain a high dependence on them. Analyses use the Hamming Weight model of 256-bit words (the full Hamming weight of x and y coordinates of the targeted EC point).

Addition in the windowed scalar multiplication. Multiple traces are emulated. In each, the trace segments where additions occur are extracted and stacked into a pool of addition traces, labeled by the corresponding window value. A CPA between traces and point coordinates shows a significant correlation at two similar patterns. These patterns indicate that the point coordinates leak in Hamming weight, directly tied to the modular multiplications that form the core of the point addition operation.

Point coordinate conversion. Similarly, the leakage analysis is performed on each point's coordinate conversion. The CPA reveals clear correlation peaks concentrated around the modular multiplication operations. These leakage analyses confirm the findings from the code analysis: the point addition in the scalar multiplication loop and the affine coordinate conversion in the precomputation step both expose exploitable information.

7.3 Attack Description

This attack detects whether the same EC point is processed during table precomputation and during scalar multiplication. If the precomputed point $[i]P$ is the one selected as $[w_j]P$ in the j-th iteration, their side-channel signatures during the affine coordinate conversion and point addition may be similar. This attack can be formalized as a collision-classification problem [11]: each addition operation is matched against precomputation candidates $i \in [2, 15]$, an oracle distinguishes collision events from mismatches. In previous work, correlations were used to quantify the collision between two similarly-shaped parts of traces. Because, in Botan, the conversion outputs the EC point while the addition uses it as operand, they are intrinsically of different nature, and we do not observe comparable lengths and leakage models between both operations. The key idea of this new technique is to use a neural network to learn non-trivial similarities between conversion outputs and addition inputs. For that, chunks of traces from both operations are paired together via concatenation, and a DNN is trained as the oracle to recognize when collision happens.

Training phase. For each candidate i, we build multiple training pairs from a single scalar multiplication trace. We extract the sub-trace covering the affine conversion of $[i]P$ and concatenate it with the sub-trace of the point addition at each of the 79 windows iterations[2]. The training label is 1 if i matches the actual window value w_j at that iteration, and 0 otherwise. Because collisions are rare ($p = 1/16$), we subsample negative results to balance the dataset. We repeat across multiple executions, always using the same input point P so that the conversion of $[i]P$ produces consistent leakage patterns.

Attack phase. Given a new trace with an unknown blinded scalar, we extract the 15 conversion sub-traces and the 79 addition sub-traces. Each conversion sub-trace of $[i]P$ is concatenated with each addition sub-trace and fed to the corresponding network. The network output gives a collision probability for each candidate i, identifying the window value at every iteration. Note that $i = 0$ and $i = 1$ are handled separately: $[0]P = \mathcal{O}$ is not stored in the table, and $[1]P = P$ is handled separately.

7.4 Attack Validation on Emulation

Training phase. We trained MLPs with architecture and training parameters given in Appendix C. As an illustration, the training metrics for the case $i = 2$ are shown in Figure 12. The same applies for all 14 networks and they all successfully learn to detect collisions with their respective precomputation.

The case $i = 1$ is handled separately. The library converts this point to Jacobian coordinates without randomizing Z_P, so $(x_P, y_P) = (X_P : Y_P : 1)$.

[2] The first iteration assigns the initial point rather than performing an addition.

Therefore, (x_P, y_P) are manipulated both in the precomputation (calculating $[2]P$) and in scalar multiplication (adding P). We train an MLP with the same structure; the learning accuracy is similar to that of the other training.

Attack phase. The attack takes a single trace, and for $i \in [1, 15]$ concatenates the signal part i of the coordinate conversion (and the point doubling for $i = 1$ with each signal part j of the point addition during scalar multiplication. Each concatenated signal is fed to the 15 networks; the one with the highest probability gives the candidate for the j-th window value. The resulting confusion matrix on the validation dataset is in Figure 13. The attack success rate is 84%, validating the attack scheme.

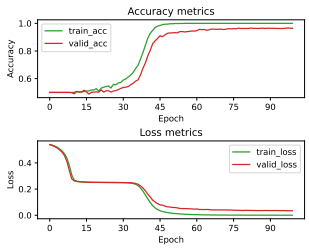


Fig. 12. Emulation: neural network training for collision detection, window value 2.

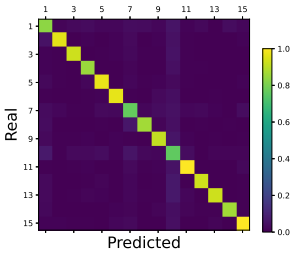


Fig. 13. Emulation: collision attack validation confusion matrix.

7.5 Attack Validation on Real Device

The EM trace during the projective-to-affine point coordinate conversion consists of 14 repeated patterns. We split each trace into 14 pieces (for the points $[15]P, [14]P, \ldots, [2]P$). A Correlation leakage analysis is performed with Hamming weight on 64 bits (architecture word size) and reveals blurry leakage peaks during the affine coordinate points conversion. The leakage peaks also appears during point addition in the scalar multiplication loop. Nevertheless, the leakages are lower than in the emulation. We ran a hyperparameter search, resulting in a 5-layer MLP with 200 nodes per layer (see Appendix C for details). However, the models failed to learn, indicating either the need for a different network or a lack of leakage.

The applicability of this attack path on real device remains to be demonstrated. While it should work in principle and it was validated on emulated traces, the accuracy on real hardware may be limited to a low signal-to-noise ratio and the presence of micro-jitter. We believe that further improvement of the acquisition setup and NN collision oracle may lead to improved results.

7.6 Implication: Precomputation Leakage via Supervised Profiling

This vulnerability exploits the table precomputation step of ECDH scalar multiplication, where precomputed table points are combined with the running accumulator to produce intermediate values that leak information about the secret scalar. This is a supervised profiled attack: an attacker must first characterize the device using known scalars before transferring the model to an unknown target and identifying the value of the windows in a single trace. On emulated traces the attack reaches 84% success rate, but on real device EM traces the elevated noise floor obscures the collision-based leakage patterns, making neural network training ineffective. Bridging this emulation-to-real gap remains an open direction for future work.

8 Proposed Mitigation

For the non-constant-time zero-window failure, the root cause is the special-case handling of the point at infinity. A direct fix is to ensure the point at infinity never enters the arithmetic: for example, by computing $T[i] + P$ instead of $T[i]$ so every table entry is a valid point with non-zero coordinates. This must be paired with a strictly constant-time dummy addition logic, but should thwart both timing and value differences previously caused by the manipulation of $\mathcal{O}$.

The table lookup leakage is rooted in the deterministic access pattern of `ct_select`. Randomly shuffling the order in which table entries are read during the main loop breaks the temporal correlation between leakage traces and scalar bits, mitigating a supervised profiled attack.

The precomputation table collision attacks similarly exploit the fixed ordering of point arithmetic during table construction and batch affine conversion. Applying the same shuffling principle to the table-building and coordinate conversion steps randomizes the time-domain alignment of these leakages, preventing an adversary from aligning traces to the sensitive intermediate values.

9 Conclusion

This work presents an end-to-end side-channel evaluation of Botan 3.7.1's ECC implementation, combining code analysis, emulated leakage assessment, and real-device validation on the NXP *i.MX* 8. We identified and successfully exploited three vulnerabilities: a non-constant-time failure in zero-window handling, a table lookup leakage enabling supervised deep learning, and a collision-based attack on precomputed tables. This study demonstrates that even subtle implementation choices in core cryptographic operations can lead to exploitable leakage, despite the use of countermeasures against side-channels.

Unlike typical evaluations, our methodology combined leakage simulation and real-world measurements. Our emulation tool provides a noise-free baseline that allows rapid validation of attack scenarios (Attack 2) before deploying on real hardware. The real-device experiments revealed a microarchitectural timing

leak (Attack 1) that our emulator could not predict, highlighting the necessity of physical validation for complete assurance. Our deep-learning-based collision oracle (Attack 3) further extends correlation-based techniques [11] by learning non-trivial similarities between trace segments of different nature, enabling a novel class of single-trace attacks.

Beyond these specific findings, the study opens to broader long-term challenges: maintaining side-channel testing pipelines as cryptographic libraries evolve. Botan 3.8.0 version switched to a Booth scalar recoding: while it renders our attack scripts obsolete, the identified vulnerabilities likely subsist. Unitary security testing and validation of sensitive subroutines may be achievable, easing the analysis of subsequent library versions. Tools such as [21] paired with our emulated leakage assessment can help automate the pipeline. Yet, profound implementation changes, such as the recoding scheme, demands validation with dedicated SCA expertise. Moreover, as demonstrated with our timing attack on this constant-time implementation, considering microarchitectural effects in the security model of cross-platform cryptographic libraries further extends the testing required to achieve resistance against physical attacks.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Bundesamt für Sicherheit für Informationstechnik. Guideline for Evaluating SCA and FA Resistance of EC Implementations (AIS 36, version 3.0). Bonn (2024). https://www.bsi.bund.de/EN/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Kryptografie/Seitenkanalresistenz/seitenkanalresistenz.html
2. Abarzúa, R., Valencia, C., López, J.: Survey for performance and security problems of passive side-channel attacks countermeasures in ECC. Cryptology ePrint Archive, Report 2019/419 (2019)
3. Ahn, Y., Moon, S., Park, J.: A new side-channel attack on secret key recovery using randomized m-ary method. In: Kim, K. (ed.) ICISC 2003. LNCS, vol. 2975, pp. 80–94. Springer, Heidelberg (2004). https://doi.org/10.1007/3-540-44843-8_36
4. Brier, E., Joye, M.: Weierstraß elliptic curves and side-channel attacks. In: Parry, D. (ed.) PKC 2002. LNCS, vol. 2274, pp. 285–299. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45664-3_24
5. Coron, J.S.: Resistance against differential power analysis for elliptic curve cryptosystems. In: Kocher, D., Katz, R. (eds.) CHES 1999. LNCS, vol. 1717, pp. 292–302. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48059-5_25
6. De Micheli, G. and Heninger, N., 2020. Recovering cryptographic keys from partial information, by example.
7. Goudarzi, D., Rivain, M. and Vergnaud, D., 2016, August. Lattice attacks against elliptic-curve signatures with blinded scalar multiplication. In International Conference on Selected Areas in Cryptography (pp. 120-139). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-69453-5_7

8. Feix, B., Roussellet, M., Venelli, A.: Side-channel analysis on blinded regular scalar multiplications. In: Canteaut, A. (ed.) INDOCRYPT 2014. LNCS, vol. 8970, pp. 3–20. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-13039-2_1
9. Fouque, P.A., Réal, D., Valette, F., Drissi, M.: The carry leakage on the randomized exponent countermeasure. In: Fischer, V., Homma, N. (eds.) CHES 2008. LNCS, vol. 5154, pp. 92–106. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85053-3_13
10. Hankerson, D., Menezes, A., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer, New York (2004). https://doi.org/10.1007/0-387-21846-7_3
11. Järvinen, K., Balasch, J.: Single-trace side-channel attacks on scalar multiplications with precomputations. In: Poschmann, A., Steinfeld, M. (eds.) CARDIS/LESYCO 2016. LNCS, vol. 10146, pp. 115–132. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-54669-8_9
12. Jin, S., Lee, S., Cho, S.M., Kim, H., Hong, S.: Novel key recovery attack on secure ECDSA implementation by exploiting collisions between unknown entries. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(4), 1–26 (2021). https://doi.org/10.46586/tches.v2021.i4.1-26
13. Joye, M., Yen, S.M.: The Montgomery powering ladder. In: Oswald, E., Kohlbrenner, P. (eds.) CT-RSA 2002. LNCS, vol. 2271, pp. 291–305. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36400-5_22
14. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_25
15. Liardet, P.Y., Smart, N.P.: Preventing SPA/DPA in ECC systems using the Jacobi form. In: Kocher, D., Katz, R. (eds.) CHES 2001. LNCS, vol. 2162, pp. 137–145. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44709-1_32
16. Möller, B.: Securing elliptic curve point multiplication against side-channel attacks. In: Fossorier, M.P.C., Hori, H., Matsumoto, T., Norishige, S. (eds.) ICICS 2001. LNCS, vol. 2229, pp. 309–320. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45439-X_22
17. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Math. Comput. **48**(177), 243–264 (1987). https://doi.org/10.1090/S0025-5718-1987-0866113-7
18. Okeya, K., Takagi, T.: The width-w NAF method provides small memory and fast elliptic scalar multiplications secure against side channel attacks. In: Koç, C.K., Naccache, D., Pachman, C.T. (eds.) CT-RSA 2003. LNCS, vol. 2612, pp. 256–272. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36563-X_23
19. Perin, G., Chmielewski, L., Batina, L., Picek, S.: Keep it unsupervised: horizontal attacks meet deep learning. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(4), 1–24 (2021). https://doi.org/10.46586/tches.v2021.i1.343-372
20. Roche, T., Imbert, L. and Lomné, V., 2019, November. Side-channel attacks on blinded scalar multiplications revisited. In International Conference on Smart Card Research and Advanced Applications (pp. 95-108). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-42068-0_6
21. Suchanek, V., Jancar, J., Kvapil, J., Svenda, P., Chmielewski, L.: ECTester: reverse-engineering side-channel countermeasures of ECC implementations. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2025**(4), 290–316 (2025). https://doi.org/10.46586/tches.v2025.i4.290-316
22. N. Howgrave-Graham and N. P. Smart, "Lattice Attacks on Digital Signature Schemes," *Designs, Codes and Cryptography*, vol. 23, no. 3, pp. 283–290, 2001.

A Botan's Compilation Options

Botan library offers a large parametrization in the compilation process. The `Makefile` is created using the provided `configure.py`:

```
python3 ./configure.py \
  --os=android \
  --cpu=arm64 \
  --cc-bin=aarch64-linux-gnu-g++ \
  --ar-command=aarch64-linux-gnu-ar \
  --without-os-feature=threads,arc4random \
  --disable-shared-library
```

B Point Coordinates Leakages

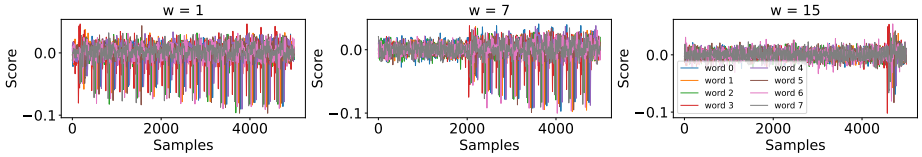


Fig. 14. Real device: leakages during `ct_select` for window values $w = 1, 7$ and 15.

C MLP Architectures and Parameters

Table 1. MLP architecture and training parameters for the second attack (real device).

```
MLP(
  (linear1): Linear(in_features=1000, out_features=1000)
  (linear2): Linear(in_features=1000, out_features=500)
  (linear3): Linear(in_features=500, out_features=200)
  (linear4): Linear(in_features=200, out_features=16)
  (act_func): ReLU()
)
```

Parameter	Value
Batch size	1000
Learning rate	0.0005
Loss function	MSELoss
Optimizer	RAdam
Epochs	20

Table 2. MLP architecture and training parameters for the third attack (emulation).

```
MLP(
  (linear1): Linear(in_features=1400, out_features=100)
  (linear2): Linear(in_features=100, out_features=100)
  (linear3): Linear(in_features=100, out_features=100)
  (linear4): Linear(in_features=100, out_features=100)
  (linear5): Linear(in_features=100, out_features=100)
  (linear6): Linear(in_features=100, out_features=2)
  (act_func): ReLU()
)
```

Parameter	Value
Batch size	200
Learning rate	0.00005
Loss function	MSELoss
Optimizer	RAdam
Epochs	100

Table 3. MLP architecture and training parameters for the third attack (real device).

```
MLP(
  (linear1): Linear(in_features=5040, out_features=200)
  (linear2): Linear(in_features=200, out_features=200)
  (linear3): Linear(in_features=200, out_features=200)
  (linear4): Linear(in_features=200, out_features=200)
  (linear5): Linear(in_features=200, out_features=2)
  (act_func): ReLU()
)
```

Parameter	Value
Batch size	1000
Learning rate	0.000001
Loss function	MSELoss
Optimizer	AdamW
Epochs	150