

ECTester: Reverse-engineering side-channel countermeasures of ECC implementations

Vojtech Suchanek, Jan Jancar, Jan Kvapil, Petr Svenda and Łukasz Chmielewski

Masaryk University, Brno, Czechia

Abstract. Developers implementing elliptic curve cryptography (ECC) face a wide range of implementation choices created by decades of research into elliptic curves. The literature on elliptic curves offers a plethora of curve models, scalar multipliers, and addition formulas, but this comes with the price of enabling attacks to also use the rich structure of these techniques. Navigating through this area is not an easy task and developers often obscure their choices, especially in black-box hardware implementations. Since side-channel attackers rely on the knowledge of the implementation details, reverse engineering becomes a crucial part of attacks.

This work presents **ECTester** – a tool for testing black-box ECC implementations. Through various test suites, **ECTester** observes the behavior of the target implementation against known attacks but also non-standard inputs and elliptic curve parameters. We analyze popular ECC libraries and smartcards and show that some libraries and most smartcards do not check the order of the input points and improperly handle the infinity point. Based on these observations, we design new techniques for reverse engineering scalar randomization countermeasures that are able to distinguish between group scalar randomization, additive, multiplicative or Euclidean splitting. Our techniques do not require side-channel measurements; they only require the ability to set custom domain parameters, and are able to extract not only the size but also the exact value of the random mask used. Using the techniques, we successfully reverse-engineered the countermeasures on 13 cryptographic smartcards from 5 major manufacturers – all but one we tested on. Finally, we discuss what mitigations can be applied to prevent such reverse engineering, and whether it is possible at all.

Keywords: elliptic-curve cryptography · black-box implementations · testing · vulnerabilities · reverse-engineering · ECDH · ECDSA

1 Introduction

Elliptic curve cryptography (ECC) implementations have been hard hit by bugs and vulnerabilities, likely due to their complexity. Focusing just on input validation in the elliptic curve Diffie-Hellman protocol (ECDH) alone, there has been the small-subgroup attack [LL97a], twist curve attack [FLR⁺08], invalid curve attack [BMM00; ABM⁺03] and degenerate curve attack [NT16]. TLS servers and libraries in the wild were found vulnerable to the invalid curve attack in 2015 [JSS15]. The same then happened to JSON Web Token implementations in 2017 [Ngu17; San17a]. Staying with ECDH, there were several cases where an innocuous miscomputation bug lead to key-recovery, first in OpenSSL in 2012 [BBP⁺12], and then in the Go standard library in 2017 [Val17].

Expanding the lens to include ECDSA and similar signature schemes, we see more issues. Input validation issues lead to such curious cases, where the (0,0) signature was deemed valid for any message by any key. This issue was dubbed “psychic signatures”, discovered in OpenJDK in 2022 and surfaced both before and after in cryptocurrency-related implementations [Mad22; NCC21; Ngu21]. Failure to properly validate certificate contents lead to the “CurveBall” vulnerability in Microsoft’s CryptoAPI, which allowed trivial certificate spoofing (CVE-2020-0601). Finally, we invite an interested reader to examine the list of bugs found by the CryptoFuzz project [Vra19; Moz24].

This unending stream of issues and vulnerabilities raises the question: *Do current implementations of ECC properly defend against known vulnerabilities?* Bug recurrence, i.e., a regression, is quite common in software development, as is bug co-occurrence, as demonstrated by the several different implementations vulnerable to invalid curve attacks at different points in time. The efforts that uncovered the above issues were usually focused only on a single or a few implementations, with the exception of the CryptoFuzz project. However, all of them were focused on open-source software implementations. To answer the posed question, we systematically test open-source ECC libraries and smartcards against a plethora of tests constructed from known attacks in our tool **ECTester**.

Recent work [JSS⁺24] has shown that the space of all possible ECC implementations is vast enough to warrant reverse engineering. All the possible combinations of different coordinate systems, formulas, scalar multipliers, and their parameters yielded almost 140 thousand configurations. Additionally, their analysis of open-source ECC libraries has shown that this is not true just in theory, but that developers do leverage all of the options and even extend them by modifying existing techniques. The authors proposed several methods for reverse-engineering ECC implementations using side-channel attacks with a high success rate, though *not* applicable to implementations using scalar randomization.

This raises yet another question: *Can the behavior of implementations under tests be used for reverse-engineering?* We answer this in the positive, by designing several techniques for reverse engineering scalar randomization countermeasures that are able to distinguish between group scalar randomization, additive, multiplicative, or Euclidean splitting. The techniques do not require side-channel measurements and are immune to other side-channel attack countermeasures. Furthermore, the techniques are able to recover the random mask size, and with some probability also the mask value in case of two of the countermeasures. Such mask value recovery on black-box targets allows for more side-channel analysis than was possible before, including mounting the learning phase of profiled attacks. Our techniques question the practice of rewarding information hiding and security by obscurity in security certifications such as Common Criteria [CC; JIL].

Contributions.

- The **ECTester** tool¹ for testing of ECC libraries and JavaCards, supporting 12 popular libraries and any JavaCard, offering a unified API for testing. Released under a permissive open-source license. Its repository is available here: <https://github.com/crocs-muni/ECTester>.
- Techniques for reverse-engineering of scalar randomization countermeasures that only require control over the domain parameters, but require no side-channel measurements or faults, and can recover the mask size and value.
- Reverse-engineering of countermeasures on 13 cryptographic smartcards from major manufacturers, many of which are certified under Common Criteria or FIPS 140.

Outline. In Section 2 we give background on elliptic curve cryptography, implementation attacks on ECC as well as side-channel attack countermeasures. Then we present related work on testing cryptographic implementations and reverse-engineering in Section 3. Section 4 presents our methodology for testing ECC implementations, while Section 5 contains a selection of our results. In Section 6 we present our techniques for reverse engineering scalar randomization countermeasures and results on smartcards from major manufacturers. We discuss the implications of our findings and mitigations in Section 7.

Responsible disclosure. We contacted three out of the five manufacturers of smartcards we were able to reverse-engineer and shared our findings with them. Despite trying, we were unable to establish security contacts with the remaining two manufacturers.

¹The tool was previously used in the discovery of the Minerva group of vulnerabilities [JSS⁺20], yet not described. We have expanded it significantly since then, and present its other results.

2 Background

We focus on short Weierstrass curves $y^2 = x^3 + ax + b$ over a prime field \mathbb{F}_p with $p > 3$. The set of points $E(\mathbb{F}_p)$ defined over \mathbb{F}_p form a group with neutral element \mathcal{O} . ECC protocols work with curves that contain a large subgroup of prime order n and generator G . Denote h the cofactor, i.e., $\#E(\mathbb{F}_p) = nh$. Assuming that n and h are coprime then any point on $P \in E(\mathbb{F}_p)$ can be expressed as the sum $P_n + P_h$ of points of orders n, h respectively, with either possibly equal to \mathcal{O} .

The group addition $P + Q$ is defined by rational functions (also called the addition law) in the coordinates of the points P and Q . Usually, points on an elliptic curve are represented using some form of projective coordinates (e.g., Jacobian, modified Jacobian, standard projective, etc.). In these coordinate systems, there exist addition laws defined by polynomial functions that allow to avoid costly divisions. Unfortunately, every addition law has a pair of exceptional points in $(E \times E)(\overline{\mathbb{F}_p})$ on which the law does not work. There are two approaches to solving this issue. First, one can use at least two addition laws to cover all the necessary points, e.g., an add formula and a doubling special case. Second, some addition laws [RCB16] have been designed to have the exceptional points outside of the main subgroup of order n . The Explicit Formulas Database (EFD) [BL07] collects formulas of various coordinate systems for addition as well as for the special case of doubling ($P = Q$), differential addition, ladder, and others.

The addition naturally defines scalar multiplication $kP = P + \dots + P$. Similarly as for the formulas, the literature contains numerous algorithms for scalar multiplication. They usually scan through the representation of the scalar k and perform additions corresponding to the values of the digits. The double-and-add multiplier scans the bits of k in either left-to-right (LTR) or right-to-left (RTL) fashion, adding P to an accumulator depending on the bits. Since the subtraction of a point has essentially the same cost as addition, a speed-up can be gained by expressing the scalar in the non-adjacent form (NAF) with signed bits 1 and -1 . A generalization of this are the window-based scalar multipliers, where the scalar is split into windows of size w and for each window a precomputed multiple mP , $m < 2^w$ is added to the accumulator. The window multipliers are either sliding or fixed, which describes the relationship of the window to the scalar bits. The comb multipliers follow a similar idea as the window-based with the difference that windows are not consecutive subsequences of bits of k , but are distributed in a regular comb-like pattern across the whole binary representation of k . Ladder multipliers scan the bits of k too, but keep a pair of accumulators and update them using a special ladder formula. See [HMOV04] for a detailed overview of popular scalar multipliers.

2.1 Weak curves

The underlying hard problem that forms the basis for all classical ECC protocols is the elliptic curve discrete logarithm problem (ECDLP): Given a multiple of the generator $P = kG$, find k . If the base field \mathbb{F}_p is large enough (i.e., p has > 160 bits), the problem is hard in general. However, there are known classes of weak curves with easier ECDLP:

- Curves with smooth cardinality can be attacked by the Pohlig-Hellman attack that breaks the ECDLP for each factor of the cardinality [PH78].
- Anomalous curves, for which $n = p$, are vulnerable to the Semaev-Satoh-Araki-Smart attack [Sem98; TA98; Sma99].
- When the embedding degree (the order of $n \in \mathbb{F}_p^\times$) is small (< 20) the Menezes-Okamoto-Vanstone attack can be used to reduce the ECDLP to an easier finite field DLP [MVO91].

2.2 ECC protocols

Private keys in ECC are represented as scalars $d \in \mathbb{Z}_n$ and the corresponding public key for each d is the point $P = dG$. Recovering the private key d from the public key P is protected by the ECDLP. The targets of our testing are the elliptic curve Diffie-Hellman key exchange (ECDH) protocol, the digital signature algorithm ECDSA, and their variants.

The ECDH protocol can be divided into two subroutines. Using the **Keygen** subroutine, both parties, A and B , generate their private keys d_A, d_B as random numbers from the interval $[1, n)$, where n is the order of the large subgroup generated by G . Then they compute and exchange the corresponding public keys $P_A = d_A G, P_B = d_B G$. In the **Derive** subroutine, both parties compute $S = d_B P_A = d_A P_B$. The resulting shared secret is the value $\text{KDF}(S)$, where KDF is an appropriate key derivation function, usually some hash of the x -coordinate of the point S . Furthermore, there is *ephemeral* ECDH where a keypair is used only once and *static* ECDH where a keypair is reused.

Apart from the Keygen, the ECDSA protocol is based on two subroutines: **Sign** and **Verify**. To sign a hash of a message $H(m)$, the owner of the private key d generates a random nonce k and computes $R = kG, r = x_R \pmod{n}$ and $s = k^{-1}(H(m) + dr) \pmod{n}$. The signature is the pair (r, s) that can be verified by computing the x -coordinate x of the point $H(m)s^{-1}G + rs^{-1}P$ and checking that $x \pmod{n}$ is equal to r .

2.3 Public key validation

Both parties in the ECDH protocol take as an input the public key P of the other party and perform a scalar multiplication dP using their private key d . To protect d , it is crucial to properly validate the point P . According to several major ECC standards (SECG SEC1 [SEC09], NIST SP 800-56 [NIST18; NIST23], ANSI X9.62 [ANSI05], IEEE P1363 [IEEE00]), the validation is composed of three steps.

$P \neq \mathcal{O}$. While every affine point can be represented using the x, y coordinates, for the infinity point \mathcal{O} one needs a projective coordinates $(X : Y : Z)$ in which \mathcal{O} lies in the projective plane given by $Z = 0$. The precise representation depends on the coordinate system (e.g., $\mathcal{O} = (0 : 1 : 0)$ for the standard projective and $\mathcal{O} = (1 : 1 : 0)$ for the Jacobian coordinates). The standards SECG, NIST, X9.62 define the zero byte string as \mathcal{O} and any affine point has the first byte nonzero (indicating the potential compression). Thus, checking that $P \neq \mathcal{O}$ is trivial.

P lies on the curve. The public key point contains the affine coordinates x, y in either compressed or uncompressed form. The implementation should check that x, y represent elements of the finite field \mathbb{F}_p (i.e., check that $x, y \in [0, p)$). Any pair (x, y) that then satisfies the curve equation $y^2 = x^3 + ax + b$ is an affine point on the curve. Failure to test this can lead to the invalid or twist curve attack (see Section 2.5).

P is in the correct subgroup. The point P should lie in the subgroup of order n . Otherwise the small subgroup attack is applicable for ECDH and a number of bits of the private key can be recovered depending on the size of the cofactor h . There are four common ways how implementations deal with this after receiving an affine point P :

- Compute $R = nP$ and reject the point if $R \neq \mathcal{O}$. This is recommended by several standards including NIST FIPS, SECG, X9.62, and P1363.
- Compute $R = (h^{-1} \pmod{n})hP$ and proceed the protocol with R . This computation clears out the cofactor part of the point, i.e., if $P = G + P_h$, then $(h^{-1} \pmod{n})h(G + P_h) = G + (h^{-1} \pmod{n})\mathcal{O} = G$. Depending on the protocol, the details of this might differ. For instance, in the computation of the shared secret kP in the Diffie-Hellman protocol, we can replace k with $((h^{-1} \pmod{n})hk)$. The costly inverse h^{-1} can be precomputed during the domain parameter set up.

- Use a modified version of the protocol which deals with the cofactor as part of the protocol. This is the approach of the Cofactor Diffie-Hellman protocol. The shared secret is khP instead of kP . Similarly, the popular X25519 protocol assumes that the private key is a multiple of the cofactor $h = 8$, i.e., $k = 8k_0$ for some k_0 .
- Ignore the subgroup check. This is not necessarily a sign of negligence. All of the above validation options are expensive (i.e., as expensive as the ECDH secret derivation). Moreover, the most popular NIST curves have cofactor $h = 1$. Finally, the small subgroup attack (as described in Section 2.4) that is possible because of the lack of validation can only recover a few bits of the private key. Note, however, that other variants of the attack [LL97b] aim for key reuse rather than key recovery.

2.4 Small subgroup attacks

The small subgroup attacks target the ECDH protocol that works over a curve with a nontrivial cofactor $h \neq 1$. The attacker sends to the victim B their public key P_A as part of the ECDH protocol. The attacker assumes that the victim omits the check that P_A lies in the subgroup of order n . If the point P_A has order h , then $d_B P_A \in \{P_A, \dots, hP_A\}$, where $d_B P_A$ is computed by B with their private key d_B . As a result, the shared secret $\text{KDF}(d_B P_A)$ can be then enumerated (and checked with the actual shared secret computed by victim) by the attacker which leaks $d_B \pmod{h}$ to them [LL97a].

Contrary to the statement on the SafeCurves website [BL17; BL24], it is not enough to reject points P_A with $hP_A = \mathcal{O}$, though it would be faster than computing and rejecting points with $nP_A \neq \mathcal{O}$. The attacker can send a mixed order point $P_A = G + P_h$, where P_h is a point of order h . The party B then computes $d_B P_A = d_B G + d_B P_h = P_B + d_B P_h$. The space of possible shared secrets $\{\text{KDF}(P_B + P_h), \dots, \text{KDF}(P_B + hP_h)\}$ can be easily enumerated, which, again, leaks $d_B \pmod{h}$.

2.5 Invalid curve attacks

The invalid curve attack targets the ECDH protocol with the assumption that the victim implementation does not check that the point lies on the curve. The attack is then based on the observation that addition and doubling formulas often do not contain the Weierstrass parameter b (the addition formulas usually do not contain the parameter a either). This means that the algorithm for scalar multiplication using these formulas correctly computes kP for a point P on any curve with the same parameter a .

The victim expects a public key point on a curve $E : y^2 = x^3 + ax + b$. The attacker generates and sends a point P_A on a different curve $E' : y^2 = x^3 + ax + b'$, with a small subgroup of order h . The victim computes the shared secret on the curve E' . Similarly as in the small subgroup attack, since h is small, the possible values for the derived secret computed by the victim can be then enumerated. This gives the attacker $d_B \pmod{h}$. The attacker repeats this for several curves obtaining $d_B \pmod{h_i}$ for each h_i , until the whole private key d_B is recovered using the Chinese remainder theorem [BMM00; ABM⁺03].

The degenerate curve attack [NT16] also tricks the victim to perform the ECDH protocol on a different group. The difference from the invalid curve is that this group is not an elliptic curve group, rather the multiplicative group of the underlying base field.

2.6 Point compression

To save space, implementations often store and transmit points (public keys or generators) in compressed form. For any point P , the x_P coordinate determines the y_P coordinate up to a sign through the curve equality $y_P^2 = x_P^3 + ax_P + b$. Hence, the point P can be represented using x_P and a single bit signaling the sign of y_P . The X9.62 standard

defines the compressed format specifically as $PC|x_P$, where PC is a single byte with $PC = 2$ if the least significant bit of y_P is 0 and $PC = 3$, otherwise. The uncompressed form is represented as $4|x_P|y_P$, with 4 represented by a single byte and the coordinates zero-padded to the field length in bytes.

2.7 Side-channel countermeasures

Let us now briefly present several side-channel attack countermeasures important to our work. Namely, several scalar randomization countermeasures that will be the targets of our reverse-engineering as well as the use of dummy operations, complete formulas and point blinding. See [DGH⁺13] for a detailed overview of this area.

Dummy operations. Some scalar multipliers are not regular, i.e., the sequence of `add` and `dbl` formula applications depends on the processed scalar. A simple double-and-add multiplier is an example of this. To protect these multipliers from simple power analysis, dummy operations can be used. This transforms the simple double-and-add multiplier to double-and-add-always. This has the effect of introducing dummy values in the scalar multiplication that are not used to compute the final result.

Complete formulas. While complete formulas are not a side-channel countermeasure as such, they allow simpler exception-less handling of the point at infinity. For example, avoiding Minerva-style [JSS⁺20] leakage of the bit-length of the scalar in a side-channel-free manner is much easier with complete formulas. Furthermore, using complete formulas, scalar multipliers can safely handle scalars larger than the curve order without issues due to the potential to reach the point at infinity.

Scalar randomization. There are several side-channel countermeasures that randomize the scalar used in scalar multiplication. They sample some additional randomness and expand a single scalar multiplication into several multiplications or a single larger one. Figure 1 shows an overview of the techniques. We consider four popular techniques: group scalar randomization (GSR) [Cor99], additive [CJ01], multiplicative [TB02], and Euclidean [CJ03] splitting. Note that the countermeasures have different *expansion ratio*, the amount and size of scalar multiplications they expand a single l bit scalar multiplication into. For GSR and multiplicative splitting, assuming they sample b bits of randomness, the ratios are $l \rightarrow 1 \times (l + b)$ and $l \rightarrow (1 \times b) + (1 \times l)$, respectively. For additive splitting the ratio is $l \rightarrow 2 \times l$ (i.e., two l bit scalar multiplications). For Euclidean splitting, the ratio is $l \rightarrow 3 \times \frac{1}{2}l$.

Coordinate and curve randomization. Apart from randomizing the scalar, one can also randomize the point coordinates, as a single affine point can be represented by many projective ones. In standard projective coordinates, point $P = (X : Y : Z)$ can be randomized using $\lambda \xleftarrow{\$} \mathbb{F}_p^*$ as $(\lambda X : \lambda Y : \lambda Z)$. This works analogously in other coordinate systems. Similarly, one can randomize computation on a whole curve by taking an isogenous curve, transferring the computation there, and transferring the result back.

Point blinding. Finally, the point blinding countermeasure, of which there are several variants, modifies the scalar multiplication of $[k]P$ by adding a random point R to point P before the start of the multiplication and then subtracts $S = [k]R$ at the end. The points R and S may be precomputed for a fixed keypair and adjusted after each multiplication. Some variants of this countermeasure subtract the point S directly during the main scalar multiplication loop and do not require an additional operation.

3 Related work

There is relatively little work in the area of testing of cryptographic implementations that is directly related to our approach.

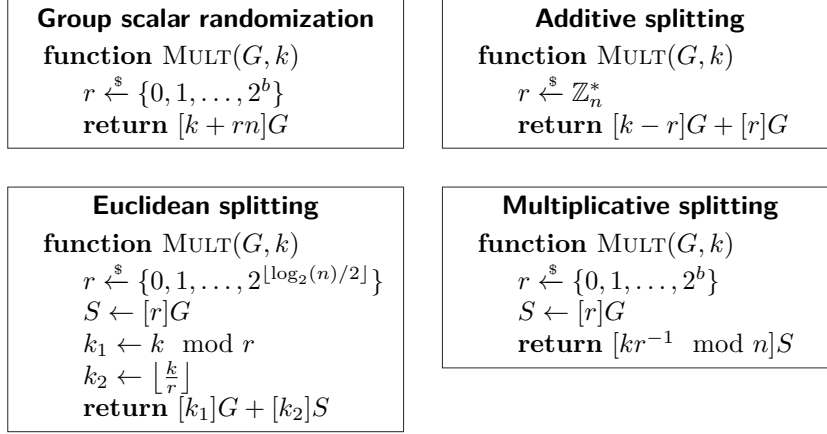


Figure 1: An overview of scalar randomization countermeasures, showing the multiplication of point G with scalar k while sampling a random mask of b bits.

- Firstly, one may get assurances of security through code review and analysis. However, this is only applicable in a white-box scenario with available source code, whereas our approach works also for black-box implementations.
- Secondly, one may perform implementation testing, for example using test-vectors such as the Wycheproof project [C2S16] or by fuzzing such as the Cryptofuzz project [Vra19; Moz24]. Our approach in **ECTester** fits this category. In contrast to Wycheproof, which recently moved to only focus on aggregating test-vector data, we support testing many native libraries directly as well as any compatible JavaCards. The future of the Cryptofuzz project is uncertain, as its original author took down the repository and only a fork under the Mozilla Security organization is available.
- Thirdly, one can test artifacts produced by the implementation (e.g., keys). This is the approach taken by Google’s project Paranoid [Goo16] and also in the *CurveSwap* work [VSS⁺18]. Our approach is able to exercise the target implementation and gain more information on it than possible by purely analyzing its keys.

In the area of reverse-engineering of public key cryptography implementations there have been some notable examples of both manual and automated reverse-engineering. Amiel et al. [AFV07] presented a technique for reverse-engineering the word size and modular multiplication algorithm in RSA implementations. Remarkable examples of manual reverse-engineering can be found in the “Side Journey to Titan” [RLM⁺21] and “EUCLEAK” [Roc24] works, which demonstrate complex attacks on ECC implementations in real-world hardware.

The recently introduced **pyecsca** [JSS⁺24] tool is able to automatically reverse-engineer ECC implementation details via side-channel attacks. While it targets the scalar multiplier and addition formulas used by the implementation, its reverse-engineering does not work in the presence of scalar randomization countermeasures. In contrast, our techniques are able to reverse-engineer the scalar randomization countermeasures and do not require side-channel measurements. We utilize the **pyecsca** toolkit for simulations and demonstration.

4 Methodology

In this section we present our methodology for testing black-box elliptic curve cryptography implementations using our **ECTester** tool. We do not discuss the reverse-engineering of randomization countermeasures in this section and leave it for Section 6.

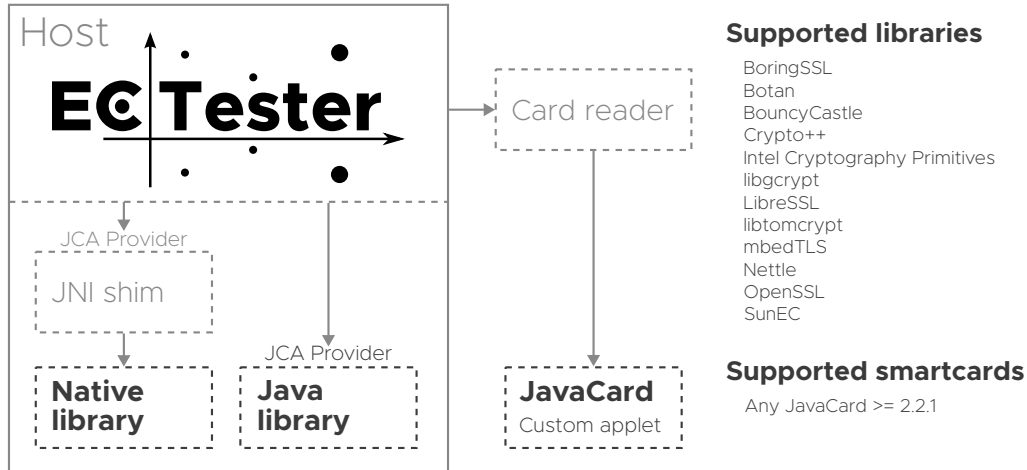


Figure 2: Architecture of the ECTester tool, which is able to interact with ECC implementations in Java libraries, native libraries through JNI shims, and JavaCards. Targets are displayed in bold.

4.1 Overview

The **ECTester** tool comprises of several components, see Figure 2:

Standalone The standalone component focuses on testing ECC libraries, both native and Java-based ones. It uses the Java Cryptography Architecture (JCA) providers of the Java-based ECC libraries to interact with them. However, the large variety of native libraries we target does not offer any common API we could use. Thus, for each native library we implement a *shim* using the Java Native Interface (JNI) that uses the library to offer a common JCA provider interface. The standalone component offers a unified command-line interface for working with, and testing, the libraries.

Reader The reader component focuses on testing JavaCard-based smartcards. To do so, it requires that a custom applet (see Applet component below) is installed on the JavaCard. It offers a similar command-line interface as the standalone component.

Applet The applet component contains the applet code that is loaded on tested JavaCards and provides a rich interface to their ECC implementations. It uses only standard JavaCard APIs and not the proprietary ones as those are often only available under a non-disclosure agreement with the card vendor. Since this component is compiled for JavaCards it needs to conform to the JavaCard subset of Java.

Common The common component contains all of the test data: elliptic curves, points, private and public keys, signatures, and test results. It also contains common functionality shared between the reader and standalone components, such as I/O or test evaluation. It supports test result output in YAML/XML and rich text formats, allowing for post-processing and further analysis.

Finally, all of the components are complemented by a set of Jupyter notebooks for data and results analysis as well as visualization. Currently, the tool supports 10 native libraries and 2 Java ones, listed in Figure 2. Adding additional libraries involves writing a simple shim which implements a conversion layer between **ECTester** and the native library. However, as libraries evolve over time and their API changes, we found that we need to update these shims to new library versions quite often. Furthermore, the changes to the shim necessary to make it work with a new library version may not be backwards

compatible with old library versions. We use the `nix` tool and `nixpkgs` repository to enable testing of historic and current library versions and simplify the library build system.

4.2 Test suites

ECTester contains a wide range of test-suites which we document here. A full listing of the tests is out-of-scope of this work, but can be found in our [repository](#).

Default: Tests support for ECC and the presence of any default curves on the target. Given that we also target JavaCards, these might not be present or the target might not even support ECC. It also tests keypair allocation, generation, ECDH, and ECDSA. ECDH is first tested with two valid generated keypairs, then with a compressed public key to test support for valid compressed points.

Test-Vectors: Tests ECDH using known test vectors provided by NIST [Kel11], SECG [Cer99], and Brainpool [Har13; ML13]. This test-suite also performs a cross-validation test of ECDH and ECDSA, using the target and validating its output using a bundled version of the BouncyCastle library.

Compression: Tests support for the compression of public points in ECDH as specified in ANSI X9.62 including the compressed and hybrid form. Also tests target response to a hybrid point with a wrong y coordinate and to the point at infinity (as public key in ECDH). Tests ECDH with an invalid compressed point, where x does not lie on the curve.

Miscellaneous: Tests ECDH and ECDSA over weak curves such as super-singular curves, anomalous curves, or Barreto-Naehrig curves with small embedding degree and CM discriminant. Also tests ECDH over MNT curves, M curves, and Curve25519 transformed into short Weierstrass form.

Signature: Tests ECDSA verification, with well-formed but invalid signatures and a wide range of malformed signatures. This includes $r, s \in \{0, 1, n\}$, taking r, s random, malformed ASN.1 format or a negated public key point. This test suite also tests signature malleability, i.e., whether a correct signature modified by adding the curve order to one of the values r, s still verifies.

Wrong: Tests behavior on ill-defined curves with parameters that do not follow the standard definition of a curve. The base-field is invalid, i.e., the number defining the prime field is not a prime or the polynomial defining the binary field is reducible. Tests also include the generator point not on the curve or not a valid affine point. Tests include a curve where the order or the cofactor is zero or one.

Composite: Tests using various composite order curves, including pseudoprime orders.

Invalid: Tests the invalid curve attack [BMM00; JSS15] using known named curves from several categories (SECG, NIST, and Brainpool) against pre-generated invalid public keys. Similar tests are implemented for the **Twist** curve attack and for the **Degenerate** curve attack in respective test-suites.

Cofactor: Tests the small-subgroup attack, i.e., whether the target correctly rejects points that lie on the curve but not on the subgroup generated by the specified generator during ECDH. This is done with curves where the cofactor subgroup has small order, then with curves that have order equal to the product of two large primes. The test sets the generator with order of one prime and tries points on the subgroup of the other prime order.


Edge-Cases: Tests various inputs to ECDH which may cause an implementation to achieve a certain edge-case state during ECDH. Some of the data is from Paranoid Crypto [Goo16] or Wycheproof [C2S16]. Tests include CVE-2017-10176 and CVE-2017-8932 and an OpenSSL modular reduction bug presented in [BBP⁺12]. Various custom edge private

key values are also tested. CVE-2017-10176 was in implementation issue in the SunEC Java library and NSS (CVE-2017-7781), that caused the implementation to reach the point at infinity during ECDH computation [San17b]. CVE-2017-8932 was an implementation issue in the Go standard library, in particular its scalar multiplication algorithm on the P-256 curve which leaked information about the private key [Val17].

5 Results

In this section we present results obtained from running **ECTester** test-suites on supported libraries and several JavaCards. We do not include results from every test-suite due to space and pick only “interesting” results from the point-of-view of an attacker. For an overview of the tested cards, see Table 1. For an overview of the tested library versions (and supported versions) see Table 6.

Table 1: Smartcards analyzed in this work. The CPLC column presents the ICFabricator, ICType, OperatingSystemID, and OperatingSystemReleaseDate values from the smartcard.

	Manufacturer	Model	Chip	CPLC
N1	NXP	JCOP J3A081*	P5	4790.5168.4791.0078
N2	NXP	JCOP J2D081*	P5	4790.5167.4791.0078
N3	NXP	JCOP21 J2E145G*	P5	4790.5167.4791.2348
N4	NXP	JCOP3 J3H145*	P60	4790.0503.8211.6351
N6	NXP	JCOP4 J3R180*	P71	4790.D321.4700.0000
N9	NXP	JCOP31 J3A081*	P5	4790.5040.4791.8102
I1	Infineon	SECORA IDS SLJ52GDT120CS*	-	4090.1912.4090.9078
I2	Infineon	CJTOP SLJ52GLA080AL*	-	4090.7165.544C.2151
A1	Athena	IDProtect†	AT90	4180.010B.8211.0352
G1	G&D	Smartcafe 7.0	-	0005.0056.D001.4212
G2	G&D	Smartcafe 6.0†	-	4790.5037.1671.1146
S2	TaiSYS	SIMoME†	-	FFFF.FFFF.FFFF.FFFF
F1	Feitian	JavaCOS A22 CR*	-	4090.7892.86AA.7068
F2	Feitian	JavaCOS JC30M48 CR	-	-

*/† Denotes Common Criteria or FIPS 140 certified smartcards, respectively.

The identification of the chip, or even exact smartcard model, or certification status in Table 1 presents our best-effort and might contain mis-identifications. Precisely identifying the exact smartcard model and certificate coverage without access to manufacturer documentation is challenging. We used the sec-certs project to speed-up our search of certification documents [JJS⁺24].

5.1 Test-vector failures

We encountered several unexpected test-vector failures in our testing of JavaCards. The **A1** card produced a wrong ECDH secret in the **brainpoolP512r1** test, as well as the **P-521 DHC** test from NIST. Similarly, it failed to produce the correct shared secret on the **secp521r1**, **brainpoolP512r1**, and **brainpoolP512t1** curves. The **G1** card produced wrong ECDH secrets on 512-bit curves, yet handled 521-bit ones fine. The **G2** card outputs only 20 bytes of the derived secret if the used keypair was generated on the card, even if the requested ECDH version is plain (not hashed) and the curve is larger.

5.2 Timing leakage

The **ECTester** tool was used to discover a group of timing vulnerabilities in ECDSA known as Minerva [JSS⁺20]. This shows the versatility of the tool in uncovering diverse issues like timing attacks which we do not focus on in this work. The functionality of the tool, with respect to test suites and usability was expanded heavily since then.

5.3 Psychic signatures

The “psychic signatures” bug, named after Doctor Who’s psychic paper, was a bug in ECDSA verification in Java 15–18 that allowed a signature of r, s both equal to zero to pass as valid for any message. This bug was discovered in the Java case by Madden [Mad22] in 2022 and found in other cryptocurrency-related ECDSA implementations in 2021 [NCC21]. A test for such a case was present in the **Signature** test-suite in **ECTester** since 2018, however the tool – written in Java – only supported Java 8 at the time. Support for newer Java versions was added in 2024.

5.4 Domain parameter validation

The **Composite** test suite tests the set up of elliptic curve parameters with composite order n . Curves with composite order are vulnerable to the Pohlig-Hellman attack and any implementation that supports custom curves and wants to avoid it should test primality of the order or use standard named curves. Column ‘prime n ’ in Table 2 shows the results for twelve popular ECC libraries and thirteen JavaCards. For eleven cards and eight libraries, we were able to set-up a composite order curve and perform either ECDSA or ECDH. For the rest, we failed to do so. The results for the **A1**, **G1** and **I2** cards agree with the findings from [SJS20] that both **A1**, **I2** contain some primality tests on n and **G1** does not.

Next, we tested the validation of the generator point. Any generator of the prime subgroup should have prime order n . Since any point on a prime order curve will be a generator, we selected a curve with cofactor $h > 1$ that we concealed from the implementation (i.e., set to $h = 1$ in the parameters). We set the curve order to n and tested ECDH and ECDSA with a generator of order nh . Additionally, we used the secp256r1 curve and changed the claimed prime order n to another prime. We were successful for all except for two cards (column ord G in Table 2) and for five libraries. Inspection of the code showed that the three libraries that support custom curves and rejected the generator (OpenSSL, LibreSSL and Intel Crypto) correctly check the order of the generator.

5.5 Cofactor validation

The **Cofactor** test suite tests how implementations deal with public key points in the wrong subgroup. This is relevant for the Derive subroutine in ECDH and Verify in ECDSA. We selected curves with cofactor $h > 1$ and gave the implementations public key points of an incorrect order nh (the expected public key should have order n). We considered two options to fully understand the behavior with incorrect options: conceal the cofactor in the curve specification and pretend that $h = 1$ or transparently present the true cofactor h . In the first case, all of the cards except for **A1** and **G1** accepted the point as well as all of the libraries that supported custom curves with the addition of **libtomcrypt** (see Table 2). The **libtomcrypt** library (the latest version v1.18.2) did not support cofactor curves, but it also did not validate that the point is lying on the curve. Hence, a small order point from a different curve could be used. The lack of the check makes the library vulnerable to the invalid curve attack. Furthermore we observed that repeated runs on the cards produced different results. For the fixed input point P of order nh , we were getting results

Table 2: Results of **ECTester** with invalid parameters on popular libraries and JavaCards. The symbols \checkmark , \times denote **ECTester** managed (or failed) to pass a curve with composite order (prime n) or a public key or generator with invalid order (ord P and ord G).

Card	ord P	ord G	prime n	Library	ord P	ord G	prime n
N1	\checkmark	\checkmark	\checkmark	BoringSSL	\checkmark	\checkmark	\checkmark
N2	\checkmark	\checkmark	\checkmark	Botan	\checkmark	\checkmark	\checkmark
N3	\checkmark	\checkmark	\checkmark	BouncyCastle	\checkmark	\checkmark	\checkmark
N4	\checkmark	\checkmark	\checkmark	Crypto++	\checkmark	\checkmark	\checkmark
N6	\checkmark	\checkmark	\checkmark	Intel Crypto	\checkmark	\times	\checkmark
N9	\checkmark	\checkmark	\checkmark	libgcrypt	\times	\times	\times
I1	\checkmark	\checkmark	\checkmark	LibreSSL	\checkmark	\times	\checkmark
I2	\checkmark	\checkmark	\times	libtomcrypt	\checkmark	\times	\times
A1	\times	\times	\times	mbedTLS	\checkmark	\checkmark	\checkmark
G1	\times	\times	\checkmark	Nettle	\times	\times	\times
S2	\checkmark	\checkmark	\checkmark	OpenSSL	\checkmark	\times	\checkmark
F1	\checkmark	\checkmark	\checkmark	SunEC	\times	\times	\times
F2	\checkmark	\checkmark	\checkmark				

of the form $P, P + nP, \dots, P + (h - 1)nP$. This pointed to a randomization of the scalar multiplication that we will explore in the further sections.

Then we repeated the same tests with a point of order nh but with correctly set cofactor h . Surprisingly, all of the cards continued to accept the point. One explanation might be that the user is supposed to correctly choose the correct type of ECDH depending on the cofactor and in case of $h > 1$ use the Cofactor Diffie-Hellman (CDH) protocol. This is what we saw in the **OpenSSL** library, which offers both ECDH and CDH, leaving the choice up to the user by setting the `ecdh-cofactor-mode` flag. **Crypto++** and **Intel Crypto** implement CDH as well and **Botan** uses the inverse version of CDH to be compatible with the regular ECDH. **BouncyCastle** implements the cofactor check, but offers the option to omit it by setting a corresponding flag for the validation function. **MbedTLS** states in the documentation that the cofactor check “is expensive, is not required by standards, and should not be necessary if the group used has a small cofactor”. For **libgcrypt** and **LibreSSL** we were not able to confidently determine their approach.

Next, we tested the implementations with points of small order h . We observed that the libraries and cards very often did not successfully finish with a (correct) ECDH secret given a point of small order as the public key. This is not surprising as when the scalar is divisible by the order, the result of the scalar multiplication is the infinity point that cannot be transformed to an affine form and thus an ECDH secret. If the input point P has order l this happens with probability $\frac{1}{l}$ given random scalars. This is indeed what we saw in most of the libraries (Figure 3). However, three libraries, **libtomcrypt**, **BouncyCastle**, **mbedTLS** had different distributions. For instance, while the rest of the libraries erred in roughly $\frac{1}{3}$ of the cases for points of order 3 and very rarely for points of order 37, **mbedTLS** erred almost always for the same points of order 3 and points of order 37.

Closer inspection of **BouncyCastle** and **mbedTLS** revealed a simple explanation. **BouncyCastle** uses the Window-NAF multiplier for scalar multiplication which begins with the precomputation of the a few small multiples of the input point that are then transformed to an affine form. Similarly, **mbedTLS** uses the Comb multiplier with a precomputation. The transformation to the affine form then causes the errors.

For **libtomcrypt**, the explanation turned out to be in the formula that is used for point addition (`add-1998-hnm`). If either of the input points has the projective z -coordinate equal to zero then the result satisfies the same regardless of the second points. In particular,

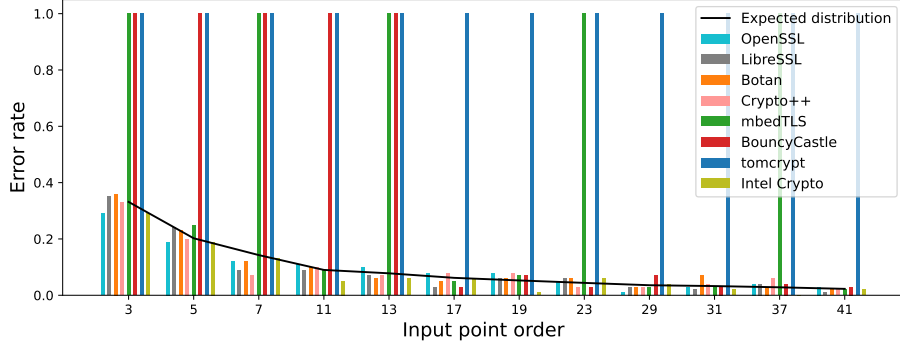


Figure 3: Error distribution of ECDH runs given an input point of low order.

if one of the input points is \mathcal{O} (which is defined as $x = 0, y = 1, z = 0$) then the result also has $z = 0$. This means that if such a point appears during the scalar multiplication, it propagates through the scalar multiplication till the end where the implementation attempts to transform it to an affine point resulting in an error. Since the input point, say P , has low order h , very likely a multiple of the point, mP , with m divisible by h appears during the scalar multiplication. Consequently, the point mP is \mathcal{O} which causes the error.

We observed similar behavior with the cards including **N1** and **N3**, though some of the cards stopped responding and so we did not do further measurements. The frequency of the errors was significantly higher than the expected $\frac{1}{h}$. We concluded that the implementations on the card suffer from the same problem with incomplete formulas as we saw in `libtomcrypt` and they do not handle the infinity point well. Moreover, the errors did not appear deterministically across multiple measurements with the same input including a fixed key. This agreed with the observation that some sort of (scalar) randomization is present on the card that we also saw in the results of the cofactor testing.

As a final note for this part, we will comment on the vulnerability to the small subgroup attack. It is known that a few bits of a private static ECDH key can be recovered using this attack. However, the lack of cofactor validation makes the verification of ECDSA signatures also vulnerable by the following simple attack. To verify a valid ECDSA signature (r, s) with a corresponding public key Q , one checks that the x -coordinate of the point $s^{-1}H(m)G + s^{-1}rQ$ is equal to r modulo n . Let T be a point of order 2 (i.e., we assume that the cofactor is even). Assuming that $s^{-1}r$ is an even number, the point $Q + T$ will also pass as a public key for the verification of the signature (r, s) . We experimentally confirmed this on the **N1**, **N2**, **N3**, **N9**, and **N4** cards. Attempts on **I1**, **I2** failed, and so we concluded that the cards have some check (here we did not conceal the cofactor) or the verification is done in a different way. Note that the lack of cofactor validation for verification means that the signatures are not bound to the public key Q , i.e., the signature scheme is not *strongly binding* as defined in [CGN20].

5.6 $n > p$ overflow

The **Edge-Cases** test suite revealed that the **N4** card did not support all the private keys from the full interval $[1, n - 1]$ for the `secp160r1` curve. This curve has a 160-bit base-field prime $p = 2^{160} - 2^{31} - 1$ and 161-bit order $n = p + 1 + t$ where t has 81 bits. Based on the tests in the suite, we concluded that the private key must have 160-bits for the implementation to accept it. This results in a small bias as the values from the interval $[2^{160}, n - 1]$ are never used. Assuming that the implementation applies the same limit on the signature nonces, this would lead to systematic nonce leakage of 1 bit, though with the negligible probability smaller than 2^{-79} .

6 Reverse-engineering scalar randomization

ECTester results on JavaCards presented in the previous section show that the tested cards usually do not validate the domain parameters and the public point. Incorrect order of the curve, cofactor, or generator with a wrong order can be passed to the card and used in ECC protocols implemented on the card. Observation of the test results already gave us evidence of side-channel countermeasures such as scalar randomization or improper handling of the infinity point. In this section, we push these ideas further and design methods for reverse-engineering of scalar randomization countermeasures (and random mask sizes and values) based on the behavior of the implementation under invalid inputs.

We will target a scalar multiplication kP using an unknown multiplier, where P is a point (either the generator for KeyGen and Sign or the public key point for Derive) and k is the used scalar (either the nonce or the private key). The computation of kP is randomized by an unknown scalar randomization ρ . We denote the randomization as $\rho(k, P)$, where under valid inputs $\rho(k, P) = kP$. We will assume that k and P are either known or can be set, but this will depend on the individual tests presented below. Finally, we will assume that ρ is one of the scalar randomization techniques described in Section 2.7. We will also briefly comment on the point blinding at the end of this section.

The general idea is to set up the curve and the point P such that $\rho(k, P) \neq kP$ and the value of $\rho(k, P)$ leaks information about the randomization ρ . This will very often require to conceal the true subgroup order n and the cofactor h from the implementation. We will denote \bar{n} , \bar{h} , \bar{G} , and \bar{P} the claimed (possibly incorrect) order, cofactor, generator, and public key, respectively. Table 3 summarizes the setting of the parameters for our tests.

Table 3: Setting of ECC parameters for our reverse-engineering of scalar randomization.

Test	Target	$ E(\mathbb{F}_p) $	n^*	\bar{n}	\bar{h}	$\text{ord}(\bar{G})$	$\text{ord}(\bar{P})$	k
$3n$	Derive Sign/Keygen	$3n$	✓	n	1	n $3n$	$3n$ n	fixed any
composite	Derive Sign/Keygen	n	✗	n	1	n	n	fixed any
$k = 10$	Derive	n	✓	n	1	n	n	10
$n + \epsilon$	Derive Sign/Keygen	n	✗	prime $n + \epsilon$	1	n	n	fixed any
EPA	Derive	$373n$	✓	n	1	n	373	fixed

* Denotes whether n is a prime or not.

Table 4: Behavior of our tests with the common scalar randomization techniques.

Test	Mask	GSR	Additive split	Euclidean split	Multiplicative split	None
$3n$		$\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$	$\frac{1}{2}, \frac{1}{2}, 0$	$1, 0, 0$	$\frac{5}{9}, \frac{2}{9}, \frac{2}{9}$	$1, 0, 0$
composite		100%	100%	100%	$< 100\%^\dagger$	100%
$k = 10$		$< 100\%^*$	100%	$< 100\%^*$	100%	100%
$n + \epsilon$	size	Yes	No	No	Yes	-
	value	100%	-	-	$\approx 10\%$	-
EPA		> 0	> 0	> 0	> 0	0

* Depends on the scalar multiplier.

† Depends on the field inversion algorithm.

6.1 Test $3n$

The test uses an elliptic curve of order $3n$, where n is a large prime, and an input point P of order $3n$. We will show that $\rho(k, P) = kP + snP$ for $s \in \{0, 1, 2\}$ for each of the scalar randomization technique ρ . More importantly, the distribution of s is different for each technique ρ , giving us a unique fingerprint for each of them. The test will do multiple measurements and output the distribution (three probabilities for the three possible values of s). This measured distribution is then compared with the theoretical distributions that we will now derive. The summary of the theoretical distributions is shown in Table 4.

For GSR, $\rho(k, P) = (k + rn)P = (k + sn)P$, where s is equal to $r \pmod{3}$ and thus uniformly sampled from $\{0, 1, 2\}$, i.e., each with probability $\frac{1}{3}$. For the additive splitting, $\rho(k, P) = (k - r)P + rP$. If $k > r$ then $\rho(k, P) = kP$. In the opposite case, the negative scalar $(k - r)$ is reduced modulo n and $\rho(k, P) = (k + n)P$. Assuming $k \sim n \sim r$, these two cases should happen with $\frac{1}{2}$ probability each. It could also be the case that the negative integer $(k - r)$ is not reduced (and perhaps the sign of P is changed instead), in which case $\rho(k, P)$ would be always kP . The Euclidean splitting does not use the value of the order n in any way and so $\rho(k, P) = kP$.

Finally, the multiplicative split will satisfy $\rho(k, P) = k'rP$, where $k' = kr^{-1} \pmod{n}$ and r^{-1} is the inverse of $r \pmod{n}$. For some integer t , we can write $\rho(k, P) = (kr^{-1} + tn)rP$. Since P has order $3n$, $\rho(k, P) = kP + snP$ for $s \in \{0, 1, 2\}$, where s satisfies $(kr^{-1} + tn)r = k + sn \pmod{3}$. Since r is a random integer, all three values r , r^{-1} and t follow a uniform distribution modulo 3. Assuming that $k \pmod{3}$ is fixed, $(kr^{-1} + tn)$ follows this distribution as well. Then the distribution of the product $(kr^{-1} + tn)r$, and consequently of s , is 0, 1, 2 with probabilities $\frac{5}{9}$, $\frac{2}{9}$, $\frac{2}{9}$, respectively. Note that this holds for a fixed $k \pmod{3}$ and so for Keygen and Sign, the nonce and private key must be grouped based on the value of $k \pmod{3}$ when observing the distribution.

6.2 Test composite

We can use a composite order curve to detect the inversion $r^{-1} \pmod{n}$ computed in the multiplicative splitting randomization, but not others. If r and n share a nontrivial factor, the computation of $r^{-1} \pmod{n}$ can then either lead to an infinite loop, an error, or an undefined value, as the inverse does not exist. If the extended Euclidean algorithm is used, then the result will be incorrect or produce an error with probability $\frac{\phi(n)}{n}$. If the Fermat's little theorem is used with $r^{-1} = r^{n-2} \pmod{n}$ then it is unlikely that any of the inversions will be correct, though errors are not expected. Hence, the measurement is repeated and the output of the test is the rate of successful results. The rest of the countermeasures do not use the primality of the order, and so they will have a 100% success rate. A success rate lower than 100% is indicative of the multiplicative splitting.

Beware that in Sign this test will detect the inversion of the nonce. To determine whether we are detecting any further inversions (in the multiplicative splitting), we have to consider the expected frequency of errors and compare them with the measurements.

Note that the frequency of the errors can be adjusted for both inversion algorithms. To modify the error rate of the inversion using Fermat's little theorem, we can use Carmichael numbers that will satisfy $r^{n-2} = r^{-1} \pmod{n}$ if and only if r and n are coprime. The probability of error is then the same as for the Euclidean algorithm: $\frac{\phi(n)}{n}$. By selecting n with a different number of factors we can increase or decrease the probability.

6.3 Test $k = 10$

This test is designed for Derive with key $k = 10$ to detect the GSR countermeasure, where $\rho(k, P) = (k + rn)P = (10 + rn)P$. Simulations with the **pyecscsca** tool showed that scalar multipliers tend to compute rnP as an intermediate value while building up the

resulting $10P + rnP$. Naturally, this tends to happen for any small k . This can lead to an error if the implementation does not handle $rnP = \mathcal{O}$ well. At the same time, other countermeasures do not use scalars larger than n and are therefore unlikely to cause a similar error. The only assumption of this test is that we can fix $k = 10$ and so this can be set up on any curve including `secp256r1`, which we used. The result of the test is the rate of the correctly computed shared secrets. However, not every multiplier has to cross the point rnP and so a 100% success rate does not necessarily mean that GSR is not present in the implementation. Other small scalars can be used to increase the confidence.

6.4 Test $n + \epsilon$

The $n + \epsilon$ test is aimed at reverse-engineering the randomization mask used in GSR or in the multiplicative splitting. It works under the assumption that the random mask r is sampled from $[1, 2^b]$ where b is much smaller than $\log_2(n)$. The idea is to set a curve with order n and claim a different order $n + \epsilon$ for small ϵ , where $n + \epsilon$ is a prime to pass any primality test. The GSR randomization then computes $\rho(k, P) = (k + r(n + \epsilon)) = (k + r\epsilon)P = dP$, where $d = k + r\epsilon \pmod{n}$ and P has order n . Since r is significantly smaller than n then the equality $d = k + r\epsilon$ holds over integers, giving us also the mask as $r = \frac{d-k}{\epsilon}$. The only unknown (besides the mask) is the discrete logarithm d of $\rho(k, P)$ with respect to P . If we set a weak curve with composite n , we can easily find the discrete logarithm d and find the mask r . We used a curve with order n equal to a product of 32-bit primes. Note that we need access to the result point $(k + \epsilon r)P$, which holds for Keygen, Sign, and plain Derive where the point can be recovered from the shared secret.

The multiplicative splitting behaves with the $n + \epsilon$ test in the following way. The randomization is $\rho(k, P) = k'rP$, where $k'r = k \pmod{n + \epsilon}$. The last modular equality can be written as $k'r = k + t(n + \epsilon)$ for some integer t . We estimate the size of t by comparing the sizes of both sides of the equality. Specifically, $k'r \sim tn$ where $k' < n + \epsilon$ and so t is expected to be smaller than r and consequently significantly smaller than n . If we find the discrete logarithm d of $k'rP$ with respect to P , we get $d = k'r = k + \epsilon t \pmod{n}$. Both k and d are smaller than n and ϵt is smaller than n (here we use the assumption on the size of r and t), and so the equality $d = k + \epsilon t$ holds in the integers. We can then compute $t = \frac{d-k}{\epsilon}$. The size of t alone gives us a reasonable guess on the size of r as on average $k' \sim n$, and so r should be only slightly larger than t . To recover the value of r , we need to use the fact that r divides $k + t(n + \epsilon)$. We factor $k + t(n + \epsilon)$ and look for all divisors of similar size as t . Unfortunately, this only gives us a set of candidates for r that cannot be verified. To understand how often we can expect to have just one candidate, we simulated this test for different sizes of the mask r in the range that we saw in the cards: 32, 64, 96, 128, and 160 bits. The probability of a single candidate for r was 10%, 9.8%, 7.2%, 8.4% and 7.8%, respectively for individual bit-lengths of r . Hence, to collect a single mask value, it is expected to be enough to do 13 measurements.

The reason why $n + \epsilon$ is chosen as a prime is to pass any primality tests the implementation can have on the order. Other conditions can be imposed on the value as well. For instance, take the case of the implementation of Derive that does not check order primality but checks that the generator vanishes when multiplied by the claimed order $n + \epsilon$. We can then select ϵ such that $n + \epsilon$ and n share a factor f and set \bar{G} as any point of order f . Such a generator will pass the check and the rest of the method remains the same.

6.5 Test EPA

The last test leverages the observation of **ECTester** that a point of small order can be passed to the implementations leading to errors during the scalar multiplication. The occurrence of the errors depends on the scalar and changes with each randomization. Two runs of the same algorithm with the same setting (including a fixed scalar), one with an

error and the other without, is clear evidence of randomization. As the input point P , we select a point of small order l so that the point $lP = \mathcal{O}$ sporadically appears during the scalar multiplication regardless of the multiplier, causing an error. Nevertheless, one might accidentally select a point of order l that always/never causes an error for some multipliers so it is better to use several points of different orders. Formally, we will define the result of the test to be the variance in the measured boolean output (error or no error). Nonzero variance indicates that a scalar randomization is present. The opposite (zero variance) does not necessarily mean a lack of randomization. The implementation can handle the infinity point well as discussed in Section 5. Another possibility is that the number of measurements or the number of used points was not enough to detect a small variance. This test is applicable only to Derive as Sign and Keygen contain randomization by design (generation of nonces and private keys). Note that the name of the test comes from the Exceptional Procedure Attack by Izu and Takagi [IT03] which in the same principle invokes errors in the scalar multiplication to gain information about the private key.

Remark. The presented tests were designed to distinguish the behavior of different scalar randomization techniques. All of the tests use correctly defined points on the claimed curve (Table 3) and use their order to leak information about the techniques. This is invariant under the coordinate and curve randomization. Hence, the tests will work even if these countermeasures are implemented together with the scalar randomization. Similarly, the point blinding, as described in Section 2.7, uses random points on the curve for randomization and not the order n or the cofactor. For any set up of the tests, $\rho(k, P)$ will always correctly compute kP under point blinding and so our tests will be unaffected by point blinding as well. The exception is the EPA test which uses exceptional points of the formulas to produce points not on the curve and consequently errors. The EPA test can reveal point blinding with the same principle as with the scalar randomization.

6.6 Scalar randomization on JavaCard smartcards

Table 5 presents the results of our reverse-engineering tests on thirteen JavaCard smartcards. Overall we managed to reverse-engineer the scalar randomization on every tested card except for **A1**, which refused the inputs of all five tests. On most of the cards, we managed to recover the randomization technique for all three functions, Derive, Sign, and Keygen, which turned out to all use the same technique. On six of the cards, we recovered the value of the randomization mask. We intentionally did not run all the tests on all the cards to avoid breaking the cards when the randomization technique was already clear, as some of the tests were likely triggering internal fault attack detection mechanisms that were disabling the cards. This was mainly the case of test composite and test EPA. Tests $3n$ and $n + \epsilon$, which on their own were often able to recover the randomization technique and the mask, were causing no errors. Overall, we disabled only four cards during the testing.

Most of the cards use GSR, as the results of test $3n$ follow the uniform distribution $\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$ distribution (recall Table 4). The GSR was also confirmed by the $k = 10$ test on four cards. **N4** uses multiplicative splitting as it exhibits the $\frac{5}{9}, \frac{2}{9}, \frac{2}{9}$ distribution. There was also evidence of the multiplicative splitting on **G1** by the composite test, but the errors did not appear close to the expected frequency and so the result was not conclusive.

The test $3n$ also showed that **S2** and **F2** do not use scalar randomization at all. EPA test measured high-frequency errors for **F2** and **N1**. In the case of **F2**, they were constant for fixed inputs, which confirmed no randomization. On **N1** the errors were non-deterministic, changing for fixed inputs, confirming the existence of randomization. For both **N1** and **F2**, the errors appeared more frequently than the expected distribution, i.e., for points of order $l = 3$ or $l = 5$ the errors appeared almost always and not with the probability $\frac{1}{l}$. As discussed in Section 5 this points to improper handling of the infinity point. On **S2** the errors appeared with the expected probability $\frac{1}{l}$.

Table 5: Results of our reverse-engineering of scalar randomization on JavaCards.

Card	Target	$3n$	Composite	$k = 10$	EPA	ρ	Mask
N1	Derive	0.34, 0.33, 0.32	100%	86%	>0	GSR	X
	Sign	0.31, 0.31, 0.38	83%	-		GSR	160
	Keygen	0.32, 0.33, 0.35	100%	-		GSR	160
N2	Derive	0.37, 0.33, 0.30	100%	98%		GSR	X
	Sign	0.37, 0.31, 0.31	83%	-		GSR	160
	Keygen	0.35, 0.35, 0.31	100%	-		GSR	160
N3	Derive	0.33, 0.32, 0.35	100%	98%		GSR	32
	Sign	0.31, 0.30, 0.39	85%	-		GSR	160
	Keygen	X	X	-		X	X
N4	Derive	0.22, 0.56, 0.22	82%	100%		Mult	64
	Sign	0.23, 0.23, 0.54	-	-		Mult	?
	Keygen	X	X	-		X	X
N6	Derive	0, 0, 1	100%	100%	\square	Euc.?	2
	Sign	0, 0.52, 0.48	71%	-		Euc.?	2
	Keygen	0, 0.51, 0.49	100%	-		Euc.?	2
N9	Derive	0.32, 0.34, 0.35	100%	99%		GSR	X
	Sign	0.29, 0.35, 0.35	84%	-		GSR	160
	Keygen	0.34, 0.33, 0.32	100%	-		GSR	160
I1	Derive	0.37, 0.32, 0.31	100%	100%		GSR	X
	Sign	0.30, 0.33, 0.37	92%	-		GSR	X
	Keygen	0.35, 0.30, 0.35	100%	-		GSR	X
I2	Derive	0.31, 0.36, 0.33	X	100%		GSR	X
	Sign	0.32, 0.32, 0.36	X	-		GSR	64
	Keygen	0.32, 0.33, 0.35	X	-		GSR	32
A1	Derive	X	X	100%		X	X
	Sign	X	X	-		X	X
	Keygen	X	X	-		X	X
G1	Derive	X	99%	100%		Mult	X
	Sign	X	X	-		X	X
	Keygen	X	X	-		X	X
S2	Derive	1, 0, 0	100%	100%	0	None	-
	Sign	1, 0, 0	80%	-		None	-
	Keygen	\square	\square	\square		\square	\square
F1	Derive	0.33, 0.34, 0.33	100%	100%		GSR	X
	Sign	0.33, 0.33, 0.35	78%	-		GSR	X
	Keygen	0.34, 0.31, 0.36	100%	-		GSR	X
F2	Derive	1, 0, 0	100%	100%	0	None	-
	Sign	X	X	-		X	X
	Keygen	X	X	-		X	X

The results of test $3n$ on **N6** point to either Euclidean splitting or no scalar randomization at all. Furthermore, the $n + \epsilon$ mask recovery revealed the implementation adds n to the scalar if the scalar is odd and adds $2n$ otherwise. Initial runs of the EPA test on Derive showed some evidence of randomization, but the card soon stopped responding. We concluded that it is most likely the Euclidean splitting, but further measurements would be needed for a definitive answer.

Apart from **N6**, our tests disabled one **N3**, one **N4** and one **S2** card. The symbol \square in Table 5 denotes that the card stopped responding before we had a chance to run the test.

Mask recovery. For **N1**, **N2**, **N3**, **N4**, **N9** and **I2**, we recovered the value of the randomization mask using the $n + \epsilon$ test for both the GSR and the multiplicative splitting. The sizes of the masks were the expected 32 and 64 bits (a trade-off between security and efficiency) as well as 160 bits. One explanation for a 160-bit mask is that many standard curves have orders n with special form close to a power of two. GSR blinding with rn for small r might not mask the whole scalar as shown in [FRV14].

Note that on **I2** and **N3** the sizes of the mask differed between the different functions. The $n + \epsilon$ test failed on Sign on **N4** for unknown reasons that might be worth investigating further. The $3n$ test shows that multiplicative splitting is used on this card. Recall that if k is the used nonce, d is the discrete logarithm between the generator G and the output point kG then $d = k + \epsilon t$ for some integer t of size similar to the size of the mask. Using this equality, we can recover t and then use it to get the candidate set for the mask. This approach was successful in Derive and showed that the mask has 64 bits. However, for Sign, the term $d - k = 92t$ turned out to be a large 256-bit number and often not divisible by 92, making the equality incorrect in this case. Yet, the results showed some signal as $d - k$ was divisible by 92 much more often than a random integer (roughly 12% of time). Perhaps the multiplicative splitting was combined with some other countermeasure that we were not able to recover.

7 Impact, limitations and discussion

Impact. Our findings have impacts on several levels. First, it is now possible to mount the learning phase of profiled attacks on black-box ECC targets using scalar randomization via our mask recovery techniques if the targets allow the setting of custom domain parameters. This is the case for the JavaCard platform, as we demonstrated reverse-engineering and mask recovery on several certified smartcards. The learning phase will have to be performed on a different (non-standard) curve than the attack phase, but the curve can be made similar to the target one (same prime, same a parameter). Second, knowledge of scalar randomization countermeasures can no longer be assumed secret in security evaluations of JavaCard smartcards unless they also mitigate the reverse-engineering techniques, including the ones listed in this paper. This may impact the JIL attack rating of certified (or to be certified) products [JIL]. Finally, our findings open up possibilities for more reverse-engineering. Perhaps it is possible to reverse-engineer scalar multipliers even if scalar randomization countermeasures are present.

Limitations. While we managed to recover the scalar randomization on almost all of the cards, several of the tests failed. In some cases, the cards explicitly validated the parameters (e.g., the primality of the order). In others, the implementation errored out on invalid inputs and gave us no meaningful information. Moreover, some of the cards likely have an internal fault detection, as repeated errors caused the cards to stop responding temporarily or even permanently. We encountered this behavior for the composite and EPA tests, but not for the $3n$ and $n + \epsilon$ tests.

The $n + \epsilon$ test is capable of fully recovering the mask value for GSR. However, for the multiplicative splitting, the test only offers a set of candidates, any of which can be the

true masking value. In roughly 10% of the cases, there is only one (true) candidate.

One of the main limitations of our work is the presence of a combination of countermeasures. We have designed individual tests to distinguish the four main scalar randomization techniques. Any implementation that uses two or more together may fool our tests. This might have been the case of the results on **N4** with Sign. The $3n$ test indicated the multiplicative splitting, but the $n + \epsilon$ test gave nonsensical results for the mask.

We have implemented countermeasures composed of two scalar randomization techniques (their precise description can be found in our repository) using **pyecsc**. This amounted to 15 combinations since for each pair of the four main techniques, there were more than one possible way to combine them. This by no means is meant to be an exhaustive list and only aims to provide a better understanding of our limitations and to possibly open up paths for future work. We ran test $3n$, test composite and test $k = 10$ on each combination with the results in Table 7 in the appendix. The results of test $3n$ on each combination of techniques corresponded to the expected result on one of the techniques. For instance, the test on all three combinations of GSR and the multiplicative splitting (GM-1, GM-2, GM-3) output the distribution $\frac{2}{9}, \frac{2}{9}, \frac{5}{9}$, which is the expected distribution for the multiplicative splitting. There were two exceptions though. First, one combination of the additive and multiplicative splitting slightly deviated from $\frac{2}{9}, \frac{2}{9}, \frac{5}{9}$. Closer examination showed that it rather follows the distribution $\frac{8}{27}, \frac{8}{27}, \frac{11}{27}$. Second, one combination of GSR and Euclidean splitting followed the distribution $\frac{2}{9}, \frac{2}{9}, \frac{5}{9}$. In all combinations, the composite test correctly identified the presence of the multiplicative splitting. Hence, the combination of these two tests would either signal that the implementation does not contain one simple technique or would at least identify one of them. Surprisingly, the results of test $k = 10$ did not follow any clear pattern.

Mitigations. Our reverse-engineering methods are observing the behavior of the implementations under invalid inputs. An obvious countermeasure is proper validation of the domain parameters and the inputs, as suggested in [SJS20]. Both the prime p defining the base field and the prime field order n need to be verified by a primality test to avoid composite order curves. The generator G and any point must be checked that they belong on the curve and their order is n . The correct way to do so is to multiply and compare $nG = \mathcal{O}$. Since n is prime, this implies that G must have the correct order, and the curve truly contains a subgroup of order n . The order of the public key points should be verified as well to avoid any points outside of the main subgroup. Other solutions to cofactor validation, like cofactor Diffie-Hellman or the inverse trick, rely on the assumption that the claimed cofactor is correct. This precise assumption by implementations is violated by our inputs and made most of our tests work. Specifically, tests $3n$, $n + \epsilon$, and composite concealed the true cofactor, used points outside of the claimed group, and composite curve order n . When implemented, the mentioned checks would make these tests ineffective.

Primality testing and domain parameter validation of custom curves come with a high cost in performance. A clear solution is to restrict the API to standard named curves. Such curves will not contain any hidden small subgroups. Not only does this avoid any costly checks, these curves are optimized for higher performance. Popular examples are the prime order NIST curves or the cofactor curves Curve25519 and Ed25519. Note that this needs to be a strict limitation on an API level and not just an option (as is the case for JavaCard API since version 3.1), as an attacker will simply opt for optional setting of custom domain parameters to make our techniques work.

The EPA and $k = 10$ tests relied on improper handling of the infinity point by the target during scalar multiplication. Not every formula computing point addition $P + Q$ is prepared for special cases like $P = \pm Q$ or $P = \mathcal{O}$, which results in an undefined output that causes an error. One solution is for the implementation to check for these special cases and solve them separately. This might involve dummy operations to preserve constant-time execution. A more straightforward, and secure, solution is to use complete

formulas [RCB16]. Note that even these formulas contain exceptional points that are outside of the main subgroup, and so they must be used together with cofactor validation.

We would like to stress that the mitigations described above will only thwart methods presented in this paper and in their current form. Reverse engineering of countermeasures, in general, likely cannot be fully prevented, as evidenced also by the Side Journey to Titan [RLM⁺21] and EUCLEAK papers [Roc24]. The assumption that implementation details can be kept secret relies on security by obscurity in contrary to Kerckhoff’s principle. While obscurity typically increases the difficulty of mounting an attack in the short term due to an additional layer of uncertainty about the implementation, it also decreases availability for public scrutiny. The lack of public audits is possibly harming the security in the long run – a timeframe especially relevant for cryptographic smartcards used as, e.g., electronic IDs with the expected lifetime of a decade or more. The advent of open-source cryptographic hardware designs like LowRISC’s OpenTitan [Ope] or TropicSquare’s TROPIC01 [Tro] may offer interesting tests of the feasibility of open-source secure hardware design.

Acknowledgements

We would like to thank Thomas Roche for his insightful comments and discussion. Jan Jancar was supported by Red Hat Czech. J. Jancar, V. Suchanek, P. Svenda, and Ł. Chmielewski were supported by the AI-SecTools (VJ02010010) project.

References

- [ABM⁺03] Adrian Antipa, Daniel R. L. Brown, Alfred Menezes, René Struik, and Scott A. Vanstone. Validation of elliptic curve public keys. In Yvo Desmedt, editor, *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, volume 2567 of *Lecture Notes in Computer Science*, pages 211–223. Springer, 2003. DOI: [10.1007/3-540-36288-6_16](https://doi.org/10.1007/3-540-36288-6_16).
- [AFV07] Frédéric Amiel, Benoit Feix, and Karine Villegas. Power analysis for secret recovering and reverse engineering of public key algorithms. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *Selected Areas in Cryptography, 14th International Workshop, SAC 2007, Ottawa, Canada, August 16-17, 2007, Revised Selected Papers*, volume 4876 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2007. DOI: [10.1007/978-3-540-77360-3_8](https://doi.org/10.1007/978-3-540-77360-3_8).
- [ANSI05] ANSI. American National Standard X9.62-2005, Public key cryptography for the financial services industry: the elliptic curve digital signature algorithm (ECDSA). Standard, Accredited Standards Committee X9, 2005.
- [BBP⁺12] Billy Bob Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. Practical realisation and elimination of an ECC-related software bug attack. In Orr Dunkelman, editor, *Topics in Cryptology - CT-RSA 2012 - The Cryptographers’ Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings*, volume 7178 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2012. DOI: [10.1007/978-3-642-27954-6_11](https://doi.org/10.1007/978-3-642-27954-6_11).
- [BL07] Daniel J. Bernstein and Tanja Lange. Explicit-formulas database. 2007. URL: <https://hyperelliptic.org/EFD/> (visited on 07/10/2024).
- [BL17] Daniel J. Bernstein and Tanja Lange. Safecurves: choosing safe curves for elliptic-curve cryptography. January 2017. URL: <https://safecurves.cr.yp.to/> (visited on 04/06/2025).

- [BL24] Daniel J. Bernstein and Tanja Lange. Safe curves for elliptic-curve cryptography. *IACR Cryptol. ePrint Arch.*:1265, 2024. URL: <https://eprint.iacr.org/2024/1265>.
- [BMM00] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential fault attacks on elliptic curve cryptosystems. In Mihir Bellare, editor, *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, volume 1880 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2000. DOI: [10.1007/3-540-44598-6_8](https://doi.org/10.1007/3-540-44598-6_8).
- [C2S16] C2SP. Wycheproof, 2016. URL: <https://github.com/C2SP/wycheproof> (visited on 05/20/2024).
- [CC] Common Criteria. ISO/IEC 15408 Information technology — Security techniques — Evaluation criteria for IT security. In *ISO/IEC 15408-1:2022*. ISO/IEC, 2022.
- [Cer99] Certicom Research. Test vectors for SEC 1. Standards for Efficient Cryptography Group, 1999. URL: <http://rfc.nop.hu/secg/gec2.pdf> (visited on 02/20/2025).
- [CGN20] Konstantinos Chalkias, François Garillot, and Valeria Nikolaenko. Taming the many EdDSAs. In Thyla van der Merwe, Chris J. Mitchell, and Maryam Mehrnezhad, editors, *Security Standardisation Research - 6th International Conference, SSR 2020, London, UK, November 30 - December 1, 2020, Proceedings*, volume 12529 of *Lecture Notes in Computer Science*, pages 67–90. Springer, 2020. DOI: [10.1007/978-3-030-64357-7_4](https://doi.org/10.1007/978-3-030-64357-7_4).
- [CJ01] Christophe Clavier and Marc Joye. Universal exponentiation algorithm. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 300–308. Springer, 2001. DOI: [10.1007/3-540-44709-1_25](https://doi.org/10.1007/3-540-44709-1_25).
- [CJ03] Mathieu Ciet and Marc Joye. (virtually) free randomization techniques for elliptic curve cryptography. In Sihan Qing, Dieter Gollmann, and Jianying Zhou, editors, *Information and Communications Security, 5th International Conference, ICICS 2003, Huhehaote, China, October 10-13, 2003, Proceedings*, volume 2836 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 2003. DOI: [10.1007/978-3-540-39927-8_32](https://doi.org/10.1007/978-3-540-39927-8_32).
- [Cor99] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 1999. DOI: [10.1007/3-540-48059-5_25](https://doi.org/10.1007/3-540-48059-5_25).
- [DGH⁺13] Jean-Luc Danger, Sylvain Guilley, Philippe Hoogvorst, Cédric Murdica, and David Naccache. A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards. *J. Cryptogr. Eng.*, 3(4):241–265, 2013. DOI: [10.1007/S13389-013-0062-6](https://doi.org/10.1007/S13389-013-0062-6).
- [FLR⁺08] Pierre-Alain Fouque, Reynald Lercier, Denis Réal, and Frédéric Valette. Fault attack on elliptic curve montgomery ladder implementation. In *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 92–98, 2008. DOI: [10.1109/FDTC.2008.15](https://doi.org/10.1109/FDTC.2008.15).

- [FRV14] Benoit Feix, Mylène Roussellet, and Alexandre Venelli. Side-channel analysis on blinded regular scalar multiplications. In Willi Meier and Debdeep Mukhopadhyay, editors, *Progress in Cryptology - INDOCRYPT 2014 - 15th International Conference on Cryptology in India, New Delhi, India, December 14-17, 2014, Proceedings*, volume 8885 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2014. DOI: [10.1007/978-3-319-13039-2_1](https://doi.org/10.1007/978-3-319-13039-2_1).
- [Goo16] Google. Paranoid crypto, 2016. URL: https://github.com/google/paranoid_crypto (visited on 05/20/2024).
- [Har13] Dan Harkins. RFC 6932: Brainpool elliptic curves for the internet key exchange (IKE) group description registry, 2013.
- [HMOV04] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Professional Computing. Springer, 2004. ISBN: 978-0-387-95273-4. DOI: [10.1007/b97644](https://doi.org/10.1007/b97644).
- [IEEE00] IEEE. IEEE Standard Specifications for Public-Key Cryptography. Standard, IEEE Std 1363-2000 Working Group, 2000.
- [IT03] Tetsuya Izu and Tsuyoshi Takagi. Exceptional procedure attack on elliptic curve cryptosystems. In Yvo Desmedt, editor, *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, volume 2567 of *Lecture Notes in Computer Science*, pages 224–239. Springer, 2003. DOI: [10.1007/3-540-36288-6_17](https://doi.org/10.1007/3-540-36288-6_17).
- [JIL] Senior Official Group Information Systems Security. Application of Attack Potential to Smartcards and Similar Devices. Joint Interpretation Library, November 2022. URL: <https://www.sogis.eu/documents/cc/domains/sc/JIL-Application-of-Attack-Potential-to-Smartcards-v3.2.pdf>.
- [JJS⁺24] Adam Janovsky, Jan Jancar, Petr Svenda, Łukasz Chmielewski, Jiri Michalik, and Vashek Matyas. sec-certs: Examining the security certification practice for better vulnerability mitigation. *Comput. Secur.*, 143:103895, 2024. DOI: [10.1016/J.COSE.2024.103895](https://doi.org/10.1016/J.COSE.2024.103895).
- [JSS⁺20] Jan Jancar, Vladimir Sedlacek, Petr Svenda, and Marek Šýs. Minerva: The curse of ECDSA nonces; Systematic analysis of lattice attacks on noisy leakage of bit-length of ECDSA nonces. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(4):281–308, 2020. DOI: [10.13154/TCHES.V2020.I4.281-308](https://doi.org/10.13154/TCHES.V2020.I4.281-308).
- [JSS⁺24] Jan Jancar, Vojtech Suchanek, Petr Svenda, Vladimir Sedlacek, and Łukasz Chmielewski. pyecsc: Reverse engineering black-box elliptic curve cryptography via side-channel analysis. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(4):355–381, 2024. DOI: [10.46586/TCHES.V2024.I4.355-381](https://doi.org/10.46586/TCHES.V2024.I4.355-381).
- [JSS15] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. Practical invalid curve attacks on TLS-ECDH. In Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl, editors, *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I*, volume 9326 of *Lecture Notes in Computer Science*, pages 407–425. Springer, 2015. DOI: [10.1007/978-3-319-24174-6_21](https://doi.org/10.1007/978-3-319-24174-6_21).
- [Kel11] Sharon S Keller. The elliptic curve cryptography cofactor Diffie-Hellman (ECC CDH) primitive validation system (ECC_CDHVS). *NIST Information Technology Laboratory*, 2011. URL: <https://csrc.nist.gov/csrc/media/projects/cryptographic-algorithm-validation-program/documents/components/ecccdhvs.pdf>.

- [LL97a] Chae Hoon Lim and Pil Joong Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 249–263. Springer, 1997. DOI: [10.1007/BFB0052240](https://doi.org/10.1007/BFB0052240).
- [LL97b] Chae Hoon Lim and Pil Joong Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In *Advances in Cryptology—CRYPTO'97: 17th Annual International Cryptology Conference Santa Barbara, California, USA August 17–21, 1997 Proceedings 17*, pages 249–263. Springer, 1997.
- [Mad22] Neil Madden. Psychic signatures in java. 2022. URL: <https://neilmadden.blog/2022/04/19/psychic-signatures-in-java/> (visited on 04/03/2025).
- [ML13] Johannes Merkle and Manfred Lochter. RFC 7027: Elliptic curve cryptography (ECC) Brainpool curves for transport layer security (TLS), 2013.
- [Moz24] Mozilla Security. Cryptofuzz. <https://github.com/MozillaSecurity/cryptofuzz>, 2024. (Visited on 04/01/2025).
- [MVO91] Alfred Menezes, Scott A. Vanstone, and Tatsuaki Okamoto. Reducing elliptic curve logarithms to logarithms in a finite field. In Cris Koutsougeras and Jeffrey Scott Vitter, editors, *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, May 5-8, 1991, New Orleans, Louisiana, USA*, pages 80–89. ACM, 1991. DOI: [10.1145/103418.103434](https://doi.org/10.1145/103418.103434).
- [NCC21] NCC Group. Arbitrary signature forgery in Stark Bank ECDSA libraries. <https://www.nccgroup.com/us/research-blog/technical-advisory-arbitrary-signature-forgery-in-stark-bank-ecdsa-libraries-cve-2021-43572-cve-2021-43570-cve-2021-43569-cve-2021-43568-cve-2021-43571/>, 2021. (Visited on 04/08/2025).
- [Ngu17] Quan Nguyen. Practical cryptanalysis of JSON web token and Galois counter mode's implementations, 2017. URL: <https://research.google/pubs/practical-cryptanalysis-of-json-web-token-and-galois-counter-modes-implementations/>. Presented on the Real World Cryptography 2017 conference.
- [Ngu21] Quan Thoi Minh Nguyen. 0. *IACR Cryptol. ePrint Arch.*:323, 2021. URL: <https://eprint.iacr.org/2021/323>.
- [NIST18] NIST. SP 800-56A Rev. 3: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography. Standard, National Institute for Standards and Technology, April 2018. URL: <https://doi.org/10.6028/NIST.SP.800-56Ar3>.
- [NIST23] NIST. SP 800-186: Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters. Standard, National Institute for Standards and Technology, February 2023. URL: <https://doi.org/10.6028/NIST.SP.800-186>.
- [NT16] Samuel Neves and Mehdi Tibouchi. Degenerate curve attacks - extending invalid curve attacks to edwards curves and other models. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part II*, volume 9615 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2016. DOI: [10.1007/978-3-662-49387-8_2](https://doi.org/10.1007/978-3-662-49387-8_2).

- [Ope] OpenTitan. Open source silicon root of trust (rot). URL: <https://opentitan.org/> (visited on 04/15/2025).
- [PH78] Stephen Pohlig and Martin Hellman. An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance. *IEEE Transactions on Information Theory*, 24(1):106–110, January 1978. DOI: [10.1109/TIT.1978.1055817](https://doi.org/10.1109/TIT.1978.1055817).
- [RCB16] Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 403–428. Springer, 2016. DOI: [10.1007/978-3-662-49890-3_16](https://doi.org/10.1007/978-3-662-49890-3_16).
- [RLM⁺21] Thomas Roche, Victor Lomné, Camille Mutschler, and Laurent Imbert. A side journey to Titan. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 231–248. USENIX Association, 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/roche>.
- [Roc24] Thomas Roche. EUCLEAK. *IACR Cryptol. ePrint Arch.*:1380, 2024. URL: <https://eprint.iacr.org/2024/1380>.
- [San17a] Antonio Sanso. Critical vulnerability in JSON web encryption (JWE). <https://blog.intothesyymetry.com/2017/03/critical-vulnerability-in-json-web.html>, 2017. (Visited on 04/08/2025).
- [San17b] Antonio Sanso. CVE-2017-7781/CVE-2017-10176: issue with elliptic curve addition in mixed Jacobian-affine coordinates in Firefox/Java, 2017. URL: <https://blog.intothesyymetry.com/2017/08/cve-2017-7781cve-2017-10176-issue-with.html> (visited on 04/08/2025).
- [SEC09] SECG. SEC 1: Elliptic Curve Cryptography. Standard, Standards for Efficient Cryptography Group, May 2009. URL: <http://www.secg.org/sec1-v2.pdf>.
- [Sem98] Igor A. Semaev. Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p. *Math. Comput.*, 67(221):353–356, 1998. DOI: [10.1090/S0025-5718-98-00887-4](https://doi.org/10.1090/S0025-5718-98-00887-4).
- [SJS20] Vladimir Sedlacek, Jan Jancar, and Petr Svenda. Fooling primality tests on smartcards. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part II*, volume 12309 of *Lecture Notes in Computer Science*, pages 209–229. Springer, 2020. DOI: [10.1007/978-3-030-59013-0_11](https://doi.org/10.1007/978-3-030-59013-0_11).
- [Sma99] Nigel P. Smart. The discrete logarithm problem on elliptic curves of trace one. *J. Cryptol.*, 12(3):193–196, 1999. DOI: [10.1007/S001459900052](https://doi.org/10.1007/S001459900052).
- [TA98] Satoh T. and Kiyomichi Araki. Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves. *Commentarii Math. Univ. St. Pauli.*, 47:81–92, 1998. ISSN: 0010258X. URL: <https://cir.nii.ac.jp/crid/1570009752659281152>.

- [TB02] Elena Trichina and Antonio Bellezza. Implementation of elliptic curve cryptography with built-in counter measures against side channel attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2002. DOI: [10.1007/3-540-36400-5_9](https://doi.org/10.1007/3-540-36400-5_9).
- [Tro] TropicSquare. Tropic Square. URL: <https://tropicsquare.com/> (visited on 04/15/2025).
- [Val17] Filippo Valsorda. Squeezing a key through a carry bit, 2017. URL: <https://fahrplan.events.ccc.de/congress/2017/Fahrplan/events/9021.html> (visited on 04/08/2025).
- [Vra19] Guido Vranken. Cryptofuzz, 2019. URL: <https://github.com/guidovranken/cryptofuzz> (visited on 05/20/2024).
- [VSS⁺18] Luke Valenta, Nick Sullivan, Antonio Sanso, and Nadia Heninger. In search of curveswap: measuring elliptic curve implementations in the wild. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 384–398. IEEE, 2018. DOI: [10.1109/EUROSP.2018.00034](https://doi.org/10.1109/EUROSP.2018.00034).

Appendix

Table 6: The version ranges of libraries currently supported by **ECTester**. The third column shows the versions used for the results presented in this paper. As BoringSSL is not versioned we test roughly every 40th commit, for about 100 times, from the most recent one. For libgcrypt the version 1.9.0 is suggested not to be used and thus is not supported. Reasons for unsupported versions are usually API-breaking changes in the library or unresolved issues during the build of the library or shim. For some versions, these issues may be resolvable with extra effort.

Library	Supported versions	Evaluated version
BoringSSL	r76bb141–r2eb2889, rde43457–r2be18f5	r67422ed
Botan	2.0.0–2.19.5	2.19.1
BouncyCastle	1.69–1.80	1.77
Crypto++	6.1.0–8.9.0	8.6.0
Intel Crypto	2020–2021.2, 2021.5–2021.12.1	2021.7
libgcrypt	1.8.0–1.10.3	1.9.4
LibreSSL	2.6.2–4.0.0	3.9.0
libtomcrypt	1.18.0-rc4–1.18.2	1.18.2
mbedTLS	2.7.19–3.6.2	3.5.2
Nettle	3.1.1, 3.{4, 5, 8}.1, 3.7.{1, 2, 3}	3.7.3
OpenSSL	1.1.0–1.1.0j, 1.1.1–3.4.1	3.2.0-dev
SunEC	OpenJDK 15–23	17

Table 7: The results of our reverse-engineering tests on the combinations of scalar randomization techniques. The precise description of the algorithms can be found in our repository. Each algorithm is labeled XY-N, where X,Y are the first letters of the four used techniques (GSR, Additive, Multiplicative and Euclidean splitting) and N is a numerical identifier. Individual algorithms were implemented and simulated using **pyecsca**.

Combination	Test $3n$	Test composite	Test $k = 10$
GA-1	$\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$	100%	100%
GA-2	$\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$	100%	100%
GM-1	$\frac{2}{9}, \frac{2}{9}, \frac{5}{9}$	91%	100%
GM-2	$\frac{2}{9}, \frac{2}{9}, \frac{5}{9}$	91%	100%
GM-3	$\frac{2}{9}, \frac{2}{9}, \frac{5}{9}$	91%	100%
GE-1	$\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$	100%	100%
GE-2	$\frac{2}{9}, \frac{2}{9}, \frac{5}{9}$	100%	0%
GE-3	$\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$	100%	0%
GE-4	$\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$	100%	0%
AM-1	$\frac{8}{27}, \frac{8}{27}, \frac{11}{27}$	84%	100%
AM-2	$\frac{2}{9}, \frac{2}{9}, \frac{5}{9}$	90%	100%
AE-1	$\frac{1}{2}, \frac{1}{2}, 0$	100%	100%
AE-2	$\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$	100%	100%
ME-1	$\frac{2}{9}, \frac{2}{9}, \frac{5}{9}$	91%	46%
ME-2	$\frac{2}{9}, \frac{2}{9}, \frac{5}{9}$	84%	0%