Attacks Against White-Box ECDSA and Discussion of Countermeasures

A Report on the WhibOx Contest 2021

Sven Bauer[1], Hermann Drexler[1], Max Gebhardt[2], Dominik Klein[2], Friederike Laus[2] and Johannes Mittmann[2]

[1] Giesecke+Devrient Mobile Security GmbH, Munich, Germany
firstname.lastname@gi-de.com
[2] Bundesamt für Sicherheit in der Informationstechnik (BSI), Bonn, Germany
firstname.lastname@bsi.bund.de

Abstract. This paper deals with white-box implementations of the Elliptic Curve Digital Signature Algorithm (ECDSA): First, we consider attack paths to break such implementations. In particular, we provide a systematic overview of various fault attacks, to which ECDSA white-box implementations are especially susceptible. Then, we propose different mathematical countermeasures, mainly based on masking/blinding of sensitive variables, in order to prevent or at least make such attacks more difficult. We also briefly mention some typical implementational countermeasures and their challenges in the ECDSA white-box scenario. Our work has been initiated by the CHES challenge WhibOx Contest 2021, which consisted of designing and breaking white-box ECDSA implementations, so called *challenges*. We illustrate our results and findings by means of the submitted challenges and provide a comprehensive overview which challenge could be solved in which way. Furthermore, we analyze selected challenges in more details.

Keywords: White-box cryptography · Deterministic ECDSA · Computation analysis · Fault analysis · Countermeasures · CHES Challenge · WhibOx Contest 2021

1 Introduction

In a nutshell, white-box cryptography is about finding software implementations of cryptographic algorithms that embed a secret key in such a way that it is impossible to extract this secret key from the software. There have been a number of approaches to capturing this notion formally, see, for example, [DLPR14, ABABM20]. White-box cryptography is nowadays an active area of research with various practical applications such as digital rights management, software licensing, mobile payment systems, mobile contract signing or authentication tokens without the need for special secure hardware that are used for instance in the context of cryptocurrencies and blockchain technologies. There are several vendors offering white-box cryptography libraries with support for a range of cryptographic algorithms.

Starting with the seminal work [CEJO02, CEJvO03], numerous white-box designs of symmetric algorithms, in particular DES and AES, have been proposed – and subsequently been broken: The rough idea behind the constructions in [CEJO02, CEJvO03] is to implement a cipher as a network of precomputed and randomly encoded lookup tables, such that an adversary is confused by seemingly useless intermediate values in the memory. However, the underlying techniques were soon broken [JBF02, BGEC04], which motivated further approaches [LN05, BCD06, XL09, Kar10]. But also these proposals were eventually shown to be vulnerable as well [GMQ07, WMGP07, MWP10, MRP12, DMRP13, LRM+13, LR13], and the design of secure white-box implementations remains a cat-and-mouse game. Besides practical designs, only few attempts to a formalization of white-box cryptography were proposed so far. The authors of [SWP09] showed how security notions from black-box models [BGI+01] can be adapted to the white-box setting, while [DLPR13] formalized the basic unbreakability property and introduced several other notions such as one-wayness, incompressibility and traceability for symmetric ciphers.

At the same time, white-box implementations found different practical applications and thereby raised increasing industrial interest. To summarize the academic and industrial experiences in this field, the ECRYPT CSA project organized a WhibOx workshop [Cry16] in 2016. At this occasion, the idea arose to organize a contest on white-box cryptography to give a playground for "researchers and practitioners to confront their (secretly designed) white-box implementations to state-of-the-art attackers". As a consequence, one year later the so-called WhibOx Contest was launched by ECRYPT CSA as the CHES 2017 capture the flag challenge [CSA17], and given the success of the first edition, a second edition organized by CryptoExperts and Cybercrypt followed as the CHES 2019 capture the flag challenge [CC19].

The competitions have not missed their target, so that for getting an overview of current state-of-the-art white-box implementations of symmetric ciphers, one might have a look at the papers that the winners of the previous editions published subsequent to the WhibOx Contests: In [BU18], the authors of the winning submission of the WhibOx Contest 2017 present their considerations regarding the effectiveness of different masking schemes, while in [GPRW19] and [GRW20], the successful breakers of the WhibOx Contest 2019 summarize common white-box countermeasures and explain their approach to break the winning implementations.

While numerous academic works address symmetric white-box cryptography, there are up to now only few publications targeting asymmetric cryptographic mechanisms in a white-box setting: In [FHW+19] and [ZHH+20], the authors propose a white-box implementation of Shamir's identity-based signature scheme and of the identity-based signature scheme in the IEEE P1363 standard, respectively. More generally, [Bar20b, Bar20a] proposes a white-box design for an asymmetric lattice based scheme by combining techniques used in AES white-box designs (lookup tables) with different homomorphic encodings and additional countermeasures. The recent paper [ZBJ20] considers a white-box implementation of ECDSA. However, the authors of [ZBJ20] assume, among other things, that a trusted third party and a cloud server are available, which are strong assumptions that do most likely not meet reality. [DGH21] gives an overview of some of the challenges when attempting to defend an ECDSA implementation against a white-box attacker and suggests some countermeasures. The authors of [GG22] present a concrete white-box implementation of the Hidden Field Equation (HFE) signature algorithm for a specific set of internal polynomials. Furthermore, they formulate more precise definitions of the concepts *unbreakability* and *incompressibility*

To stimulate the research in the field of asymmetric white-box cryptography, a third edition of the WhibOx Contest took place prior to the (virtual) CHES conference in September 2021. As the previous editions, it consisted of a capture the flag challenge with two parts: The first part ("designer") was to design a white-box implementation computing an ECDSA signature, where the underlying elliptic curve was the NIST P-256 curve [KG13]. The signature algorithm should be deterministic with a freely chosen nonce derivation mechanism. In the second part ("attacker"), the participants were invited to break the submitted implementations by extracting their hard-coded signing key. In total, there were 83 registered users, 97 submitted challenges and 898 successful breaks.[1]

[1] Sven Bauer and Hermann Drexler won as team `bananaramadama` the attacker challenge; Max Gebhardt, Dominik Klein, Friederike Laus and Johannes Mittmann won together with further colleagues Tobias

Challenges had to be submitted as C source code and had to comply with several competition rules, among them several requirements on execution time, the code size and the RAM usage of the compiled executable. No external dependencies were allowed except for usage of the GNU Multiple Precision Arithmetic Library (GMP).[2] The exact rules of the competition can be found on the WhibOx Contest 2021 website.[3] The fact that attackers had C source code of the implementation is of course a major difference to attacking a real-world application, where, for example, the code for ECDSA would be compiled into a larger application. However, the organizers of the competition apparently wanted to focus on the mathematical aspect of white-box cryptography and avoid the challenges becoming exercises in reversing binaries.

Based on the previously mentioned criteria, the authors of a challenge earned strawberries according to a performance score that depended on the execution time, the code size and the RAM usage of the executable. Challenges that were either faster, smaller or less memory-consuming got a higher performance score. The amount of strawberries increased quadratically with time as long as the challenge remained unbroken, and it symmetrically decreased back to zero after the first break. The designers of the challenge were awarded with the number of strawberries reached at the time the challenge was broken.

Attackers could gain bananas for the successful break of a challenge; they obtained the number of bananas corresponding to the number of strawberries the challenge had at the time the attacker broke it. In particular, the attacker that first broke a challenge got the highest number of bananas for this challenge, while no bananas were awarded once the strawberry score of a challenge had dropped to zero some time after it was first broken.

The designer of the challenge with the highest strawberry score won the design part of the challenge and similarly, the attacker with the highest banana score won the attack part. Note that in both categories, strawberries respective bananas of different challenges were not accumulated, but their maximum was taken, so it was not advantageous so submit or break as many challenges as possible.

These special rules had a number of consequences:

- New challenges were not published at a fixed time each day, but as soon as possible after they passed the server test checking the fulfillment of the performance constraints. Thus, depending on the time zone they live in, some attackers might have had a significant advantage for solving a challenge compared to others, and similarly, some designer simply might have had luck since they posted their challenge (possibly on purpose) at a time when presumably they assumed most of the attackers to sleep (considering e.g. the typical distribution of the CHES participants across the world). These time differences are even more significant since no challenge survived longer than 33 hours, so that a difference of several hours could possibly have made a large difference in the final ranking.

- Since bananas were not accumulated across different challenges, it was not honored if an attacker tried to break as many challenges as possible, thereby showing power and flexibility. For the same reason, the number of total breaks of a challenge is not necessarily a measure of its strength, although it probably gives a first hint.

- An attacker might have held back a successful attack before submitting it (either until the challenge had reached a certain number of strawberries or in general as long as possible) in order to gain more bananas. This of course comes along with the risk that someone else submits a solution, in which case however one could directly submit afterwards and by doing so basically get the same number of bananas.

Damm, Aron Gohr, Dennis Kügler and Vivien Thiel as team `auguste` the second prize in both categories. After the end of the contest, the two teams got in touch with each other, which resulted in this paper.

[2] https://gmplib.org/

[3] https://whibox.io/contests/2021/rules

- Overall, the computational constraints, in particular the time constraint, seemed to favor the attackers, as they made strong computational countermeasures and obfuscation strategies rather impossible. Furthermore, the system that validated whether the constraints are fulfilled, seemed to behave non-deterministically, potentially depending on the server load.

All in all, these aspects make an objective assessment of the strength of the submitted challenges as well as of the power of an attacker based on the strawberry score, respectively the banana score, quite difficult, although a rough distinction between challenges that were rather easy to break (many breaks within a short period after the publication) and harder challenges (longer survival and potentially fewer breaks) can be made. It is therefore one of the main intentions of this paper to provide a comprehensive analysis of the given task – a white-box implementation of ECDSA – and to apply this analysis in a second step to the submitted implementations. To this aim, we first present a systematic overview of possible computation and fault attacks, to which ECDSA is particularly susceptible. Then, in a second step, we propose some countermeasures and obfuscation techniques that aim at preventing the presented attacks. Finally, we provide a detailed evaluation which of the submissions can be attacked by which kind of attack and analyze selected challenges in more details.

The recent publication [BBD+22] also analyzes the results of the WhibOx Contest 2021. We describe additional attacks and show how to break some challenges with simpler methods than the ones presented in [BBD+22]. To the best of our knowledge, our work is also the first to name at least one successful automated attack for each challenge in the WhibOx Contest 2021 (see Table 2). Interestingly, [BBD+22, Table 5] gives the secret keys for challenges `#305` and `#346` but does not state an attack vector.

The outline of the remaining paper is as follows: First, we introduce in Section 2 preliminaries on ECDSA on the curve P-256 and on signature equations. In Section 3 we analyze which kind of information can be revealed from ECDSA by means of computation analysis, before different types of fault attacks, to which deterministic signature schemes such as dECDSA are particularly susceptible, are considered in Section 4. Next, we give in Section 5 an overview of which challenges were vulnerable to which attacks and provide an in-depth analysis of selected challenges. In Section 6, three classes of countermeasures are discussed, before finally conclusions and an outline of future work are drawn in Section 7.

2 Preliminaries

2.1 Deterministic ECDSA on P-256

For the sake of completeness and to introduce the used notation, we briefly recall the specification of ECDSA on the curve P-256 (see [KG13]). It is defined over a finite field $\mathbb{F}_p$ with characteristic

$$\begin{aligned} p &= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \\ &= \texttt{FFFFFFFF 00000001 00000000 00000000 00000000 FFFFFFFF FFFFFFFF FFFFFFFF}_{16}\,. \end{aligned}$$

The elliptic curve P-256 is defined by its *Weierstrass form* $y^2 = x^3 + ax + b$ with the parameters

$$\begin{aligned} a &= p - 3 \\ &= \texttt{FFFFFFFF 00000001 00000000 00000000 00000000 FFFFFFFF FFFFFFFF FFFFFFFC}_{16}\,, \\ b &= \texttt{5AC635D8 AA3A93E7 B3EBBD55 769886BC 651D06B0 CC53B0F6 3BCE3C3E 27D2604B}_{16}\,. \end{aligned}$$

The base point G with coordinates

$$G_x = \texttt{6B17D1F2 E12C4247 F8BCE6E5 63A440F2 77037D81 2DEB33A0 F4A13945 D898C296}_{16} ,$$
$$G_y = \texttt{4FE342E2 FE1A7F9B 8EE7EB4A 7C0F9E16 2BCE3357 6B315ECE CBB64068 37BF51F5}_{16}$$

generates a cyclic subgroup of prime order

$$q = \texttt{FFFFFFFF 00000000 FFFFFFFF FFFFFFFF BCE6FAAD A7179E84 F3B9CAC2 FC632551}_{16} .$$

An ECDSA key pair consists of a private key $d \in \mathbb{F}_q^*$ and the public key $P = d\,G$. The ECDSA specification [KG13] uses a random ephemeral key in the signature generation process, so even signing the same message twice with the same key will, with overwhelming probability, give two different signatures. In the strict white-box model, there is xcept for the input no external source of entropy. We model the ephemeral key derivation mechanism as a deterministic random number generator $(\mathsf{seed}, \mathsf{rand})$, where $\mathsf{seed}\colon \{0,1\}^{256} \to Z$ and $\mathsf{rand}\colon Z \to \mathbb{F}_q^* \times Z$ are functions for some finite set Z of internal states.

A deterministic version of ECDSA is standardised in RFC 6979 [Por13], where the ephemeral key derivation mechanism uses the secret key d in addition to the message hash h for seeding. In our model, we consider the ephemeral key derivation mechanism itself as a secret, but it could also be a public mechanism parameterized by the secret key d or another secret. With this model, Algorithm 1 describes a deterministic version of ECDSA signature generation and signing the same message twice with the same key with Algorithm 1 will give the same signature twice.

The functions $\mathsf{os2int}$ and $\mathsf{int2os}_{32}$ used in Algorithm 1 denote conversion functions between octet strings in $\{0,1\}^{256}$ and integers in $\{0,\dots,2^{256}-1\}$. Even though we have

$$\mathsf{os2int}(\mathsf{int2os}_{32}(0)) = \mathsf{os2int}(\mathsf{int2os}_{32}(q)) \pmod q ,$$

for instance, we will usually identify the octet strings h_{os} and $(r_{\mathsf{os}}, s_{\mathsf{os}})$ with their counterparts $h \in \mathbb{F}_q$ and $(r,s) \in \mathbb{F}_q^2$ throughout this paper, if a distinction is not relevant for the discussion.

Algorithm 1: White-Box dECDSA Signature Generation on P-256.

Embedded Secrets: An ephemeral key derivation mechanism $(\mathsf{seed}, \mathsf{rand})$ and a private key $d \in \mathbb{F}_q^*$.

Input: A message hash $h_{\mathsf{os}} \in \{0,1\}^{256}$.

Output: The signature $(r_{\mathsf{os}}, s_{\mathsf{os}}) \in \{0,1\}^{256} \times \{0,1\}^{256}$ for h_{os}.

S1. Set $h \leftarrow \mathsf{os2int}(h_{\mathsf{os}})$ and set $z \leftarrow \mathsf{seed}(h_{\mathsf{os}})$.

S2. Set $(k, z) \leftarrow \mathsf{rand}(z)$.

S3. Set $r \leftarrow \big((k\,G)_x \bmod p\big) \bmod q$.

S4. Set $s \leftarrow k^{-1}(rd + h) \bmod q$.

S5. If $r = 0$ or $s = 0$, go to step **S2**, otherwise return $\big(\mathsf{int2os}_{32}(r), \mathsf{int2os}_{32}(s)\big)$.

Remark 1. The interface of Algorithm 1 enables attackers to perform chosen-hash attacks.

Remark 2. Some challenges tried to use entropy input. There were two types of “illegal“ sources of non-determinism used in the WhibOx Contest 2021:

(a) The function `time()` of the C standard library was used by 29 challenges, see Table 2 in Subsection 5.1 under the identifier ND_1. These challenges can easily be derandomized by patching the call to `time()` with a constant function.

(b) Six challenges used uninitialized variables, see Table 2 in Subsection 5.1 under the identifier ND_2. They can be derandomized using QEMU user-mode emulation[4].

Using the described derandomization methods, we may assume for the rest of the paper that all challenges are deterministic.

The signature verification process detailed in Algorithm 2 is identical for ECDSA signatures generated with a truly random ephemeral key and for deterministic ones.

Algorithm 2: ECDSA Signature Verification on P-256.

Input: A message hash $h_{\mathsf{os}} \in \{0,1\}^{256}$, a signature $(r_{\mathsf{os}}, s_{\mathsf{os}}) \in \{0,1\}^{256} \times \{0,1\}^{256}$, and a public key $P_{\mathsf{os}} = (P_{\mathsf{os},x}, P_{\mathsf{os},y}) \in \{0,1\}^{256} \times \{0,1\}^{256}$.

Output: The symbol $\top$ (true) if the signature is correct and $\perp$ (false) otherwise.

V1. Set $h \leftarrow \mathsf{os2int}(h_{\mathsf{os}})$, set $r \leftarrow \mathsf{os2int}(r_{\mathsf{os}})$, set $s \leftarrow \mathsf{os2int}(s_{\mathsf{os}})$, and set $P \leftarrow \big(\mathsf{os2int}(P_{\mathsf{os},x}), \mathsf{os2int}(P_{\mathsf{os},y})\big)$.
If $r \notin \mathbb{F}_q^*$ or $s \notin \mathbb{F}_q^*$ or P is not on the elliptic curve P-256, return $\perp$ and stop.

V2. Set $u \leftarrow s^{-1} r \bmod q$, set $v \leftarrow s^{-1} h \bmod q$, and set $Q \leftarrow u\,P + v\,G$.
If $Q = \mathcal{O}$, return $\perp$ and stop.

V3. If $r = (Q_x \bmod p) \bmod q$, return $\top$, otherwise return $\perp$.

2.2 Signature Equations

A signature (r, s) for a message hash h gives rise to the $\mathbb{F}_q$-linear *signature equation*

$$r\,d - s\,k = -h \tag{1}$$

with unknown variables d and k. More generally, signatures $(r_1, s_1), \dots, (r_m, s_m)$ for message hashes $h_1, \dots, h_m$ give rise to the $\mathbb{F}_q$-linear system

$$\begin{aligned} r_1\,d - \;\; s_1\,k_1 &= -\,h_1 \\ &\;\vdots \\ r_m\,d - s_m\,k_m &= -\,h_m \end{aligned} \tag{2}$$

of m equations with unknown variables d and $k_1, \dots, k_m$. Since this linear system is underdetermined, it cannot efficiently be solved for the private key d without additional information on d or $k_1, \dots, k_m$. An attacker may acquire that additional information through computation analysis or fault analysis, which we consider in Section 3 and Section 4, respectively.

3 Computation Analysis

The publication [BHMT16] introduced *differential computation analysis* as a software analogue to differential power analysis of hardware implementations. Computation analysis may exploit any information that is processed during the execution of a software program such as memory contents, accessed memory addresses, function calls, etc.

[4] https://www.qemu.org/

3.1 Explicit Information on Intermediate Values

An unprotected implementation of Algorithm 1 contains several sensitive intermediate values in $\mathbb{F}_q$. First of all, the private key d itself is embedded in the program. The intermediate values rd, $rd + h$, k, and k^{-1} are equally sensitive, since, given an input hash h with corresponding signature (r, s), the private key d can easily be computed from any of those values either directly or via the signature equation (1). Hence, a candidate for those sensitive variables yields a candidate $\widetilde{d}$ for d, which can be checked by testing whether $\widetilde{d}\,G$ equals the public key P. Note that if the public key P would not be available, the candidate public key $\widetilde{P} := \widetilde{d}\,G$ could be verified against a few valid signatures using Algorithm 2 instead.

Since k is used in the scalar multiplication $k\,G$ in step **S3** of Algorithm 1, its bits can also be revealed through the sequence of elliptic curve operations or branching in the scalar multiplication.

3.2 Implicit Information on Intermediate Values

In this subsection we consider implicit information on intermediate values. As explained in Remark 2, implementations could easily be forced to use the message hash as their only source of entropy. In this case, weaknesses in the ephemeral key derivation mechanism $(\mathsf{seed}, \mathsf{rand})$ in Algorithm 1 can be exploited. A heuristic method to detect implementations with this weakness is to request signatures of values h with low Hamming weight or, more generally, with pairwise small Hamming distance.

Ephemeral key collisions. The most obvious weakness is a re-use of an ephemeral key k for different message hashes h. This *ephemeral key collision* is easily spotted because in this case there are two different message hashes h_1, h_2 with signatures (r_1, s_1), (r_2, s_2), respectively, and $r_1 = r_2$. The equation system (2) then simplifies to

$$\begin{aligned} r_1\,d - s_1\,k &= -h_1\,, \\ r_2\,d - s_2\,k &= -h_2\,, \end{aligned} \tag{3}$$

which can easily be solved for k and the private key d.

The reference implementation provided by the organizers of the WhibOx Contest 2021 could be broken by this attack as the example code signed the hash values 0 and 1 with the same ephemeral key. Some further challenges could be broken exactly the same way, i.e. an ephemeral key collision occurred for the same values $h_1 = 0$ and $h_2 = 1$. Presumably, these challenges were obfuscated derivatives of the reference implementation.

In total, 33 out of the 97 challenges of the WhibOx Contest 2021 were susceptible to ephemeral key collisions, see Table 2 in Subsection 5.1 under the identifiers C_1 and C_2. The identifier C_1 denotes challenges with a fixed ephemeral key and C_2 denotes challenges with colliding ephemeral keys for the special inputs $h_1 = 0$ and $h_2 = 1$.

Cross-challenge ephemeral key collisions. For some groups of challenges, the same hash value resulted in the same ephemeral key – for different challenges. In some cases this may have occurred because the same author published two challenges that used the same functions seed and rand in their implementations of Algorithm 1. In this case, two challenges can be attacked simultaneously. The equations (2) result in two systems of equations

$$\begin{aligned} r_1\,d_1 \qquad\quad - s_1\,k_1 \qquad\quad &= -h_1\,, \\ r_2\,d_1 \qquad\qquad\qquad\quad - s_2\,k_2 &= -h_2\,, \\ r_3\,d_2 - s_3\,k_1 \qquad\quad &= -h_1\,, \\ r_4\,d_2 \qquad\quad - s_4\,k_2 &= -h_2\,, \end{aligned} \tag{4}$$

which can be solved together for k_1, k_2 and the two private keys d_1, d_2.

Among the challenges of the WhibOx Contest 2021, we identified six groups of challenges with mutually colliding ephemeral keys. In total, 40 out of 97 challenges were affected by this attack, see Table 2 in Subsection 5.1 under the identifiers $\mathsf{XC}_1, \ldots, \mathsf{XC}_6$. The largest group is labeled by XC_1 and contains (amongst others) challenges derived from the reference implementation. We also used special inputs $h_1 = 0$ and $h_2 = 1$, causing some groups with non-identical ephemeral key derivation mechanisms to merge.

Ephemeral key differential collisions. Another type of weak entropy extraction produces related ephemeral keys. Suppose for four pairwise different message hashes h_1, h_2, h_3, h_4, the entropy extractor produces ephemeral keys $k_1, k_2 = k_1 + t, k_3, k_4 = k_3 + t$, respectively, so that $k_2 - k_1 = k_4 - k_3 = t$. We will refer to this as an *ephemeral key differential collision.* In this case, equation system (2) turns into

$$\begin{aligned} r_1\, d - s_1\, k_1 & & &= -h_1\,,\\ r_2\, d - s_2\, k_1 & & - s_2\, t &= -h_2\,,\\ r_3\, d & - s_3\, k_3 & &= -h_3\,,\\ r_4\, d & - s_4\, k_3 & - s_4\, t &= -h_4\,, \end{aligned} \tag{5}$$

which can easily be solved for k_1, k_3, t and the private key d.

This situation occurs if the entropy extraction process is such that some bits of the ephemeral key k only depend on a proper subset of the bits of the input hash value h. More precisely, let us assume the set of input hash values splits as $U \times V \times W$ with

$$U = \{0,1\}^i, \quad V = \{0,1\}^j \quad \text{and} \quad W = \{0,1\}^{256-i-j}$$

for some $i > 0, j > 0$ with $i + j \leq 256$ and there are functions

$$f\colon U \times W \to \mathbb{F}_q, \quad g\colon V \times W \to \mathbb{F}_q$$

such that the derivation of the ephemeral key k from the input hash value h can be written as

$$k = f(h_u, h_w) + g(h_v, h_w)$$

with $h = (h_u, h_v, h_w) \in U \times V \times W$. If the attacker manages to find four hash values with $h_1 = (h_{u,1}, h_{v,1}, h_w), h_2 = (h_{u,1}, h_{v,2}, h_w), h_3 = (h_{u,2}, h_{v,1}, h_w), h_4 = (h_{u,2}, h_{v,2}, h_w)$, then the corresponding ephemeral keys are

$$\begin{aligned} k_1 &= f(h_{u,1}, h_w) + g(h_{v,1}, h_w)\,, & k_2 &= f(h_{u,1}, h_w) + g(h_{v,2}, h_w)\,,\\ k_3 &= f(h_{u,2}, h_w) + g(h_{v,1}, h_w)\,, & k_4 &= f(h_{u,2}, h_w) + g(h_{v,2}, h_w)\,. \end{aligned}$$

Now, the attacker can use (5) with $t = g(h_{v,2}, h_w) - g(h_{v,1}, h_w)$. To test for and exploit this type of weakness, the attacker signs a fixed hash value h_1, the 256 hash values with exactly one bit in h_1 flipped and another 32,640 hash values with bit-flips in exactly two distinct bit positions in h_1. The attacker then groups the hash values and their signatures into quadruples consisting of the hash value h_1, a hash value with a single bit-flip at position u (this is h_2), a hash value with a single bit-flip at position v, $v \neq u$, and a hash value with bit-flips in positions u and v (these are h_3 and h_4). Afterwards, the attacker solves equation (5) for each of these quadruples and tests whether the resulting private key candidate d corresponds to the given public key. The attack is successful if u is a bit index in U and v is a bit index in V. Observe that this attack only works if the attacker can choose the hash value that is to be signed, rather than choosing the message before it is hashed.

Using this attack we could solve 49 out of the 97 challenges of the WhibOx Contest 2021, see Table 2 in Subsection 5.1 under the identifier DC. Note that this includes the 33 challenges susceptible to plain ephemeral key collisions C_1 and C_2 (in this case, we have $t = 0$).

Implicit information from computation analysis. Next, we consider the general situation, where the attacker obtains coefficients $c_{i,1},\ldots,c_{i,n} \in \mathbb{F}_q$ such that the ephemeral key k_i for input hash h_i is given by

$$k_i = c_{i,1}\,\ell_1 + \cdots + c_{i,n}\,\ell_n\,, \tag{6}$$

where $\ell_1,\ldots,\ell_n \in \mathbb{F}_q$ are values unknown to the attacker. The coefficients $c_{i,j}$ could be obtained by some hypothesis about the implementation as in the previous attacks in this subsection or by computation analysis. Substituting (6) into (2), we obtain the linear system

$$\begin{aligned} r_1\,d - s_1 c_{1,1}\,\ell_1 - \cdots - s_1 c_{1,n}\,\ell_n &= -h_1\,,\\ &\vdots\\ r_m\,d - s_m c_{m,1}\,\ell_1 - \cdots - s_m c_{m,n}\,\ell_n &= -h_m\,, \end{aligned} \tag{7}$$

which in general can be solved for $d, \ell_1, \ldots, \ell_n$ if $m \geq n+1$.

In Subsubsection 5.2.1 and Subsubsection 5.2.2 we provide concrete examples of this attack type using information from source-code and computation analysis, solving six challenges of the WhibOx Contest 2021.

3.3 Partial Information on Intermediate Values

It is well known that knowledge of a few most or least significant bits of the ephemeral keys of several ECDSA signatures is sufficient to recover the private key. This can be accomplished by solving an instance of the Hidden Number Problem [BV96] using lattice basis reduction [HGS01], [NS03]. In the case of curve P-256, also middle bits can be efficiently exploited [vdPSY15]. Finally, implicit partial information on ephemeral keys may be used as well, see [FGR12]. We also refer to the survey [MH20] of methods for key recovery from various kinds of partial information. These techniques, however, were not necessary to break the challenges of the WhibOx Contest 2021 (see Section 5), so we refer here only to the aforementioned literature.

Note that partial information on the intermediate values $rd \bmod q$ and $rd + h \bmod q$ can be utilized in the same way as partial information on k, but partial information on $k^{-1} \bmod q$ seems to be more difficult to exploit.

4 Fault Analysis

Deterministic signature schemes such as dECDSA are highly susceptible to fault attacks (see, e.g., [BP16], [ABF+18], [PSS+18], [RP17], [SB18], [CSC+22]). Consequently, fault attacks can be mounted against unprotected white-box implementations of dECDSA with minimal reverse-engineering efforts.

4.1 White-Box Fault Model

White-box implementations can be altered by attackers at will. Therefore, we extend the term “fault“ to include any kind of modification of a white-box implementation. We refer to the outputs of a modified dECDSA implementation as *faulty signatures*. Faulty signatures can provide information on the secrets embedded in the dECDSA implementation.

In contrast to conventional fault attack settings (e.g. laser fault attacks against hardware implementations), faults can easily be induced in white-box implementations in a controlled and deterministic manner. A modified dECDSA implementation yields a deterministic function that maps given or chosen hash values to faulty signatures.

We consider the following fault model: We assume that an intermediate value $v \in \mathbb{F}_q$ computed or stored by Algorithm 1, such as k, k^{-1}, r, d, rd, h, or $rd + h$, is replaced by a faulty value e (value fault) or by $v + e$ (differential fault), where $e \coloneqq f(h_{\mathsf{os}})$ for some function $f\colon \{0,1\}^{256} \to \mathbb{F}_q$.

In Subsection 4.2, we revisit a simple fault attack that works for almost any function f (*uncontrolled faults*) and requires only one correct/faulty signature pair. Since f is arbitrary, value faults and differential faults are equivalent in this case. In Subsection 4.3 and Subsection 4.4, we consider fault attacks with value faults and differential faults, respectively, that exploit collisions of f. Collisions of f can, for instance, easily be found if the image $F \coloneqq f(\{0,1\}^{256})$ is small. Unlike some previously proposed attacks that require the set F to be known (*controlled faults*) or the fault value e to be recovered by computation analysis, our attack variants are based on collisions of fault values and work without knowledge of f.

As before, we denote the secret key of the white-box implementation by $d \in \mathbb{F}_q^*$. For a hash value $h_i \in \mathbb{F}_q$, we denote the corresponding ephemeral key by $k_i \in \mathbb{F}_q^*$ and the correct signature, computed by the original implementation on input h_i, by $(r_{\mathsf{c},i}, s_{\mathsf{c},i}) \in \mathbb{F}_q^* \times \mathbb{F}_q^*$. The faulty signature, computed by the modified implementation on input h_i, will be denoted by $(r_{\mathsf{f},i}, s_{\mathsf{f},i}) \in \mathbb{F}_q \times \mathbb{F}_q$, and the corresponding fault value by $e_i \in \mathbb{F}_q$.

4.2 Simple Fault Attack

Let $f\colon \{0,1\}^{256} \to \mathbb{F}_q$ be an arbitrary function.

Uncontrolled fault in r (faulty r returned). We assume that an uncontrolled fault is induced in r such that the faulty value of r is returned as part of the faulty signature. In our fault model, step **S3** of Algorithm 1 is replaced by

S3'. Set $r \leftarrow f(h_{\mathsf{os}})$.

In particular, we assume that the ephemeral key k computed in step **S2** remains unchanged. This fault attack could by realized by inducing a fault in the elliptic-curve part of the signature generation algorithm, for instance, by changing the prime p (if used explicitly) or the reduction modulo p, the curve coefficients a, b (if used explicitly), the base point G, the point operations, or the scalar multiplication algorithm. This shows that the attack surface for this fault attack is quite large.

The signature equations (1) of the correct and faulty signature for h_i give rise to the $\mathbb{F}_q$-linear system

$$\begin{aligned} r_{\mathsf{c},i}\, d - s_{\mathsf{c},i}\, k_i &= -h_i\,, \\ r_{\mathsf{f},i}\, d - s_{\mathsf{f},i}\, k_i &= -h_i\,, \end{aligned} \tag{8}$$

which can be solved for d, k_i.

This simple fault attack, using only one correct/faulty signature pair, is possible because the attacker obtains the fault value as first part of the faulty signature and knows that this value is related to the second part of the signature via the signature equation. In the following two subsections we consider fault attacks in which the fault values are not revealed to the attacker.

4.3 Collision Fault Attacks

Let $f\colon \{0,1\}^{256} \to \mathbb{F}_q$ be a function. We assume that collisions of f can easily be found, e.g. that collisions occur with high probability for random inputs or that collisions occur for special inputs such as bit strings of low Hamming weight. This is for instance the case if the image $F \coloneqq f(\{0,1\}^{256})$ is small or, in particular, if $F = \{e\}$ is a singleton. The function f, however, can be unknown.

Value fault in k or k^{-1}. First, we assume that a value fault $e = f(h_{\mathsf{os}})$ is induced in k or k^{-1}. A value fault in k can happen before or after the computation of r. We model the first case by replacing step **S2** of Algorithm 1 by

S2'. Set $(k, z) \leftarrow \mathsf{rand}(z)$ and set $k \leftarrow f(h_{\mathsf{os}})$.

This fault attack could be realized by fixing the input or output of seed in step **S1** such that k becomes constant, but h remains unchanged when used in step **S4**.

A value fault in k after the computation of r can be modeled by replacing step **S4** of Algorithm 1 by

S4'. Set $s \leftarrow f(h_{\mathsf{os}})^{-1}(rd + h) \bmod q$.

Similarly, a value fault in k^{-1} can be modeled by replacing step **S4** by

S4'. Set $s \leftarrow f(h_{\mathsf{os}})(rd + h) \bmod q$.

These fault attacks could be realized by skipping the multiplication by k^{-1} in step **S4** (special case $e = 1$).

For a value fault in k, the signature equation (1) of the faulty signature for h_i yields the $\mathbb{F}_q$-linear equation

$$r_{\mathsf{f},i}\, d - s_{\mathsf{f},i}\, e_i = -h_i \tag{9}$$

with unknowns d, e_i. If we find two inputs $h_i \neq h_j$ with $e_i = e_j$, we can solve the combined linear system for $(d, e_i) = (d, e_j)$. We call this a *(value) fault collision.*

Note that this fault attack does not even require the correct signature. However, since we have $r_{\mathsf{c},i} \neq r_{\mathsf{f},i}$ if the fault happens before the computation of r and $r_{\mathsf{c},i} = r_{\mathsf{f},i}$ otherwise, we can use the correct signature to narrow down the fault location.

Similarly, for a value fault in k^{-1} we obtain the linear equation

$$r_{\mathsf{f},i}\, d - s_{\mathsf{f},i}\, e_i^{-1} = -h_i \tag{10}$$

with unknowns d, e_i^{-1}. If we find two inputs $h_i \neq h_j$ with $e_i = e_j$, we can solve the combined linear system for $(d, e_i^{-1}) = (d, e_j^{-1})$.

Note that a combined system of (9) is solvable if and only if the combined system of (10) is solvable (e_i and e_j are replaced by e_i^{-1} and e_j^{-1}).

Furthermore, faulty signatures $(r_{\mathsf{f},i}, s_{\mathsf{f},i})$ with a fault in k before the computation of r are valid ECDSA signatures, albeit not the ones intended by the implementation. In particular, those faults in k cannot be detected by trial signature verification.

Value fault in r (correct r returned) or rd. Next, we assume that a value fault $e = f(h_{\mathsf{os}})$ is induced in r such that the correct value of r is returned as part of the faulty signature or in rd. We model the first case by replacing step **S4** of Algorithm 1 by

S4'. Set $s \leftarrow k^{-1}\big(f(h_{\mathsf{os}})d + h\big) \bmod q$.

This fault attack could be realized by skipping the multiplication by r in step **S4** (special case $e = 1$).

A value fault in rd can be modeled by replacing step **S4** of Algorithm 1 by

S4'. Set $s \leftarrow k^{-1}\big(f(h_{\mathsf{os}}) + h\big) \bmod q$.

This fault attack could be realized by skipping the addition of rd in step **S4** (special case $e = 0$).

For a value fault in r, the signature equations (1) of the correct and faulty signature for h_i yield the $\mathbb{F}_q$-linear system

$$\begin{aligned} r_{\mathsf{c},i}\, d - s_{\mathsf{c},i}\, k_i \qquad &= -h_i\,, \\ - s_{\mathsf{f},i}\, k_i + e_i d &= -h_i \end{aligned} \tag{11}$$

with unknowns $d, k_i, e_i d$. If we find two inputs $h_i \neq h_j$ with $e_i = e_j$, we can solve the combined linear system for $(d, k_i, k_j, e_i d) = (d, k_i, k_j, e_j d)$.

Similarly, for a value fault in rd we obtain the linear system

$$\begin{aligned} r_{\mathsf{c},i}\, d - s_{\mathsf{c},i}\, k_i \quad &= -h_i\,, \\ - s_{\mathsf{f},i}\, k_i + e_i &= -h_i \end{aligned} \tag{12}$$

with unknowns d, k_i, e_i. If we find two inputs $h_i \neq h_j$ with $e_i = e_j$, we can solve the combined linear system for $(d, k_i, k_j, e_i) = (d, k_i, k_j, e_j)$.

Note that a combined system of (11) is solvable if and only if the combined system of (12) is solvable ($e_i d$ and $e_j d$ are replaced by e_i and e_j).

Value fault in d. Now, we assume that a value fault $e = f(h_{\mathsf{os}})$ is induced in d. In our fault model, step **S4** of Algorithm 1 is replaced by

S4'. Set $s \leftarrow k^{-1}(rf(h_{\mathsf{os}}) + h) \bmod q$.

This fault attack could be realized by skipping the multiplication by d in step **S4** (special case $e = 1$).

The signature equations (1) of the correct and faulty signature for h_i yield the $\mathbb{F}_q$-linear system

$$\begin{aligned} r_{\mathsf{c},i}\, d - s_{\mathsf{c},i}\, k_i \quad &= -h_i\,, \\ - s_{\mathsf{f},i}\, k_i + r_{\mathsf{f},i}\, e_i &= -h_i \end{aligned} \tag{13}$$

with unknowns d, k_i, e_i. If we find two inputs $h_i \neq h_j$ with $e_i = e_j$, we can solve the combined linear system for $(d, k_i, k_j, e_i) = (d, k_i, k_j, e_j)$.

Value fault in h. Here we assume that a value fault $e = f(h_{\mathsf{os}})$ is induced in h. In our fault model, either step **S1** of Algorithm 1 is replaced by

S1'. Set $h \leftarrow f(h_{\mathsf{os}})$ and set $z \leftarrow \mathsf{seed}(h_{\mathsf{os}})$.

or step **S4** is replaced by

S4'. Set $s \leftarrow k^{-1}(rd + f(h_{\mathsf{os}})) \bmod q$.

In particular, we assume that the input of seed in step **S1** remains unchanged. This fault attack could be realized by skipping the addition of h in step **S4** (special case $e = 0$).

The signature equations (1) of the correct and faulty signature for h_i yield the $\mathbb{F}_q$-linear system

$$\begin{aligned} r_{\mathsf{c},i}\, d - s_{\mathsf{c},i}\, k_i \quad &= -h_i\,, \\ r_{\mathsf{f},i}\, d - s_{\mathsf{f},i}\, k_i + e_i &= 0 \end{aligned} \tag{14}$$

with unknowns d, k_i, e_i. If we find two inputs $h_i \neq h_j$ with $e_i = e_j$, we can solve the combined linear system for $(d, k_i, k_j, e_i) = (d, k_i, k_j, e_j)$.

Value fault in $rd + h$. Finally, we assume that a value fault $e = f(h_{\mathsf{os}})$ is induced in $rd + h$. In our fault model, step **S4** of Algorithm 1 is replaced by

S4'. Set $s \leftarrow k^{-1} f(h_{\mathsf{os}}) \bmod q$.

This fault attack could be realized by skipping the multiplication by $rd + h$ in step **S4** (special case $e = 1$).

The signature equations (1) of the correct and faulty signature for h_i yield the $\mathbb{F}_q$-linear system

$$\begin{aligned} r_{\mathsf{c},i}\, d - s_{\mathsf{c},i}\, k_i \quad &= -h_i\,, \\ - s_{\mathsf{f},i}\, k_i + e_i &= 0 \end{aligned} \tag{15}$$

with unknowns d, k_i, e_i. If we find two inputs $h_i \neq h_j$ with $e_i = e_j$, we can solve the combined linear system for $(d, k_i, k_j, e_i) = (d, k_i, k_j, e_j)$.

4.4 Differential Collision Fault Attacks

Let $f\colon \{0,1\}^{256} \to \mathbb{F}_q^*$ be a function. As in Subsection 4.3, we assume that collisions of f can easily be found, but the function f can be unknown.

Differential fault in k. First, we assume that a differential fault $e = f(h_{\mathsf{os}})$ is induced in k. In our fault model, either step **S2** of Algorithm 1 is replaced by

S2'. Set $(k, z) \leftarrow \mathsf{rand}(z)$ and set $k \leftarrow k + f(h_{\mathsf{os}})$.

or **S4** is replaced by

S4'. Set $s \leftarrow (k + f(h_{\mathsf{os}}))^{-1}(rd + h) \bmod q$.

The signature equations (1) of the correct and faulty signature for h_i yield the $\mathbb{F}_q$-linear system

$$\begin{aligned} r_{\mathsf{c},i}\, d - s_{\mathsf{c},i}\, k_i \qquad\quad &= -h_i\,, \\ r_{\mathsf{f},i}\, d - s_{\mathsf{f},i}\, k_i - s_{\mathsf{f},i}\, e_i &= -h_i \end{aligned} \tag{16}$$

with unknowns d, k_i, e_i. If we find two inputs $h_i \neq h_j$ with $e_i = e_j$, we can solve the combined linear system for $(d, k_i, k_j, e_i) = (d, k_i, k_j, e_j)$. We call this a *differential fault collision.*

Note that we have $r_{\mathsf{c},i} \neq r_{\mathsf{f},i}$ if the fault happens before the computation of r, and $r_{\mathsf{c},i} = r_{\mathsf{f},i}$ otherwise.

Differential fault in k^{-1}. Next, we assume that a differential fault $e = f(h_{\mathsf{os}})$ is induced in k^{-1}. In our fault model, step **S4** of Algorithm 1 is replaced by

S4'. Set $s \leftarrow (k^{-1} + f(h_{\mathsf{os}}))(rd + h) \bmod q$.

The correct and faulty signature for h_i satisfy the equations $s_{\mathsf{c},i} = k_i^{-1}(r_{\mathsf{c},i}\, d + h_i)$ and $s_{\mathsf{f},i} = (k_i^{-1} + e_i)(r_{\mathsf{f},i}\, d + h_i)$. Since $r_{\mathsf{c},i} = r_{\mathsf{f},i}$, we get $s_{\mathsf{c},i} - s_{\mathsf{f},i} = -e_i(r_{\mathsf{c},i}\, d + h_i)$. Rearranging yields the $\mathbb{F}_q$-linear equation

$$r_{\mathsf{c},i}\, d + (s_{\mathsf{c},i} - s_{\mathsf{f},i})\, e_i^{-1} = -h_i \tag{17}$$

with unknowns d, e_i^{-1}. If we find two inputs $h_i \neq h_j$ with $e_i = e_j$, we can solve the combined linear system for $(d, e_i^{-1}) = (d, e_j^{-1})$.

Differential fault in r (correct r returned). Now, we assume that a differential fault $e = f(h_{\mathsf{os}})$ is induced in r such that the correct value of r is returned as part of the faulty signature. In our fault model, step **S4** of Algorithm 1 is replaced by

S4'. Set $s \leftarrow k^{-1}((r + f(h_{\mathsf{os}}))d + h) \bmod q$.

The signature equations (1) of the correct and faulty signature for h_i yield the $\mathbb{F}_q$-linear system

$$\begin{aligned} r_{\mathsf{c},i}\, d - s_{\mathsf{c},i}\, k_i \qquad &= -h_i\,, \\ r_{\mathsf{f},i}\, d - s_{\mathsf{f},i}\, k_i + e_i d &= -h_i \end{aligned} \tag{18}$$

with unknowns $d, k_i, e_i d$. If we find two inputs $h_i \neq h_j$ with $e_i = e_j$, we can solve the combined linear system for $(d, k_i, k_j, e_i d) = (d, k_i, k_j, e_j d)$.

Differential fault in d. Here we assume that a differential fault $e = f(h_{\mathsf{os}})$ is induced in d. In our fault model, step **S4** of Algorithm 1 is replaced by

S4'. Set $s \leftarrow k^{-1}(r(d + f(h_{\mathsf{os}})) + h) \bmod q$.

The signature equations (1) of the correct and faulty signature for h_i yield the $\mathbb{F}_q$-linear system

$$\begin{aligned} r_{\mathsf{c},i}\, d - s_{\mathsf{c},i}\, k_i \qquad &= -h_i\,, \\ r_{\mathsf{f},i}\, d - s_{\mathsf{f},i}\, k_i + r_{\mathsf{f},i}\, e_i &= -h_i \end{aligned} \tag{19}$$

with unknowns d, k_i, e_i. If we find two inputs $h_i \neq h_j$ with $e_i = e_j$, we can solve the combined linear system for $(d, k_i, k_j, e_i) = (d, k_i, k_j, e_j)$.

Note that a combined system of (13) is solvable if and only if the combined system of (19) is solvable (e_i and e_j are replaced by $e_i + d$ and $e_j + d$).

Differential fault in rd, h, or $rd + h$. Finally, we assume that a differential fault $e = f(h_{\mathsf{os}})$ is induced in rd, h, or $rd + h$. In our fault model, step **S4** of Algorithm 1 is replaced by

S4'. Set $s \leftarrow k^{-1}(rd + h + f(h_{\mathsf{os}})) \bmod q$.

For a differential fault in h, we could also replace step **S1** by

S1'. Set $h \leftarrow \mathsf{os2int}(h_{\mathsf{os}}) + f(h_{\mathsf{os}})$ and set $z \leftarrow \mathsf{seed}(h_{\mathsf{os}})$,

where we assume that the input of seed in step **S1** is unchanged.

The signature equations (1) of the correct and faulty signature for h_i yield the $\mathbb{F}_q$-linear system

$$\begin{aligned} r_{\mathsf{c},i}\, d - s_{\mathsf{c},i}\, k_i \qquad &= -h_i\,, \\ r_{\mathsf{f},i}\, d - s_{\mathsf{f},i}\, k_i + e_i &= -h_i \end{aligned} \tag{20}$$

with unknowns d, k_i, e_i. If we find two inputs $h_i \neq h_j$ with $e_i = e_j$, we can solve the combined linear system for $(d, k_i, k_j, e_i) = (d, k_i, k_j, e_j)$.

Note that a combined system of (18) is solvable if and only if the combined system of (20) is solvable ($e_i d$ and $e_j d$ are replaced by e_i and e_j).

4.5 Comparison with Previous Work

Several differential fault attacks on deterministic ECDSA and EdDSA were introduced in [ABF+18], which form the foundation of our approach. In particular, attacks with uncontrolled faults in the base point and the scalar multiplication are presented there, which can be subsumed by an uncontrolled fault in r (with faulty r returned). They also consider uncontrolled faults that lead to a constant but unknown ephemeral key, which is a value fault collision in k in our notation. Finally, they present controlled faults with fault values from a small and known set as well as faults induced by operation skipping during the computation of s.

In this paper, we extend the fault attacks presented in [ABF+18]. We systematically consider value fault collisions and differential fault collisions at every step of Algorithm 1, which leads to additional attacks that do not require knowledge on the set of fault values. Operation skipping attacks are subsumed as special cases of value fault collisions in our framework (at the cost of generating an extra correct/faulty signature pair).

In [CSC+22], lattice-based fault attacks on deterministic ECDSA and EdDSA are investigated. There, the authors consider a fixed hash value and induce several random faults, which result in a number of faulty signatures for the given hash value. Based on these faulty signatures, the problem of recovering the private key is then reduced to solving an instance of the Hidden Number Problem (see Subsection 3.3).

By contrast, in this paper we investigate scenarios, where we consider deterministic faults for two distinct hash values $h_i \neq h_j$ and for different steps of the signature computation, and we exploit these faults by using simple linear algebra.

5 Attacking Challenges of the WhibOx Contest 2021

5.1 Automated Attacks

Many of the computation and fault attacks presented in Section 3 and Section 4 can be automated. First, potential entropy input can be removed from the challenges as described in Remark 2. Most of the computation attacks presented in Subsection 3.2 do not require side-channel information and can readily be automated.

To automate fault attacks, we compile each program and compute signatures for a small number of fixed hash values. Next, we iterate through the program and subsequently replace each assembly instruction with one or more `NOP` instructions. In addition, we induce faults in the data segment of the binary. Note that the susceptibility of the binary to attacks with these kinds of fault induction may depend on the compiler and the options used to generate the binary. We generate faulty signatures for all the fixed hash values and for any generated fault, we apply the methods described in Section 4. For the (differential) collision fault attacks, we only test a small number of pairs of correct/faulty signatures. Our results below demonstrate, that (differential) fault collisions are practical in the white-box setting without costly collision search.

The computation and fault attacks considered in our automated attacks are summarized in Table 1 and the results of the automated attacks are shown in Table 2. As we have just mentioned, we had to limit the size of the search space. So if an attack is not listed for a challenge, this does not necessarily imply that the challenge is not vulnerable to this attack.

Table 1: Description of attack and note identifiers.

Id	Description	Reference
C_1	Ephemeral key collision (constant ephemeral key)	(3)
C_2	Ephemeral key collision (chosen hashes)	(3)
XC_i	Cross-challenge ephemeral key collision (collision group i)	(4)
DC	Ephemeral key differential collision (chosen hashes)	(5)
F	Uncontrolled fault in r (faulty r returned)	(8)
FC_1	Value fault in r (correct r returned) or rd	(11), (12)
FC_2	Value/differential fault in d	(13), (19)
FC_3	Value fault in h	(14)
FC_4	Value fault in $rd + h$	(15)
FC_5	Value fault in k or k^{-1}	(9), (10)
FDC_1	Differential fault in r (correct r returned), rd, h, or $rd + h$	(18), (20)
FDC_2	Differential fault in k	(16)
FDC_3	Differential fault in k^{-1}	(17)
ND_1	Non-deterministic challenge (use of `time()`)	Remark 2(a)
ND_2	Non-deterministic challenge (use of uninitialized variables)	Remark 2(b)

Table 2: Results of our automated attacks, see Table 1 for a description of the attack and note identifiers.

Challenge			Attacks		Note
Id	Name	User	CA	FA	
#3	hopeful_liskov	Account	C_2, XC_1, DC	F, $FC_{1,2,3,5}$	
#4	vibrant_jackson	Cronokirby	C_2, XC_1, DC	F, $FC_{1,2,3,5}$, FDC_2	
#8	trusting_bhabha	Cronokirby	C_2, XC_1, DC	F, $FC_{2,3,5}$, FDC_2	
#10	sad_curran	Account	C_2, XC_1, DC	F, $FC_{1,3,5}$, FDC_2	
#11	festive_jennings	Account	C_2, XC_1, DC	F, $FC_{1,3,5}$, FDC_2	
#12	vigilant_wescoff	checc	C_1, XC_1, DC	$FC_{1,2,5}$, FDC_1	
#13	gracious_mcnulty	checc	C_1, XC_1, DC	FC_5	
#15	cool_dubinsky	checc	C_1, XC_1, DC	FC_5	
#16	stupefied_varahamihira	checc	C_1, XC_1, DC	$FC_{1,2,5}$, FDC_1	
#32	clever_hoover	0o0o00o00	C_1, XC_2, DC	F, $FC_{1,2,5}$, $FDC_{1,2}$	
#33	keen_berson	bluecat	DC	F, $FC_{1,3,4,5}$, $FDC_{1,2}$	
#34	determined_goldwasser	Cronokirby	C_2, DC	F, $FC_{1,3,5}$, FDC_2	
#36	frosty_rosalind	Sir Kwit	—	F, FC_2, FDC_1	
#38	epic_dijkstra	Sir Kwit	—	F, FC_2, FDC_1	
#42	practical_franklin	Sir Kwit	—	F, FC_2, FDC_1	
#44	agitated_ritchie	Sir Kwit	—	F, FC_2, FDC_1	
#45	quirky_keller	0o0o00o00	C_1, XC_2, DC	F, $FC_{1,2,5}$, $FDC_{1,2}$	
#50	flamboyant_engelbart	Sir Kwit	—	F, FC_2, FDC_1	
#54	ecstatic_brattain	0o0o00o00	C_1, DC	F, $FC_{1,2,3,4,5}$, $FDC_{1,2}$	ND_1
#55	famous_stonebraker	0o0o00o00	—	F, $FC_{1,2,3,5}$, FDC_2	ND_1
#57	thirsty_fermat	0o0o00o00	—	F, $FC_{2,3,5}$, $FDC_{1,2}$	ND_1
#58	tender_goodall	0o0o00o00	XC_3	F, $FC_{2,3,5}$	ND_1
#61	nostalgic_noether	0o0o00o00	XC_3	F, $FC_{2,3,5}$, FDC_1	ND_1
#62	objective_goldberg	0o0o00o00	—	F, $FC_{2,3,5}$, FDC_1	ND_1
#66	affectionate_johnson	0o0o00o00	XC_4	F, $FC_{2,3,5}$	ND_1
#70	smart_blackwell	0o0o00o00	XC_4	F, $FC_{2,3,5}$	ND_1
#71	sharp_wright	0o0o00o00	XC_4	F, $FC_{2,3,5}$	ND_1
#72	jolly_lamport	0o0o00o00	XC_4	F, $FC_{2,3,5}$, FDC_1	ND_1
#73	heuristic_nobel	0o0o00o00	XC_4	F, $FC_{2,3,5}$, FDC_1	ND_1
#74	bright_lumiere	0o0o00o00	—	F, $FC_{2,3,5}$, FDC_1	ND_1
#76	relaxed_noyce	0o0o00o00	—	F, $FC_{1,2,3,5}$, FDC_1	ND_1
#77	optimistic_booth	0o0o00o00	—	F, $FC_{1,2,3,5}$, FDC_1	ND_1
#78	kind_kalam	0o0o00o00	—	F, $FC_{1,2,3,5}$, FDC_1	ND_1
#79	quizzical_newton	0o0o00o00	—	F, $FC_{1,2,3,5}$, FDC_1	ND_1
#80	suspicious_minsky	0o0o00o00	—	F, $FC_{1,2,3,5}$, FDC_1	ND_1
#81	confident_benz	0o0o00o00	—	F, $FC_{1,2,3,5}$, FDC_1	ND_1
#84	mystifying_galileo	0o0o00o00	—	F, $FC_{1,2,3,5}$, FDC_1	ND_1
#85	boring_knuth	01010	C_1, XC_2, DC	F, $FC_{1,2,5}$, $FDC_{1,2}$	
#87	eager_euler	0o0o00o00	—	F, $FC_{2,3,5}$, FDC_1	ND_1
#89	cocky_hopper	0o0o00o00	—	F, FC_5	ND_1
#94	loving_pasteur	0o0o00o00	—	F, FC_5	ND_1
#96	zen_bardeen	0o0o00o00	—	F, $FC_{3,5}$	ND_1
#97	admiring_lamarr	01010	C_2, XC_1, DC	F, $FC_{1,3,5}$, FDC_2	
#100	hopeful_kirch	Sir Kwit	—	F, FDC_1	
#101	vibrant_morse	Sir Kwit	—	F, FDC_1	
#103	wizardly_allen	0o0o00o00	—	F, FC_5	ND_1
#104	angry_meninsky	0o0o00o00	—	F, $FC_{1,5}$	ND_1
#105	trusting_mestorf	0o0o00o00	—	F, $FC_{1,5}$	ND_1
#107	sad_einstein	0o0o00o00	—	F, $FC_{1,5}$	ND_1
#108	festive_bohr	Sir Kwit	—	F, FDC_1	
#114	condescending_torvalds	BugsBunny	DC	F, $FC_{1,2,3,5}$, FDC_2	
#127	modest_colden	edgarcuisantes	DC	F	
#135	epic_borg	k0bEbRyAnT	XC_5, DC	F, $FC_{1,2,3,5}$, FDC_2	
#136	competent_heyrovsky	k0bEbRyAnT	XC_5, DC	F, $FC_{1,2,3,5}$, FDC_2	ND_1
#139	practical_cori	k0bEbRyAnT	XC_5, DC	F, $FC_{1,2,5}$, $FDC_{1,2}$	ND_1
#153	gallant_ramanujan	mochilo	C_2, XC_1, DC	F, $FC_{1,3,5}$, FDC_2	
#157	happy_carson	bad_rutabaga	C_2, XC_1, DC	F, $FC_{1,3,5}$, FDC_2	
#165	nervous_joliot	bad_rutabaga	C_2, XC_1, DC	F, $FC_{1,2,3,5}$, $FDC_{1,2}$	
#166	hungry_liskov	BugsBunny	—	F, $FC_{3,5}$	
#172	dreamy_curie	bad_rutabaga	C_2, DC	F, $FC_{1,2,3,5}$, $FDC_{1,2}$	

Table 2: Results of our automated attacks, see Table 1 for a description of the attack and note identifiers.

Challenge			Attacks		Note
Id	Name	User	CA	FA	
#174	optimistic_jennings	TENET	C_2, XC_1, DC	F, $FC_{1,3,5}$, FDC_2	
#185	amazing_aryabhata	TENET	C_2, XC_1, DC	F, $FC_{1,3,5}$, FDC_2	
#187	wonderful_roentgen	TENET	C_2, XC_1, DC	F, $FC_{1,3,5}$, FDC_2	
#192	fervent_montalcini	BugsBunny	XC_6	F, $FC_{1,2,3,4,5}$	
#193	zen_clarke	BugsBunny	XC_6	F, $FC_{1,2,3,4,5}$, FDC_1	
#209	cool_panini	mcs	DC	—	
#212	elegant_bell	BugsBunny	—	F, $FC_{1,3,4,5}$	ND_2
#226	clever_kare	zerokey	—	F	
#227	keen_ptolemy	zerokey	DC	F	
#228	determined_jones	GMorseCode	DC	F	
#231	musing_bhaskara	auguste	C_2, XC_1, DC	F, $FC_{1,3,5}$, FDC_2	
#235	nifty_lamport	TENET	C_2, XC_1, DC	F, $FC_{1,3,5}$, FDC_2	
#251	thirsty_mcclintock	bad_rutabaga	C_2, XC_1, DC	F, $FC_{1,2,3,5}$, $FDC_{1,2}$	
#253	priceless_feynman	bad_rutabaga	C_2, DC	FC_5	ND_2
#256	objective_swanson	auguste	—	F, FC_1	
#261	hardcore_kowalevski	auguste	DC	F, $FC_{2,3,5}$, FDC_1	
#262	nervous_davinci	auguste	—	F	
#264	smart_morse	bad_rutabaga	C_2, DC	FC_5	ND_2
#267	heuristic_meninsky	yww	C_2, XC_1, DC	F, $FC_{1,3,5}$, FDC_2	
#274	suspicious_lichterman	TENET	C_2, XC_1, DC	F, $FC_{1,3,5}$, FDC_2	
#283	cocky_bartik	bluecat	DC	F, FC_5	
#299	trusting_heyrovsky	from0to1	C_1, XC_2, DC	F, $FC_{1,2,5}$, $FDC_{1,2}$	
#304	gracious_wilson	bluecat	DC	F, FC_5	
#305	goofy_mayer	Maidei	—	F, FC_2	
#307	stupefied_kepler	BugsBunny	—	F, $FC_{2,3,4,5}$, FDC_1	ND_2
#308	condescending_boyd	GMorseCode	DC	F, FDC_1	
#314	upbeat_banach	bluecat	—	F, $FC_{1,2,3,4,5}$, FDC_2	
#320	gifted_carson	from0to1	C_1, XC_2, DC	F, $FC_{1,2,3,4,5}$, $FDC_{1,2}$	
#321	modest_darwin	from0to1	C_1, XC_2, DC	F, $FC_{1,2,3,4,5}$, $FDC_{1,2}$	
#323	clever_hypatia	from0to1	C_1, XC_2, DC	F, $FC_{1,2,3,4,5}$, $FDC_{1,2}$	
#325	determined_yonath	bluecat	—	F, $FC_{2,3,5}$, FDC_2	
#327	frosty_albattani	scnucrypto	—	F, $FC_{1,2,3,4,5}$, FDC_1	
#328	musing_joliot	mcs	DC	—	
#335	agitated_curie	scnucrypto	—	F, $FC_{1,2,3}$	
#336	quirky_curran	BlackSea	DC	—	
#345	ecstatic_khorana	auguste	DC	F, FC_2, FDC_1	
#346	famous_gary	Maidei	—	F, FC_2	

All our automated fault attacks are *static*, i.e. we modify the program binary prior execution. Clearly, changing both program and data *during* execution provides a much larger attack surface. To give an intuitive example, suppose that statically modifying a program binary results in an invalid output value of a particular function. Then, during execution that function will always return the invalid output. A dynamic fault attack that modifies the execution trace allows to inject an invalid output for only one or some particular executions of the function.

Dynamic fault attacks on binaries, however, require significant more implementation effort, which is why we omit them here. Nevertheless, in Subsubsection 5.2.3 we provide an example of successfully applying additional dynamic fault attacks by manual source-code modification. During the WhibOx Contest 2021 we also solved challenge `#336` (`quirky_curran`), which seems not to be susceptible to our automated fault attacks, with FDC_1 using manual source-code modifications.

5.2 Closer Look at Selected Challenges

In this subsection we take a closer look at selected challenges of the WhibOx Contest 2021.

5.2.1 Challenges by mcs

The user `mcs` submitted the challenges `#209` (`cool_panini`) and `#328` (`musing_joliot`). We exploited the following peculiarity of these challenges: Each program contains a table of 1024 points $P_1, \dots, P_{1024}$ on P-256. Depending on the input hash h, the program computes 64 pairwise distinct indices $j_1, \dots, j_{64} \in \{1, \dots, 1024\}$ and computes

$$r \leftarrow \left(P_{j_1} + \cdots + P_{j_{64}}\right)_x \bmod q\,.$$

Without further source code analysis, it is unclear how the corresponding ephemeral key k is computed. However, we can write it as $k = \ell_{j_1} + \cdots + \ell_{j_{64}}$, where $\ell_j \in \mathbb{F}_q$ denotes the unknown discrete logarithm of P_j with respect to the base point G, i.e. we have $P_j = \ell_j\, G$ for all $j = 1, \dots, 1024$.

We instrumented the program such that, on input h, it outputs the indices $j_1, \dots, j_{64}$ in addition to the signature (r, s) for h. Using this extra information the challenges can be broken as follows: We pick $m \geq 1025$ random hashes $h_1, \dots, h_m$. For each hash h_i, we compute the corresponding signature (r_i, s_i) and indices $j_{i,1}, \dots, j_{i,64}$ using the instrumented program. We have the signature equation $r_i\, d - s_i\, k_i = -h_i$, where the unknown ephemeral key k_i can be written as $k_i = \ell_{j_{i,1}} + \cdots + \ell_{j_{i,64}}$. We obtain the $\mathbb{F}_q$-linear system

$$\begin{aligned} r_1\, d - \ s_1\, \ell_{j_{1,1}} - \cdots - \ s_1\, \ell_{j_{1,64}} &= -h_1\,, \\ &\ \vdots \\ r_m\, d - s_m\, \ell_{j_{m,1}} - \cdots - s_m\, \ell_{j_{m,64}} &= -h_m \end{aligned}$$

with m equations and unknowns $d, \ell_1, \dots, \ell_{1024}$, which is an instance of (7) for suitably defined coefficients $c_{i,j} \in \{0, 1\}$. Since $m \geq 1025$, this linear system can generally be solved for $d, \ell_1, \dots, \ell_{1024}$.

Another way to break these challenges is the ephemeral key differential collision attack described in Subsection 3.2. The nonce k is the sum of $64 = 256/4$ numbers ℓ_j, so looking for differential collisions with the method of Subsection 3.2, one would expect to find $256 - 4 = 252$ bit-flip positions v for every bit-flip position u. However, it turns out that for most bit-flip positions u there are actually 254 bit-flip positions v that lead to a successful attack. A more detailed analysis shows that the ℓ_j are not chosen randomly, but differences $\ell_j - \ell_{j'}$ are the same for many pairs j, j', leading to additional differential collisions.

5.2.2 Challenges by bluecat

The user `bluecat` submitted (in addition to `#33`) the challenges `#283` (`cocky_bartik`), `#304` (`gracious_wilson`), `#314` (`upbeat_banach`), and `#325` (`determined_yonath`). These challenges can be broken in a similar way as those in Subsubsection 5.2.1.

First, we consider the programs of the challenges `#283` and `#304`. Each program contains two points P_1, P_2 on P-256. On input h, the program computes

$$\begin{aligned} h' &\leftarrow h \oplus m \bmod q\,, \\ r &\leftarrow \left(h'\, P_1 + P_2\right)_x \bmod q\,, \end{aligned}$$

where $m \in \{0, 1\}^{256}$ is a fixed and known mask value. The computation of s (and, in particular, k) is obfuscated using a custom virtual machine, which would be more difficult to reverse-engineer. We denote by $\ell_1, \ell_2 \in \mathbb{F}_q$ the unknown discrete logarithms of P_1, P_2 with respect to G, i.e. we have $P_j = \ell_j\, G$ for $j = 1, 2$. Therewith, we can write the corresponding ephemeral key as $k = h'\, \ell_1 + \ell_2$.

We pick three random hashes h_1, h_2, h_3 and compute the corresponding signatures (r_i, s_i) for $j = 1, 2, 3$ using the program. Then, we can solve the $\mathbb{F}_q$-linear system

$$\begin{aligned} r_1\, d - s_1 h_1'\, \ell_1 - s_1\, \ell_2 &= -h_1\,, \\ r_2\, d - s_2 h_2'\, \ell_1 - s_2\, \ell_2 &= -h_2\,, \\ r_3\, d - s_3 h_3'\, \ell_1 - s_3\, \ell_2 &= -h_3 \end{aligned}$$

for the unknowns d, ℓ_1, ℓ_2.

Next, we consider the programs of the challenges `#314` and `#325`. Each program contains three points P_1, P_2, P_3 on P-256. On input h, the program computes

$$r \leftarrow \left(h^2\, P_1 + h\, P_2 + P_3\right)_x \bmod q$$

using Horner's scheme. We write $P_j = \ell_j\, G$ for $j = 1, 2, 3$ and the corresponding ephemeral key can be written as $k = h^2\, \ell_1 + h\, \ell_2 + \ell_3$.

We pick four random hashes $h_1, \ldots, h_4$ and compute the corresponding signatures (r_i, s_i) for $j = 1, \ldots, 4$ using the program. Then, we can solve the $\mathbb{F}_q$-linear system

$$\begin{aligned} r_1\, d - s_1 h_1^2\, \ell_1 - s_1 h_1\, \ell_2 - s_1\, \ell_3 &= -h_1\,, \\ r_2\, d - s_2 h_2^2\, \ell_1 - s_2 h_2\, \ell_2 - s_2\, \ell_3 &= -h_2\,, \\ r_3\, d - s_3 h_3^2\, \ell_1 - s_3 h_3\, \ell_2 - s_3\, \ell_3 &= -h_3\,, \\ r_4\, d - s_4 h_4^2\, \ell_1 - s_4 h_4\, \ell_2 - s_4\, \ell_3 &= -h_4 \end{aligned}$$

for the unknowns $d, \ell_1, \ell_2, \ell_3$.

Note that the challenges `#283` and `#304` are also susceptible to the black-box attack DC, but the challenges `#314` and `#325` are not (see Table 2). The reason is that the ephemeral keys of the former challenges have a "linear" dependence on the input hash h (a mixture of $\mathbb{F}_q$- and $\mathbb{F}_2$-linearity due to the mask m), while the ephemeral keys of the latter challenges have a quadratic dependence on h.

5.2.3 Challenges by Sir Kwit

The user `Sir Kwit` submitted the 8 challenges `#36`, `#38`, `#42`, `#44`, `#50`, `#100`, `#101`, and `#108`. The challenge programs are obfuscated, but they contain a main loop that is easy to understand. In the main loop, some kind of straight-line program (SLP) is executed that uses 7 types of instructions on 32-bit words. The instructions and their operands are decoded from a large binary array in a not-so-obvious way, but by writing the instructions and operands to a file during program execution, it is possible to obtain a cleaned-up version of the SLP without understanding the decoding mechanism. Due to its simple instruction set, an interpreter for the extracted SLP can easily be written in any programming language using just a few lines of code.

The SLP can be described in our own notation as follows: We parse the input hash $h \in \{0,1\}^{256}$ as a sequence $h = (h_7, \ldots, h_0)$ of 32-bit words $h_j \in \{0,1\}^{32}$, which corresponds to the integer $\sum_{j=0}^{7} h_j 2^{32j}$. We denote the length of the SLP by N. The SLP uses an array `MEM` of N 32-bit words as memory. In each step $i = 0, 1, \ldots, N-1$ of the SLP, the array element `MEM`$[i]$ is updated using one of the following instructions with one or two operands:

Instruction	Operand(s)	Meaning	Notes
INP	j	$\texttt{MEM}[i] \leftarrow h_j$	$0 \leq j < 8$
ADD	j_0 , j_1	$\texttt{MEM}[i] \leftarrow (\texttt{MEM}[j_0] + \texttt{MEM}[j_1]) \bmod 2^{32}$	$0 \leq j_0, j_1 < i$
CAR	j_0 , j_1	$\texttt{MEM}[i] \leftarrow (\texttt{MEM}[j_0] + \texttt{MEM}[j_1]) \operatorname{div} 2^{32}$	$0 \leq j_0, j_1 < i$
MLO	j_0 , j_1	$\texttt{MEM}[i] \leftarrow (\texttt{MEM}[j_0] \cdot \texttt{MEM}[j_1]) \bmod 2^{32}$	$0 \leq j_0, j_1 < i$
MHI	j_0 , j_1	$\texttt{MEM}[i] \leftarrow (\texttt{MEM}[j_0] \cdot \texttt{MEM}[j_1]) \operatorname{div} 2^{32}$	$0 \leq j_0, j_1 < i$
NOT	j	$\texttt{MEM}[i] \leftarrow \texttt{MEM}[j] \oplus \texttt{0xFFFFFFFF}$	$0 \leq j < i$
CST	c	$\texttt{MEM}[i] \leftarrow c$	$0 \leq c < 2^{32}$

In a final step, the signature is extracted from memory as

$$(r, s) \leftarrow \big(\texttt{MEM}[o_0], \ldots, \texttt{MEM}[o_{15}]\big) \in \{0,1\}^{512} ,$$

where $0 \leq o_0, \ldots, o_{15} < N$ are given output memory locations.

Note that every MEM-element is written exactly once. Therefore, at the end of the computation, the array MEM contains the complete history of values encountered during the computation. This type of SLP can also be considered as some kind of arithmetic circuit (read the user name `Sir Kwit` out loud).

Our attempts at computation analysis were not successful against these challenges. We just mention that the constants of the SLPs contain the words of $\lfloor 2^{512}/q \rfloor$, which indicates that Barrett's reduction is used for multiplications modulo q. To fully understand the white-box ECDSA implementation, more reverse-engineering efforts would be required.

In addition to the automated attacks reported in Subsection 5.1, we mounted fault attacks against challenge #108 (`festive_bohr`) using manual program modifications. First, we appended the operation $\texttt{MEM}[i_0] \leftarrow \texttt{MEM}[i_0] + 1$ to a fixed step i_0 of the SLP. We tested a subset of the $N = 2,127,669$ possible locations for i_0 of challenge #108 and found that this challenge is susceptible to the fault attacks F, FDC_1, and FDC_3. Observe that FDC_3 is the only fault attack that was never successful in our automated attacks (see Table 2). Inserting the operations $\texttt{MEM}[i_0] \leftarrow 0$ or $\texttt{MEM}[i_0] \leftarrow 1$ at a fixed step i_0 additionally enabled the fault attack FC_2, if the special input hashes $h_1 = 1$ and $h_2 = 2$ were used.

5.2.4 Challenges by zerokey

It is interesting to have a closer look at the challenges #226 (`clever_kare`) and #227 (`keen_ptolemy`) by `zerokey`, as they are the winning challenges and the authors give a description of the implementation and an analysis of potential weaknesses in [BBD+22]. Both challenges can be broken with an uncontrolled fault in r (see Table 2).

Concretely, for challenge #226 removing line 5053

```
__gmpz_mod(l___24989->f___9, (o___22 )(l___24989->f___9), (o___22 )(o___89));
```

produces an uncontrolled fault in r when signing the hash value 1, and the private key can be recovered as described in Subsection 4.2.

For challenge #227 the location of a successful fault depends on the hash value that is to be signed. When signing $h = p$, for example, removing line 3084

```
__gmpz_mod(l___24929->f___9, (o___21 )(l___24929->f___9), (o___21 )(o___78));
```

results in a faulty signature that can be exploited.

This is surprising, because according to Algorithm 4 in [BBD+22] both challenges verify the generated signature. One possible explanation is that the fault is actually induced in the signature verification. The loop in Algorithm 4 keeps modifying an intermediate result until the verification passes. We conjecture that the fault leads to an incorrect signature passing verification and being output in line 7. In our experiments, the faulty first signature component was always off by a small multiple of $p - q$. We assume that the modular reductions we skip to induce the fault is the reduction mod p at the end of the calculation of the x-coordinate that is compared to r to verify the signature. More precisely, if (r, s) is a valid ECDSA signature and we set $r' = r + t(p - q)$ for some t and compute the corresponding $s' = k^{-1}(h + r'd)$, then step **V2** in the signature verification (Algorithm 2) becomes

$$Q = (s')^{-1}r'\,P + (s')^{-1}h\,G = k(h + r'd)^{-1}r'd\,G + k(h + r'd)^{-1}h\,G = k\,G\,.$$

We have that $Q_x = r$; if the reduction $Q_x \bmod p$ is skipped, then we may obtain r' in step **V3** and the verification passes.

6 Discussion of Countermeasures

In this section, we sketch some countermeasures that our designs were built upon. None of them is sufficient to protect the implementation alone, and combining them to achieve a sufficient level of protection is a non-trivial task as the susceptibility of our and other designs against attacks greatly illustrates. Additionally, a countermeasure that is intended to prevent a special kind of attack may open the door for a different attack path.

We focus on three classes of countermeasures: those that make computational analysis difficult, those that impede fault analysis, and standard program obfuscation techniques.

6.1 Countermeasures Against Computation Analysis

Any computational analysis will likely target intermediate values. It is therefore important to ensure that intermediate values do not leak, i.e. do not appear in memory during the computation.

Let $x \in \mathbb{F}_q$ and represent x with two *shares* (a, b) such that $x = a + b$. Transforming x into shared form can be achieved by selecting a random value r and representing x as the share $(x - r, r)$. All field operations in $(\mathbb{F}_q, +, \cdot\,)$ can then be transformed into operations that operate on shares:

$$\begin{aligned}
(a_1, a_2) +_s (b_1, b_2) &:= (a_1 + b_1, a_2 + b_2)\,,\\
(a_1, a_2) \cdot_s (b_1, b_2) &:= (a_1 \cdot b_1 + a_2 \cdot b_2, a_1 \cdot b_2 + a_2 \cdot b_1)\,,\\
(a_1, a_2)^{-1_s} &:= \begin{cases} \left(\frac{a_1}{a_1^2 - a_2^2}, \frac{-a_2}{a_1^2 - a_2^2}\right) & \text{if } a_1^2 \neq a_2^2\,,\\ \left(\frac{a_1 + 1}{(a_1+1)^2 - (a_2-1)^2}, \frac{-(a_2-1)}{(a_1+1)^2 - (a_2-1)^2}\right) & \text{if } a_1 = a_2\,,\\ \text{undefined} & \text{if } a_1 = -a_2\,. \end{cases}
\end{aligned}$$

Theorem 1. *For the map* $\mathsf{unshare}\colon \mathbb{F}_q \times \mathbb{F}_q \to \mathbb{F}_q,\ (u, v) \mapsto u + v$ *and shares* $x, y \in \mathbb{F}_q \times \mathbb{F}_q$, *it holds:*

(a) We have $\mathsf{unshare}(x +_s y) = \mathsf{unshare}(x) + \mathsf{unshare}(y)$.

(b) We have $\mathsf{unshare}(x \cdot_s y) = \mathsf{unshare}(x) \cdot \mathsf{unshare}(y)$.

(c) If $\mathsf{unshare}(x) \neq 0$, *we have* $\mathsf{unshare}(x^{-1_s}) = \mathsf{unshare}(x)^{-1}$.

Similarly, we can extend scalar multiplication to operate on shares. Let Q be an elliptic curve point and (a, b) a share over $\mathbb{F}_q^*$. Then $(a, b)\,Q$ is defined as $a\,Q + b\,Q$. One can directly apply this idea to the signature computation, i.e. convert the input hash into shares, compute steps **S2** and **S3** and turn r into a share (r_1, r_2), compute step **S4** over shares and finally convert (r_1, r_2) and (s_1, s_2) back to the normal representation (r, s) with $r = (r_1 + r_2, s_1 + s_2)$. Using this straight-forward approach, however, leaks r during step **S3**, as it appears as an intermediate value in memory. Therefore, we make use of the following fact

$$\begin{aligned}\big((x_1 + x_2) \bmod p\big) \bmod q &= \left(\left(\left(x_1 - \left\lfloor \frac{x_1 + x_2}{p} \right\rfloor p\right) + x_2\right) \bmod p\right) \bmod q \\ &= \left(\left(x_1 - \left\lfloor \frac{x_1 + x_2}{p} \right\rfloor p\right) + x_2\right) \bmod q \\ &= \left(\left(\left(x_1 - \left\lfloor \frac{x_1 + x_2}{p} \right\rfloor p\right) \bmod q\right) + (x_2 \bmod q)\right) \bmod q\,,\end{aligned}$$

which gives

Theorem 2. *Let $x_1, x_2 \in \{0 \dots p-1\} = \mathbb{F}_p$. Let $y_1, y_2 \in \{0 \dots q-1\} = \mathbb{F}_q$ be defined as*

$$y_2 = x_2 \bmod q \qquad \textit{and} \qquad y_1 = \begin{cases} (x_1 - p) \bmod q & \textit{if } x_2 \geq p - x_1\,, \\ x_1 \bmod q & \textit{otherwise}\,. \end{cases}$$

Then $\big((x_1 + x_2) \bmod p\big) \bmod q = (y_1 + y_2) \bmod q$.

Using $x_2 \geq p - x_1$ instead of $x_1 + x_2 \geq p$, the value $x_1 + x_2$ does not have to be calculated explicitly. From an abstract point of view, Theorem 2 can be seen as changing from a masking over $\mathbb{F}_p$ to a masking over $\mathbb{F}_q$, akin to mask switching from Boolean to arithmetic masking for symmetric ciphers, as observed by Goubin [Gou01].

With these building blocks we propose the following sequence of operations to compute step **S3**:

- Convert the base point G into (randomized) projective coordinates G'.
- Apply scalar multiplication on the $\mathbb{F}_q$-share (k_1, k_2) of k to obtain $(x, y, z) = k\,G' = k_1\,G' + k_2\,G'$.
- Select a random $u \in \mathbb{F}_p$ and compute $\big(u_1, u_2\big) := \big(u, (x - u) \bmod p\big)$.
- Compute in $\mathbb{F}_p$ the share $(w_1, w_2) := (u_1 \cdot z^{-1}, u_2 \cdot z^{-1})$.
- Apply Theorem 2 on the $\mathbb{F}_p$-share (w_1, w_2) to obtain the $\mathbb{F}_q$-share (r_1, r_2).

In this way we computed the signature value $r = (r_1 + r_2) \bmod q$ without r appearing as an intermediate value.

Nevertheless, it remains challenging to implement this sequence of operations without creating attack vectors with respect to computational analysis. In particular, akin to classical side-channel analysis, branching can be utilized to identify key parts of the implementation. Here, we mention the following additional measures:

- For (projective) point addition, a branchless implementation can be utilized, as for instance introduced in [RCB16].
- A straight-forward implementation of double & add for scalar multiplication with dedicated functions is not only susceptible to side-channel attacks, but an attacker with access to the function call graph can furthermore easily extract the scalar value. Since computational resources on a standard computer are not as limited as on an embedded system – such as smartcards – one can implement scalar multiplication as add & add instead of classical double & add. The call graph will then show only a sequence of calls of add.
- Sacrificing even more computational resources, one can further implement add & add always to hide any correlation between the scalar bits and called functions.

6.2 Countermeasures Against Fault Analysis

Extensive research has been conducted on constructing fault-proof ECC implementations. However, they are usually motivated from the world of smartcards and embedded systems, where the attack surface is usually much smaller. Using electromagentic or laser fault attacks one can assume that it is possible to induce uncontrolled faults into intermediate values, but one can usually exclude the possibility that an attacker can either read out values from memory, and it is usually also much more difficult to change values in a controlled manner. Therefore, the main focus is often to avoid uncontrolled faults [RP17] and to implement a fault-resistant scalar multiplication, as done e.g. in [FPBS16] and [Joy20].

Redundant computation. It is not difficult to modify a program with faulty values. However, if two related values exist, it usually requires a significant reverse engineering effort in order to identify these two related values and to modify two corresponding values in a controlled manner. One can thus implement the signature computation in a redundant way, compare the results and potentially also intermediate results, and only output the signature value if the comparisons succeeded. Constant values can be further verified using checksums. Nevertheless, once the location of the comparison is identified, a fault attack can again be mounted.

Infective computation. As a consequence of the above considerations it desirable to modify the signature computation by introducing additional pseudo-random values to ensure that the system of equations that originates from faults becomes unsolvable. We first illustrate this approach by adapting the infective computation countermeasuse described in [RP17] from EdDSA to dECDSA with the aim to prevent uncontrolled faults in r with faulty r returned (see Subsection 4.2).

We write the private key as $d = d_1 + d_2$ and assume that the static additive shares d_1, d_2 are embedded in the implementation. Furthermore, the static shares will be re-randomized as $d_1 - v$ and $d_2 + v$ using a pseudo-random value v that changes for different input hashes. We compute r twice as r' and r'' using different implementations and assume that faults resulting in faulty values $r'_{\mathsf{f}}, r''_{\mathsf{f}}$ of these variables with $r'_{\mathsf{f}} = r''_{\mathsf{f}} \neq r$ cannot be induced without changing k. The value r' is output as first part of the signature and s is computed using both r' and r'' as follows:

1. Set $h \leftarrow \mathsf{os2int}(h_{\mathsf{os}})$ and set $z \leftarrow \mathsf{seed}(h_{\mathsf{os}})$.

2. Set $(v, z) \leftarrow \mathsf{rand}(z)$, set $d' \leftarrow d_1 - v$, and set $d'' \leftarrow d_2 + v$.

3. Set $(k, z) \leftarrow \mathsf{rand}(z)$.

4. Compute $r' \leftarrow \big(k\, G^{(1)}\big)_x$ and $r'' \leftarrow \big(k\, G^{(2)}\big)_x$ using different implementations, where $G^{(1)}, G^{(2)}$ denote copies of the base point.

5. Set $s \leftarrow k^{-1}(r'd' + r''d'' + h)$.

6. Return $\big(\mathsf{int2os}_{32}(r'), \mathsf{int2os}_{32}(s)\big)$.

We have $d = d_1 + d_2 = d' + d''$. If no faults occur, we have $r = r' = r''$ and $s = k^{-1}(rd_1 + rd_2 + h) = k^{-1}(rd + h)$, hence we obtain a correct signature.

Next, we assume that an uncontrolled fault is induced in r' (but not in k) such that the faulty value is output as first part of the signature. In this case, r'' is equal to the correct r. With the notation of Section 4, we obtain the linear system

$$\begin{aligned} r_{\mathsf{c},1}\, d_1 + r_{\mathsf{c},1}\, d_2 - s_{\mathsf{c},1}\, k_1 \qquad\qquad\qquad\quad &= -h_1\,, \\ r_{\mathsf{f},1}\, d_1 + r_{\mathsf{c},1}\, d_2 - s_{\mathsf{f},1}\, k_1 + (r_{\mathsf{c},1} - r_{\mathsf{f},1})\, v_1 &= -h_1\,, \\ r_{\mathsf{c},2}\, d_1 + r_{\mathsf{c},2}\, d_2 - s_{\mathsf{c},2}\, k_2 \qquad\qquad\qquad\quad &= -h_2\,, \\ r_{\mathsf{f},2}\, d_1 + r_{\mathsf{c},2}\, d_2 - s_{\mathsf{f},2}\, k_2 + (r_{\mathsf{c},2} - r_{\mathsf{f},2})\, v_2 &= -h_2 \end{aligned}$$

with unknowns $d_1, d_2, k_1, k_2, v_1, v_2$. This system of equations is underdetermined due to the additional unknowns v_1, v_2. With additional correct/faulty signature pairs, the system remains underdetermined, because every new input hash h_i introduces two new unknowns k_i, v_i. Note that if the static shares d_1, d_2 are not re-randomized (i.e., if $v = 0$), this linear system could be solved.

However, other fault attacks presented in Section 4 remain still possible. For instance, if faults are induced such that $r' = r''$ (both correct or both faulty), the contribution of v in step 5 will typically cancel out.

In general, additive blinding is not effective against differential faults in the additive shares, because these faults are equivalent to differential faults in the original variables. Therefore, we combine the approach adapted from [RP17] with multiplicative blinding. To this end, we augment the interface of the deterministic random number generator $(\mathsf{seed}, \mathsf{rand})$ by a procedure $\mathsf{reseed}\colon \{0,1\}^{256} \times Z \to Z$ that allows to update the internal state of the generator using additional input. Just before the derivation of the (multiplicatively blinded) ephemeral key, we will reseed the random number generator using previously computed variables, in order to make the ephemeral key dependent on faults that might already have occurred. The signature generation works as follows:

1. Set $h \leftarrow \mathsf{os2int}(h_{\mathsf{os}})$, set $z \leftarrow \mathsf{seed}(h_{\mathsf{os}})$, set $(u', z) \leftarrow \mathsf{rand}(z)$, set $(u'', z) \leftarrow \mathsf{rand}(z)$, set $u \leftarrow u'u''$, set $u'^{(1)} \leftarrow u'^{(2)} \leftarrow u'$, set $u'_{\text{inv}} \leftarrow (u')^{-1}$, and set $u^{(1)}_{\text{inv}} \leftarrow u^{(2)}_{\text{inv}} \leftarrow u^{-1}$.

2. Set $(v, z) \leftarrow \mathsf{rand}(z)$, set $d' \leftarrow u''\, d_1 - v$, and set $d'' \leftarrow u''\, d_2 + v$.

3. Set $(h', z) \leftarrow \mathsf{rand}(z)$ and set $h'' \leftarrow uh - h'$.

4. For all $c \in \{h', h'', d', d'', u'^{(1)}, u'^{(2)}, u'_{\text{inv}}, u^{(1)}_{\text{inv}}, u^{(2)}_{\text{inv}}\}$, update $z \leftarrow \mathsf{reseed}(\mathsf{int2os}_{32}(c), z)$. Finally, set $(k', z) \leftarrow \mathsf{rand}(z)$.

5. Compute $r' \leftarrow u'^{(1)} \cdot \big(u^{(1)}_{\text{inv}}(k'\, G^{(1)})\big)_x$ and $r'' \leftarrow u'^{(2)} \cdot \big(k'(u^{(2)}_{\text{inv}}\, G^{(2)})\big)_x$ using different implementations, where $G^{(1)}, G^{(2)}$ denote copies of the base point.

6. Set $s \leftarrow (k')^{-1}\big((r'd' + h') + (r''d'' + h'')\big)$.

7. Return $\big(\mathsf{int2os}_{32}(u'_{\text{inv}}\, r'), \mathsf{int2os}_{32}(s)\big)$.

First, we show that these steps generate a valid signature: We have $d' + d'' = u''d_1 + u''d_2 = u''d$ (step 2) and $h' + h'' = uh$ (step 3). If we interpret k' (generated in step 4) as uk, that means the ephemeral key is defined as $k \coloneqq u^{-1}k'$, we get $r' = r'' = u'r$ (step 5). Therefore, we obtain $r = (u')^{-1}\, r'$ and

$$s = (k')^{-1}(r'd' + h' + r''d'' + h'') = (uk)^{-1}(u'ru''d + uh) = k^{-1}(rd + h)\,, \tag{21}$$

as required.

As before, we assume that it is infeasible to induce faults such that k' remains unchanged, but both

$$r^{(1)} \coloneqq \big(u^{(1)}_{\text{inv}}(k'\, G^{(1)})\big)_x \quad \text{and} \quad r^{(2)} \coloneqq \big(k'(u^{(2)}_{\text{inv}}\, G^{(2)})\big)_x$$

are altered to the same faulty value. This should be ensured by implementing two different scalar multiplications. If no fault occurs, we have $r^{(1)} = r^{(2)} = r$.

In the following, we illustrate how the proposed algorithm prevents several fault attacks we covered in Section 4. Observe that for an input hash h_i, we always have the signature equation $r_{\mathsf{c},i}\, d - s_{\mathsf{c},i}\, k_i = -h_i$ of the correct signature $(r_{\mathsf{c},i}, s_{\mathsf{c},i})$ with unknowns d and k_i. The goal of an attacker is to induce faults such that the faulty signature $(r_{\mathsf{f},i}, s_{\mathsf{f},i})$ for h_i yields an additional equation making the combined linear system solvable, in particular with respect to d. The additional equation can be utilized if it contains only the unknowns d (or d_1 and d_2) and k_i or an additional unknown consisting of the fault value e_i (or $c\, e_i$ or $c\, e_i^{-1}$ for some unknown constant c) that can be eliminated by a second correct/faulty signature pair for $h_j \neq h_i$ when a fault collision $e_i = e_j$ occurs (see Subsection 4.3 and Subsection 4.4). If, on the other hand, the additional equation depends on a monomial that involves a blinding value that changes with every input h_i, the combined linear system will remain underdetermined and is deemed unsolvable by our analysis.

We first observe that we may disregard faults in the variables d_1, d_2, h, because this would lead to a different ephemeral key being derived due to reseeding in step 4. Note, however, that we cannot rule out faults in the variables used for reseeding to occur *after* reseeding. Therefore, at least the intermediate values $u'^{(1)}$, $u'^{(2)}$, $r^{(1)}$, $r^{(2)}$, r', r'', k', $(k')^{-1}$, h', h'', d', d'', $r'd'$, $r''d''$, $r'd' + h'$, $r''d'' + h''$, and $r'd' + h' + r''d'' + h''$ remain to be considered as fault locations for value and differential faults. Additionally, uncontrolled faults in the values $r^{(1)}$, r', and $u'^{(1)}$ might potentially leak information on the fault value via r_{f}. Here, we limit ourselves to analyze a few typical cases:

- **Uncontrolled fault in $\boldsymbol{r^{(1)}}$, $\boldsymbol{r'}$, or $\boldsymbol{u'^{(1)}}$.** At first we investigate whether the simple fault attack presented in Subsection 4.2 is applicable in this setting. Assume that $r^{(1)}$ is replaced by a fault value e in step 5. Then $r_{\mathsf{f}} = e$, hence r' gets $u' r_{\mathsf{f}}$ after the fault. By assumption, we have $r_{\mathsf{c}} = r^{(2)} \neq r_{\mathsf{f}}$. We obtain the linear equation

  $$r_{\mathsf{f}}\, d_1 + r_{\mathsf{c}}\, d_2 - s_{\mathsf{f}}\, k + (r_{\mathsf{c}} - r_{\mathsf{f}})(u'')^{-1} v = -h\,,$$

  which cannot be utilized due to the unknown $(u'')^{-1}v$ with non-zero coefficient.
  Now assume that r' is replaced by a fault value e in step 5. Then $r_{\mathsf{f}} = (u')^{-1}e$, hence r' equals $u' r_{\mathsf{f}}$ after the fault, as before.
  Next, assume that $u'^{(1)}$ is replaced by a fault value e in step 5. Then $r_{\mathsf{f}} = (u')^{-1} e r_{\mathsf{c}}$, hence r' gets $e r_{\mathsf{c}} = u' r_{\mathsf{f}}$ after the fault, as before.

- **Value fault in $\boldsymbol{r^{(2)}}$.** Now we assume that $r^{(2)}$ is replaced by a fault value $e \neq r_{\mathsf{c}}$. By assumption, we have $r_{\mathsf{f}} = r^{(1)} = r_{\mathsf{c}}$. We obtain the linear equation

  $$r_{\mathsf{c}}\, d_1 + e\, d_2 - s_{\mathsf{f}}\, k + (e - r_{\mathsf{c}})(u'')^{-1} v = -h\,,$$

  which cannot be utilized due to the unknown $(u'')^{-1}v$. Note that this term only vanishes if $e = r_{\mathsf{c}}$, i.e. if no fault occurs.

- **Value fault in $\boldsymbol{k'}$ or $\boldsymbol{(k')^{-1}}$.** Now we assume that k' is replaced by a fault value e. Then $k = u^{-1}e$ and we obtain the linear equation

  $$r_{\mathsf{f}}\, d - s_{\mathsf{f}}\, u^{-1} e = -h\,,$$

  which cannot be utilized due to the unknown $u^{-1}e$. For a value fault in $(k')^{-1}$ the situation is similar (e is replaced by e^{-1}).

- **Value fault in $\boldsymbol{r'd' + h' + r''d'' + h''}$.** Now we assume that $r'd' + h' + r''d'' + h''$ is replaced by a fault value e. We obtain the linear equation

  $$-s_{\mathsf{f}}\, uk + e = 0\,,$$

  which cannot be utilized due to the unknown uk.

- **Differential fault in $(k')^{-1}$.** Now we assume that a fault value e is added to $(k')^{-1}$. Then $s_\mathsf{f} = (k^{-1} + ue)(r_\mathsf{f} d + h)$ and as in (17) we obtain the linear equation

$$r_\mathsf{c}\, d + (s_\mathsf{c} - s_\mathsf{f})\, u^{-1} e^{-1} = -h\,,$$

  which cannot be utilized due to the unknown $u^{-1}e^{-1}$.

- **Differential fault in $r'd'$, $r''d''$, h', h'', $r'd' + h'$, $r''d'' + h''$, or $r'd' + h' + r''d'' + h''$.** Now we assume that a fault value e is added to $r'd' + h' + r''d'' + h''$ or, equivalently, to one of its summands. We obtain the linear equation

$$r_\mathsf{f}\, d - s_\mathsf{f}\, k + u^{-1} e = -h\,,$$

  which cannot be utilized due to the unknown $u^{-1}e$.

The omitted cases admit a similar analysis. The analysis demonstrates that many fault attacks would be possible if multiplicative blinding were not used (i.e., if $u = u' = u'' = 1$), hence additive blinding with v alone is not an effective countermeasure against fault attacks.

Of course, other faults attacks outside our fault model might still apply. In particular, we didn't consider attacks using multiple faults and attacks combining fault with computation analysis.

6.3 Program Obfuscation

In order to hide the implementation and make it harder for an attacker to make sense of the implemented functions, several obfuscation methods are available. They are heavily employed in the field of malware creation and digital rights management. Examples include flattening the control flow graph [Wan00], self-modifying code or virtualization [Rol09]. Since all these techniques affect the control flow, they can on the downside make the generated binary more susceptible to faults. Consider for example Tigress [Col18][5], a source-to-source virtualizer which was used by many submissions in the WhibOx Contest 2021 and the simple C-program

```
int x = 7;
printf("The result is: %i\n", x);
```

A straight-forward method of inducing faults is to replace a valid instruction by a `NOP` instruction. Here, the compiled code includes only three instructions at `0x1155`, `0x115c` and `0x115f` that affect the output of the program. These instructions set up the constant value `0x7` as an argument to the `printf` function.

```
1155:     c7 45 fc 07 00 00 00    movl   $0x7,-0x4(%rbp)
115c:     8b 45 fc                mov    -0x4(%rbp),%eax
115f:     89 c6                   mov    %eax,%esi
                                  [...]
116d:     e8 de fe ff ff          callq  1050 <printf@plt>
```

If we consider replacing each instruction with a `NOP`, three distinct faults with incorrect output values can be generated. Virtualizing this function with Tigress, however, yields a complex control structure with several possible continuations, depending on the executed VM instruction. Inducing faults by replacing instructions with `NOP` then generates more than 22 different faulty output values. We also tested our proposed countermeasure of Section 6 with respect to such an attack. Whereas the raw implementation was not susceptible to static faults by `NOP`-ing out instructions, the virtualized version employing Tigress could be broken by an uncontrolled fault in r.

[5] https://tigress.wtf/

7 Conclusion

In this paper we provided a systematic overview of different computational and fault attacks that are relevant for white-box implementations of ECDSA and proposed different countermeasures to prevent and/or complicate them. We applied the attacks to evaluate the submissions of the WhibOx Contest 2021 and our analysis showed that all challenges could be broken, often by several attacks, indicating that asymmetric white-box cryptography is a challenging task where much further research is needed.

There are at least two different directions for future work: On the one hand, we intend to investigate in further countermeasures to prevent the presented attacks (and possibly other ones as well). On the other hand, it might be interesting to generalize further classes of attacks, e.g. from the domain of side channel analysis, to the asymmetric ECDSA white-box setting. Indeed, in particular differential computational analysis, the white-box analogue of differential power analysis, has proven to be powerful analysis tool for symmetric white-box implementations, see e.g. [BHMT16, BRVW19]. It is, however, not straightforward to generalize it to the asymmetric ECDSA case. In general, in an asymmetric setting, the difficulty arises which part of the algorithm to target, which is not the case for symmetric algorithms, where usually S-boxes constitute promising targets.

References

[ABABM20] Estuardo Alpirez Bock, Alessandro Amadori, Chris Brzuska, and Wil Michiels. On the security goals of white-box cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):327–357, 2020.

[ABF+18] Christopher Ambrose, Joppe W. Bos, Björn Fay, Marc Joye, Manfred Lochter, and Bruce Murray. Differential attacks on deterministic signatures. In Nigel P. Smart, editor, *Topics in Cryptology – CT-RSA 2018*, volume 10808 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2018.

[Bar20a] Lucas Barthelemy. *A First Approach To Asymmetric White-Box Cryptography and a Study of Permutation Polynomials Modulo* 2^n *in Obfuscation*. PhD thesis, Sorbonne Université, Paris, France, 12 2020. PhD Thesis.

[Bar20b] Lucas Barthelemy. Toward an asymmetric white-box proposal. Cryptology ePrint Archive, Report 2020/893, 2020. https://ia.cr/2020/893.

[BBD+22] Guillaume Barbu, Ward Beullens, Emmanuelle Dottax, Christophe Giraud, Agathe Houzelot, Chaoyun Li, Mohammad Mahzoun, Adrián Ranea, and Jianrui Xie. ECDSA white-box implementations: Attacks and designs from whibox 2021 contest. Cryptology ePrint Archive, Report 2022/385, 2022. https://ia.cr/2022/385.

[BCD06] Julien Bringer, Hervé Chabanne, and Emmanuelle Dottax. White box cryptography: Another attempt. Cryptology ePrint Archive, Report 2006/468, 2006. https://ia.cr/2006/468.

[BGEC04] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a white box AES implementation. In *International workshop on selected areas in cryptography*, pages 227–240. Springer, 2004.

[BGI+01] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Annual international cryptology conference*, pages 1–18. Springer, 2001.

[BHMT16] Joppe W. Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. Differential computation analysis: Hiding your white-box designs is not enough. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, pages 215–236, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[BP16] Alessandro Barenghi and Gerardo Pelosi. A note on fault attacks against deterministic signature schemes. In Kazuto Ogawa and Katsunari Yoshioka, editors, *Advances in Information and Computer Security*, volume 9836 of *Lecture Notes in Computer Science*, pages 182–192. Springer, 2016.

[BRVW19] Andrey Bogdanov, Matthieu Rivain, Philip S. Vejre, and Junwei Wang. Higher-order DCA against standard side-channel countermeasures. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 118–141. Springer, 2019.

[BU18] Alex Biryukov and Aleksei Udovenko. Attacks and countermeasures for white-box designs. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 373–402. Springer, 2018.

[BV96] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In *Annual International Cryptology Conference*, pages 129–142. Springer, 1996.

[CC19] CryptoExperts and Cybercrypt. CHES 2019 capture the flag challenge – the whibOx contest – edition 2. ECRYPT CSA, 2019. https://whibox.io/contests/2019/.

[CEJO02] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C Van Oorschot. White-box cryptography and an AES implementation. In *International Workshop on Selected Areas in Cryptography*, pages 250–270. Springer, 2002.

[CEJvO03] Stanley Chow, Phil Eisen, Harold Johnson, and Paul C. van Oorschot. A white-box DES implementation for DRM applications. In Joan Feigenbaum, editor, *Digital Rights Management*, pages 1–15. Springer Berlin Heidelberg, 2003.

[Col18] Christian Collberg. Tigress: A source-to-source-ish obfuscation tool. Proc. 8th Workshop on Software Security Protection, and Reverse Engineering, 2018.

[Cry16] CryptoExperts. White-box cryptography and obfuscation. ECRYPT CSA, 2016. https://www.cryptoexperts.com/whibox2016/.

[CSA17] ECRYPT CSA. CHES 2017 capture the flag challenge – the whibOx contest, 2017. https://whibox.io/contests/2017/.

[CSC+22] Weiqiong Cao, Hongsong Shi, Hua Chen, Jiazhe Chen, Limin Fan, and Wenling Wu. Lattice-based fault attacks on deterministic signature schemes of ECDSA and EdDSA. In *Topics in Cryptology – CT-RSA 2022*, volume 13161, page 169. Springer Nature, 2022.

[DGH21] Emmanuelle Dottax, Christophe Giraud, and Agathe Houzelot. White-box ECDSA: challenges and existing solutions. In Shivam Bhasin and Fabrizio De Santis, editors, *Constructive Side-Channel Analysis and Secure Design – COSADE 2021*, volume 12910 of *Lecture Notes in Computer Science*, pages 184–201. Springer, 2021.

[DLPR13] Cécile Delerablée, Tancrède Lepoint, Pascal Paillier, and Matthieu Rivain. White-box security notions for symmetric encryption schemes. In *International Conference on Selected Areas in Cryptography*, pages 247–264. Springer, 2013.

[DLPR14] Cécile Delerablée, Tancrède Lepoint, Pascal Paillier, and Matthieu Rivain. White-box security notions for symmetric encryption schemes. In Tanja Lange, Kristin Lauter, and Petr Lisoněk, editors, *Selected Areas in Cryptography – SAC 2013*, pages 247–264, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[DMRP13] Yoni De Mulder, Peter Roelse, and Bart Preneel. Revisiting the BGE attack on a white-box AES implementation. Cryptology ePrint Archive, Report 2013/450, 2013. https://ia.cr/2013/450.

[FGR12] Jean-Charles Faugere, Christopher Goyet, and Guénaël Renault. Attacking (EC)DSA given only an implicit hint. In *International Conference on Selected Areas in Cryptography – SAC 2012*, pages 252–274. Springer, 2012.

[FHW+19] Qi Feng, Debiao He, Huaqun Wang, Neeraj Kumar, and Kim-Kwang Raymond Choo. White-box implementation of Shamir's identity-based signature scheme. *IEEE Systems Journal*, 14(2):1820–1829, 2019.

[FPBS16] Apostolos P Fournaris, Louiza Papachristodoulou, Lejla Batina, and Nicolas Sklavos. Residue number system as a side channel and fault injection attack countermeasure in elliptic curve cryptography. In *2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–4. IEEE, 2016.

[GG22] Pierre Galissant and Louis Goubin. Resisting key-extraction and code-compression: a secure implementation of the HFE signature scheme in the white-box model. Cryptology ePrint Archive, Report 2022/138, 2022. https://ia.cr/2022/138.

[GMQ07] Louis Goubin, Jean-Michel Masereel, and Michaël Quisquater. Cryptanalysis of white box DES implementations. In *International workshop on selected areas in cryptography*, pages 278–295. Springer, 2007.

[Gou01] Louis Goubin. A sound method for switching between Boolean and arithmetic masking. In *Cryptographic Hardware and Embedded Systems - CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2001.

[GPRW19] Louis Goubin, Pascal Paillier, Matthieu Rivain, and Junwei Wang. How to reveal the secrets of an obscure white-box implementation. *Journal of Cryptographic Engineering*, pages 1–18, 2019.

[GRW20] Louis Goubin, Matthieu Rivain, and Junwei Wang. Defeating state-of-the-art white-box countermeasures with advanced gray-box attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):454–482, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8597/8164.

[HGS01] Nick A Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography*, 23(3):283–290, 2001.

[JBF02] Matthias Jacob, Dan Boneh, and Edward Felten. Attacking an obfuscated cipher by injecting faults. In *ACM workshop on digital rights management*, pages 16–31. Springer, 2002.

[Joy20] Marc Joye. Protecting ECC against fault attacks: the ring extension method revisited. *Journal of Mathematical Cryptology*, 14(1):254–267, 2020.

[Kar10] Mohamed Karroumi. Protecting white-box AES with dual ciphers. In *International conference on information security and cryptology*, pages 278–291. Springer, 2010.

[KG13] Cameron F. Kerry and Patrick D. Gallagher. FIPS PUB 186-4 Digital Signature Standard (DSS), 2013.

[LN05] Hamilton E Link and William D Neumann. Clarifying obfuscation: improving the security of white-box DES. In *International Conference on Information Technology: Coding and Computing*, volume 1, pages 679–684. IEEE, 2005.

[LR13] Tancrède Lepoint and Matthieu Rivain. Another nail in the coffin of white-box AES implementations. Cryptology ePrint Archive, Report 2013/455, 2013. https://ia.cr/2013/455.

[LRM+13] Tancrède Lepoint, Matthieu Rivain, Yoni De Mulder, Peter Roelse, and Bart Preneel. Two attacks on a white-box AES implementation. In *International conference on selected areas in cryptography*, pages 265–285. Springer, 2013.

[MH20] Gabrielle De Micheli and Nadia Heninger. Recovering cryptographic keys from partial information, by example. Cryptology ePrint Archive, Report 2020/1506, 2020. https://ia.cr/2020/1506.

[MRP12] Yoni De Mulder, Peter Roelse, and Bart Preneel. Cryptanalysis of the Xiao–Lai white-box AES implementation. In *International conference on selected areas in cryptography*, pages 34–49. Springer, 2012.

[MWP10] Yoni De Mulder, Brecht Wyseur, and Bart Preneel. Cryptanalysis of a perturbated white-box AES implementation. In *International Conference on Cryptology in India*, pages 292–310. Springer, 2010.

[NS03] Phong Q Nguyen and Igor E Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Designs, codes and cryptography*, 30(2):201–217, 2003.

[Por13] Thomas Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979, 2013.

[PSS+18] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 338–352. IEEE, 2018.

[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*, volume 9665 of *Lecture Notes in Computer Science*, pages 403–428. Springer, 2016.

[Rol09] Rolf Rolles. Unpacking virtualization obfuscators. In *Proc. 3rd USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[RP17] Yolan Romailler and Sylvain Pelissier. Practical fault attack against the Ed25519 and EdDSA signature schemes. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 17–24. IEEE, 2017.

[SB18] Niels Samwel and Lejla Batina. Practical fault injection on deterministic signatures: the case of EdDSA. In *Progress in Cryptology – AFRICACRYPT 2018*, pages 306–321. Springer, 2018.

[SWP09] Amitabh Saxena, Brecht Wyseur, and Bart Preneel. Towards security notions for white-box cryptography. In *International Conference on Information Security*, pages 49–58. Springer, 2009.

[vdPSY15] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In Kaisa Nyberg, editor, *Topics in Cryptology - CT-RSA 2015*, volume 9048 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2015.

[Wan00] Chenxi Wang. *A scecurity architecture for survivability mechanism*. PhD thesis, University of Virginia, Charlottesville, USA, 2000. PhD Thesis.

[WMGP07] Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel. Cryptanalysis of white-box DES implementations with arbitrary external encodings. In *International workshop on selected areas in cryptography*, pages 264–277. Springer, 2007.

[XL09] Yaying Xiao and Xuejia Lai. A secure implementation of white-box AES. In *2009 2nd International Conference on Computer Science and its Applications*, pages 1–6. IEEE, 2009.

[ZBJ20] Jie Zhou, Jian Bai, and Meng Shan Jiang. White-box implementation of ECDSA based on the cloud plus side mode. *Secur. Commun. Networks*, 2020:8881116:1–8881116:10, 2020.

[ZHH+20] Yudi Zhang, Debiao He, Xinyi Huang, Ding Wang, Kim-Kwang Raymond Choo, and Jing Wang. White-box implementation of the identity-based signature scheme in the IEEE P1363 standard for public key cryptography. *IEICE TRANSACTIONS on Information and Systems*, 103(2):188–195, 2020.