# ECDSA White-Box Implementations: Attacks and Designs from WhibOx 2021 Contest

Guillaume Barbu[1], Ward Beullens[2], Emmanuelle Dottax[1], Christophe Giraud[1], Agathe Houzelot[1], Chaoyun Li[3], Mohammad Mahzoun[4], Adrián Ranea[3], and Jianrui Xie[3]

[1] IDEMIA, Cryptography & Security Labs, Pessac, France.
firstname.lastname@idemia.com
[2] IBM Research, Zurich, Switzerland
wbe@zurich.ibm.com
[3] imec-COSIC, KU Leuven, Belgium
firstname.lastname@esat.kuleuven.be
[4] Eindhoven University of Technology, Netherlands
m.mahzoun@tue.nl

**Abstract.** Despite the growing demand for software implementations of ECDSA secure against attackers with full control of the execution environment, the scientific literature on white-box ECDSA design is scarce. To assess the state-of-the-art and encourage practical research on this topic, the WhibOx 2021 contest invited developers to submit white-box ECDSA implementations and attackers to break the corresponding submissions.

In this work we describe several attack techniques and designs used during the WhibOx 2021 contest. We explain the attack methods used by the team TheRealIdefix, who broke the largest number of challenges, and we show the success of each method against all the implementations in the contest. Moreover, we describe the designs, submitted by the team zerokey, of the two winning challenges; these designs represent the ECDSA signature algorithm by a sequence of systems of low-degree equations, which are obfuscated with affine encodings and extra random variables and equations.

The WhibOx contest has shown that securing ECDSA in the white-box model is an open and challenging problem, as no implementation survived more than two days. To this end, our designs provide a starting methodology for further research, and our attacks highlight the weak points future work should address.

**Keywords:** ECDSA · White-box Cryptography · WhibOx

## 1 Introduction

Cryptographic techniques are primarily designed to be secure in a context where the confidentiality of secret keys is ensured with *black-box* access to the algorithm – only inputs and outputs are available to the attacker. Confidence in security is built from detailed studies, carefully defined security notions, and security proofs. Such a strong level of confidence is now a standard expectation. However, real-life scenarios for implementations might jeopardize initial assumptions, where attackers have access to additional information via side channels (e.g., timing or power consumption) or can modify the algorithm's execution and exploit faulty results. This is called the *grey-box* model. Developers have to put countermeasures in place to reach the originally expected security level.

In the context of mobile applications – contactless payments, cryptocurrency wallets, streaming services – or connected objects, devices often lack secure storage to protect secret keys, and their generally open execution environment exposes a large attack surface. This hostile environment is captured by the white-box model, which assumes an attacker having control of every aspect of the implementation: execution flow, memory content and addresses. The first white-box implementations were proposed in the early 2000s by Chow et al. [14,15], and the field has continuously developed since then, with design proposals [7,12,20,32,39,41,45], attacks [2,8,17,18,26,27,34,36,44] and efforts to define security notions [1,19,40].

The industry is increasingly interested in white-box cryptography owing to the widespread usage of security-related applications on connected devices. The WhibOx contest, attached to the CHES conference, has been held biennially since 2017 to encourage practical experiments both from the designer and attacker perspectives. It lasts for months, inviting coders to post white-box implementations and attackers to break them. Participants can remain anonymous and silent about any detail on their work. The first two editions in 2017 and 2019 focused on white-box implementations of AES and exhibited the community's strong interest in this subject. Some candidates survived all attacks in the second edition in 2019, showing a certain maturity for this algorithm. In 2021, organisers changed the target for the third edition and decided to consider the ECDSA signature algorithm, whose white-box implementation is of substantial interest for industry but virtually lacks scientific literature.

From May $17^{\text{th}}$ to August $22^{\text{nd}}$ 2021, 97 candidate implementations were submitted for scrutiny by 37 (teams of) attackers. All challenges were broken within 35 hours, suggesting the difficulty of achieving a secure white-box implementation of ECDSA. Thus, studying the attacks helps to discern weak points inside the implementations and deduce where to commit with robust countermeasures. Besides, the core of these challenges successfully defeating most attackers would also shed light on subsequent designs. The above sums up the purpose of revealing in what follows how the team TheRealIdefix who broke the most challenges (92) and the team zerokey who proposed the winning challenge and the most resistant one proceeded during the contest.

The paper is organized as follows. Section 2 outlines the rules of the WhibOx 2021 contest. Section 3 recalls the ECDSA algorithm and the state-of-the-art regarding white-box implementations. Section 4 presents the different methods that have been used by the team TheRealIdefix to break the implementations and some statistics regarding their success. Section 5 discloses the designs of Challenges 226 and 227 by the team zerokey, and Section 6 concludes this paper.

## 2 Rules of the WhibOx 2021 Contest

Designers were required to post challenges computing ECDSA signatures on the NIST P256 curve under a hard-coded, freely chosen key and given as input any message digest. At the same time, attackers were encouraged to extract the signing keys. In addition, acceptance of submitted implementations was conditional on compliance with some requirements:

- the public key corresponding to the embedded private key, as well as a proof of knowledge of the private key, had to be provided,
- submissions had to be source code in portable C,
- linking to external libraries was forbidden, except for the GNU Multi Precision library [28],
- the signature algorithm had to be deterministic,
- the execution time was limited to 3 seconds, the program size to 20 MB, and the RAM usage to 20 MB as well.

There was an elaborate system with scoreboards to reward designers and attackers. A challenge gains strawberries as time goes by till broken. Challenges with a higher performance score (measured in terms of execution time, code size, and RAM usage) gain strawberries faster. Accordingly, when submitting a matching signing key to the system, attackers receive bananas, the number of which is influenced by that of the strawberries of the challenge. More detailed information can be found on the contest website [13].

## 3 ECDSA and White-box Implementations

### 3.1 ECDSA

In 1992, Vanstone introduced a variant of DSA based on elliptic curve cryptography. The resulting public key signature algorithm is called the Elliptic Curve Digital Signature Algorithm (ECDSA) [43]. Its parameters are an elliptic curve $E$ over a field $\mathbb{F}_q$, a point $G$ of prime order $n$ and a cryptographic hash function $H$. The secret key $d$ is randomly drawn from $[\![1, n-1]\!]$ and the public key consists of the point $Q = [d]G$ where $[d]G$ corresponds to the scalar multiplication of the point $G$ by the scalar $d$. The ECDSA signature is described in Algorithm 1.

Note that the key $d$ is not the only sensitive value in that scheme. Indeed, the recovery of the nonce $k$ allows the computation of $d$ from the signature $(r, s)$ and the hash $e$:

$$d = (ks - e)r^{-1} \bmod n \ . \tag{1}$$

**Algorithm 1:** ECDSA signature

**Input :** the message $m$
**Output:** the signature $(r, s)$

$e \leftarrow H(m)$
$k \xleftarrow{\$} [\![1, n-1]\!]$
$R = (R_x, R_y) \leftarrow [k]G$
$r \leftarrow R_x \bmod n$
$s \leftarrow k^{-1}(e + rd) \bmod n$
**if** $r = 0$ *or* $s = 0$ **then**
  Go to step 2
**end**
Return $(r, s)$

The nonce must not only remain secret but also differ for each execution of the algorithm. Indeed, an efficient way to recover its value is to find another signature of a different message $e' \neq e$ using the same nonce, that is with $k' = k$. In that case, we also have $r' = r$, so the adversary may compute

$$k = (e' - e)(s' - s)^{-1} \bmod n \ . \tag{2}$$

In the black-box model, the security of ECDSA is widely believed to reduce to the difficulty of the Elliptic Curve Discrete Logarithm Problem (ECDLP), that is, on the difficulty of computing the scalar $k$ (resp. $d$) from the points $G$ and $R = [k]G$ (resp. $Q = [d]G$). To ensure that this problem is difficult to solve, there are several standards to define elliptic curves, e.g. [24, 31, 35, 42]. However, there is a gap between the security of ECDSA in theory and that of practical implementations. Many grey-box attacks have been described in the literature (see for example [22]). Some of them directly target the key $d$ while others aim at recovering some information on the nonce $k$. As explained previously, the knowledge of the nonce used in a signature allows an adversary to compute the secret key. Even the recovery of a few bits of several nonces for several executions may be enough for an attacker. Indeed, this allows the generation of a system of equations that can be solved using lattice-based algorithms [11, 30] or Bleichenbacher's Fourier analysis-based attacks [4]. These bits could, for example, be recovered via side-channel analysis if the implementation is not protected or simply guessed if the nonce is not drawn uniformly at random. These attacks show that it is already not straightforward to make a secure ECDSA implementation in the grey-box model, and of course, things get worse in the white-box context.

### 3.2 White-box Implementation of ECDSA

The white-box model assumes that the attacker has total access to the executable: he can read and modify it at will. He also has access to all the memory used during execution, so a white-box designer does not only have to protect his

implementation against grey-box attacks but also against an adversary who can dump the memory and search for sensitive values as $k$ or $d$. Techniques to prevent secret data from appearing in plain were introduced by Chow *et al.* in [15]. The key is embedded into the algorithm, and each operation is performed with the help of look-up tables protected by carefully crafted *encodings*. Informally, the algorithm is broken into low-level operations, and each operation *op* is replaced by $f^{-1} \circ op \circ f'$, where $f$ and $f'$ are bijections called respectively *input* and *output encodings*. The drawback of this technique is that the memory needed drastically increases with the algorithm's complexity, so applying it to operations as complex as scalar multiplications or inversions while staying efficient is a real challenge.

Another challenge in white-box cryptography is the impossibility of relying on any external source of randomness. Indeed, such a source could be simply disabled by an attacker fixing its output to a constant value. In the context of AES, for example, this reduces the efficiency of the countermeasures against side-channel or fault attacks based on randomization techniques. Nevertheless, this is not the only problem when one considers ECDSA signatures. Disabling the source of randomness yields multiple uses of the same nonce and, thus, easy recovery of the secret key. The seemingly only solution is to compute $k$ from the only source of randomness available, the hash $e$ of the message. The idea is that two different hash values imply two different nonces $k_1 = f(e_1)$ and $k_2 = f(e_2)$. To ensure the security of this deterministic scheme, the function $f$ must be unknown to the attacker and produce a random uniform distribution [5]. We will see in the next section that many challenges of the WhibOx competition did not fulfill this requirement.

## 4 Breaking the Challenges

The contest call for submissions encouraged the developers to rely on encoding and/or other theoretically sound approaches to secure their white-box designs. However, browsing the submitted source files reveals extensive use of obfuscation techniques. In these circumstances, independently reverse-engineering each challenge appears too time-consuming, and so we considered some attack methods that can be easily automated.

In this section we describe these automated methods, give some rationales to explain why we chose to discard some of them, and, finally, analyze the results of the different methods on the whole set of submissions.

### 4.1 Attack Methods

**Hooking shared libraries** In the context of the WhibOx contest, the definition of the rules was a clear incentive for developers to use the GMP library for big number arithmetic operations. A first attempt to break the submitted challenges was then to search if sensitive values were manipulated in clear by the GMP

library. In order to perform this automatically, our approach was to hook the calls to GMP functions thanks to the so-called *LD_PRELOAD trick*.

Pre-loading is a feature of the dynamic linker on UNIX systems that allows loading a specific shared library before all other libraries linked to a given executable binary[5]. In our specific case, we built a shared library defining the same function as the GMP library (e.g. `mpz_mul`, `mpz_mod` or `mpz_invert`). Each of these functions simply updates a log of the given parameters before calling the real GMP function, explicitly using the dynamic linker (thanks to the `<dlfcn.h>` module) to ensure the correct execution of the white-box implementation.

It is then only necessary to add our shared library to the *LD_PRELOAD* environment variable of the dynamic linker on our system before calling the ECDSA binary to have our custom functions called in place of the genuine GMP ones. The corresponding log is analysed in a second step to eventually reveal the secret key if $d$, $k$ or related values such as $r \cdot d$ or $e + r \cdot d$ are found in the log. Such an approach allowed us to break 32% of the challenges.

Besides, we stress the fact that the *LD_PRELOAD trick* also jeopardizes implementations relying on system-dependent random generators such as `srand` or `mpz_XrandomX` functions, and even on random sources such as `time`.

**Biased nonces** As explained in Sect. 3.2, the nonce $k$ is generated from the hash $e$, i.e. $k = f(e)$. However, if the white-box designer is not very careful about the selection of the function $f$, it could happen that the $k_i$'s generated from different $e_i$ are not independent.

In the worst case, it exists two different hash values $e_0$ and $e_1$ producing two nonces $k_0$ and $k_1$ such that $k_0 = k_1$. If such a collision occurs, then one can recover the private key $d$ as explained in Sect. 3.1. One of the main challenges from the attacker's perspective is finding such hash values. Indeed, it is not possible to check each and every possible value for $e$ since there are $2^{256}$ possibilities. The strategy we used during this competition to find a collision in a reasonable amount of time is to use hashes $e_i$ such that the Hamming weight of $e_i$ is equal to 1 or 2. We restricted ourselves to 32 896 hash values and were able to break 60% of the challenges.

We did not find any collision in some cases, but we assumed that the nonces were biased. To exploit such a potential weakness, we used well-known lattice attacks derived from [37] and [23]. Such attacks can recover an ECDSA private key only with the knowledge of a few bits of the ephemeral keys of several signatures.

A concrete example showing why such techniques can be successful in our context consists in considering $f = Id$. Then $k_i = e_i$ and with providing $e_i$ ranging from 0 to 99 we obtained 100 signatures for which the 249 most-significant bits of the nonces are 0. This bias is more than enough for a lattice attack to recover the private key $d$.

[5] `https://man7.org/linux/man-pages/man8/ld.so.8.html`

Lattice-based attacks can also be applied when the ephemeral key is the product of a small random $\kappa$ by another (large) constant scalar $t$. Such a design allows to efficiently perform the scalar multiplication as $R = [\kappa]T = [k]G$, with $T = [t]G$ a precomputed value. The point is that the small size of $\kappa$ reduces the cost of the scalar multiplication.

To sum up, the relations we used for our lattice attacks are the following (with $e_i$ ranging from 0 to 999):

- Assuming $l$ known most- or least- significant bits of the ephemeral key :

$$k_{high}2^l + k_{low} = s^{-1}(e + rd) \bmod n \ , \tag{3}$$

  with $l = 6$ (for known MSB) or $l = 250$ (for known LSB),
- Assuming the ephemeral keys are $k_i = t\kappa_i$:

$$\begin{cases} t &= \kappa_0^{-1}(e_0 + r_0 d) \bmod n, \\ \kappa_i &= t^{-1}(s_i^{-1}e_i - s_i^{-1}r_i r_0^{-1}e_0) + \kappa_0(s_i^{-1}r_i r_0^{-1}s_0) \bmod n, \end{cases} \tag{4}$$

  with $\kappa_i < 2^{248}$ and $t$ an unknown constant scalar.

Such an approach allowed us to break 72% of the challenges.

**DCA** In 2016, Bos et al. showed that although firstly described for the grey-box context, the well-known side-channel attacks could be very well adapted to the white-box model. The resulting attack [10] is called DCA (Differential Computational Analysis). The principle is very similar to classical side-channel attacks: secret values are extracted from leakage traces obtained during several executions of a cryptographic algorithm with the help of statistical tools. The only difference relies upon the nature of the traces. While in the grey-box context, one records the device's power consumption in which the algorithm is implemented or its electromagnetic emanations. For example, a white-box attacker can simply use software execution traces. Indeed, by instrumenting the binary, he can record traces of all accessed addresses and data over time. This leads to much more efficient attacks since, contrary to the traces obtained in the grey-box model, these are completely noiseless.

In theory, this attack is particularly devastating since it can be fully automated and does not require an earlier reverse engineering step. In practice, it is quite difficult to apply because of the size of the traces, in particular for time-consuming cryptosystems such as ECDSA. Indeed if the whole white-box execution were to be recorded, each trace would easily reach several gigabytes. A first step of reverse engineering allowing to select a smaller window of the implementation for the attack may thus be needed, and this explains why we did not use this technique to break the challenges of the WhibOx contest.

**Fault Injections** Another attack method derived from the grey-box model consists of disturbing the algorithm's execution and exploiting the resulting faulty

cipher/signature. In the white-box context, faults can be induced easily since the attacker is allowed to modify the binary or use debugging tools to stop the execution and, for example, skip an instruction or modify the value of a particular register. Again, this attack can be automated and does not require an earlier reverse engineering step.

In the case of an ECDSA white-box implementation, the fault can be induced on different variables to give an exploitable result. All the fault attacks that can be performed in the grey-box context are also a potential threat here. The most obvious attack is to force the use of a weak elliptic curve during the scalar multiplication by disturbing the curve parameters [6] in order to solve the discrete logarithm problem easily. The attacker can also force the use of a biased nonce, for instance, by sticking a 32-bit word of $k$ at zero during several executions. The corresponding signatures can then be used to obtain information on the key using lattice-based algorithms. Finally, modifying one byte of $d$ during the computation of $rd$ may allow one to recover information on the key as shown in [25].

Furthermore, the white-box model comes with new possibilities of fault attacks [3, 21, 38]. They arise from the impossibility of using the usual sources of randomness in that context, forcing the designers to implement deterministic versions of the scheme. When the algorithm is used twice on the same message, the same nonce $k$ is derived. The attacker may thus obtain a correct signature $s = k^{-1}(e + rd) \bmod n$ for a given message $m$, and sign it again, but this time modifying an intermediate variable. To break the challenges of the WhibOx contest, we mainly disturbed the computation of the first part of the signature $r$, obtaining a faulty result $\tilde{s} = k^{-1}(e + \tilde{r}d) \bmod n$. Some secret information can be deduced from the correct/faulty signatures:

$$(r - \tilde{r})(s - \tilde{s})^{-1} \equiv (r - \tilde{r})(k^{-1}d(r - \tilde{r}))^{-1} \equiv kd^{-1} \bmod n \ . \tag{5}$$

Let $\alpha = kd^{-1} \bmod n$. The adversary can compute

$$d = e(\alpha^{-1}s - r)^{-1} \bmod n \ , \tag{6}$$

It is also possible to disturb other variables, but the faulty value must be known to exploit the result. Interestingly, when one modifies the first part of the signature, if no countermeasure is implemented, the faulty value is just given to the attacker as part of the output. Furthermore, the attack surface is huge: the fault may happen anywhere during the scalar multiplication. This is why we considered only this perturbation in the context of this competition. This approach is the most successful one as it allowed us to break 75% of the challenges.

### 4.2 Attacks Results

When applying the various attack methods described in Sect. 4.1, we obtain the results presented in Table 1. We observe that lattice and fault attacks are very efficient. Collision attacks also give good results.

**Table 1.** Success rate of each attack on the 97 challenges.

| Attack type | | Percentage of broken challenges |
|---|---|---|
| Hooking | | 32% |
| Bad nonce | Collision | 60% |
| | Lattice | 72% |
| Fault Injection | | 75% |

We give in Appendix A the specific vulnerabilities for each of the 97 submitted challenges as well as the corresponding private key.

However, one could notice that many challenges have a low level of security. Some of them are even plain implementations. We think these challenges are due to the fact that the designers test the submission process with a toy example before submitting their real challenge. Indeed, the WhibOx submission process is difficult, and no proper explanation is given when a challenge is rejected. Moreover, if the designer wants to resubmit its challenge again, he would have to change the key, which is a complex process on white-box implementations. In order to focus on the *strongest* white-box ECDSA implementations, we rejected the challenges where the nonce and/or the private key is manipulated in plain. By using such a criteria, we rejected 30 challenges[6] and we analysed in Table 2 the efficiency of the attacks presented in Sect. 4.1 on the 67 remaining challenges. We observe that hooking is useless, collision and lattice attacks are significantly less efficient, and fault injection seems the most powerful attack.

**Table 2.** Success rate of each attack on the 67 strongest challenges.

| Attack type | | Percentage of broken challenges |
|---|---|---|
| Hooking | | 1% |
| Bad nonce | Collision | 49% |
| | Lattice | 61% |
| Fault Injection | | 69% |

Among the 67 strongest challenges, the challenges 226 and 227 are the winning ones. In the next section, we present the design of these two white-box implementations.

[6] The challenges 3, 4, 8, 10, 11, 32, 45, 54, 55, 57, 85, 97, 114, 135, 136, 139, 153, 157, 174, 185, 187, 231, 235, 267, 274, 299, 307, 320, 321, and 323 are considered as weak.

## 5 Design of the Winning Challenges

In this section we describe the designs of the two winning challenges of the WhibOx contest: Challenge 226 `clever_kare` and Challenge 227 `keen_ptolemy`. The designs of both challenges were inspired from the implicit white-box implementation framework [29], which allows encoding the whole state with large affine permutations efficiently. We implemented both challenges with the same methodology; they only differ in some additional countermeasures used.

As we mention in Section 2, in the WhibOx contest, a challenge gains strawberries as time goes by, from the moment it is submitted until it is broken. Challenges with a higher performance score (measured in terms of the execution time, code size, and RAM usage) gain strawberries faster. As a result, we strategically posted two challenges with different trade-offs between security level and implementation cost. Challenge 227, our lightweight variant, was the winning implementation of the contest, obtaining the highest number of strawberries (20.39). On the other hand, Challenge 226, our hardened but heavier variant, achieved second place in the contest with the second-highest number of strawberries (11.19). However, it was the challenge standing the longest time unbroken (35 hours).

This section first introduces the (white-box) implicit framework, then describes the shared design approach of both challenges, and finally explains the additional countermeasures used in each challenge.

### 5.1 Implicit White-box Implementations

The implicit framework is a method to obtain an implicit white-box implementation of a block cipher. Its main idea is to represent the round functions of the cipher by implicit functions of low degree and protect these implicit functions with large affine encodings. Before introducing implicit white-box implementations, we need to introduce the notions of encoding, encoded implementation, and quasilinear implicit functions. While these notions are originally defined in [29] for vectorial functions over the binary field, we will extend these notions for an arbitrary finite field.

Let $\mathbb{F}_q$ be the finite field with $q$ elements. A vectorial function $F$ from the vector space $(\mathbb{F}_q)^l$ to $(\mathbb{F}_q)^{l'}$ will be called a $(l, l')$ function over $\mathbb{F}_q$, and its $l'$ component functions will be denoted by $(F_1, F_2, \ldots, F_{l'})$. The degree of an $(l, l')$ function $F$ denotes the maximum polynomial degree of the $l'$ multi-variate polynomials uniquely representing the component functions of $F$.

**Definition 1.** *Let $F$ be an $(l, l')$ function over $\mathbb{F}_q$, $A$ be an $(l, l)$ permutation over $\mathbb{F}_q$ and $B$ be an $(l', l')$ permutation over $\mathbb{F}_q$. The function $\overline{F} = B \circ F \circ A$ is called an* encoded function *of $F$, and $A$ and $B$ are called the input and output encodings respectively.*

**Definition 2.** *Let $F = F^{(t)} \circ F^{(t-1)} \circ \cdots \circ F^{(1)}$ be a vectorial function over $\mathbb{F}_q$. An encoded implementation of $F$, denoted by $\overline{F}$, is an encoded function of $F$*

*composed of encoded functions of* $F^{(i)}$*, that is,*

$$\overline{F} = \overline{F^{(t)}} \circ \cdots \circ \overline{F^{(1)}} = (B^{(t)} \circ E^{(t)} \circ A^{(t)}) \circ \cdots \circ (B^{(1)} \circ F^{(1)} \circ A^{(1)}),$$

*where the input and output encodings* $(A^{(i)}, B^{(i)})$ *are permutations over* $\mathbb{F}_q$ *such that* $A^{(r+1)} = \left(B^{(r)}\right)^{-1}$*. The first and last encodings* $(A^{(1)}, B^{(t)})$ *are called the* external encodings*.*

**Definition 3.** *Let* $F$ *be an* $(l, l')$ *function over* $\mathbb{F}_q$*. A* $(l + l', l'')$ *function* $T$ *is called an implicit function of* $F$ *if it satisfies*

$$T(u_1, u_2, \ldots, u_l, v_1, v_2, \ldots, v_{l'}) = 0 \iff F(u_1, u_2, \ldots, u_l) = v_1, v_2, \ldots, v_{l'}\,.$$

*In this case,* $T$ *is said to be quasilinear if for any* $(u_1, u_2, \ldots, u_l) \in (\mathbb{F}_q)^l$*, the function* $(v_1, v_2, \ldots, v_{l'}) \mapsto T(u_1, u_2, \ldots, u_l, v_1, v_2, \ldots, v_{l'})$ *is affine over* $\mathbb{F}_q$*.*

The following lemma from [29] describes how the composition of affine permutations translates to implicit functions.

**Lemma 1.** *Let* $F$ *be an* $(l, l')$ *function over* $\mathbb{F}_q$ *and* $T$ *be a quasilinear implicit* $(l + l', l'')$ *function of* $F$*. Let* $A$ *be an affine* $(l, l)$ *permutation over* $\mathbb{F}_q$*,* $B$ *be an affine* $(l', l')$ *permutation over* $\mathbb{F}_q$*, and* $M$ *be a linear* $(l'', l'')$ *permutation over* $\mathbb{F}_q$*. Then,* $T' = M \circ T \circ (A, B^{-1})$ *is a quasilinear implicit function of* $F' = B \circ F \circ A$*.*

The quasilinear property allows the *implicit evaluation* of $F$ in a point $(u_1, u_2, \ldots, u_l)$ by solving the affine system $T(u_1, u_2, \ldots, u_l, v_1, v_2, \ldots, v_{l'}) = 0$ for the variables $v_1, v_2, \ldots, v_{l'}$. We are ready to present the definition of an implicit implementation.

**Definition 4.** *Let* $F = F^{(t)} \circ F^{(t-1)} \circ \cdots \circ F^{(1)}$ *be a vectorial function over* $\mathbb{F}_q$*, and let* $\overline{F} = \overline{F^{(t)}} \circ \overline{F^{(t-1)}} \circ \cdots \circ \overline{F^{(1)}}$ *be an encoded implementation of* $F$*. An implicit (white-box) implementation of* $F$ *with underlying encoded implementation* $\overline{F}$ *is a set of quasilinear implicit functions* $\{\overline{T^{(1)}}, \overline{T^{(2)}}, \ldots, \overline{T^{(t)}}\}$ *where* $\overline{T^{(i)}}$ *is an implicit function of* $\overline{F^{(i)}}$*.*

### 5.2 White-boxing ECDSA Signature Algorithm Using the Implicit Framework

In the WhibOx contest, designers submitted white-box implementations of the ECDSA signature algorithm on the NIST P256 curve. As opposed to the standard ECDSA algorithm (Algorithm 1), the algorithm for the WhibOx contest (hereafter denoted by $E$) takes as input the 256-bit message digest. The private key is not an input of the algorithm; it is freely chosen by the designer, but it is fixed (hard-coded) in the implementation. Algorithm 2 depicts a high-level overview of this deterministic variant of ECDSA, where the deterministic nonce derivation mechanism is chosen freely by the designer.

**Algorithm 2:** Deterministic ECDSA signature algorithm for WhibOx contest

**Input :** 256-bit hashed message digest $e$
**Output:** the signature $(r, s)$

1 $state \leftarrow e$
2 $k, state \leftarrow$ `NonceDerivation`$(state)$
3 $R = (R_x, R_y) \leftarrow [k]G$
4 $r \leftarrow R_x \bmod n$
5 $s \leftarrow k^{-1}(e + rd) \bmod n$
6 **if** $r = 0$ *or* $s = 0$ **then**
7 | Go to step 2
8 **end**
9 Return $(r,\ s)$

The main steps of $E$ can be represented by the functions $E^{(1)}$ and $E^{(2)}$. The $\mathbb{F}_p$-function $E^{(1)}$ is given by

$$E^{(1)}(e) = (R_x, k, e) \ , \tag{7}$$

which takes as input $e \in \mathbb{F}_p$ and computes the scalar multiplication $R = [k]G$ over $\mathbb{F}_p$. On the other hand, the $\mathbb{F}_n$-function $E^{(2)}$ can be written as

$$E^{(2)}(R'_x, k', e') = (r, s) \ , \tag{8}$$

which takes as input $(R'_x, k', e') = (R_x \mod n, k \mod n, e \mod n)$ and computes $(r, s) = (R'_x, k^{-1}(e + rd))$ over $\mathbb{F}_n$.

Inspired from the implicit framework, we built the white-box implementations of the Challenges 226 and 227 by encoding $E^{(1)}$ and $E^{(2)}$ with affine permutations and obtaining low-degree implicit round functions of $\overline{E^{(1)}}$ and $\overline{E^{(2)}}$, the encoded functions of $E^{(1)}$ and $E^{(2)}$. We will first describe the implicit implementation of $E^{(1)}$ and then that of $E^{(2)}$.

### White-boxing the Scalar Multiplication

To build an implicit implementation of $E^{(1)}$, we need first to decompose $E^{(1)}$ as the composition of $\mathbb{F}_p$-functions, that we will call round functions. Then we will explain how to encode these round functions and how to obtain low-degree quasilinear implicit function of the encoded round functions.

*Decomposing $E^{(1)}$ into round functions.* The function $E^{(1)}(e) = (R_x, k, e)$, mainly consists of the scalar multiplication $r = [k]G$ of the nonce $k$ and the point $G$. For the scalar multiplication, we perform the following subroutine. First, we precompute and store a list of $t$ random point pairs on the curve, i.e., $(G_{i0} = [k_{i0}]G, G_{i1} = [k_{i1}]G)$ for $1 \leq i \leq t$. Then, for each pair we select one of the two points together with its logarithm, denoted as $(G_{ib_i},\ k_{ib_i})$,

where $b_i \in \{0,1\}$ and $1 \leq i \leq t$. We add the selected points and the selected logarithms, obtaining the scalar multiplication

$$G_{1b_1} + \cdots + G_{tb_t} = [k_{1b_1} + \cdots + k_{tb_t}]G = [k]G \ , \tag{9}$$

where $k = k_{1b_1} + \cdots + k_{tb_t}$. This selection is done in a deterministic way depending on the bits $(e_1, e_2, \ldots, e_{256})$ of the hash $e$, the only source of entropy in the algorithm. Moreover, the selection is done with $\mathbb{F}_p$-arithmetic operations rather than with conditional instructions, so that each iteration only performs $\mathbb{F}_p$ operations. The subroutine is given in Algorithm 3.

It is worth pointing out that the values $k_{ij}$ are chosen such that the sum of $\max(k_{i0}, k_{i1})$ for all $i$ is always smaller than $n$. That is, we have $k < n$. Hence, $r$ and $s$ are never 0. In this way, we avoid the trivial case, i.e., avoid going to Step 7 in Algorithm 2.

By considering the precomputed points and their logarithms as fixed function parameters, we can represent each iteration or round as a vectorial function $F^{(i)}$ over $\mathbb{F}_p$, i.e.,

$$E^{(1)} = F^{(t)} \circ \cdots \circ F^{(2)} \circ F^{(1)} \ , \tag{10}$$

where the input value is $(0,0,0,0,e_1)$. The components of the each round function $F^{(i)}(u_1,u_2,u_3,u_4,u_5)$ are given by

$$\begin{aligned}
F^{(i)}_{1,2}(u_1,u_2,u_3,u_4,u_5) &= (u_1,u_2) + [1-u_5]G_{i0} + [u_5]G_{i1} \\
F^{(i)}_{3}(u_1,u_2,u_3,u_4,u_5) &= u_3 + (1-u_5)k_{i0} + u_5 k_{i1} \, , \\
F^{(i)}_{4}(u_1,u_2,u_3,u_4,u_5) &= u_4 + u_5 \cdot 2^i \, , \\
F^{(i)}_{5}(u_1,u_2,u_3,u_4,u_5) &= e_{i+1} \, ,
\end{aligned} \tag{11}$$

where the component function $F^{(i)}_{1,2}$ denotes a point on the elliptic curve with $F^{(i)}_1$ and $F^{(i)}_2$ the $x$- and $y$- coordinates respectively. The last round function $F^{(t)}$ only outputs four components. i.e., $(F^{(t)}_1, F^{(t)}_2, F^{(t)}_3, F^{(t)}_4)$. It is worth pointing out that $k_{ij}$ and $G_{ij}$ are not inputs but fixed values, and that the elliptic curve additions can be represented by operations over $\mathbb{F}_p$.

The component function $F^{(i)}_4$ recovers one bit of the hash at a time and passes the currently computed message to the next round. This is done because $E^{(1)}$ needs to output the hash $e$ for $E^{(2)}$.

*Encoding the round functions.* To protect the round functions, we encode each round with random $\mathbb{F}_p$-affine permutations $A^{(i)}$, obtaining the encoded round functions

$$\overline{F^{(i)}} = A^{(i)} \circ F^{(i)} \circ (A^{(i-1)})^{-1}, \ 1 \leq i \leq t \ . \tag{12}$$

In other words, the input and output encodings of $F^{(i)}$ are $\left((A^{(i-1)})^{-1}, A^{(i)}\right)$, and the composition of the round functions cancels all intermediate encodings except $(A^{(0)})^{-1}$ and $A^{(t)}$, that is,

$$\overline{E^{(1)}} = \overline{F^{(t)}} \circ \cdots \circ \overline{F^{(2)}} \circ \overline{F^{(1)}} = A^{(t)} \circ F^{(t)} \circ \cdots \circ F^{(2)} \circ F^{(1)} \circ (A^{(0)})^{-1} \ , \tag{13}$$

**Algorithm 3:** Round-based scalar multiplication used in $E^{(1)}$

**Input :** the bits $(e_1, e_2, \ldots, e_{255})$ of the hash $e$ (little-endian order)
**Output:** $x$-coordinate of $[k]G$ and the message-dependent scalar $k$

`/* Round 1: input` $e_1$`,` $k_{10}, k_{11}, G_{10}, G_{11}$ `embedded values */`
$R \leftarrow [1 - e_1]G_{10} + [e_1]G_{11}$
$k \leftarrow (1 - e_1)k_{10} + e_1 k_{11}$

`/* Round` $i$`: input` $(R, k, e_i)$`,` $k_{i0}, k_{i1}, G_{i0}, G_{i1}$ `embedded values */`
**for** $2 \leq i \leq t$ **do**
  $R \leftarrow R + [1 - e_i]G_{i0} + [e_i]G_{i1}$
  $k \leftarrow k + (1 - e_i)k_{i0} + e_i k_{i1}$
**end**
**return** $R_x, k$ `//` $(R_x, R_y) = R = [k]G$

where $t$ is the number of rounds. The input encoding $(A^{(0)})^{-1}$ of $\overline{F^{(1)}}$ is set as the identity mapping to preserve the input-output behaviour of $E$.

*Obtaining the implicit round functions.* Now we proceed to obtain an implicit round function $\overline{T^{(i)}}$ of each encoded round function $\overline{F^{(i)}}$. To this end, we need to first deal with the implicit implementation of elliptic curve additions.

Let $\texttt{ADD}(P_x, P_y, Q_x, Q_y) = (R_x, R_y)$ be the vectorial $\mathbb{F}_p$-function denoting the elliptic curve addition $P + Q = R$, that is,

$$\begin{aligned} R_x &= (Q_y - P_y)^2((Q_x - P_x)^2)^{-1} - P_x - Q_x \\ R_y &= (Q_y - P_y)(P_x - R_x)(Q_x - P_x)^{-1} - P_y\,, \end{aligned} \tag{14}$$

more details can been found in [33]. While ADD has a high degree due to the inversion over $\mathbb{F}_p$, it is easy to show that the function $\texttt{IMP}(P_x, P_y, Q_x, Q_y, R_x, R_y) = (\texttt{IMP}_0, \texttt{IMP}_1)$ defined by

$$\begin{aligned} \texttt{IMP}_0 &= (Q_y - P_y)^2 - (P_x + Q_x + R_x)(Q_x - P_x)^2 \\ \texttt{IMP}_1 &= (Q_y - P_y)(P_x - R_x) - (R_y + P_y)(Q_x - P_x) \end{aligned} \tag{15}$$

is a quasilinear implicit round function of ADD with degree 3.

From the above implicit function of the elliptic curve addition, it is easy to derive a quasilinear implicit function $T^{(i)}$ of each round function $F^{(i)}$. Then, we sample a linear permutation $M^{(i)}$ for each round $i$, and by Lemma 1 the function

$$\overline{T^{(i)}} = M^{(i)} \circ T^{(i)} \circ \left((A^{(i-1)})^{-1}, (A^{(i)})^{-1}\right) \tag{16}$$

is a quasilinear implicit function of $\overline{F^{(i)}}$ for $1 \leq i \leq t$.

The white-box implementations of the Challenges 226 and 227 contain this implicit implementation of $E^{(1)}$, with underlying encoded implementation $\overline{E^{(1)}}$, given by the $t$ implicit round functions $\{\overline{T^{(1)}}, \ldots, \overline{T^{(t)}}\}$ in Eq. (16). Moreover, $\overline{E^{(1)}}$ is evaluated in our white-box implementations by implicitly evaluating the

encoded round functions $\overline{F^{(i)}}$. In other words, given the output $\mathbf{u}$ of the round $i-1$, the output $\mathbf{v}$ of the $i$th round is computed by finding the solution of the affine system $\overline{T^{(i)}}(\mathbf{u};\mathbf{v}) = 0$ for $\mathbf{v}$.

### White-boxing the Computation of $s$

Now we turn our attention to $E^{(2)}$, the second step of the signing algorithm, where we compute $r = R_x \mod n$ and $s = k^{-1}(e + dR_x) \mod n$, and output the signature $(r, s)$. As opposed to $E^{(1)}$, we do not decomposed $E^{(2)}$, and we build a single (vectorial) quasilinear implicit function of $\overline{E^{(2)}} = E^{(2)} \circ (A^{(t)})^{-1}$, the encoded version of $E^{(2)}$.

The vectorial $\mathbb{F}_n$-function $T^{(t+1)}$ defined as

$$\begin{cases} T_1^{(t+1)}(R_x, R_y, k, e; s, r) = ks - e - dR_x \\ T_2^{(t+1)}(R_x, R_y, k, e; s, r) = r - R_x \end{cases} . \tag{17}$$

is a quasilinear implicit function of $E^{(2)}$. In other words, the polynomial system $T^{(t+1)} = \{T_1^{(t+1)}, T_2^{(t+1)}\}$ implicitly defines $E^{(2)}$ because $(s, r) = E^{(2)}(R_x, R_y, k, e)$ if and only if $T^{(t+1)}(R_x, R_y, k, e; s, r) = 0$. Moreover, the system is affine in $r$ and $s$, so after plugging in values for $R_x, R_y, k$ and $e$ the system can be solved for $r, s$ efficiently.

The encoded version $\overline{E^{(2)}}$ gets as input $A^{(t)}(R_x, R_y, k, e)$, where $A^{(t)}$ is the affine function that protects the last round of $E^{(1)}$. By Lemma 1, we build the implicit round function of $\overline{E^{(2)}}$ as

$$\overline{T^{(t+1)}}(\mathbf{u}; r, s) = M \cdot T^{(t+1)}((A^{(t)})^{-1}(\mathbf{u}); r, s) , \tag{18}$$

where $(A^{(t)})^{-1}$ is the inverse of $A^{(t)} \mod n$, and where $M$ is a random invertible 2-by-2 matrix mod $n$. The function $\overline{T^{(t+1)}}$ is quasilinear, and we can implicitly evaluate $\overline{E^{(2)}}$ on input $\mathbf{u} = A^{(t)}(R_x, R_y, k, e)$ by plugging $\mathbf{u}$ in the first slot of $\overline{T^{(t+1)}}$ and solving the remaining system (which is affine) for $r$ and $s$ over $\mathbb{F}_n$.

However, the fact that $E^{(1)}$ works in $\mathbb{F}_p$, and $E^{(2)}$ works in $\mathbb{F}_n$ causes some problems. The problem is that the input to $\overline{E^{(2)}}$ is $\mathbf{u} = A^{(t)}(R_x, R_y, k, e)$ *reduced mod* $p$, so $(A^{(t)})^{-1}(\mathbf{u})$ is in general not equal to $(R_x, R_y, k, e) \mod n$ if there are overflows in the computation of $\mathbf{u}$. Let $\mathbf{o}$ be the vector of overflows mod $p$, such that

$$\mathbf{u} = A^{(t)}(R_x, R_y, k, e) - p\mathbf{o} , \tag{19}$$

then $(A^{(t)})^{-1}(\mathbf{u}) = (R_x, R_y, k, e) - pL_t^{-1}(\mathbf{o}) \mod n$, where $L_t$ is the linear part of the affine map $A^{(t)}$ (i.e., $A^{(t)}(x) = L_t(x) + c$ for some constant term $c$).

To deal with this problem, we will correct for the overflow mod $p$, by guessing the overflow vector $\mathbf{o}$, and setting $\mathbf{u}' = \mathbf{u} + p\mathbf{o}$ before plugging $\mathbf{u}'$ into $\overline{T^{(t+1)}}(\mathbf{u}; r, s)$ and solving for $(r, s)$. If the guess is correct, then $\mathbf{u}'$ is equal to $A^{(t)}(R_x, R_y, k, e)$ over the integers, so the correct $r, s$ will be recovered. Therefore, we repeatedly run the last step with random guesses of $\mathbf{o}$, to get a candidate

signature $(r, s)$. Then we run the verification algorithm on $(r, s)$ and we output the first $(r, s)$ for which the verification algorithm succeeds. Note that we do not need to protect the verification algorithm because it does not use secret information.

If $A^{(t)}$ was a random affine map with entries of size up to $p$, then guessing $\mathbf{o}$ correctly would be very unlikely. Therefore, we choose the affine map $A^{(t)}$ with small entries. For example, we could use

$$A^{(t)}(R_x, R_y, k, e) = \begin{pmatrix} 1\,0\,1\,2 \\ 1\,1\,2\,0 \\ 0\,1\,2\,1 \\ 1\,2\,0\,1 \end{pmatrix} \begin{pmatrix} R_x \\ R_y \\ k \\ e \end{pmatrix} + \mathbf{c} \ . \tag{20}$$

With this choice, the weight of each row is four, so there are at most four overflows mod p in each entry of $\mathbf{u}$, which means $\mathbf{o}$ can be guessed more easily. Not all guesses are equally likely, (e.g., $\mathbf{o} = [4, 4, 4, 4]$ only occurs if $R_x, R_y, k, e$ are all quite big, which is unlikely.) Rather than guessing $\mathbf{o} \in [0, 4]^4$ at random, which is still quite inefficient, we precompute a list of guesses $\mathcal{L}$ ordered from more likely to be correct to less likely, and we iterate through the list of guesses in that order.

The white-box implementations of the Challenges 226 and 227 contain the implicit function $T^{(t+1)}$, which allows the implicit evaluation of $\overline{E^{(2)}}$, together with the correction for the overflow mod $p$ described above and summarized in Algorithm 4.

Note that the severe restriction on the size of the entries of $A^{(t)}$ makes the conversion from $\mathbb{F}_p$ to $\mathbb{F}_n$ one of the most vulnerable points in the white-box implementation. In particular, an attacker knowing the specifications of the design can easily recover $A^{(t)}$ by exhaustive search if no additional countermeasures are used.

**Algorithm 4:** White-box implementation of ECDSA signature algorithm for winning challenges

**Input :** 256-bit hashed message digest $e$
**Output:** the signature $(r, s)$

$e \leftarrow e \mod p$
$(v_1, v_2, v_3) \leftarrow \overline{E^{(1)}}(e)$ `// implicit evaluation`
**for** $\mathbf{o}$ *in* $\mathcal{L}$ **do**
  $(u_1, u_2, u_3) \leftarrow (v_1, v_2, v_3) + p \cdot \mathbf{o}$
  $(r, s) \leftarrow \overline{E^{(2)}}(u_1, u_2, u_3)$ `// implicit evaluation`
  **if** *VerifySignature*$(r, s, e) =$ valid **then**
    **return** $(r, s)$
  **end**
**end**

### 5.3 Additional Countermeasures

The representation of the implicit round functions as systems of multivariate polynomials allows applying countermeasures from multivariate public-key cryptosystems. In fact, the challenges 227 and 226 only differ in the additional countermeasures used.

In particular, we considered two techniques. First, we obfuscated the components (seen as polynomials) of the implicit round functions $\overline{T^{(i)}}$ by multiplying them with random polynomials in the input variables. Note that the multiplication of inputs variables preserves the quasilinear property. Moreover, the image of a random polynomial is non-zero with high probability, and multiplying an equation with a non-zero value does not change its solution set. In the unlikely case that one of the added polynomials vanishes, the output of the corresponding implicit function will be invalid, and no valid signature will be obtained. To prevent this extreme case, we made the first implicit round function dependent on an initial value; if no valid signature is found, we simply repeated the whole process with a different initial value.

This first technique increases the degree of the implicit round functions, significantly increasing the implementation size. Thus, for the lightweight Challenge 227 we only applied this technique to raise the degree of the components to the total degree of the functions, but for Challenge 226 we multiplied with polynomials of higher degree to increase the total degrees of the implicit round functions. The final degrees are listed in Table 3 and Table 4.

The second technique we used was adding additional variables and components to the implicit round functions but preserving the input-output behaviour of the underlying encoded round functions. Since this technique also introduces significant overhead in the implementation size, we only applied it to Challenge 226. In particular, we added two variables and two equations in the implicit round functions of $\overline{E^{(1)}}$, and two variables and one equation in those of $\overline{E^{(2)}}$.

We also used Tigress [16] for both challenges to obfuscate the C source code. Tigress is an obfuscator for C language that protects programs against dynamic and static reverse engineering attacks. We used the transformations[7] Flatten (flattens the code to remove structured flow), AntiTaintAnalysis (disrupts tools that make use of dynamic taint analysis), AddOpaque (adds opaque predicates), EncodeLiterals (replaces integers and strings with run-time expressions) and CleanUp (renames variables and functions).

### 5.4 Challenge 227: The Winner

**Description** Following the method described above, we built Challenge 227 (`keen_ptolemy`) as a lightweight white-box implementation without additional countermeasures increasing the number of equations, the degree, or the number of variables of the implicit round functions. Challenge 227 was the winning implementation of the WhibOx contest; it achieved the highest number of strawberries (20.39), and it stood for 33 hours as the second longest.

[7] `https://tigress.wtf/transformations.html`

Table 3 describes the memory complexity of $\{\overline{T^{(1)}}, \ldots, \overline{T^{(t)}}\}$ and $\overline{T^{(t+1)}}$, the implicit round functions of $\overline{E^{(1)}}$ and $\overline{E^{(2)}}$ respectively, of Challenge 227 (after applying the additional countermeasures). The number of coefficients in Table 3 denotes the maximum number of non-zero coefficients of a quasilinear vectorial function with a given number of input variables, components, and degrees. If each coefficient is represented with 256 bits, $\{\overline{T^{(1)}}, \ldots, \overline{T^{(t)}}\}$ and $\overline{T^{(t+1)}}$ require in total roughly 4 MB.

**Table 3.** Information of the implicit round functions $\overline{T^{(i)}}$ of Challenge 227.

| | $\overline{T^{(1)}}$ | $\{\overline{T^{(2)}}, \ldots, \overline{T^{(t-1)}}\}$ | $\overline{T^{(t)}}$ | $\overline{T^{(t+1)}}$ |
|---|---|---|---|---|
| input variables | 2+4 | 5+4 | 5+3 | 3+2 |
| number of components | 4 | 4 | 3 | 2 |
| degree | 3 | 3 | 4 | 2 |
| number of coefficients | $27 \times 4$ | $130 \times 4$ | $255 \times 3$ | $18 \times 2$ |

After obfuscating the code with Tigress, the size of the final C source code of Challenge 227 is 4.4 MB. In a modern personal laptop with the environment[8] provided by the competition, the size of the compiled binary is 4.42 MB, and the average running time and RAM consumed is 0.04 s and 6.14 MB respectively. The code obfuscation did not impact the running time but increased the binary size by 8% and the average RAM by 3%.

**Security Analysis** Challenge 227 can be broken in different ways. Here, we describe two methods that can be more or less automated.

*Finding the inversion of the nonce.* During the computation of $s$, the nonce $k$ must be inverted modulo $n$. This operation is very sensitive and can be quite difficult to protect. In Challenge 227, it is not performed in the clear: the value of $k$ is protected by the encoding function $A^{(t)}$ and by the matrix $M$. If we note $M = \begin{pmatrix} m_0 & m_1 \\ m_2 & m_3 \end{pmatrix}$, the system that is solved for the computation of $r$ and $s$ is

$$\begin{cases} m_0(ks - e - dR_x) + m_1(r - R_x) = 0 \\ m_2(ks - e - dR_x) + m_3(r - R_x) = 0 \end{cases}. \tag{21}$$

We stress that $k, e$ and $R_x$ do not appear in the clear. They are expressed as linear combinations of the input $\mathbf{u} = A^{(t)}(k, e, R_x, R_y)$ of $\overline{E^{(2)}}$. Nevertheless, the factor of $s$ has to be inverted at some point of the computation, so $(m_0 k)^{-1} \bmod n$ appears in the clear, no matter how $m_0 k$ is computed. If the attacker finds this value during the computation of two different signatures, let's say of $(r_1, s_1)$ and $(r_2, s_2)$, he may compute $\alpha_1 = m_0 k_1 \bmod n$ and $\alpha_2 = m_0 k_2 \bmod n$ in order

[8] `https://github.com/CryptoExperts/whibox_contest_submission_server`

to solve the following system of two equations with two unknowns ($m_0$ and $d$) in $\mathbb{F}_n$:

$$\begin{cases} m_0(e_1 + r_1 d) = \alpha_1 s_1 \\ m_0(e_2 + r_2 d) = \alpha_2 s_2 \end{cases} . \tag{22}$$

Therefore, recovering the value $(m_0 k_t)^{-1} \bmod n$ for two different signatures allows an attacker to compute the private key.

This attack may seem very specific but protecting the inversion by multiplying the nonce with a constant may look like an easy countermeasure for designers, so it can possibly break several implementations. Also, finding the interesting values inside the white-box may seem difficult without a reverse engineering step, but the attack turns out to be easily automated on the challenges which use the GMP library, as Challenge 227. Indeed, finding the inversion is easy when one can simply trace the calls to the function *mpz_invert*(). This attack can thus be efficiently applied on Challenge 227 without requiring a reverse engineering step.

*With lattice reduction.* There exists a more generic way of breaking challenge 227. Indeed, the way the ephemeral key is constructed (see Sect.9) opens the way for an attack using lattice reduction techniques.

Given that the ephemeral key $k$ is obtained by summing 256 scalars $k_{i,j}$ according to each bit of the input, one can obtain the following signatures by selecting couple of hashes $(e_0, e_i)$, with $e_0 = 0$ and $e_i = 2^i$ :

$$\begin{cases} s_0 = (\sum_{j=0}^{255} k_{j,0})^{-1}(e_0 + r_0 d) \bmod n \\ s_i = (k_{i,1} + \sum_{j=0, j\neq i}^{255} k_{j,0})^{-1}(e_0 + r_0 d) \bmod n \end{cases} , \tag{23}$$

which allow us to construct 256 equations involving only one of the $k_{i,j}$:

$$\begin{aligned} k_{i,1} + \sum_{j=0, j\neq i}^{255} k_{j,0} - \sum_{j=0}^{255} k_{j,0} &= s_i^{-1}(e_i + r_i d) - s_0^{-1}(e_0 + r_0 d) \bmod n \\ k_{i,1} - k_{i,0} &= s_i^{-1} e_i - s_0^{-1} e_0 + (r_i - r_0) d \bmod n \end{aligned} . \tag{24}$$

Now, the additional constraint $k < n$ lets us estimate that each $k_{i,j}$ is sampled from $[0, \lfloor n/256 \rfloor]$. Consequently,

$$|k_{i,1} - k_{i,0}|_n = |s_i^{-1} e_i - s_0^{-1} e_0 + (r_i - r_0) d|_n < \lfloor \frac{n}{256} \rfloor , \tag{25}$$

with $|y|_n := \min_{a \in \mathbb{Z}} |y - an|$ to denote the distance of $y \in \mathbb{R}$ to the closest integer multiple of $n$.

We recognize in Eq. 25 an instance of the Hidden Number Problem (HNP) [9]. Indeed, we are given many HNP inequalities of the form:

$$|\alpha t_i - u_i|_n < \lfloor \frac{n}{256} \rfloor , \tag{26}$$

with $t_i = r_i - r_0$, $u_i = s_0^{-1}e_0 - s_i^{-1}e_i$ and the hidden number $\alpha$ is the private key $d$.

Solving HNP instances in the context of ECDSA given inequalities such as Eq. 26, has been described numerous times in the literature. We refer the reader to [30] for a more detailed description[9]. In particular, the authors detail the reduction of the HNP instance to a Closest Vector Problem instance in a specific lattice as well as the construction of this lattice.

Finally, we use 75 relations such as Eq. 25 (out of the 255 we can establish) to build a lattice whose reduction allows us to recover the private key $d$.

### 5.5 Challenge 226: The Most Resistant

**Description** Challenge 226 (`clever_kare`) was the second white-box implementation that we built following the method described above and including all the additional countermeasures. While this challenge stood for the longest (35 hours), Challenge 226 achieved the second-highest number of strawberries due to its higher time and memory complexity than Challenge 227.

Table 4 describes the memory complexity of $\{\overline{T^{(1)}}, \ldots, \overline{T^{(t)}}\}$ and $\overline{T^{(t+1)}}$ of Challenge 226 after applying the additional countermeasures. Given each coefficient as a 256-bit value, $\{\overline{T^{(1)}}, \ldots, \overline{T^{(t)}}\}$ and $\overline{T^{(t+1)}}$ require in total roughly 15 MB.

**Table 4.** Information of the implicit round functions $\overline{T^{(i)}}$ of Challenge 226.

| | $\overline{T^{(1)}}$ | $\{\overline{T^{(2)}}, \ldots, \overline{T^{(t-1)}}\}$ | $\overline{T^{(t)}}$ | $\overline{T^{(t+1)}}$ |
|---|---|---|---|---|
| input variables | 2+6 | 7+6 | 7+5 | 5+2 |
| number of components | 6 | 6 | 5 | 2 |
| degree | 3 | 3 | 4 | 5 |
| number of coefficients | $37 \times 6$ | $322 \times 6$ | $854 \times 5$ | $504 \times 2$ |

The size of the final C source code of Challenge 266 is 17.54 MB, the size of the compiled binary is 15.44 MB, and the average running time and RAM consumed are 0.15 s and 17.27 MB respectively. The code obfuscation did not significantly impact the performance of Challenge 266; the running time, the binary size, and the average RAM increased by less than 1%.

## 6 Conclusion

This work describes several attack techniques and designs used in the WhibOx 2021 contest. We explained the attack methods used by the team TheRealIdefix,

[9] We also highlight that the authors of [30] made their code available at `https://github.com/crocs-muni/minerva`.

who broke the largest number of challenges, and we showed the success of each method against all the implementations in the contest. Fault attacks were the most efficient and effective ones; collision and lattice attacks were slightly less efficient, and hooking succeeded against weak implementations only.

Among the three white-box implementations that resisted these attacks, the one with the highest score was Challenge 226 (`clever_kare`). This challenge, together with Challenge 227 (`keen_ptolemy`), was submitted by the team zerokey, and they obtained the second-highest and the highest score in the contest respectively. In this work, we described the design methodology of these two challenges, which was inspired by the implicit white-box framework. We also described the additional countermeasures used in each challenge.

The large number of implementations broken by our automated attacks and the fact that no challenge survived more than two days show that securing ECDSA in the white-box model is a challenging problem. White-box attacks benefit from the huge progress in side-channel and fault attacks against ECDSA implementations, but not much research has been done on the design part. To this end, our designs provide insightful examples for future works, and our attacks highlight the weak points future research should address.

One of the main challenges specific to white-boxing ECDSA is the conversion from $\mathbb{F}_p$ to $\mathbb{F}_n$. While grey-box countermeasures can protect this step (e.g. Arithmetic to Boolean and Boolean to Arithmetic mask conversions), these techniques rely on randomness, which is ineffective in white-box implementations. In particular, the conversion from $\mathbb{F}_p$ to $\mathbb{F}_n$ is one of the weakest points in our designs, and further research in white-boxing the field conversion is needed.

## Acknowledgment

The authors would like to thank the other members of TheRealIdefix team: Yannick Bequer, Luk Bettale, Laurent Castelnovi, Thomas Chabrier, Nicolas Debande, Roch Lescuyer, Sarah Lopez and Nathan Reboud. Adrián Ranea is supported by a PhD Fellowship from the Research Foundation – Flanders (FWO). Chaoyun Li is an FWO post-doctoral fellow under grant No. 1283121N. Ward Beullens is an FWO post-doctoral fellow under grant No. 1S95620N.

## References


1. E. Alpirez Bock, A. Amadori, C. Brzuska, and W. Michiels. On the Security Goals of White-Box Cryptography. Cryptology ePrint Archive, Report 2020/104, 2020. https://eprint.iacr.org/2020/104.
2. A. Amadori, W. Michiels, and P. Roelse. A DFA Attack on White-Box Implementations of AES with External Encodings. In K. G. Paterson and D. Stebila, editors, *SAC 2019*, volume 11959 of *LNCS*, pages 591–617. Springer, Heidelberg, Aug. 2019.
3. C. Ambrose, J. W. Bos, B. Fay, M. Joye, M. Lochter, and B. Murray. Differential attacks on deterministic signatures. In *Cryptographers' Track at the RSA Conference*, pages 339–353. Springer, 2018.
4. D. F. Aranha, F. R. Novaes, A. Takahashi, M. Tibouchi, and Y. Yarom. LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage. In J. Ligatti, X. Ou, J. Katz, and G. Vigna, editors, *ACM CCS 20*, pages 225–242. ACM Press, Nov. 2020.
5. E. Barker and J. Kelsey. Recommendation for Random Number Generation Using Deterministic Random Bit Generators. Technical report, NIST, 2015. https://doi.org/10.6028/NIST.SP.800-90Ar1.
6. I. Biehl, B. Meyer, and V. Müller. Differential Fault Attacks on Elliptic Curve Cryptosystems. In M. Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 131–146. Springer, Heidelberg, Aug. 2000.
7. O. Billet and H. Gilbert. A Traceable Block Cipher. In C.-S. Laih, editor, *ASIACRYPT 2003*, volume 2894 of *LNCS*, pages 331–346. Springer, Heidelberg, Nov. / Dec. 2003.
8. O. Billet, H. Gilbert, and C. Ech-Chatbi. Cryptanalysis of a White Box AES Implementation. In H. Handschuh and A. Hasan, editors, *SAC 2004*, volume 3357 of *LNCS*, pages 227–240. Springer, Heidelberg, Aug. 2004.
9. D. Boneh and R. Venkatesan. Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes. In N. Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 129–142. Springer, Heidelberg, Aug. 1996.
10. J. W. Bos, C. Hubain, W. Michiels, and P. Teuwen. Differential Computation Analysis: Hiding Your White-Box Designs is Not Enough. In B. Gierlichs and A. Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 215–236. Springer, Heidelberg, Aug. 2016.
11. J. Breitner and N. Heninger. Biased Nonce Sense: Lattice Attacks Against Weak ECDSA Signatures in Cryptocurrencies. In I. Goldberg and T. Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 3–20. Springer, Heidelberg, Feb. 2019.
12. J. Bringer, H. Chabanne, and E. Dottax. White Box Cryptography: Another Attempt. Cryptology ePrint Archive, Report 2006/468, 2006. https://eprint.iacr.org/2006/468.
13. CHES 2021 Challenge - WhibOx Contest. https://whibox.io/contests/2021/.
14. S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot. A White-Box DES Implementation for DRM Applications. In J. Feigenbaum, editor, *Security and Privacy in Digital Rights Management, ACM CCS-9 Workshop, DRM 2002*, volume 2696 of *LNCS*, pages 1–15. Springer, Heidelberg, Nov. 2002.
15. S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot. White-Box Cryptography and an AES Implementation. In K. Nyberg and H. M. Heys, editors, *SAC 2002*, volume 2595 of *LNCS*, pages 250–270. Springer, Heidelberg, Aug. 2003.

16. C. Collberg. The Tigress C Diversifier/Obfuscator. `https://tigress.wtf`.
17. Y. De Mulder, P. Roelse, and B. Preneel. Cryptanalysis of the Xiao-Lai White-Box AES Implementation. In L. R. Knudsen and H. Wu, editors, *SAC 2012*, volume 7707 of *LNCS*, pages 34–49. Springer, Heidelberg, Aug. 2013.
18. Y. De Mulder, B. Wyseur, and B. Preneel. Cryptanalysis of a Perturbated White-Box AES Implementation. In G. Gong and K. C. Gupta, editors, *INDOCRYPT 2010*, volume 6498 of *LNCS*, pages 292–310. Springer, Heidelberg, Dec. 2010.
19. C. Delerablée, T. Lepoint, P. Paillier, and M. Rivain. White-Box Security Notions for Symmetric Encryption Schemes. In T. Lange, K. Lauter, and P. Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 247–264. Springer, Heidelberg, Aug. 2014.
20. P. Derbez, P.-A. Fouque, B. Lambin, and B. Minaud. On Recovering Affine Encodings in White-Box Implementations. Cryptology ePrint Archive, Report 2019/096, 2019. `https://eprint.iacr.org/2019/096`.
21. E. Dottax, C. Giraud, and A. Houzelot. White-Box ECDSA: Challenges and Existing Solutions. In S. Bhasin and F. D. Santis, editors, *COSADE 2021*, volume 12910 of *LNCS*, pages 184–201. Springer, 2021.
22. J. Fan and I. Verbauwhede. An updated survey on secure ECC implementations: Attacks, countermeasures and cost. In *Cryptography and Security: From Theory to Applications*, pages 265–282. Springer, 2012.
23. J.-C. Faugère, C. Goyet, and G. Renault. Attacking (EC)DSA Given Only an Implicit Hint. In L. R. Knudsen and H. Wu, editors, *SAC 2012*, volume 7707 of *LNCS*, pages 252–274. Springer, Heidelberg, Aug. 2013.
24. FIPS PUB 186-4. *Digital Signature Standard*. National Institute of Standards and Technology, July 19, 2013.
25. C. Giraud and E. W. Knudsen. Fault Attacks on Signature Schemes. In H. Wang, J. Pieprzyk, and V. Varadharajan, editors, *ACISP 04*, volume 3108 of *LNCS*, pages 478–491. Springer, Heidelberg, July 2004.
26. L. Goubin, J.-M. Masereel, and M. Quisquater. Cryptanalysis of White Box DES Implementations. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *SAC 2007*, volume 4876 of *LNCS*, pages 278–295. Springer, Heidelberg, Aug. 2007.
27. L. Goubin, M. Rivain, and J. Wang. Defeating State-of-the-Art White-Box Countermeasures with Advanced Gray-Box Attacks. Cryptology ePrint Archive, Report 2020/413, 2020. `https://eprint.iacr.org/2020/413`.
28. T. Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.2.1 edition, 2020. `http://gmplib.org/`.
29. Implicit White-Box Implementations: White-Boxing ARX Ciphers, 2022. Submitted.
30. J. Jancar, V. Sedlacek, P. Svenda, and M. Sys. Minerva: The curse of ECDSA nonces. *IACR TCHES*, 2020(4):281–308, 2020. `https://tches.iacr.org/index.php/TCHES/article/view/8684`.
31. JORF n°0241. Avis relatif aux paramètres de courbes elliptiques définis par l'État français, Oct. 16, 2011.
32. M. Karroumi. Protecting White-Box AES with Dual Ciphers. In K. H. Rhee and D. Nyang, editors, *ICISC 10*, volume 6829 of *LNCS*, pages 278–291. Springer, Heidelberg, Dec. 2011.
33. J. Katz and Y. Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

34. T. Lepoint, M. Rivain, Y. De Mulder, P. Roelse, and B. Preneel. Two Attacks on a White-Box AES Implementation. In T. Lange, K. Lauter, and P. Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 265–285. Springer, Heidelberg, Aug. 2014.
35. M. Lochter. RFC 5639: ECC Brainpool Standard Curves and Curve Generation, 2010. `https://tools.ietf.org/pdf/rfc5639.pdf`.
36. W. Michiels, P. Gorissen, and H. D. L. Hollmann. Cryptanalysis of a Generic Class of White-Box Implementations. In R. M. Avanzi, L. Keliher, and F. Sica, editors, *SAC 2008*, volume 5381 of *LNCS*, pages 414–428. Springer, Heidelberg, Aug. 2009.
37. P. Q. Nguyen and I. E. Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Des. Codes Cryptogr.*, 30(2):201–217, 2003.
38. D. Poddebniak, J. Somorovsky, S. Schinzel, M. Lochter, and P. Rösler. Attacking deterministic signature schemes using fault attacks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 338–352. IEEE, 2018.
39. M. Rivain and J. Wang. Analysis and Improvement of Differential Computation Attacks against Internally-Encoded White-Box Implementations. Cryptology ePrint Archive, Report 2019/076, 2019. `https://eprint.iacr.org/2019/076`.
40. A. Saxena, B. Wyseur, and B. Preneel. Towards Security Notions for White-Box Cryptography. In P. Samarati, M. Yung, F. Martinelli, and C. A. Ardagna, editors, *ISC 2009*, volume 5735 of *LNCS*, pages 49–58. Springer, Heidelberg, Sept. 2009.
41. O. Seker, T. Eisenbarth, and M. Liskiewicz. A White-Box Masking Scheme Resisting Computational and Algebraic Attacks. Cryptology ePrint Archive, Report 2020/443, 2020. `https://eprint.iacr.org/2020/443`.
42. Standards for Efficient Cryptography Group (SECG). *SEC 2 Ver 2.0 : Recommended Elliptic Curve Domain Parameters*. Certicom Research, Jan. 27, 2010.
43. S. Vanstone. Responses to NIST's Proposal. *Communications of the ACM*, 35:50–52, 1992.
44. B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel. Cryptanalysis of White-Box DES Implementations with Arbitrary External Encodings. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *SAC 2007*, volume 4876 of *LNCS*, pages 264–277. Springer, Heidelberg, Aug. 2007.
45. Y. Xiao and X. Lai. A Secure Implementation of White-Box AES. In *2nd International Conference on Computer Science and its Applications*, pages 1–6. IEEE, 2009.

# A Attacks Summary Table

Table 5 presents for each challenge submitted to WhibOx 2021, the successful attacks and the value of the corresponding key.

**Table 5.** Vulnerabilities of the various challenges.

| Challenge | Hooking | Collision | Fault | Lattice | Key |
|---|---|---|---|---|---|
| 3 | ✓ | ✓ | ✓ | ✓ | 45189C81EADEE03202BFA06EAA15831789F0C76575508A563E1A739CA37B87BE |
| 4 | ✓ | ✓ | ✓ | ✓ | 22BEF7AC4C31B2B98227D95B5EB49AF23343004CF2713FED48BEC3B5B7C3D24D |
| 8 | ✓ | ✓ | ✓ | ✓ | F484955872415A32B1B5B731EA1A8C729458055C17DC5FE9C57BCB39D1A40BFE |
| 10 | ✓ | ✓ | ✓ | ✓ | 32D67733DF0D0257DA78E92752494CFD5112E303BA1413388126EA33BB60AEFC |
| 11 | ✓ | ✓ | ✓ | ✓ | E7F3287D91B528D78BF19D5E62828C845E1A4027A3E1F988B62B7407EBF5CF38 |
| 12 | | ✓ | | ✓ | 773F0C0FFACB531F50FAE0987D2B8972FE1B9231BBF46859F475BAFB45257FED |
| 13 | | ✓ | | ✓ | 034332A23341538143FDB88F314FD942501FF8B6BA6A14D5013F1FC0984924BE |
| 15 | | ✓ | | ✓ | 3F77C51259E1C8CC48217A66998CCF3212A17120B0FCA09163E300576DFCD9E7 |
| 16 | | ✓ | | | 23773F0BECFACB534250FAE0987D2B8969D1AFD7EF942F148746DC73A3C6B39A |
| 32 | ✓ | ✓ | | ✓ | 32D67733DF0D0257DA78E92752494FFD5112E303BA14133FF126EA33BB60AEFC |
| 33 | | | ✓ | | CD9540B70C2F92B2894594CABC4E724203A615B9144C459714758BC3CAA12242 |
| 34 | | ✓ | ✓ | ✓ | 70253E6587D04D7A9A30A1461A80FCD235B28FBFC11FE8534CDFCE0A341C9257 |
| 36 | | | ✓ | | 10D7EF92F06DF6EB94F2F344085DAD51D3A550E24A4569922460F579CB5DF11A |
| 38 | | | ✓ | | 70C3A9F11773C8DD795FD7942B5DB448FDFA5D12E6EC387691A19B6E523AE6AE |
| 42 | | | ✓ | | 1BEDDC1DD79F8856BF2E1FD66EB194073D60FEC658C5D0E2C8BAE02DC72ADF65 |
| 44 | | | ✓ | | B519BB44EC5BF3380CB2DF555F39ED836CDBF4961E43A66C218FADB211BF468C |
| 45 | ✓ | ✓ | ✓ | ✓ | 32D67733DF3D0257DA78E92752494FFD5112E303BA14133FF126EA33BB60AEFC |
| 50 | | | ✓ | | 7A7AA97370B1EE16D64C71C7C5BC8C9F9456FBEA603780883399D89DA43F8A15 |
| 54 | ✓ | ✓ | ✓ | ✓ | 32D67733DF3D0257DA78E92752494FFD5112E22222222222F126EA33F6E49790 |
| 55 | ✓ | ✓ | ✓ | ✓ | 0049859485984958495 4E92752494FFD5112E2222222EE22F126EA33F6E49790 |
| 57 | ✓ | ✓ | ✓ | ✓ | 7D1BBD475A8EB5AF7DDB238CD8A67F86B601E0EA101C04036849B31F96CA6083 |
| 58 | | ✓ | ✓ | ✓ | BD3026C700A75B5970807802E2B47C2A892DF85E3CE57366D335EEBABCAAE255 |
| 61 | | ✓ | ✓ | ✓ | F4DDC95A88146CF52DEC752E737F8E3FB16AE4F6B7E726068946F3B0BA0C8E95 |
| 62 | | ✓ | ✓ | ✓ | 0A99EB20F9DE4DD7607288B8B766F6217FE5D2CE6DDD51C6159941066AF192ED |
| 66 | | ✓ | ✓ | ✓ | 8836AC84AA148440A20628810CA65EB038BB625841275CC11590D8F5BC7F1BAC |
| 70 | | ✓ | ✓ | ✓ | 21A35C57E23B2D23ADDA19EA30325F1B532DA645489E29E47A13E92CA1F6670C |
| 71 | | ✓ | ✓ | ✓ | 588BEED930355AF54EEBAFAA46A7D26DA378A36EF5CD15D1F876D753A395F8AF |
| 72 | | ✓ | ✓ | ✓ | B7A9B0F7661FC9A1DEC001F2C2C9EAE08748AEB187E1247726663E3DD1AB36BF |
| 73 | | ✓ | ✓ | ✓ | 4DAA29CBD634F28137499B9557104FDD36D4D4EFDFE87EFC0D8BD03555F8497F |
| 74 | | ✓ | ✓ | ✓ | 12691AAC55A079F529FE81205DF775EF297A14CA81499BF0857643E694CF8816 |
| 76 | | ✓ | ✓ | ✓ | F5178EEC7A9779E13CE01B35C8264BF32C094B172051CA32156DC61485718318 |
| 77 | | ✓ | ✓ | ✓ | A0543814F86D1C4AF6A08094CD0246F606F7E76CEE47EC052B62328038146D93 |
| 78 | | ✓ | ✓ | ✓ | 511128DCBF369E985B99D07CC1668A2D28F4BA535CF7AC7926D4C5F696C3D35F |
| 79 | | ✓ | ✓ | ✓ | 595AD4C8A0EB2FDA798BC01D322F4C5ED098A2E749004B2B54FD815215F46686 |
| 80 | | ✓ | ✓ | ✓ | 8E938EA9BE9E51A28DFD30BD6EDB9D6765C1272B8F7048CE81021194759C3E52 |
| 81 | | ✓ | ✓ | ✓ | F134975C5A989635F1D9FA7469C848A953622E9DA1BED7E12455DCD2AFA070BE |
| 84 | | ✓ | ✓ | ✓ | 36A990B9F35B79934FB25C64681DE3A83FC178DC2383C585FFCFDDD7C1F6C2B7 |
| 85 | ✓ | ✓ | ✓ | ✓ | AB700D75274336FD26A1FE49D400ACEAE89F0FDBFE4BDE9A70373CA693003CA8 |
| 87 | | ✓ | ✓ | ✓ | 9A4D4A94A1FE0FA1C559764C85D06496BD752498E0B5A2459624211013B9A088 |
| 89 | | ✓ | | ✓ | C80682FCB2D78B2515A70A70D17C47A8512E24A127E797C073566D54586B9482 |
| 94 | | ✓ | | ✓ | A04B6199A1DFE39EF35F6302454D71C872771A2F02A27AB5EC8130DA226F6F90 |
| 96 | | ✓ | | ✓ | AFAAABE59B2EBB4FE15274E4EB5D1999C0554CC2D498BC92C59A3F6CD8FE2BC0 |
| 97 | ✓ | ✓ | ✓ | ✓ | 0754CA8EA936675EC3F64782A14E1A75B3D357044D4B2C434C6011279D17E829 |
| 100 | | | ✓ | | 7F58EDB783C1F3FA7FF424CF7F5DF6D4BCDCF18D8A98CE4559EC22EB17030578 |

| Challenge | Hooking | Collision | Fault | Lattice | Key |
|---|---|---|---|---|---|
| 101 | | | ✓ | | 25D31D3AFF5773799ECF43DEC1882B8F05D9231697BDDA5482DE05B14FB8A63B |
| 103 | | ✓ | | ✓ | CC977E0748722D615B845C1B10EA554B69DFCA640440CA5C468BBEF84B8C0442 |
| 104 | | ✓ | ✓ | ✓ | 638C9DFBF9F376CBB3E3B01DF27960EC53A689D2FF4DFF23D97EE5351ED4A3D0 |
| 105 | | ✓ | ✓ | ✓ | D29E9D130016D930BF830BCAD071BC6503F877FB207922A9E495CF71A79631FE |
| 107 | | ✓ | ✓ | ✓ | 4E420B6AA9E9F07F19CF7ED97497871C1223BC2A68E83716575C235DE6D63E17 |
| 108 | | | ✓ | | 60609404F0B9086D3A995AF0680D048724CF2B1AF2B33CEA8DD4AF4B62A5DDBB |
| 114 | ✓ | | ✓ | ✓ | 0000000000000000000000000000000000000000000000000000000000000005 |
| 127 | | | ✓ | | 1144D82B9568581405D10CF8B219FF7E94E4559E0832B06056F1F87D43C75777 |
| 135 | ✓ | ✓ | ✓ | ✓ | 0C2A5692FE1A7F9B8EE7EB4A7CD59CD62BCE33576B3123CECBB6406837BF51F5 |
| 136 | ✓ | | ✓ | ✓ | 0C2A5692FE1A7F9B8EE7EB4A7CD59CD62BCE33476B3123CECBB6406837BF51F4 |
| 139 | ✓ | | ✓ | ✓ | 000000000000000000000000000000004319055358E8617B0C46353D039CDAA9 |
| 153 | ✓ | | | ✓ | 9C29EDDAEF2C2B4452052B668B83BE6365004278068884FA1AC3F6D0622875C3 |
| 157 | ✓ | ✓ | ✓ | ✓ | F04DBFD1147F9D43747538C1C9256DD2BC20562F9D92B83E9AFA751299B160A4 |
| 165 | | ✓ | ✓ | ✓ | 84DAF8B6620FC6669BF1EE264D1B214A4FBECACEADDFDC0DCBC89CF4B6E3232B |
| 166 | ✓’ | | ✓ | | C746740A4A6BCBD462D9041023A0FEF5CCF0328FF80D9C50132682030D77D33C |
| 172 | | ✓ | ✓ | ✓ | 285E57F7BDDAAA6201D8870A0B9B168C7A5D8200085F62504EE3EBFCC11EF150 |
| 174 | ✓ | ✓ | ✓ | ✓ | 9C29EDDAEF2C2B4452052B668B83BE6365004278068884FA1AC3F6D0622875EC |
| 185 | ✓ | ✓ | ✓ | ✓ | 7729EDDAEF2C2B4452052B668B83BE6365004278068884FA1AC3F6D0622875EC |
| 187 | ✓ | ✓ | ✓ | ✓ | 7779EDDAEF2C2B4452052B668B83BE6365004278068884FA1AC3F6D0622875EC |
| 192 | | | ✓ | | 09302BDFA5313312B9A665316F7E9365DCC57DA7E21FD8612CDCD553BABB51FE |
| 193 | | | ✓ | | E0FE06BE0684455EDD2F5134A3AE8B9F6852561C821672FA16606986233BF811 |
| 209 | | | | ✓ | 6E3A09F8EC613B8A524F7608CB80B2D3C510E27506AD84FA14C3B6D018E659F7 |
| 212 | | | ✓ | | D663E156F036F11D4E73CC0EC09A952DEAED316947DF73EB28467EC623C5740D |
| 226 | | | | | 6F1D9093F3D5AE7C5F133659295914C9AF22E54B4ADE38CA421CA9BBD3D48A50 |
| 227 | | | | ✓ | ADA6C6A1049825989811C9495D83681A68C67AB5E8EBDDC126CEE77056A7BB27 |
| 228 | | | | ✓ | EA7BA345EB9D99F54261D01AE6319B184769E5745621706D77018E0DB46DDAFA |
| 231 | ✓ | ✓ | ✓ | ✓ | 8ADE24EE6413C6E408784DBB4D81D04F33238AB503CBE35C77400517EE5ABC96 |
| 235 | ✓ | ✓ | ✓ | ✓ | 0000000000000000000000000000000000000000000000000000D0FACADE0DEFACED |
| 251 | | ✓ | ✓ | ✓ | DDE098A74086ECBB4DBA1848511BEA924145D1A9ED2EC9E64E0C5934BAAC97AE |
| 253 | | ✓ | | ✓ | B22DB44C9E66D567B3B2CBB3C720309D1EEAD38717017F5E79F05274F289A52C |
| 256 | | | ✓ | | F1662664E7E303740C0CA3927F9870A789978DAE95892302E73C85E3993B4CC9 |
| 261 | | | ✓ | ✓ | 3266C9F6379DFDAE4AA763E8E6BA94526504CA364C482306829D4BF1E97BFF92 |
| 262 | | | ✓ | | A0F00DCAA5DAB169FD4DFE2186BCBCBD22631AB68BFEFF1FC19306174EAF8970 |
| 264 | | | | ✓ | D0EE17829A397C18074EA3888057AE815B5336773F9668E6CE4464D4B2B05F1F |
| 267 | ✓ | ✓ | ✓ | ✓ | C17536B60BCF94326A9C8CA17E0FC4EDBD76822532B350E8237CA2D8CF9C74B0 |
| 274 | ✓ | ✓ | ✓ | ✓ | 0080ECD2A00080ECD2A00080ECD2A00080ECD2A00080ECD2A00080ECD2A00080 |
| 283 | | | ✓ | | 79FE8D884DC2F7440824DE79C9F7C513C2B4549631D343523C73CB8F85983A4F |
| 299 | ✓ | ✓ | ✓ | ✓ | 3A0F803A874CD5B826023F2073FF200371D399E76E66B05E1241AA787B0564D6 |
| 304 | | | ✓ | | EE8942A527CA1A58B8A8EA369441CB8518836DDB98F6380B8008B6053BC8182C |
| 305 | | | | | 311EA92FBCDD3C6A29D269589A9E71F13A231FFEC85FF36B398967EC9934805E |
| 307 | ✓ | | ✓ | | FA3FCDE70679E7E44391F7157E2B5822F5B9B9C93ADD95C2BA90FF4B95C8A6BB |
| 308 | | | ✓ | ✓ | 84CCCAA904CB397F41A36FF9E05D4EB6C58B8E203E02373C465B6C3F03280C82 |
| 314 | | | | ✓ | 7E045DB89DD77BD6B2EAF23172A89A656B5084748642DB82BBAE931E737560C2 |
| 320 | ✓ | ✓ | | ✓ | D235C2B1D089F158A0AE4E7799C2DCA9985E3D44C8F243BAD8B5E1A4EB647E1B |
| 321 | ✓ | ✓ | ✓ | ✓ | BA15757E1B0DB122F349C0C50C97071A4CFFF4FD2875B4A092FBDD985E8595DE |
| 323 | ✓ | ✓ | ✓ | ✓ | C7491BBC530FFA9DDCF3E7D732536FACF04239693D549C50DDAD41931A6244C2 |
| 325 | | | ✓ | ✓ | 6902CD65AE124A45B9DD16BAEFD26D9CFFB5C291DC1E256D9CCE17BE3CF11775 |
| 327 | | | | ✓ | 2BC6F2467C7F8DFA164EDC68DDCF65E795B8A2153182565481D8D6878D80EA81 |
| 328 | | | | ✓ | 37170CF851A89AAFD3511234BE2B96C89B783A44D7A6C22E9A150872809F7CDF |
| 345 | | | ✓ | | 3266C9F6378DFDAE4AA763E9166B131E6514CA364C482306829D4BF1E97BFF92 |
| 346 | | | | | 5EE43950837D0ABA419FE5B586D1A7AA44DDAAC6327DADC3133F18A850211B9F |

## B Some Remarks on the Challenges

Among the various submissions, we notice the following facts:

- Challenges 15 and 16 have a very small code size, only 194 bytes! To obtain such tiny implementations, the designers use a fixed nonce $k = 1$ (i.e. $r = G_x$) and use a private key $d$ such that $dr \equiv 2^i \bmod n$. In such a case, the signature of a hash $e$ is equal to $(G_x, e + 2^i)$.
- Challenge 114 uses a very small private key, indeed $d_{114} = 5$
- Challenge 274 uses a sparse (and funny) private key equals to `0080ECD2A-00080ECD2A00080ECD2A00080ECD2A00080ECD2A00080ECD2A00080`.
- Despite what is indicated in the rules (cf. Sect. 2), some challenges are not deterministic[10], i.e. the two signatures of the same message could be different. All these challenges use the `time()` function to obtain some randomness. However, it is easy to hook such calls and return a constant value.

[10] Challenges 54, 55, 57, 58, 61, 62, 66, 70, 71, 72, 73, 74, 76, 77, 78, 79, 80, 81, 84, 89, 94, 96, 103, 104, 105, 107, 136 and 139 are not deterministic.