ECDSA White-Box Implementations

IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2022, No. 4, pp. 527–552. DOI:10.46586/tches.v2022.i4.527-552

ECDSA White-Box Implementations: Attacks and Designs from CHES 2021 Challenge

Guillaume Barbu[1], Ward Beullens[2], Emmanuelle Dottax[1], Christophe Giraud[1], Agathe Houzelot[1,3], Chaoyun Li[4], Mohammad Mahzoun[5], Adrián Ranea[4] and Jianrui Xie[4]

[1] IDEMIA, Cryptography & Security Labs, Pessac, France
firstname.lastname@idemia.com
[2] IBM Research, Zurich, Switzerland
wbe@zurich.ibm.com
[3] LaBRI, CNRS, Université de Bordeaux, Bordeaux, France
[4] imec-COSIC, KU Leuven, Leuven, Belgium
firstname.lastname@esat.kuleuven.be
[5] Eindhoven University of Technology, Eindhoven, 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, scientific literature on ECDSA white-box design is scarce. The CHES 2021 WhibOx contest was thus held to assess the state-of-the-art and encourage relevant practical research, inviting developers to submit ECDSA white-box implementations and attackers to break the corresponding submissions.

In this work, attackers (team TheRealIdefix) and designers (team zerokey) join to describe several attack techniques and designs used during this contest. We explain the methods used by the team TheRealIdefix, which broke the most challenges, and we show the efficiency of each of these methods against all the submitted implementations. Moreover, we describe the designs of the two winning challenges submitted by the team zerokey; 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. In this context, 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 Contest

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 execution

Received: 2022-04-15 Accepted: 2022-06-15 Published: 2022-08-31

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. [CEJvO02, CEJv03], and the field has continuously developed since then, with design proposals [BG03, BCD06, XL09, Kar11, DFLM18, RW19, SEL21, BCC21], attacks [BGEC04, GMQ07, WMGP07, MGH09, DWP10, DRP13, LRD+14, AMR19, GRW20] and efforts to define security notions [SWP09, DLPR14, AABM20].

The industry shows a growing interest 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 several 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 and decided to consider the ECDSA signature algorithm, whose white-box implementation is of substantial interest to the 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 would help to discern weak points inside the implementations. Besides, the analysis of the design of the most resistant challenges, which successfully defeated most attackers, would also give directions for future designs.

Contributions. In this paper, teams TheRealIdefix — who broke the most challenges — and zerokey — who proposed the two winning challenges — join to present how they proceeded during the contest[1]. On the attack side, we describe a strategy to achieve efficient attacks. As reverse engineering is a time-consuming task, automated attacks are desirable. We consider different attack paths against ECDSA white-boxes: the ones inherited from traditional cryptanalysis, the extensions of attacks in the grey-box model, and the logical attacks of the software. We discuss the feasibility of automating each attack path and provide detailed information regarding which attacks succeeded (or failed) on each candidate. Our results show that, with few exceptions, it was sufficient to fully recover the secret value by these automated attacks. On the design side, we describe the methodology we used to build the two winning challenges, Challenges 226 and 227. It includes modifying the implicit framework [RVP22] originally proposed for block ciphers, applying techniques from multivariate public-key cryptosystems, and obfuscating the resulting C code with a C obfuscator. Our design thus turns the ECDSA signature algorithm into a sequence of systems of low-degree equations which are obfuscated with large affine encodings and additional variables and equations. Finally, we show how to break Challenge 226 with automated attacks and how to break Challenge 227 once reverse-engineered.

[1] Guillaume Barbu, Emmanuelle Dottax, Christophe Giraud and Agathe Houzelot are part of the team TheRealIdefix; Ward Beullens, Chaoyun Li, Mohammad Mahzoun, Adrián Ranea and Jianrui Xie compose the team zerokey.

Outline. 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 various implementations and some statistics regarding the success rate of these methods. Section 5 discloses the designs of Challenges 226 and 227 proposed 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 P-256 curve under a hard-coded, freely chosen key, and accepting as input any 256-bit message digest $e = H(m)$. Notice that the cryptographic hash function H is excluded from the intended white-box implementation of ECDSA and the message m is also not provided. At the same time, attackers were encouraged to extract the private keys. In addition, acceptance of submitted implementations was conditioned on 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 [Gt20],
- 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. Eventually, the challenge with the highest number of strawberries wins the competition. Accordingly, when submitting a matching private key to the system, attackers receive bananas, the number of which is determined by the number of strawberries of the challenge at the time of the break. More detailed information can be found on the contest website [CHE].

3 ECDSA and White-box Implementations

3.1 ECDSA

In 1992, Vanstone introduced a variant of DSA based on elliptic curves. The resulting public-key signature algorithm is called *Elliptic Curve Digital Signature Algorithm* (ECDSA) [Van92]. 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 private 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, where R_x and R_y denote the coordinates of the point R.

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 message m:

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

Algorithm 1: ECDSA signature

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

1 $e \leftarrow H(m)$
2 $k \xleftarrow{\$} [\![1, n-1]\!]$
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 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 (r', s') of a different message $m' \neq m$ using the same nonce, that is with $k' = k$. In that case, we also have $r' = r$, so the adversary may compute

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

In the black-box model, the security of ECDSA is based on the difficulty of the *Elliptic Curve Discrete Logarithm Problem* (ECDLP), i.e., 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. [Loc10, Sta10, JOR11, FIP13]. However, there is a gap between the security of ECDSA in theory and that of ECDSA implementations. Many grey-box attacks have been described in the literature (see for example [FV12]). 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 allows an adversary to compute the secret key. Recovering a few bits of the nonces associated to different signatures may be enough for an attacker. Indeed, this allows the construction of a system of equations that can be solved using lattice-based algorithms [BH19, JSSS20] or Bleichenbacher's FFT-based approach [ANT+20]. 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 complicated to achieve a secure implementation of ECDSA 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 such as k or d. The first technique to prevent secret data from appearing in plain was introduced by Chow et al. in [CEJv03]. Their idea is to embed the key into the algorithm, and each operation is performed with the help of look-up tables protected by carefully crafted *encodings*. Informally, the algorithm is split 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 required memory drastically increases with the algorithm's complexity. Using it to secure operations as

complex as scalar multiplications or inversions while remaining efficient is thus a real challenge.

Another challenge in white-box cryptography is the impossibility of relying on any external source of randomness. An attacker could simply disable such a source and fix its output to a constant value. For example, in the context of AES, this renders some countermeasures against side-channel or fault attacks based on randomization techniques completely inefficient. When one considers ECDSA signatures, disabling the source of randomness yields multiple uses of the same nonce and, thus, easy recovery of the private key, as seen in Sect. 3.1. The solution is to compute k as a function of the only source of randomness available, the input message: $k = f(m)$. In order to maintain the security of the signature scheme, this mapping must be computationally indistinguishable from what a randomly and uniformly chosen function would return. We will see in the next section that many challenges of the WhibOx competition did not fulfill this requirement.

4 Breaking the Challenges

White-box implementations usually rely on encodings and other theoretically sound approaches to protect the secret values and their manipulations. It is also very often the case that code obfuscation techniques are used to make understanding the design a time-consuming and challenging task. Extensive use of such obfuscation techniques in the submitted source files causes independently reverse-engineering each challenge to be overwhelming in time. We thus focused on designing attack methods that could be efficient and easily automated.

This section looks at the different attacks that can be automated in a white-box context and gives the rationale for using and discarding them. We then present the results of applying the selected methods to the whole set of submissions.

4.1 Attack Methods

4.1.1 Hooking Shared Libraries

The contest rules were 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 has been 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[2]. 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.

As a side note, this technique also jeopardizes implementations relying on system-dependent random generators such as `srand` or `mpz_XrandomX` functions, or on other sources such as `time`.

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

4.1.2 Biased Nonces

As explained in Sect. 3.2, white-box designers usually generate the nonce k from the input. In the case of the WhibOx contest, the nonce is thus computed as a function of the hash, i.e. $k = f(e)$. However, if the function f is not carefully selected, it could happen that the k_i's generated from different e_i's are not uniformly random.

In the worst case, we have collisions such that different hash values e_0 and e_1 produce the same nonce ($k_0 = k_1$). If such a collision occurs, one can recover the private key d as explained in Sect. 3.1. Furthermore, collisions can be efficiently detected by looking at the r part of the signature. To efficiently browse a subset of hash values in search of such collisions, we limited ourselves to hash values with a Hamming weight equal to 1 or 2. We thus considered 32 896 hash values and were able to break 60% of the challenges with this technique.

In those cases where we did not find any collision, we looked for biases in the nonce generation. We used well-known lattice attacks derived from [NS03] and [FGR13] to exploit such a potential weakness. 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 succeed 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.

Lattice-based attacks can also be applied when the ephemeral key is the product of a small random κ 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 κ 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 $l = 6$ known most- or least-significant bits of the ephemeral key:

$$k_{msb}2^L + k_{lsb} = s^{-1}(e + rd) \bmod n \ , \tag{3}$$

  with $L = 256 - l$ for the MSB case and $L = l$ for the LSB case (we considered both cases where the known value is 0 or $63 = 2^6 - 1$),

- 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.

4.1.3 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 [BHMT16] is called *Differential Computational Analysis* (DCA). 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. Whereas in the grey-box context, one can record the power consumption of the device in which the algorithm is implemented, a white-box attacker can simply use software execution traces. By instrumenting the binary, he can record completely noiseless traces of all accessed addresses and data over time, leading to much more efficient attacks.

In theory, this attack is particularly devastating since it can be fully automated and does not require any earlier reverse engineering step. In practice, it is quite difficult to apply because of the size of the traces, in particular for cryptosystems such as ECDSA that have relatively long execution time. Indeed, if the whole white-box execution were to be recorded, each trace would easily reach several gigabytes. For instance, tracing n 64-bit registers on a 3GHz machine during 3 seconds would lead to a single trace of $9 * 8 * n$ Gb. Therefore, iterating over dozens of traces for a CPA would be overwhelming in time and memory. A time-consuming step of reverse engineering allowing to select a smaller window of the implementation before the attack is thus required, which is why we did not use this technique to break the challenges of the WhibOx contest.

4.1.4 Fault Injections

Another attack method is to disturb the algorithm execution and exploit the resulting faulty output. In the white-box context, faults can be easily induced since the attacker can 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.

All the fault attacks that can be performed in the grey-box context are obviously also a potential threat in the white-box context. In the case of ECDSA, different faults can be induced on different variables to give an exploitable result. The most obvious attack is to force the use of a weak elliptic curve during the scalar multiplication by disturbing the curve parameters [BMM00] in order to solve the discrete logarithm problem easily. The attacker can also force the use of biased nonces, 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 [GK04].

In addition, the white-box model offers new possibilities [PSS+18, ABF+18, DGH21]. They arise from the fact that deterministic versions of the scheme have to be implemented due to the impossibility of relying on a source of randomness in this context. When the algorithm is used twice on the same message, the same nonce k is derived. The attacker may thus obtain a correct signature for a given digest e, and an erroneous one by modifying a second execution of the same signature. To break the challenges of the WhibOx contest, we mainly disturbed the computation of the first part of the signature r, obtaining faulty results $\tilde{r}$ and $\tilde{s} = k^{-1}(e + \tilde{r}d) \bmod n$. Some secret information can be deduced from the correct and 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 then compute the private key:

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

It is also possible to disturb other variables, but still, 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, allowing us to break 75% of the challenges.

4.2 Attacks Results

When applying the various attack methods described above, 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 of each of the 97 submitted challenges as well as the corresponding private key.

However, we noticed that many challenges had a low level of security, some of which were even plain implementations. We thus excluded 30 challenges[3] where the nonce and/or the private key were manipulated in plain. Table 2 illustrates the efficiency of the attacks presented in Sect. 4.1 on the remaining 67 challenges. We observe that hooking gives no significant result anymore, collision and lattice attacks become 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, Challenges 226 and 227 are the winning ones. In the next section, we present the design of these two white-box implementations.

5 Design of the Winning Challenges

In this section, we describe the designs of the two winning challenges of the WhibOx contest: Challenges 226 and 227. The designs of both challenges were inspired from the white-box implicit framework [RVP22], 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 mentioned in Sect. 2, in the WhibOx contest, a challenge gains strawberries quadratically with time before being broken. The rule is that challenges that are either smaller, faster, or less memory-consuming gain strawberries faster. As a result, we strategically

[3]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.

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 stood unbroken for the longest time (35 hours).

Note that these challenges were specifically built for the WhibOx contest, where attackers did not know the design. Against an attacker who knows the design details, these challenges are easy to break once reverse-engineered.

This section first introduces the implicit framework, then describes the shared design approach of both challenges, and finally explains the additional countermeasures used in each challenge. For access to the underlying software used to build these challenges, please contact the authors from the team `zerokey`.

5.1 Implicit White-box Implementations

The implicit framework is a method to obtain a 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 to 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 [RVP22] for vectorial functions over the binary field, we 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 $(\mathbb{F}_q)^{l'}$ is called a (l, l') function over $\mathbb{F}_q$, and its l' component functions are 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' multivariate 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,

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

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 [RVP22] 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 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 (cf. 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 message digest e
Output : the signature (r, s)

1 $state \leftarrow e$
2 $k, state \leftarrow \texttt{NonceDerivation}(state)$
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 $(R'_x, k', e') = (R_x \bmod n, k \bmod n, e \bmod n)$ 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 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 $E^{(2)}$.

5.2.1 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 call round functions. Then we explain how to encode these round functions and how to obtain low-degree quasilinear implicit functions 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., $(G_{i,0} = [k_{i,0}]G, G_{i,1} = [k_{i,1}]G)$ for $1 \le i \le t$. Then, for each pair we select one of the two points together with its logarithm, denoted as (G_{i,b_i}, k_{i,b_i}), where $b_i \in \{0,1\}$ and $1 \le i \le t$. We add the selected points and the selected logarithms, obtaining the scalar multiplication

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

where $k = k_{1,b_1} + \cdots + k_{t,b_t}$. 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 $k_{i,j}$ are chosen such that the sum of $\max(k_{i,0}, k_{i,1})$ 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.

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

Input : the bits $(e_1, e_2, \ldots, e_{256})$ of the hash e (little-endian order)
Output : x-coordinate of $[k]G$ and the message-dependent scalar k

`/* Round 1: input` e_1, $k_{1,0}, k_{1,1}, G_{1,0}, G_{1,1}$ `embedded values */`
$R \leftarrow [1 - e_1]G_{1,0} + [e_1]G_{1,1}$
$k \leftarrow (1 - e_1)k_{1,0} + e_1 k_{1,1}$

`/* Round` i: `input` (R, k, e_i), $k_{i,0}, k_{i,1}, G_{i,0}, G_{i,1}$ `embedded values */`
for $2 \le i \le t$ **do**
  $R \leftarrow R + [1 - e_i]G_{i,0} + [e_i]G_{i,1}$
  $k \leftarrow k + (1 - e_i)k_{i,0} + e_i k_{i,1}$
end
return R_x, k `//` $(R_x, R_y) = R = [k]G$

By considering the precomputed points $G_{i,j}$ and their logarithms $k_{i,j}$ as fixed values and by representing the elliptic curve additions by operations over $\mathbb{F}_p$, we can represent $E^{(1)}$ given by Algorithm 3 as an iterated function over $\mathbb{F}_p$, that is,

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

where each $(4,4)$ round function $F^{(i)}$ is given by the following component functions

$$\begin{aligned} F^{(i)}_{1,2}(u_1, u_2, u_3, u_4) &= (u_1, u_2) + [1 - e_i]G_{i,0} + [e_i]G_{i,1} \\ F^{(i)}_{3}(u_1, u_2, u_3, u_4) &= u_3 + (1 - e_i)k_{i,0} + e_i k_{i,1} \, , \\ F^{(i)}_{4}(u_1, u_2, u_3, u_4) &= u_4 + e_i 2^i \, . \end{aligned} \tag{11}$$

Note that the input value of $F^{(1)}$ is $(0,0,0,0)$, and each round function $F^{(i)}$ takes the hash bit e_i as an additional input value. The pair of component functions $F^{(i)}_{1,2}$ return a point on the elliptic curve with $F^{(i)}_1$ and $F^{(i)}_2$ the x- and y- coordinates respectively. The component function $F^{(i)}_3$ updates and returns the current nonce, while $F^{(i)}_4$ updates the current hash. The hash e is recomputed so that $E^{(1)}$ can output the hash e and provide it in an encoded form to $E^{(2)}$.

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

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

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,

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

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 first show how to derive an implicit function of the elliptic curve addition.

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 $P + Q = R$ where P and Q are not the point at infinity and where P and Q have different x-coordinates[4]. In this case, R can be written as [KL14]

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

From Eq. (14) it is easy to see that $P + Q = R$ holds if and only if the relations

$$(P_x + Q_x + R_x)(Q_x - P_x)^2 = (Q_y - P_y)^2 \tag{15}$$

$$(R_y + P_y)(Q_x - P_x) = (Q_y - P_y)(P_x - R_x) \tag{16}$$

hold. Note that these relations have degree 3 (degree 1 over the variables R_x and R_y), while Eq. (14) has a high degree due to the inversion over $\mathbb{F}_p$.

Thus, the function $\texttt{IMP}(P_x, P_y, Q_x, Q_y, R_x, R_y) = (\texttt{IMP}_0, \texttt{IMP}_1)$ defined by

$$\begin{aligned} \texttt{IMP}_0 &= (Q_y - P_y)^2 - (P_x + Q_x + R_x)(Q_x - P_x)^2 \\ \texttt{IMP}_1 &= (Q_y - P_y)(P_x - R_x) - (R_y + P_y)(Q_x - P_x) \end{aligned} \tag{17}$$

is a quasilinear implicit round function of ADD with degree 3, assuming none of the points is the point at infinity and assuming the x-coordinates of the points are different.

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

$$\overline{T^{(i)}} = M^{(i)} \circ T^{(i)} \circ \left((A^{(i-1)})^{-1}, (A^{(i)})^{-1}\right) \tag{18}$$

[4]The only elliptic curve additions performed in $E^{(1)}$ are the additions between the random points $G_{i,j}$, and the probability that these additions involve points at infinity or points with the same x-coordinate is negligible.

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

The white-box implementations of 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. (18). 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 ith round is computed by finding the solution of the affine system $\overline{T^{(i)}}(\mathbf{u}; \mathbf{v}) = 0$ for $\mathbf{v}$.

5.2.2 White-boxing the Computation of s

Now we turn our attention to $E^{(2)}$, the second step of the signing algorithm, where we compute $r = R_x \bmod n$ and $s = k^{-1}(e + dR_x) \bmod n$, and output the signature (r, s). As opposed to $E^{(1)}$, we do not decompose $E^{(2)}$ but 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

$$\begin{cases} T_1^{(t+1)}(R_x, R_y, k, e; s, r) = ks - e - dR_x \\ T_2^{(t+1)}(R_x, R_y, k, e; s, r) = r - R_x \end{cases} \tag{19}$$

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 $\mathbf{u} = A^{(t)}(R_x, R_y, k, e)$, 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

$$\overline{T^{(t+1)}}(\mathbf{u}; s, r) = M \cdot T^{(t+1)}((A^{(t)})^{-1}(\mathbf{u}); s, r) \ , \tag{20}$$

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

$$\mathbf{u} = A^{(t)}(R_x, R_y, k, e) - p\mathbf{o} \ , \tag{21}$$

then $(A^{(t)})^{-1}(\mathbf{u}) = (R_x, R_y, k, e) - pL_t^{-1}(\mathbf{o}) \bmod n$, 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 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 $\overline{T^{(t+1)}}(\mathbf{u}; s, r)$ to solve 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 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

$$A^{(t)}(R_x, R_y, k, e) = \begin{pmatrix} 1 & 0 & 1 & 2 \\ 1 & 1 & 2 & 0 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 0 & 1 \end{pmatrix} \begin{pmatrix} R_x \\ R_y \\ k \\ e \end{pmatrix} + \mathbf{c} \ . \tag{22}$$

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 inefficiently guessing $\mathbf{o} \in [0, 4]^4$ at random, 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 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, 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 input 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 Tables 3 and 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.

In particular, to avoid the bias in the most significant part of the nonce k due to the constraint $k < n$ (see Section 5.2.1), we duplicated the nonce variable and its equations so that $\overline{E^{(1)}}$ outputs an additional nonce variable k' similarly to k,

$$k' = k'_{1,b_1} + \cdots + k'_{t,b_t}, \quad \sum_i \max(k'_{i,0}, k'_{i,1}) < n\,,$$

and $\overline{E^{(2)}}$ uses the sum of the nonce variables $k + k'$ as the final nonce. On top of that, instead of e, the input $L(e)$ is given to $\overline{E^{(1)}}$ for some hard-coded low-degree encoding L, and its inverse $L^{(-1)}$ is composed to $\overline{E^{(2)}}$ to recover e. Note that this is a minor trick since the encoding L is not merged or composed with other functions (as opposed to the other encodings $A^{(i)}$), and the computation $L(e)$ is done in clear.

Since adding additional variables and equations also introduces significant overhead in the implementation size, we only applied the second technique 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 [Col] 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[5] 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

5.4.1 Description

Following the method described in Section 5.2, we built Challenge 227 (`keen_ptolemy`) as a lightweight white-box implementation. The only additional countermeasures from Section 5.3 included in Challenge 227 are the degree increase of each component to the total degree of the corresponding vectorial function and the code obfuscation by Tigress. Challenge 227 was the winning implementation of the WhibOx contest; it achieved the highest number of strawberries (20.39) and stood for 33 hours as the second-longest.

Table 3 describes the memory complexity of Challenge 227 (after applying the additional countermeasures) by describing $\{\overline{T^{(1)}}, \ldots, \overline{T^{(t)}}\}$ and $\overline{T^{(t+1)}}$, the implicit round functions of $\overline{E^{(1)}}$ and $\overline{E^{(2)}}$ respectively. 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.

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[6] provided by the competition, the size of the compiled binary is 4.42 MB, and the average running time and

[5] https://tigress.wtf/transformations.html
[6] https://github.com/CryptoExperts/whibox_contest_submission_server

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	27×4	130×4	255×3	18×2

RAM consumed are 0.04 seconds and 6.14 MB respectively. The code obfuscation did not impact the running time but increase the binary size by 8% and the average RAM by 3%.

5.4.2 Security Analysis

Challenge 227 can be broken in several ways. Here, we explain how the attacks of Sect. 4 allow one to recover the secret key of Challenge 227 or why they do not work.

Hooking shared libraries. During the implicit evaluation of $\overline{E^{(2)}}$ for the valid input $\mathbf{u} = A^{(t)}(R_x, R_y, k, e)$, the affine system $\overline{T^{(t+1)}}(\mathbf{u}; s, r)$ is solved for r and s. By denoting the entries of M as $M = \begin{pmatrix} m_0 & m_1 \\ m_2 & m_3 \end{pmatrix}$, it is easy to see that this affine system is given by the equations

$$\begin{cases} c_1 s + c_3 r - c_5 = 0 \\ c_2 s + c_4 r - c_6 = 0 \end{cases} \tag{23}$$

where the coefficients c_i are given by

$$\begin{array}{lll} c_1 = m_0 k, & c_3 = m_1, & c_5 = m_0 e - m_0 d R_x - m_1 R_x \\ c_2 = m_2 k, & c_4 = m_3, & c_6 = m_2 e - m_2 d R_x - m_3 R_x \end{array} . \tag{24}$$

We stress that k, e, and R_x do not appear in the clear. They are expressed as linear combinations of the input $\mathbf{u} = A^{(t)}(R_x, R_y, k, e)$ of $\overline{E^{(2)}}$. Nevertheless, the coefficients c_i are operated in the clear during the Gaussian elimination, and the adversary can obtain their values.

Some of these coefficients are sensitive. In particular, if the attacker manages to find c_1 during the computation of two different signatures (r, s) and (r', s'), he may solve the following system of two equations with two unknowns (m_0 and d) in $\mathbb{F}_n$:

$$\begin{cases} m_0(e + rd) = c_1 s \\ m_0(e' + r'd) = c_1 s' \end{cases} . \tag{25}$$

Therefore, recovering the value $c_1 = m_0 k \bmod n$ for two different signatures allows an attacker to compute the private key.

Finding the interesting values inside the white-box may seem difficult without a reverse engineering step. However, the attack turns out to be easily automated on challenges that use the GMP library, such as Challenge 227. Indeed, one of the coefficients of s, say $c_1 = m_0 k$, will be inverted modulo n during the resolution of the system, and finding this inversion is easy when one can simply trace the calls to the function `mpz_invert()`. The team TheRealIdefix was thus able to efficiently apply this attack on Challenge 227 during the contest without any reverse engineering step.

This attack may seem very specific, but multiplying the nonce with a constant may appear as an easy way to protect the inversion step, and could very well be used by designers. This attack shows it is not a robust countermeasure.

Biased nonces. There exists a more generic way of breaking Challenge 227. Indeed, the way the ephemeral key is constructed (see Sect. 5.2.1) 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 a couple of hashes (e_0, e_i), with $e_0 = 0$ and $e_i = 2^i$:

$$\begin{cases} s_0 &= \left(\sum_{j=0}^{255} k_{j,0}\right)^{-1} (e_0 + r_0 d) \bmod n \\ s_i &= \left(k_{i,1} + \sum_{j=0, j\neq i}^{255} k_{j,0}\right)^{-1} (e_i + r_i d) \bmod n \end{cases} , \tag{26}$$

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

$$\begin{aligned} k_{i,1} + \sum_{j=0, j\neq i}^{255} k_{j,0} - \sum_{j=0}^{255} k_{j,0} &= s_i^{-1}(e_i + r_i d) - s_0^{-1}(e_0 + r_0 d) \bmod n \\ k_{i,1} - k_{i,0} &= s_i^{-1} e_i - s_0^{-1} e_0 + (s_i^{-1} r_i - s_0^{-1} r_0) d \bmod n \end{aligned} . \tag{27}$$

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

$$|k_{i,1} - k_{i,0}|_n = |s_i^{-1} e_i - s_0^{-1} e_0 + (s_i^{-1} r_i - s_0^{-1} r_0) d|_n < \left\lfloor \frac{n}{256} \right\rfloor , \tag{28}$$

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. (28) an instance of the Hidden Number Problem (HNP) [BV96]. Indeed, we are given many HNP inequalities of the form:

$$|\alpha t_i - u_i|_n < \left\lfloor \frac{n}{256} \right\rfloor , \tag{29}$$

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

Solving HNP instances in the context of ECDSA given inequalities such as Eq. (29) has been described numerous times in the literature. We refer the reader to [JSSS20] for a more detailed description[7]. 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. (28) (out of the 255 we can establish) to build a lattice whose reduction allows us to recover the private key d.

DCA. As explained in Sect. 4, we did not mount this side-channel attack during the contest. With the design of Challenge 227 in hand, we can see that it would have been unsuccessful, at least at the first order, thanks to the linear masking scheme used to protect all the implementation.

Fault injections. None of the faults injected on Challenge 227 during the contest were exploitable. This can be explained by the presence of the signature verification that is used to check if the guess for the overflow between E_1 and E_2 is correct. If a fault is induced, the signature is rejected and recomputed.

Of course, a reverse engineering step could be performed to get rid of this verification, but this would be quite time-consuming. Furthermore, even without this verification,

[7] We also highlight that the authors of [JSSS20] made their code available at https://github.com/crocs-muni/minerva.

the fault attack is still not trivial to perform because of the linear masking scheme. In particular, R_x is not manipulated directly in E_2. It is expressed as a linear combination of the input $A^{(t)}(R_x, R_y, k, e)$, so modifying one of the shares would probably also fault e, k or R_y, making the resulting faulty signature unexploitable.

5.5 Challenge 226: The Most Resistant

5.5.1 Description

Challenge 226 (`clever_kare`) was the second white-box implementation that we built following the method described in Section 5.2 and including all the additional countermeasures from Section 5.3. While this challenge stood for the longest (35 hours), Challenge 226 achieved the second-highest number of strawberries (11.19) 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. The impact of the additional countermeasures on the number of equations, the degree, and the number of variables can be seen by comparing this table with Table 3.

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	37×6	322×6	854×5	504×2

The size of the final C source code of Challenge 226 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 seconds and 17.27 MB, respectively. The code obfuscation did not significantly impact the performance of Challenge 226; the running time, the binary size, and the average RAM increased by less than 1%.

5.5.2 Security Analysis

During the WhibOx contest, the team theRealIdefix did not manage to break Challenge 226 with any of the automated attacks presented in Sect. 4.1. DCA and fault injection, which fail to break Challenge 227, are also not applicable to Challenge 226 since it is designed to be more secure. Moreover, the two attacks presented in Sect. 5.4.2 also fail to recover any secret information.

Hooking shared libraries. As mentioned in Sect. 5.3, Challenge 226 implements an additional countermeasure which consists in multiplying the components of the implicit round functions $\overline{T^{(i)}}$ with random polynomials in the input variables. Hence, the coefficients of s in the system $\overline{T^{(t+1)}}(\mathbf{u}; s, r)$ for the valid input $\mathbf{u}$ are no longer fixed multiples of k, and the attack cannot be mounted anymore.

Biased nonces. Likewise, the additional countermeasures implemented in Challenge 226 makes the lattice attack described in Sect. 5.4.2 fail. The additional variable k' alone would only reduce by 1 bit the bias observed in Eq. 29 and the attack would still be

practical. However, without the knowledge of the encoding L introduced in this challenge it is impossible to exhibit such a bias leading to key recovery.

As explained, none of the automated attacks that the TheRealIdefix team used during the contest were successful on Challenge 226. Nevertheless, with the design in hand, one could easily break this challenge. Indeed, knowing that the matrix M of the last affine encoding $A^{(t)}$ contains small entries, the attacker could, for example:

- Compute two signatures (r_1, s_1) and (r_2, s_2) for two messages e_1 and e_2 and extract the two valid $\overline{E^{(2)}}$ inputs $\mathbf{u}_1 = A^{(t)}(\mathbf{v}_1)$ and $\mathbf{u}_2 = A^{(t)}(\mathbf{v}_2)$ from the execution. Note that $\mathbf{v}_1 - \mathbf{v}_2$ contains the nonce difference $\kappa = k_1 - k_2$.
- Find κ by exhaustive search over M; for each guess M', obtain a candidate $\mathbf{v}_1 - \mathbf{v}_2 = (M')^{-1}(\mathbf{u}_1 - \mathbf{u}_2)$ and check if one of its entries κ satisfies $(\kappa G)_x = r_1 - r_2$.
- Solve the equation
$$s^{-1}(e_1 + r_1 d) - s^{-2}(e_2 + r_2 d) = \kappa \tag{30}$$
in d in order to obtain the secret key.

Therefore, this challenge can be easily broken once reverse-engineered. Nevertheless, such an attack is quite time-consuming, and resisting TheRealIdefix's automated attacks on ECDSA in the white-box contest is already an achievement considering that only 5 challenges resisted these attacks during the contest.

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, which broke the most 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 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.

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., with 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 the 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) under grant No. 11E1921N. 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

[AABM20] Estuardo Alpirez Bock, Alessandro Amadori, Chris Brzuska, and Wil Michiels. On the security goals of white-box cryptography. *IACR TCHES*, 2020(2):327–357, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8554.

[ABF+18] Christopher Ambrose, Joppe W Bos, Björn Fay, Marc Joye, Manfred Lochter, and Bruce Murray. Differential attacks on deterministic signatures. In *Cryptographers' Track at the RSA Conference*, pages 339–353. Springer, 2018.

[AMR19] Alessandro Amadori, Wil Michiels, and Peter Roelse. A DFA attack on white-box implementations of AES with external encodings. In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019*, volume 11959 of *LNCS*, pages 591–617. Springer, Heidelberg, August 2019.

[ANT+20] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. LadderLeak: Breaking ECDSA with less than one bit of nonce leakage. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20*, pages 225–242. ACM Press, November 2020.

[BCC21] Alberto Battistello, Laurent Castelnovi, and Thomas Chabrier. Enhanced Encodings for White-Box Designs. In Vincent Grosso and Thomas Pöppelmann, editors, *CARDIS 2021*, volume 13173 of *LNCS*, pages 254–274. Springer, 2021.

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

[BG03] Olivier Billet and Henri Gilbert. A traceable block cipher. In Chi-Sung Laih, editor, *ASIACRYPT 2003*, volume 2894 of *LNCS*, pages 331–346. Springer, Heidelberg, November / December 2003.

[BGEC04] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a white box AES implementation. In Helena Handschuh and Anwar Hasan, editors, *SAC 2004*, volume 3357 of *LNCS*, pages 227–240. Springer, Heidelberg, August 2004.

[BH19] Joachim Breitner and Nadia Heninger. Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies. In Ian Goldberg and Tyler Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 3–20. Springer, Heidelberg, February 2019.

[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, *CHES 2016*, volume 9813 of *LNCS*, pages 215–236. Springer, Heidelberg, August 2016.

[BMM00] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential fault attacks on elliptic curve cryptosystems. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 131–146. Springer, Heidelberg, August 2000.

[BV96] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 129–142. Springer, Heidelberg, August 1996.

[CEJv03] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-box cryptography and an AES implementation. In Kaisa Nyberg and Howard M. Heys, editors, *SAC 2002*, volume 2595 of *LNCS*, pages 250–270. Springer, Heidelberg, August 2003.

[CEJvO02] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. A white-box DES implementation for DRM applications. In Joan 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.

[CHE] CHES 2021 Challenge - WhibOx Contest. https://whibox.io/contests/2021/.

[Col] Christian Collberg. The Tigress C Diversifier/Obfuscator. https://tigress.wtf.

[DFLM18] Patrick Derbez, Pierre-Alain Fouque, Baptiste Lambin, and Brice Minaud. On recovering affine encodings in white-box implementations. *IACR TCHES*, 2018(3):121–149, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7271.

[DGH21] Emmanuelle Dottax, Christophe Giraud, and Agathe Houzelot. White-box ECDSA: challenges and existing solutions. In Shivam Bhasin and Fabrizio De Santis, editors, *COSADE 2021*, volume 12910 of *LNCS*, pages 184–201. Springer, 2021.

[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 Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 247–264. Springer, Heidelberg, August 2014.

[DRP13] Yoni De Mulder, Peter Roelse, and Bart Preneel. Cryptanalysis of the Xiao-Lai white-box AES implementation. In Lars R. Knudsen and Huapeng Wu, editors, *SAC 2012*, volume 7707 of *LNCS*, pages 34–49. Springer, Heidelberg, August 2013.

[DWP10] Yoni De Mulder, Brecht Wyseur, and Bart Preneel. Cryptanalysis of a perturbated white-box AES implementation. In Guang Gong and Kishan Chand Gupta, editors, *INDOCRYPT 2010*, volume 6498 of *LNCS*, pages 292–310. Springer, Heidelberg, December 2010.

[FGR13] Jean-Charles Faugère, Christopher Goyet, and Guénaël Renault. Attacking (EC)DSA given only an implicit hint. In Lars R. Knudsen and Huapeng Wu, editors, *SAC 2012*, volume 7707 of *LNCS*, pages 252–274. Springer, Heidelberg, August 2013.

[FIP13] FIPS PUB 186-4. *Digital Signature Standard*. National Institute of Standards and Technology, July 2013.

[FV12] Junfeng Fan and Ingrid 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.

[GK04] Christophe Giraud and Erik Woodward Knudsen. Fault attacks on signature schemes. In Huaxiong Wang, Josef Pieprzyk, and Vijay Varadharajan, editors, *ACISP 04*, volume 3108 of *LNCS*, pages 478–491. Springer, Heidelberg, July 2004.

[GMQ07] Louis Goubin, Jean-Michel Masereel, and Michaël Quisquater. Cryptanalysis of white box DES implementations. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *SAC 2007*, volume 4876 of *LNCS*, pages 278–295. Springer, Heidelberg, August 2007.

[GRW20] Louis Goubin, Matthieu Rivain, and Junwei Wang. Defeating state-of-the-art white-box countermeasures. *IACR TCHES*, 2020(3):454–482, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8597.

[Gt20] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.2.1 edition, 2020. http://gmplib.org/.

[JOR11] JORF n°0241. Avis relatif aux paramètres de courbes elliptiques définis par l'État français, October 2011.

[JSSS20] Jan Jancar, Vladimir Sedlacek, Petr Svenda, and Marek Sys. Minerva: The curse of ECDSA nonces. *IACR TCHES*, 2020(4):281–308, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8684.

[Kar11] Mohamed Karroumi. Protecting white-box AES with dual ciphers. In Kyung Hyune Rhee and DaeHun Nyang, editors, *ICISC 10*, volume 6829 of *LNCS*, pages 278–291. Springer, Heidelberg, December 2011.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

[Loc10] M. Lochter. RFC 5639: ECC Brainpool Standard Curves and Curve Generation, 2010. https://tools.ietf.org/pdf/rfc5639.pdf.

[LRD+14] Tancrède Lepoint, Matthieu Rivain, Yoni De Mulder, Peter Roelse, and Bart Preneel. Two attacks on a white-box AES implementation. In Tanja Lange, Kristin Lauter, and Petr Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 265–285. Springer, Heidelberg, August 2014.

[MGH09] Wil Michiels, Paul Gorissen, and Henk D. L. Hollmann. Cryptanalysis of a generic class of white-box implementations. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *SAC 2008*, volume 5381 of *LNCS*, pages 414–428. Springer, Heidelberg, August 2009.

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

[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.

[RVP22] Adrián Ranea, Joachim Vandersmissen, and Bart Preneel. Implicit white-box implementations: White-boxing ARX ciphers. In *CRYPTO*, Lecture Notes in Computer Science. Springer, 2022.

[RW19] Matthieu Rivain and Junwei Wang. Analysis and improvement of differential computation attacks against internally-encoded white-box implementations. *IACR TCHES*, 2019(2):225–255, 2019. https://tches.iacr.org/index.php/TCHES/article/view/7391.

[SEL21] Okan Seker, Thomas Eisenbarth, and Maciej Liskiewicz. A white-box masking scheme resisting computational and algebraic attacks. *IACR TCHES*, 2021(2):61–105, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8788.

[Sta10] Standards for Efficient Cryptography Group (SECG). *SEC 2 Ver 2.0 : Recommended Elliptic Curve Domain Parameters*. Certicom Research, January 2010.

[SWP09] Amitabh Saxena, Brecht Wyseur, and Bart Preneel. Towards security notions for white-box cryptography. In Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio Agostino Ardagna, editors, *ISC 2009*, volume 5735 of *LNCS*, pages 49–58. Springer, Heidelberg, September 2009.

[Van92] Scott Vanstone. Responses to NIST's Proposal. *Communications of the ACM*, 35:50–52, 1992.

[WMGP07] Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel. Cryptanalysis of white-box DES implementations with arbitrary external encodings. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *SAC 2007*, volume 4876 of *LNCS*, pages 264–277. Springer, Heidelberg, August 2007.

[XL09] Yaying Xiao and Xuejia 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. During the contest, we broke 92 out of 97 challenges. The 5 remaining challenges have been broken a posteriori.

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
101			✓		25D31D3AFF5773799ECF43DEC1882B8F05D9231697BDDA5482DE05B14FB8A63B
103		✓		✓	CC977E0748722D615B845C1B10EA554B69DFCA640440CA5C468BBEF84B8C0442
104		✓	✓	✓	638C9DFBF9F376CBB3E3B01DF27960EC53A689D2FF4DFF23D97EE5351ED4A3D0
105		✓	✓	✓	D29E9D130016D930BF830BCAD071BC6503F877FB207922A9E495CF71A79631FE

Challenge	Hooking	Collision	Fault	Lattice	Key
107		✓	✓	✓	4E420B6AA9E9F07F19CF7ED97497871C1223BC2A68E83716575C235DE6D63E17
108			✓		60609404F0B9086D3A995AF0680D048724CF2B1AF2B33CEA8DD4AF4B62A5DDBB
114	✓		✓	✓	0000000000000000000000000000000000000000000000000000000000000005
127			✓		1144D82B9568581405D10CF8B219FF7E94E4559E0832B06056F1F87D43C75777
135	✓	✓	✓	✓	0C2A5692FE1A7F9B8EE7EB4A7CD59CD62BCE33576B3123CECBB6406837BF51F5
136	✓		✓	✓	0C2A5692FE1A7F9B8EE7EB4A7CD59CD62BCE33476B3123CECBB6406837BF51F4
139	✓		✓	✓	00000000000000000000000000000004319055358E8617B0C46353D039CDAA9
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[8]					6F1D9093F3D5AE7C5F133659295914C9AF22E54B4ADE38CA421CA9BBD3D48A50
227	✓			✓	ADA6C6A1049825989811C9495D83681A68C67AB5E8EBDDC126CEE77056A7BB27
228				✓	EA7BA345EB9D99F54261D01AE6319B184769E5745621706D77018E0DB46DDAFA
231	✓	✓	✓	✓	8ADE24EE6413C6E408784DBB4D81D04F33238AB503CBE35C77400517EE5ABC96
235	✓	✓	✓	✓	000000000000000000000000000000000000000000000000D0FACADE0DEFACED
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
335			✓		EC90CC12DC70E3C5A7D47B6083A988F3F6C6B2B63EB0D8991F84B19E21ACC061
336[8]					D2E2AE325946DDADC9A67A2DFE8EE74065D8D39968707F5D818D7B62910894EA
345			✓		3266C9F6378DFDAE4AA763E9166B131E6514CA364C482306829D4BF1E97BFF92
346			✓		5EE43950837D0ABA419FE5B586D1A7AA44DDAAC6327DADC3133F18A850211B9F

[8]This challenge has not been broken by our automatic tools so we have used reverse engineering techniques.

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 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 $d_{114} = 5$
- Some designer teams modify a few bits only of the private key in several challenges (cf. Challenges 174, 185 and 187 for instance). In such a case, if one implementation is broken, then the private keys of the other challenges of the same team could be recovered by brute force search.
- Despite what is indicated in the rules (cf. Sect. 2), some challenges are not deterministic[9], 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.

[9] Challenges 54, 55, 57, 58, 61, 62, 66, 70, 71, 72, 73, 74, 76, 77, 78, 79, 80, 81, 84, 87, 89, 94, 96, 103, 104, 105, 107, 136 and 139 are not deterministic.