White-box filtering attacks breaking SEL masking: from exponential to polynomial time

Alex Charlès[1] and Aleksei Udovenko[2]

[1] DCS, University of Luxembourg, Esch-sur-Alzette, Luxembourg,
alex.charles@uni.lu
[2] SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg,
aleksei@affine.group

Abstract. This work proposes a new white-box attack technique called *filtering*, which can be combined with any other trace-based attack method. The idea is to filter the traces based on the value of an intermediate variable in the implementation, aiming to fix a share of a sensitive value and degrade the security of an involved masking scheme.
Coupled with LDA (filtered LDA, FLDA), it leads to an attack defeating the state-of-the-art SEL masking scheme (CHES 2021) of arbitrary degree and number of linear shares with quartic complexity in the window size. In comparison, the current best attacks have exponential complexities in the degree (higher degree decoding analysis, HDDA), in the number of linear shares (higher-order differential computation analysis, HODCA), or the window size (white-box learning parity with noise, WBLPN).
The attack exploits the key idea of the SEL scheme - an efficient *parallel* combination of the nonlinear and linear masking schemes. We conclude that a proper composition of masking schemes is essential for security.
In addition, we propose several optimizations for linear algebraic attacks: redundant node removal (RNR), optimized parity check matrix usage, and chosen-plaintext filtering (CPF), significantly improving the performance of security evaluation of white-box implementations.

Keywords: White-box Cryptography · Cryptanalysis · Filtering · Masking schemes · SEL · FLDA · RNR · CPF

1 Introduction

White-box cryptography is a discipline introduced by Chow *et al.* [CEJv02, CEJv03] in which it is supposed that the attacker has full access to the implementation of the cryptographic primitive. The security goals may vary (for example, incompressibility, one-wayness, traceability [DLPR14]), but in this paper, we focus on the secret key recovery of a protected implementation.

The key protection methods in this context can be categorized into two main categories: *structure-hiding* and *value-hiding* [BU18]. In the former, the designer will employ various obfuscation techniques to make the implementation resistant to circuit analysis and fault injection. The latter refers to protections against passive attacks that do not rely on the circuit structure. In this work, we focus solely on the value-hiding protections and the related attacks.

The paper of Bos, Hubain, Michiels, and Teuwen *et al.* [BHMT16] introduced the first automated passive attack in the white-box setting called *Differential Computational Analysis* (DCA), which broke existing encoding-based white-box implementations (for further analysis, see [ABMT18, RW19]). The classic Ishai-Sahai-Wagner's masking scheme

(ISW) [ISW03] can counter DCA. Still, it has been later shown in [GPRW20, BU18] that this masking scheme could also be broken in the white-box context with a linear algebraic attack called *Linear Decoding Analysis* (LDA).

Recently, Seker, Eisenbarth, and Liskiewicz [SEL21] proposed a nonlinear masking scheme (SEL), derived from the minimalist quadratic masking scheme (BU) by Biryukov and Udovenko [BU18]. Its goal is to resist DCA and LDA attacks and be more efficient than a combination of BU- and ISW-masking schemes. The SEL scheme can still be attacked by a higher order/degree version of the attacks ([BRVW19] and [GPRW20, BU21]). However, these attacks have an exponential cost with their degree/order, which should be chosen according to the number of linear shares and the degree[1] of the SEL masking scheme respectively. More recently, an attack based on the LPN problem [CU23] - learning parity with noise - was shown to be practical for high-degree instances of the SEL masking scheme. However, it has exponential complexity in the implementation size (more precisely, in the attacked window size, see Subsection 2.2).

In this work, we present *filtering*, a new methodology that can be coupled with any other trace-based attack. Filtering is a preprocessing step that can be applied before executing another attack. It has the effect of fixing an intermediate value in the implementation. For masking schemes, this leads to nullifying (at least) one of the monomials of the decoding polynomial. For instance, if the filtering methodology is used against the SEL masking scheme, it removes the decoding function's nonlinear monomial, making it linear and vulnerable to the LDA attack.

Our attacks do not contradict the security proof of the SEL scheme, since they are outside of the existing model of algebraic attacks. On the other hand, filtering particularly exploits the key idea of the SEL masking scheme: to use only a single nonlinear monomial and not split its variables linearly. It follows that both BU-masking and SEL-masking can not be used securely and must be combined with a linear masking scheme such as ISW. Unfortunately, such combinations do not have security proofs yet, and our results emphasize the need for their analysis. A more detailed discussion on countermeasures against FLDA is given in Section 7 and in Section 8.

Our contribution *(Filtering and Filtered LDA)* The main contribution of this paper is the *filtering* methodology , which consists of nullifying a *node* by choosing all the traces where it is equal to zero to reduce the security of the countermeasure. Most notably, applying it with the Linear Decoding Attack (LDA) [GPRW20] creates an attack that we call *Filtered Linear Decoding Analysis* (FLDA). We show both theoretically and in practice that it breaks the SEL masking scheme in polynomial-time complexity independently of its degree or number of linear shares, making it the best attack against this countermeasure regarding traces and time it requires. We recall that every previous attack is exponential in at least one of the parameters.

(Optimizations of algebraic attacks) In the first part of this section, we show that using the parity check matrix in the LDA attack gives an optimization reducing its time complexity from $O(\mathcal{W}^\omega + |\mathcal{K}|\mathcal{W}^2)$ to $O(\mathcal{W}^\omega + |\mathcal{K}|\mathcal{W})$, with ω the *matrix multiplication exponent*. In the second part, we observe that LDA and its derivatives only depend on the *span* of the vectors of a given window. Therefore, we propose an algorithm (redundant node removal, RNR) that removes all redundant nodes from traces (within a given distance), allowing us to perform sliding window algorithms through fewer and only relevant nodes. Finally, we propose a general technique based on chosen-plaintext traces (chosen-plaintext filtering, CPF), allowing a significant reduction of the cost of testing a large number of selection vectors. These techniques greatly boost the speed of the assessment process of white-box implementations.

[1]The authors of the SEL scheme define a general shape of the masking scheme for arbitrary degrees, but only provide concrete gadgets and security proofs for quadratic and cubic versions.

Table 1 compares the time and space complexities of the attacks from the literature with our improvements against ISW and SEL masking schemes. The time complexities are not given for a single window but rather for going through the traces over N nodes. After dealing with a window, we slide over $\mathcal{S}$ nodes, implying that we must go over $N/\mathcal{S}$ windows. Since $\mathcal{S} = \Theta(\mathcal{W})$, given a time complexity $O(\mathcal{C})$ for a single window, we thus display $O(\mathcal{C}N/\mathcal{W})$in Table 1.

Table 1: Summary of time and trace complexity of relevant trace-based white-box attacks over N nodes for a window of size $\mathcal{W}$; $\omega \approx 2.8$ is the matrix multiplication exponent; ℓ and d denote the numbers of linear and nonlinear shares respectively; $|\mathcal{K}|$ is the number of selection functions (≥ 4096 for the AES); g is the number of selection function groups (16 for the AES, see Section 6); τ is the fraction of the points of interest among all nodes (3 - 5% for CPF-LDA and 0.1 - 0.3% for CPF-FLDA); $c_d = (1 - 2^{-d})^{-1}$, $c'_d = \frac{-\ln(1-2^{-d})}{(1/2-2^{-d})^2}$.

Target	Attack	Reference	Time, $O(\cdot)$	Traces, $O(\cdot)$
ISW_ℓ	LDA	[BU18]	$N \cdot (\mathcal{W}^{\omega-1} + \lvert\mathcal{K}\rvert\mathcal{W})$	$\mathcal{W}$
	LDA	Subsection 5.1	$N \cdot (\mathcal{W}^{\omega-1} + \lvert\mathcal{K}\rvert)$	$\mathcal{W}$
	CPF-LDA	Algorithm 4	$N \cdot (\mathcal{W}^{\omega-1}g + \lvert\mathcal{K}\rvert\tau)$	$\mathcal{W}g$
$\text{SEL}_{\ell,d}$	HODCA	[BRVW19]	$N \cdot \lvert\mathcal{K}\rvert T_\ell \mathcal{W}^\ell$	T_ℓ
	HDDA	[GPRW20]	$N \cdot \lvert\mathcal{K}\rvert\mathcal{W}^{d\omega-1}$	$\mathcal{W}^d$
	HDDA	[BU18]	$N \cdot (\mathcal{W}^{d\omega-1} + \lvert\mathcal{K}\rvert\mathcal{W}^{2d-1})$	$\mathcal{W}^d$
	HDDA	Subsection 5.1	$N \cdot (\mathcal{W}^{d\omega-1} + \lvert\mathcal{K}\rvert\mathcal{W}^{d-1})$	$\mathcal{W}^d$
	WBLPN	[CU23]	$N \cdot \lvert\mathcal{K}\rvert\mathcal{W}^{\omega-2}c'_d c_d^{\mathcal{W}}$	$\mathcal{W}c'_d$
	FLDA	Subsection 4.2	$N \cdot (\mathcal{W}^\omega + \lvert\mathcal{K}\rvert\mathcal{W})$	$\mathcal{W}$
	CPF-FLDA	Algorithm 5	$N \cdot (\mathcal{W}^\omega g + \lvert\mathcal{K}\rvert\mathcal{W}\tau)$	$\mathcal{W}g$

(Other filtered attacks and discussion) We present the higher-order filtering, which fixes multiple values in the implementation at once. We then discuss the combination of filtering and higher-order filtering with other attacks and possible countermeasures.

2 Preliminaries

2.1 Notations

We denote by $\mathbb{F}_2$ the finite field of size 2 and by $\mathbb{F}_2^n$ the vector space of dimension n over $\mathbb{F}_2$. We denote by $\vec{0}$ the vector consisting of only zeroes, with the dimension given by the context. Similarly, we denote by $\vec{1}$ the vector composed of only ones. We let `len` denote the function returning the length of a list: $\texttt{len}(L) = n$.

Given a matrix M, we denote by $\ker(M)$ the (right) kernel of M, which is the vector space of all vectors $\vec{v}$ such that $M\vec{v} = \vec{0}$. A *parity check matrix* P is a basis matrix of the left kernel of M, i.e., it has the maximum number of linearly independent rows verifying $PM = 0$. Equivalently, $P\vec{v} = 0$ for every vector $\vec{v}$ from the columns span of M. We denote by ω the *matrix multiplication exponent*, $2 \leq \omega \leq 3$, such that operations on $C \times C$ matrices can be done in time $O(C^\omega)$. It varies depending on the algorithm employed, but known algorithms with smaller known exponents are impractical due to the hidden complexity constant. The common practical choice is the Strassen algorithm [Str69], which has $\omega = 2.8$.

We denote by $\binom{a}{\leq b} = \sum_{i=0}^{b} \binom{a}{i}$, $0 \leq b < a$, the sum of all binomial coefficients inferior or equal to b.

Proposition 1. *For any fixed d, it holds $O\left(\binom{n}{\leq d}\right) = O(n^d)$.*

2.2 Traces, Node Vectors and Sliding Window

White-box implementations come in different forms, typically as a compiled program. Internally, they are often based on arithmetic or Boolean circuits, since these allow masking countermeasures. The advantage of trace-based attacks is that they apply to any program. For compiled programs, the authors of [BHMT16] suggest recording traces based on the memory accesses (addresses and/or values). This works well if the program does not employ desynchronization techniques. Otherwise, the underlying circuit may be recovered by reverse-engineering efforts.

From now on, we consider a circuit-based implementation for the simplicity of exposition. We denote by N the number of nodes (Boolean intermediate values) in the implementation. To generate a *trace*, we encrypt a message and record the output values of all implementation nodes. If we generate multiple traces, the i^{th} bit contained in each of these traces will correspond to the same i^{th} *node* of the implementation. For T traces, we denote by the *node vector* $\in \mathbb{F}_2^T$ the vector consisting of all the values taken by the same *node* through all the encryptions.

Most available algorithms in the literature and this paper cannot process all of the *node vectors* at once. The common standard solution is to use a sliding window algorithm, that applies the algorithm separately on shorter consecutive parts of the traces, called *windows*. In practice, it is more effective to employ data-dependency analysis (DDA) [GRW20, TGCX23] and choose parts of the implementation based on the inter-dependencies of nodes in the structure. These methods are based on the analysis of the circuits, which puts it in the *structure hiding* [BU18] category, which is out of the scope of this paper. However, we emphasize that all of our attacks can be equally combined with the sliding window method or data-dependency techniques. The latter only dictates the *selection* of windows, while we focus on the method of *processing* the given windows.

2.3 Selection Functions and Selection Vectors

This paper will focus on attacking a white-box implementation of the AES block cipher [DR98, DR02], as it is the most often studied in the literature. Therefore, we will explain in this section a *selection function* for the AES. However, our results are not limited to this target and can be applied to others by choosing an appropriate *selection function*.

A *selection function* is a function of the plaintext and a part of the secret key. Matching a selection function with a part of the implementation suggests a candidate for the involved part of the secret key. For example, if we want to attack the first byte of the AES key of a white-box implementation, a usual selection function is the first output bit of the first S-box in the first round. Assume that we have generated T *traces* corresponding to T plaintexts. To attack the first byte of the key, for the i^{th} trace ($i \in \{1 \cdots T\}$), we will go through all of the 256 possible key byte values and compute the first output bit of the first S-box. After doing this for all of the T messages, we end up with 256 different *selection vectors* $\in \mathbb{F}_2^T$, each corresponding to a key guess. We denote the set of all considered selection functions by $\mathcal{K}$.

In a reference implementation, there must exist a *node vector* that corresponds exactly to the first bit of the output of the first S-box somewhere in the implementation. Therefore, there must exist a node vector in the implementation's traces that exactly matches the correct selection vector. With enough traces, matches with incorrect selection vectors can be excluded, leaving only the correct first key byte candidate. This process can be repeated for all of the 16 bytes of the key to recover the full master key. This attack is called *exact matching* [BU18]. For the AES, the number of selection functions is at least $16 \cdot 256 = 4096$. In practice, it is beneficial to consider more than one output bit of the S-box or even linear combinations of the output bits. This allows to catch alternative representations [Kar11] or flawed countermeasures, at the cost of increased analysis time.

The total number of such selection functions for the AES is $16 \cdot 256 \cdot 255 \approx 1$ million. Section 6 presents a general technique to deal with such large sets of selection vectors.

2.4 Masking Schemes, Shares and Decoding Function

To avoid an *exact matching* attack but especially other attacks to be presented in Section 3, the designer of a white-box implementation may apply Boolean masking schemes to provide *value hiding*. Applying a Boolean masking scheme transforms each different bit variable into *shares*. For instance, the BU masking scheme (presented more in detail in Subsection 3.2) transforms any bit variable a from the original implementation into three shares x_1, x_2 and x_3, such that $x_1 \oplus x_2 x_3 = a$.

Definition 1. *([CU23]):* A decoding function with n shares can be viewed as a polynomial P in $\mathbb{F}_2[x_1, \cdots, x_n]$ called a *decoding polynomial.* The *degree* $\deg P$ of the decoding function is the algebraic degree of the decoding polynomial. The *linear part* $\mathrm{lin}(P)$ of the decoding function is equal to the polynomial P with all monomials of degree more than 1 excluded.

3 Previous Works

3.1 DCA Attack and ISW Masking Scheme

Differential Computational Analysis (DCA) is the first automated white-box attack introduced in [BHMT16], coming from the side-channel *Differential Power analysis* [KJJ99]. This attack was introduced to break *value hiding* encoding-based methods, which were introduced in the original works by Chow, Eisen, Johnson and van Oorschot [CEJv02, CEJv03].

Instead of trying to find if one of the selection vectors matches one of the selection vectors *exactly*, the algorithm computes the absolute value of the correlation between each of the node vectors and each of the selection vectors and returns the highest value for each key byte.

In the side-channel ("grey-box") context, the main countermeasure against the DPA attack is the ISW masking scheme [ISW03], which naturally extends its application to the white-box setting. The ISW masking scheme replaces a sensitive bit variable s by n *shares* $x_1, \cdots, x_n$, with $x_1, x_2 \cdots x_{n-1}$ chosen uniformly at random, and x_n chosen such that $x_1 \oplus \cdots \oplus x_n = s$. Therefore, none of the *shares* taken individually correlate with s which could correspond to one of the selection vectors.

3.2 LDA Attack and BU Masking Scheme

Unlike the side-channel context, in white-box, the *traces* are generated without any noise, which led to the *Linear Decoding Attack* (LDA) [GPRW20]. If the ISW masking scheme with n shares has been employed in a white-box implementation, we know that there exist n node vectors $\vec{v_1}, \cdots, \vec{v_n}$ and a selection vector $\vec{s}$ satisfying $\vec{s} = \vec{v_1} \oplus \cdots \oplus \vec{v_n}$.

Consider a matrix M with columns being all the node vectors. Then, finding the node vectors corresponding to a given selection vector $\vec{s}$ is equivalent to solving the matrix equation $M\vec{x} = \vec{s}$ for $\vec{x}$. However, a white-box implementation may have a total number of nodes N that makes this linear system infeasible to solve in practice. Therefore, we have to use the sliding window approach to try to catch all of the *shares* of a sensitive bit variable s corresponding to one of the selection vectors. For each of the windows W, we then try to find a solution for $W\vec{x} = \vec{s}$, $\vec{s} \in \mathcal{K}$. If we find a solution, therefore the key byte guess corresponding to the selection vector might be the correct one.

For a window containing $\mathcal{W}$ node vectors, the LDA attack requires to have the number of traces $T > \mathcal{W}$. Indeed, if $T < \mathcal{W}$, there is a high chance of finding a false-positive for every window. In the cases where there is no solution in the window, we need to be sure that we do not find a false-positive. Assuming that the vectors contained in the window are generated uniformly at random, choosing $T = \mathcal{W} + 1$ leads to the probability $1/2$ of finding a false positive when solving $W\vec{x} = \vec{s}$. Therefore, choosing to add t supplementary traces reduces this probability to $\left(\frac{1}{2}\right)^t$. In practice, choosing $t = 30$ is sufficient.

Definition 2. For any linear algebra-based attack (*e.g.* LDA, HDDA, FLDA, WBLPN), we denote by t the number of supplementary traces ensuring the probability of finding a false positive (per window) being equal to $\left(\frac{1}{2}\right)^t$.

To avoid linear algebra-based attacks, Biryukov and Udovenko [BU18] introduced a new masking scheme called BU. This masking scheme is the first to propose a nonlinear decoding function, that is, a decoding function with the *degree* equal to at least 2 (*c.f.* Definition 1). This masking scheme transforms a sensitive bit variable s to $x_1 \oplus x_2x_3$. Therefore, to break it, we need to find three node vectors $\vec{v}_1, \vec{v}_2$ and $\vec{v}_3$ such that $\vec{v}_1 \oplus \vec{v}_2\vec{v}_3$ (coordinate-wise) is equal to one of the selection vectors. Since this operation is non-linear, we cannot find these vectors using the LDA attack. However, the DCA attack can reveal the correlation of $\vec{v}_1$ with one of the selection vectors [SEL21].

3.3 SEL Masking Scheme and HDDA, HODCA and WBLPN Attacks

To protect an implementation against both LDA and DCA, Seker *et al.* [SEL21] proposed a new masking scheme having a *linear part* (*c.f.* Subsection 2.4) containing multiple linear monomials, while having a degree 2 or more.

Definition 3. We denote by $\mathrm{SEL}_{\ell,d}$ the Boolean masking scheme with ℓ linear shares and degree $d \geq 2$, and with the *decoding polynomial* $P = x_1 \oplus \cdots \oplus x_\ell \oplus (x_{\ell+1}x_{\ell+2}\cdots x_{\ell+d})$ (*c.f.* Definition 1).

If we choose $\ell \geq 2$, then, as for the ISW masking scheme, the DCA attack will not work. Similarly, since $d \geq 2$, as for the BU masking scheme, the LDA attack will not work.

HODCA A variant of DCA has been studied in [BRVW19, GRW20, TGCX23], which can break SEL masking scheme. Using the sliding window method, before computing the correlation of the vectors, the HODCA of order $\mathcal{O}$ begins by increasing the window by adding the XOR of all combinations of $\mathcal{O}$ vectors.

Let us suppose that we want to attack $\mathrm{SEL}_{3,d}$ using HODCA of order $\mathcal{O} = 3$. If a window succeeds in catching all three *shares* of the *linear part* of the SEL masking scheme, HODCA will in particular try the correlation of the XOR of these three vectors, which will be high for one of the selection vector, thus recovering a part of the key.

For a window containing $\mathcal{W}$ elements, and for $T_\mathcal{O}$ the number of traces required to perform the attack with success depending on the order $\mathcal{O}$, with Proposition 1, we have a time complexity in $O(|\mathcal{K}|T_\mathcal{O}\binom{\mathcal{W}}{\mathcal{O}}) = O(|\mathcal{K}|T_\mathcal{O}\mathcal{W}^\mathcal{O})$ given in [BRVW19].

HDDA Similarly, HDDA of degree d is a variant of LDA that expands the window by appending all the AND combinations of d or less vectors to it. In this way, all degree-d monomials will be considered when performing LDA in the resulting window.

For instance, if we attack $\mathrm{SEL}_{\ell,3}$ with HDDA of degree $d = 3$, we may catch inside a window all shares of a sensitive variable. During the expansion of the window, we will in particular append the nonlinear monomial of the $\mathrm{SEL}_{\ell,3}$ decoding function. Then, the selection vector matching problem becomes linear and can be solved using LDA, leading to a key recovery.

For a window containing $\mathcal{W}$ elements, the complexity of HDDA of degree d has been first established in $O(|\mathcal{K}|\mathcal{W}^{d\omega})$ in [GPRW20], and can been reduced to $O(\binom{\mathcal{W}}{\leq d}^\omega + |\mathcal{K}|\binom{\mathcal{W}}{\leq d}^2) = O(\mathcal{W}^{d\omega} + |\mathcal{K}|\mathcal{W}^{2d})$ using LU-decomposition, as observed by [BU18]. We show in Section 5 that this complexity can be further improved to $O(\mathcal{W}^{d\omega} + |\mathcal{K}|\mathcal{W}^d)$ using parity check matrices.

Corollary 1. *$SEL_{\ell,d}$ is broken by HODCA of order greater or equal than ℓ and by HDDA of degree greater than or equal to d.*

However, HODCA (*resp.* HDDA) has a complexity growing exponentially with its order (*resp.* degree), being infeasible in practice for $\mathrm{SEL}_{\ell,d}$ with ℓ (*resp.* d) chosen high enough.

WBLPN Recently, Charlès and Udovenko [CU23] showed that considering the nonlinear monomial of SEL as a noise occurring with probability $(1/2)^d$, we can reduce the problem to an LPN instance to solve. This attack on $\mathrm{SEL}_{\ell,d}$ becomes more effective as d increases, which is the opposite of HDDA. However, its complexity remains exponential on the window size, which is problematic as even if having d large reduces the cost, it still implies that the SEL *gadgets* will be more complex, forcing to have a bigger window to catch all the *shares* of the *linear part.*

4 Filtered Linear Decoding Analysis

As presented in Subsection 3.3, the SEL masking scheme is broken by HDDA and HODCA which are exponential in time and trace number concerning the degree and the number of linear shares of SEL. Similarly, WBLPN [CU23] is faster than these two attacks if the degree of SEL is high enough, but it has exponential time complexity in the window size. In this section, we present the filtering preprocessing method that can be combined with any trace-based attack. Thereafter, we show that when combined with LDA it can break the SEL masking scheme in quartic time (in sliding window's size) independently from SEL's degree or number of linear shares.

4.1 The Filtering Methodology

Filtering can be described as a window preprocessing method, which can be combined with any window-based attack. Its broad goal is to deactivate a small amount of randomness in the implementation, which in turn should weaken the employed countermeasures. This is done by selecting a *node vector* (*c.f.* Subsection 2.2) in the window and *filtering* traces, keeping only the traces where the node value is equal to zero. The process is repeated for each node vector in the window as the filtering source. The algorithm allowing to perform a filtered version of any chosen attack $\mathcal{A}$ is depicted in Algorithm 1.

Example 1. Suppose an implementation is protected by the $SEL_{2,2}$ masking scheme that transforms every bit a of the implementation into shares (x_1, x_2, x_3, x_4) satisfying $a = x_1 \oplus x_2 \oplus x_3 x_4$. Then, by running a sliding window method with a window size $\mathcal{W} = 9$ and $T = 15$ traces, we will catch at some point the shares x_1, x_2, x_3, and x_4 corresponding to the *selection function* (*c.f.* Subsection 2.3) inside a window, among other unrelated nodes.

Even if we don't know which node vectors correspond to the shares of the secret variable, we can filter iteratively every node vector of the window as shown in Figure 1. Filtering a new vector corresponds to selecting a column of the window, and keeping only lines of the window in which the selected node vector is equal to zero. At some point, we will filter one of the shares corresponding to the non-linear part of the masking scheme, in our case x_3 or x_4.

$$
\begin{array}{ccccccccc}
 & x_1 & & x_2 & x_3 & & & x_4 & \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
\cancel{1} & \cancel{0} & \cancel{1} & \cancel{1} & \cancel{1} & \cancel{1} & \cancel{0} & \cancel{0} & \cancel{1} \\
0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
\cancel{1} & \cancel{0} & \cancel{0} & \cancel{0} & \cancel{1} & \cancel{1} & \cancel{1} & \cancel{1} & \cancel{1} \\
\cancel{1} & \cancel{0} & \cancel{1} & \cancel{1} & \cancel{1} & \cancel{0} & \cancel{1} & \cancel{1} & \cancel{0} \\
\cancel{1} & \cancel{1} & \cancel{1} & \cancel{1} & \cancel{1} & \cancel{0} & \cancel{0} & \cancel{0} & \cancel{0} \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
\cancel{0} & \cancel{1} & \cancel{0} & \cancel{0} & \cancel{1} & \cancel{1} & \cancel{1} & \cancel{0} & \cancel{0} \\
\cancel{0} & \cancel{1} & \cancel{1} & \cancel{1} & \cancel{1} & \cancel{1} & \cancel{1} & \cancel{1} & \cancel{1} \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\quad
\begin{array}{c}
x_1 \oplus x_2 \oplus x_3 x_4 \\
0 \\ \cancel{1} \\ 1 \\ 1 \\ 1 \\ 1 \\ \cancel{1} \\ \cancel{0} \\ \cancel{0} \\ 1 \\ 1 \\ \cancel{1} \\ \cancel{1} \\ 0 \\ 1
\end{array}
\longrightarrow
\begin{array}{ccccccccc}
 & x_1 & & x_2 & x_3 & & & x_4 & \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\quad
\begin{array}{c}
x_1 \oplus x_2 \oplus x_3 x_4 \\ = x_1 \oplus x_2 \\
0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1
\end{array}
$$

Figure 1: Filtering the fifth column (inside the rectangle) of a window corresponding to one of the non-linear shares of the decoding function of $SEL_{2,2}$, making the problem linear.

If we manage to filter one of the shares corresponding to the non-linear part (in our example x_3 or x_4) while having the shares of the linear part (x_1 and x_2) inside the window, we make the problem linear. In Figure 1, we managed to filter x_3 out, therefore the decoding function of $SEL_{2,2}$ becomes: $x_1 \oplus x_2 \oplus 0 \cdot x_4 = x_1 \oplus x_2$ and is now susceptible to Linear Decoding Analysis attacks (*c.f.* Algorithm 2). This procedure leads to our FLDA algorithm explained in the next Subsection 4.2.

Instead of filtering every node of a window, we can filter only the middle *node vector*, and reduce the sliding window step to 1 (to apply filtering to every node in the traces), improving the filtering algorithms' in-practice time. However, we consider that this optimization does not have an impact on the window complexity, as applying the algorithm to a unique window would require filtering all of its nodes.

Theorem 1. *Given a trace-based attack on a window of size $\mathcal{W}$ that has a time complexity in $O(\mathcal{C}_{\mathcal{W}})$, performing the filtered version of this attack has for time complexity $O(\mathcal{W}\mathcal{C}_{\mathcal{W}})$.*

Proof. Considering the worst-case scenario where one has to attack a single window, we cannot apply the optimization previously described. Therefore, we have to filter every $\mathcal{W}$ node of the window, which has a negligible cost, and apply the attack $\mathcal{W}$ time, for a total time complexity equal to $O(\mathcal{W}\mathcal{C}_{\mathcal{W}})$. □

Trace complexity To perform the filtered version of an attack $\mathcal{A}$ that requires T traces, we need to determine the total number of traces $F \geq T$ that will allow us to perform both the filtering and the attack. F also corresponds to the size of a node vector, and we want it to have at least T zeroes. If we suppose that the node vectors are uniformly distributed, then each of their elements is equal to zero with probability one-half, and the number of its zeroes follows a Binomial distribution. Therefore, if we denote the value of

Algorithm 1 Sliding window filtered version of an attack $\mathcal{A}$ (F$\mathcal{A}$)

Inputs:
- An attack $\mathcal{A}$ requiring T traces over N nodes
- A total number of traces F allowing to filter at least T traces (*c.f.* Theorem 2)
- An odd window size $\mathcal{W}$, and $\texttt{halfW} = \frac{\mathcal{W}-1}{2}$
- The list $\mathcal{K}$ containing the *selection vectors* $s_i \in \mathbb{F}_2^F$ (*c.f.* Subsection 2.3)

Output: The selection vectors corresponding to candidate key guesses, if any

```
1: for $n \in [\texttt{halfW}, \cdots, N - \texttt{halfW}]$ do
2:     window ← the list of the $(n - \texttt{halfW})^{th}$ to the $(n + \texttt{halfW})^{th}$ node vectors
3:     Idx ← the indexes where the $n^{th}$ node vector is equal to zero
4:     if len(Idx) $\geq T$ then
5:         $\widetilde{W}$ ← all rows of the $T \times \mathcal{W}$ window with indexes in Idx
6:         $\widetilde{\mathcal{S}}$ ← the list of selection vectors restricted to coordinates with indexes in Idx
7:         Perform the attack $\mathcal{A}$ on $\widetilde{W}$ and $\widetilde{\mathcal{S}}$
8:     end if
9: end for
```

the number of its zeroes by the variable X, we have:

$$P(X \geq T) = \frac{1}{2^F} \sum_{i=T}^{F} \binom{F}{i}$$

Theorem 2. *Given an attack that has a trace complexity in $O(\mathcal{C}_{\mathcal{W}})$, its filtered version has the same asymptotic complexity $O(2\mathcal{C}_{\mathcal{W}}) = O(\mathcal{C}_{\mathcal{W}})$.*

Proof. For an attack $\mathcal{A}$ requiring T traces, supposing that the *node vectors* takes their values in $\mathbb{F}_2$ uniformly at random, filtering would require to have on average a total number of traces for filtering $F = 2T$. With the Law of Large Numbers, the filtered version of an attack $\mathcal{A}$ of trace complexity $O(\mathcal{C}_{\mathcal{W}})$ has a total trace complexity $O(2\mathcal{C}_{\mathcal{W}}) = O(\mathcal{C}_{\mathcal{W}})$. □

Data dependency version of Filtered attacks Algorithm 1 shows how to perform the filtered version of an attack $\mathcal{A}$ with a sliding window algorithm. However, [GRW20, TGCX23] presented another method to skim through all the *nodes* of an implementation. Even though this method called *data dependency analysis* is out of the scope of this paper, the filtering method is compatible with it.

Indeed, the data dependency gives different windows to perform the attack $\mathcal{A}$ on. To achieve its filtered version, instead of filtering only one *node vector* of a window, we can iteratively filter each of the *node vectors* of a given window and perform the attack $\mathcal{A}$.

Filtering by one A simple countermeasure to filtering by zero is to change the gadgets to flip the filtering-sensitive bit variables. To thwart this potential countermeasure, an attacker can perform his attack first filtering by zero and then by one. This has no impact on the asymptotic time complexity as it only adds a factor 2 to it.

4.2 Filtering Linear Decoding Analysis against SEL masking scheme

As presented in Subsection 3.3, the SEL masking scheme has a *decoding polynomial* (*c.f.* Definition 1) $P = x_1 \oplus \cdots \oplus x_\ell \;\oplus\; x_{\ell+1} \cdots x_{\ell+d}$. Using a sliding window or a data dependency approach, let us consider that we managed to catch all the *node vectors* corresponding to the $\ell + d$ shares of the *decoding polynomial* inside of a single window.

Now, if we apply the filtering methodology before an LDA attack as shown in Example 1, we will at some point filter one of the d nodes corresponding to one of the shares of the non-linear monomial $x_{\ell+1} \cdots x_{\ell+d}$. Following Algorithm 1, we will select all the trace indexes where this *node vector* equals zero. For these traces, we then nullified the non-linear monomial, since we have: $P = x_1 \oplus \cdots \oplus x_\ell \ \oplus \ x_{\ell+1} \cdots 0 \cdots x_{\ell+d} = x_1 \oplus \cdots \oplus x_\ell$. For these traces only, the *decoding polynomial* of SEL becomes linear and therefore weak to the LDA attack. Applying this attack to this subset of traces recovers information on the key.

Complexity As discussed in Subsection 4.1, filtering increase the time complexity by a factor $\mathcal{W}$. Therefore, FLDA reaches a time complexity equal to $O(\mathcal{W}^{\omega+1} + |K|\mathcal{W}^2)$(*c.f.* Subsection 5.1), with ω the linear exponent (*c.f.* Section 2). Since LDA has a trace complexity $O(\mathcal{W} + t) = O(\mathcal{W})$ (*c.f.* Definition 2), FLDA has a trace complexity in $O(2\mathcal{W}) = O(\mathcal{W})$.

Time complexity comparison of FLDA with HDDA and HODCA HDDA and HODCA time complexities depend on the parameters d and ℓ of $\text{SEL}_{\ell,d}$ (*c.f.* Definition 3), whereas FLDA complexity is independent of it.

Moreover, even in the best situation for HDDA, when $d = 2$, its complexity reaches $O(\mathcal{W}^{2\omega} + |K|\mathcal{W}^2)$ whereas FLDA remains at $O(\mathcal{W}^{\omega+1} + |K|\mathcal{W}^2)$ for any ℓ and d, following Theorem 1. We will show in the next paragraph that FLDA is also outperforming HDDA against the SEL masking scheme for any degree d and window size in practice.

In [BRVW19], the authors estimated the time complexity of HODCA to be equal to $O(|\mathcal{K}|T_{\mathcal{O}}\binom{\mathcal{W}}{\mathcal{O}}) = O(|\mathcal{K}|T_{\mathcal{O}}\mathcal{W}^{\mathcal{O}})$ (*c.f.* Proposition 1), with $T_{\mathcal{O}}$ the number of traces required to succeed differentiating the correlation between the correct *selection vector* and the shares of the *decoding function* from the correlations between the other *node vectors* and *selection vectors* (*c.f.* Section 3). $T_{\mathcal{O}}$ grows extremely fast with the window size and the degree which slows down the attack drastically (*c.f.* [BRVW19], Table 1, with $\lambda = \mathcal{W}$ and $d = \mathcal{O}$). Even if $T_{\mathcal{O}}$ was equal to one, the time complexity of HODCA would remain exponential with its order $\mathcal{O}$ that should match the number of linear shares ℓ of SEL, while FLDA remains independent of these parameters.

Practical results and comparison of the number of binary operations The following computations were done on an AMD EPYC 3.2 GHz CPU with 1 TiB of RAM running Ubuntu 20.04 with SageMath 10.2 using Python 3.11.7. However, these experiments can be reproduced on a common laptop.

We ran FLDA[2] over the first 20% of the traces of SEL[3], BU[4] and ISW[4] masking schemes applied over a ten-round AES (respectively [SEL21], [BU18] and [ISW03]), since the key parts are located in the beginning of the traces. Given a window size $\mathcal{W}$, we ran FLDA over $2(\mathcal{W} + t) + 50$ traces, for $t = 30$ (*c.f.* Definition 2), which allowed us to filter out enough traces to perform LDA with very high probability.

Table 2 shows that FLDA can break all SEL implementations, and by extension, ISW and BU masking schemes with a time remaining low even for big window sizes. Compared to its counterpart HDDA, observing its time measurements from [CU23] (on comparable software and hardware), to break the quadratic $\text{SEL}_{5,2}$ with window size $\mathcal{W} = 50$, HDDA of degree 2 would take 6.5s per window with their implementation, compared to an average of $\frac{7659\text{s}}{240\text{k nodes}} = 33.13\text{ms}$ per window for FLDA. To break the cubic $\text{SEL}_{5,3}$, since FLDA is independent of SEL's degree, the time per window will not change, while HDDA needs to be of degree 3, which has a time per window already exceeding 2 minutes for window size $\mathcal{W} = 34$.

We can observe that SEL with a decoding polynomial of degree 3 ($\text{SEL}_{5,3}$ in Table 2, with 1161k nodes in total) has fewer nodes than its degree 2 version ($\text{SEL}_{5,2}$ with 1204k nodes in total), which is unexpected, considering that the gadget of the former is more complex than the latter[SEL21]. We investigated the official SEL implementation[3] that we used to generate our traces and found that the Refresh gadget is never recorded in

[2]https://github.com/cryptolu/whitebox-filtering
[3]https://github.com/UzL-ITS/white-box-masking
[4]https://github.com/hellman/ches2022wbc

Table 2: Time and number of bytes of the key recovered (✓ if the key is fully recovered) with FLDA over different countermeasures applied to the first 20% of the traces of ten-round AES implementations, with $2(\mathcal{W}+30)+50$ traces, using a sliding window approach and sliding by one node at a time.

		$\mathcal{W}=50$		$\mathcal{W}=100$		$\mathcal{W}=200$		$\mathcal{W}=400$	
Target	$20\%N$	time	Key	time	Key	time	Key	time	Key
ISW_3	39k	20min	5B	25min	14B	32min	✓	49min	✓
BU	168k	1h29	✓	1h45	✓	2h16	✓	3h32	✓
$\text{SEL}_{5,2}$	240k	2h08	✓	2h29	✓	3h18	✓	5h06	✓
$\text{SEL}_{5,3}$	232k	2h03	✓	2h25	✓	3h11	✓	4h57	✓

the traces, and the And gadget is only partially recorded for degree 3. Since the Refresh gadget is called within the other gadgets, the traces should be drastically heavier. This would make our attack slower especially for degree 3, since it would have to go through more nodes, while not impacting the time cost per window.

Trace complexity comparison of FLDA with HDDA and HODCA Given Theorem 2, FLDA has the same trace complexity as LDA in $O(\mathcal{W}+t)=O(\mathcal{W})$ (*c.f.* Definition 2), independently of the parameters of SEL ℓ and d. Whereas HDDA of degree d has a trace complexity in $O\left(\binom{\mathcal{W}}{\le d}+t\right)=O(\mathcal{W}^d)$ (*c.f.* Proposition 1). In [BRVW19], Table 1, with $\lambda=\mathcal{W}$ and $d=\mathcal{O}$, the authors shown that HODCA of order $\mathcal{O}$ requires drastically more traces than FLDA (*c.f.* Table 3).

Table 3: Number of traces F for FLDA with LDA on $\mathcal{W}+t$ traces for a probability of filtering enough traces equal to 0.99, using the trace formula from Subsection 4.1.

$\mathcal{W}+t$	10	20	40	80	160	320	640	1280
F	33	57	103	192	364	701	1365	2680
$\frac{F}{\mathcal{W}+t}$	3.3	2.85	2.575	2.4	2.2750	2.1906	2.1328	2.0938

Other Filtered attacks The filtering methodology can be applied to any trace-based attacks (FDCA, FHODCA, FHDDA, and FWBLPN). We will discuss them and also their Higher-Order filtered versions against combined countermeasures in Section 7.

5 Optimizations of linear algebra-based algorithms

In this section, we propose an optimization of LDA using a parity check matrix, a new algorithm called RNR, and a general optimization of processing a large number of selection functions. These improvements significantly enhance the performances of LDA, HDDA, FLDA, and WBLPN.

5.1 Optimization of LDA

As presented in [GPRW20], given a window W containing $\mathcal{W}$ elements, we try to solve $W\vec{x}=\vec{s}_i$ for $s_i\in\mathcal{K}$, which has complexity $O(|\mathcal{K}|\mathcal{W}^\omega)$ (*c.f.* Subsection 2.1), with ω the *linear complexity exponent*. The authors of [BU18] then showed how to reduce this complexity to $O(\mathcal{W}^\omega+|\mathcal{K}|\mathcal{W}^2)$ by using the LU-decomposition of a matrix. We will now show that we can reduce the complexity to $O(\mathcal{W}^\omega+|K|\mathcal{W})$.

We propose to compute once the parity check matrix `PCM` of W (*c.f.* Subsection 2.1), which, once multiplied by a vector $\vec{v}$, has the property to make the result equal to zero if and only if $\vec{v}$ is in the column space of W. Indeed, the equation $W\vec{x}=\vec{s}$ is solvable if and only if $\texttt{PCM} \cdot \vec{s} = \texttt{PCM} \cdot W \cdot \vec{x} = 0$. Therefore, once we have computed the parity check matrix of the window, we multiply it by all of the *selection vectors*. If the result is equal to zero, then the guess of the key part associated to this vector should be correct.

Since we verify that $\texttt{PCM}\,\vec{v} = \vec{0}$, we can stop the verification right when we find a single one in the resulting vector. For a random vector multiplied to the PCM, we will in average have to verify the multiplication of only two $(1+1/2+1/4+...)$ rows of the PCM by $\vec{v}$ before determining that $\vec{v}$ is not in the column space of W, which is depicted in Algorithm 2.

Algorithm 2 Revisited Linear Decoding Analysis (LDA)

Inputs:
- A window W containing $\mathcal{W}$ node vectors $\in \mathbb{F}_2^{\mathcal{W}+t}$
- A list $\mathcal{K}$ of selection vectors $\in \mathbb{F}_2^{\mathcal{W}+t}$

Output: The selection vectors corresponding to candidate key guesses, if any

```
1:  PCM ← the parity check matrix of $W$
2:  for $\vec{v}$ in $\mathcal{K}$ do
3:      for $i$ in $\{1 \cdots \mathcal{W}+t\}$ do
4:          if the multiplication of the $i^{th}$ row of PCM and $\vec{v}$ equals one then
5:              break
6:          end if
7:      end for
8:      if all the multiplications were equal to zero then
9:          Append $\vec{v}$ to the output
10:     end if
11: end for
```

Complexity comparison For a window with $\mathcal{W}$ *node vectors* over $T = \mathcal{W}+t$ traces (*c.f.* Definition 2), since computing the parity check matrix is a linear algebra operation, it costs $O(\mathcal{W}^{\omega})$ and creates a matrix with $T = \mathcal{W}+t$ columns. We then perform on average 2 multiplications of vectors of size $\mathcal{W}+t$ for each of the $|\mathcal{K}|$ key guesses, which represents $O(2|\mathcal{K}|(\mathcal{W}+t)) = O(|\mathcal{K}|\mathcal{W})$. We conclude that the complexity of LDA after our optimization is $O(\mathcal{W}^{\omega} + |K|\mathcal{W})$.

New complexity for HDDA The HDDA of degree $d \geq 2$ attacks begin by computing and adding to the window all the $\binom{\mathcal{W}}{\leq d}$ vectors resulting from the AND operation of all combinations of $\leq d$ *node vectors* of the original window. This cost is negligible compared to computing the LDA attack to the newly increased window.

In the previous paragraph, we showed that LDA applied to a window size $\mathcal{W}$ has complexity $O(\mathcal{W}^{\omega}+|K|\mathcal{W})$ with our optimization. Therefore, in the case of HDDA of a fixed degree d, we conclude that it dropped the cost of the attack from $O(\binom{\mathcal{W}}{\leq d}^{\omega} + |\mathcal{K}|\binom{\mathcal{W}}{\leq d}^{2}) = O(\mathcal{W}^{d\omega} + |\mathcal{K}|\mathcal{W}^{2d})$ to $O(\binom{\mathcal{W}}{\leq d}^{\omega} + |K|\binom{\mathcal{W}}{\leq d}) = O(\mathcal{W}^{d\omega} + |K|\mathcal{W}^{d})$.

However, this optimization does not change the trace complexity of both attacks, therefore LDA keeps its trace complexity in $O(\mathcal{W})$ and HDDA in $O(\mathcal{W}^{d})$.

5.2 Redundant Nodes Removal (RNR)

Linear algebra-based attacks (LDA, HDDA, FLDA, and WBLPN) are trying to determine if any of the linear combinations of the node vectors inside of a window corresponds to one of the $|\mathcal{K}|$ selection vectors. This is equivalent to verifying if one of the selection vectors is in the column space of the matrix corresponding to the window. To perform this verification, it is enough to use only the vectors from a *basis* of the matrix. Therefore, to decrease the window size, we can remove the vectors that are not part of the basis.

Definition 4. Given a list of node vectors $\in \mathbb{F}_2^T$ over T traces and a subset of it spanning the same vector space, we say that all node vectors that are not in this basis are *redundant*.

Redundancy is common among the node vectors since many of them correspond to the output of a XOR gate in the implementation. Indeed, given a XOR gate of an implementation computing $a \oplus b = c$, there will be three node vectors $\vec{v}_a, \vec{v}_b$ and $\vec{v}_c$ in the traces such that $\vec{v}_a \oplus \vec{v}_b = \vec{v}_c$, and therefore one of these three vectors is redundant.

We would like to remove all of these vectors, so that linear algebra-based attacks would only run on relevant vectors, increasing the effectiveness of the attack. In Subsection 5.2, we show that for the official SEL implementation and BU implementation, we can remove more than 50% of the nodes. Therefore, running an attack onto only the non-redundant vectors of these implementations will be twice as fast since there are only half of the elements to go through. Furthermore, if the attack is using a window size $\mathcal{W}$, then this window will be virtually twice as big, as half of its elements would have been irrelevant for the attack if we had kept the redundant node vectors.

Ideally, to remove these redundant vectors we would like to consider all node vectors as a big matrix and keep only vectors that span the vector space, but this would be too heavy to perform. Fortunately, the nodes interacting with each other are located relatively close in the traces, and therefore we can use a sliding window method to remove a major part of the redundant node vectors. Therefore, the RNR algorithm is similar to the LDA attack, but instead of observing if one of the selection vectors is in the vector space of a given window, we simply remove all the redundant nodes of the window before sliding, which is depicted in Algorithm 3.

Algorithm 3 Redundant Node Removal (RNR)

Inputs: A list L containing N node vectors, a window size $\mathcal{W}$ and a sliding size $\mathcal{S}$
Output: The list NRN of Non-Redundant Nodes

```
1: NRN ← $[0 \cdots N]$, stored in reverse order to prevent sliding the whole list at each removal
2: while $i \in \{0 \cdots N - \mathcal{W}\}$ do
3:     window ← the list of the $i^{th}$ to the $(i + \mathcal{W})^{th}$ node vectors from $L$
4:     Idx ← the indexes of the node vectors of the window that form a basis
5:     Remove all indexes that are in NRN and not in Idx
6:     $i \leftarrow i + \mathcal{S}$
7: end while
8: return NRN
```

Time and trace complexity Finding the *node vectors* that are linearly independent and which span the space of a matrix $\mathcal{W} \times (\mathcal{W} + t)$ is linear and therefore RNR has a time complexity in $O(\mathcal{W}^{\omega})$, with ω being dependent on the algorithm used (*c.f.* Section 2). To avoid removing nodes that are non-redundant, we should have t supplementary traces (*c.f.* Definition 2). Therefore, the trace complexity of RNR is $O(\mathcal{W} + t) = O(\mathcal{W})$.

In-practice RNR Result We ran our RNR algorithm[5] on the traces generated by the countermeasures SEL[6], BU[7], Dummy Shuffling[7] (three slots) and ISW[7] (three linear shares) applied over a ten-round AES (respectively [SEL21], [BU18], [BU21] and [ISW03]). As explained in Section 4, the official implementation of SEL is incorrect, explaining why $\mathrm{SEL}_{5,3}$ has fewer nodes than $\mathrm{SEL}_{5,2}$.

The computations of Table 4 were done on a 12th Gen Intel(R) Core i7-1265U 1.80 GHz CPU, with 32 GB of RAM on WSL 2 running Ubuntu 22.04 on Windows 10, with SageMath 10.2 [Sag23] using Python 3.11.7. The RNR implementation[5] was run with different window sizes, with $t = 30$ (*c.f.* Definition 2) and a sliding size $\mathcal{S} = \frac{\mathcal{W}}{6}$. The time is given in seconds and takes into account the time required to first open the node vectors.

[5]https://github.com/cryptolu/whitebox-filtering
[6]https://github.com/UzL-ITS/white-box-masking
[7]https://github.com/hellman/ches2022wbc

Table 4: Time in seconds and percentage of node removed for RNR algorithm ran with different window sizes and different algorithms

		$\mathcal{W} = 100$		$\mathcal{W} = 200$		$\mathcal{W} = 300$		$\mathcal{W} = 400$		$\mathcal{W} = 500$	
Target	N	time	del	time	del	time	del	time	del	time	del
Dum_3	164k	12s	45%	13s	54%	22s	55%	25s	57%	32s	59%
ISW_3	194k	13s	50%	18s	57%	24s	60%	30s	61%	36s	61%
BU	837k	57s	41%	83s	45%	111s	48%	138s	50%	176s	51%
$\text{SEL}_{5,2}$	1204k	75s	43%	126s	47%	161s	47%	196s	47%	240s	47%
$\text{SEL}_{5,3}$	1161k	70s	48%	108s	48%	142s	48%	182s	48%	225s	48%

Compatibility of RNR and correlation attacks Although it is clear that RNR enhances LDA, HDDA, and WBLPN attacks, it is not clear what impact it has on the other attacks. For instance, let us suppose that this implementation is secured with the BU masking scheme such that each bit s is replaced by three shares such that $s = a \oplus bc$. We know that the node vector corresponding to the share a is correlating to one of the selection vectors, therefore we would like to apply DCA (*c.f.* Subsection 2.3 and Subsection 3.1).

However, let us suppose that we have there *node* x_1, x_2 and x_3 in this implementation such that $x_1 \oplus x_2 = x_3$. If we apply RNR to these three nodes, one of them is redundant and will be removed. Therefore, we may be removing the node corresponding to the share a of the BU scheme, avoiding the DCA attack from working.

Compatibility of RNR and filtered attacks The same problem can be observed with filtered attacks as removing one of the nodes that corresponds to one of the shares that we would like to filter. Therefore, we do not recommend applying filtered attacks only on non-redundant nodes.

However, for FLDA, FHDDA, and Higher-Order FLDA (*c.f.* Section 7) it is possible to filter all the nodes while applying the subsequent attack only on non-redundant nodes. This way, we do not benefit from the aspect of RNR that allows us to go through only a subset of the nodes, but we benefit from the virtually increased window size.

6 Chosen-Plaintext Filtering (CPF)

Trace-based white-box attacks descended from gray-box attacks, which are typically *known-plaintext* attacks in their nature due to this setting being the most realistic in practice. In the white-box setting, however, there is no reason to follow this constraint. To this end, we present a general optimization technique exploiting the ability to record *chosen-plaintext* traces[8] for handling large sets of selection vectors efficiently. This is particularly important due to the possibility that a white-box designer uses alternative representations or linear encodings [Kar11, ABMT18], which can be attacked by considering all relevant linear combinations of basic single-output-bit selection functions (in the case of 1-round AES, $16 \times 256 \times 255 \approx 10^6$ selection functions).

The main idea of our optimization is to apply the *divide-and-conquer* principle to the selection function detection in a white-box implementation. The goal is to separate the location of *points of interest* from the recognition of the actual sensitive function. We achieve this by the means of *chosen plaintext filtering* (CPF):

1. (Location of points of interest) The first step is to split selection functions into groups and, for each group, to sample plaintexts for which all the selection functions in the

[8]We remark that chosen-plaintext attacks in the white-box setting were already considered in a few previous works [BU21, TGLZ23].

group are *constant*. In the case of the usual 1$^{\text{st}}$ round S-box targets in the AES, a group may correspond to plaintexts having one byte fixed to a constant. Indeed, all selection functions corresponding to that S-box would be constant (equal to 0 or 1) on all such plaintexts. Therefore, the set of distinct selection vectors (for that byte) consists of just the all-zero and the all-one vectors. Matching these selection vectors in the implementation would not immediately leak the secret key, but instead pinpoint potential points of interest for the second step. In the AES above, these include intermediate variables which are functions only of the chosen plaintext byte (i.e., are independent of the other input bytes), but also, more importantly, (possibly filtered) linear combinations yielding such functions.

2. (Selection function recognition) Once the points of interest have been located, the usual attack techniques can be applied to distinguish the chosen plaintext on function inside the group. This may include a direct comparison of each trace point with each selection function (including correlation or exact matching), or algebraic matching techniques (*c.f.* Subsection 5.1), possibly filtered as in the main FLDA (*c.f.* Section 4).

The grouping in the first phase can be rather broad, and thus yield many points of interest. For example, all intermediate variables in the circuit of one AES S-box in the first round (which make up for about 1/160=0.63% of a reference AES implementation) satisfy the detection criteria (being functions of the fixed input byte), but at the same time are difficult to exploit for key recovery. However, it is expected that the resulting number of interest points is much smaller than the original trace size. Therefore, the first phase should be viewed as a *trace reduction* optimization. In addition, this technique provides insights into the structure of the studied implementation, which can be useful for a human reverse engineer.

Remark 1. Although serving a similar purpose, the data-dependency analysis (DDA) methods [GRW20, TGCX23] are independent of chosen plaintext filtering and can be combined with it. Roughly speaking, CPF determines the *global* positions of interesting windows, while DDA determines the *local* structure of the windows.

Remark 2. CPF can be also "simulated" in the usual *known-plaintext* setting: from $256T$ random plaintexts we can always select at least T traces with an input byte fixed to a constant, for any input byte, by the pigeonhole principle. For $T = 512$ this requires 130 000 traces, which may take some time and storage, but should be feasible in most cases.

Importance of preprocessing As outlined above, the first phase is essentially searching for constant selection vectors inside the traces recorded for a specific set of plaintexts. For raw traces, this may yield many irrelevant points of interest, such as computed constants, negated variables, all linear gates, etc. Therefore, it is crucial to preprocess traces to remove this material. This can be done by preprocessing an additionally recorded set of traces for purely random plaintexts, recording the eliminated redundancies, and eliminating them from the group traces. For this purpose, in the case of algebraic CPF attacks, the previously developed RNR technique from Subsection 5.2 will play a crucial role. More specific details are described below on the examples of chosen-plaintext-filtered LDA and FLDA workflows.

Optimizing plaintext grouping for reducing the number of traces As currently described, the optimization requires recording T traces *for each group*, where T is the number of traces required to detect the constant selection function using the main attack (such as LDA or FLDA in our case). In the case of AES described above, the attacker needs 16 distinct sets of traces, one per plaintext byte being fixed (and an extra set for random plaintexts). However, it is possible to significantly reduce the required number of traces. The idea is to combine several groups into a few very large groups, sample large pools of plaintexts for these large groups, and extend them with a few plaintexts targeting smaller groups. The last step is needed to differentiate the smaller groups from the larger group.

Example 2. In the case of AES, the attacker can record $T - \epsilon$ traces for plaintexts with the first 8 bytes fixed to constants (and the other 8 bytes varying randomly), and $T - \epsilon$ traces for plaintexts with the last 8 bytes fixed to constants (and the other 8 bytes varying randomly). Then, for each plaintext byte, an extra ϵ traces are recorded with this byte fixed to the constant value which this byte was set to in one of the large sets. From all these traces, for each plaintext byte, it is possible to choose at least T traces for which the byte is constant (has the same value), while the other bytes are varying. As a result, the trace complexity is reduced from $16T$ to $2T + 14\epsilon$. How large should be the ϵ parameter? Since its goal is to distinguish the fixed input byte from the other bytes in the same half, the selection functions associated with the non-fixed bytes must be non-constant in the current trace set. Assuming that the sensitive functions are balanced, it is sufficient to choose, say, $\epsilon = 64$ to filter out irrelevant selection functions with probability $1 - 2^{-\epsilon} = 1 - 2^{-64}$ each. For $T = 512$, this leads to a factor of 4 improvement in the trace complexity and approaches the factor of 8 as T tends to infinity.

Remark 3. The grouping of selection functions can be nested to reduce the cost of iterating over each group of selection functions. This is similar to the optimization described above but applies to the time complexity rather than to the trace complexity. For example, in the case of AES, we have 16 selection function groups, one per each plaintext byte. This adds a factor of 16 to the cost of the first step. To reduce this factor, we can first consider only two groups (one per each half of the plaintext). This would yield more false-positive points of interest, but still only a small fraction of the full trace. Then, these points of interest could be further sieved by using more fine-grained selection groups. This would have negligible cost since the algorithm's input would be of much smaller size than the original trace. However, for simplicity, we do not consider this optimization further.

6.1 Chosen-plaintext-filtered LDA

The most natural application of the CPF method is to the LDA attack. The procedure is summarised below and in Algorithm 4:

Step 1: Record the required pool of traces, including a sufficient amount of plaintexts for each group of selection vectors and random plaintexts. If possible, this step should use the optimization described above and in Example 2.

Step 2: Perform RNR on the general trace set (random plaintexts) and eliminate the corresponding redundant node vectors *from all traces*.

Step 3: For each selection function group:

(a) Perform RNR on the corresponding set of reduced traces (for which all the selection functions in the group are constant) and record the *kernel* information (i.e., which linear combinations of node vectors sum to a constant).

(b) Apply the kernel map to the general trace set, producing a new small trace set.

(c) Apply the usual LDA attack on the final compact trace set, using selection functions from the group as target candidates.

The crucial step in the procedure is 3.a: it locates linear combinations of nodes that are constant when inside the chosen selection vector group. It is expected to significantly reduce the trace size. The exact reduction factor depends strongly on the underlying cipher, countermeasures, and design strategy.

Often in practice the RNR step in 3.a produces at most 1 linear combination per most windows, or just a few of them. In these cases, it does not make sense to pass these vectors through full LDA procedure and parity check matrix computation but rather to do an exact matching with selection vectors (which can be done in $O(\mathcal{W})$ bit operations and memory lookups using a precomputed hash table). If there is more than 1 vector in the window, it may still be cheaper to enumerate all their linear combinations exhaustively and perform an exact matching for each of the combinations. Indeed, an LDA matching requires at least 32 traces to ensure a low false-positive ratio, which leads to a complexity of about $32^3 = 2^{15}$ bit operations, and all linear combinations of 10 32-bit vectors can be enumerated with the same complexity. We emphasize that using exact matching in this step instead of LDA does not decrease the quality of the attack: the actual linear matching part has already happened in the RNR in step 3.a.

Proposition 2. *The procedure described above (and in Algorithm 4) produces equivalent results as the LDA attack applied to the full set of traces.*

Proof. It is easy to verify that a selection function that is non-constant on the full set of traces, but which is recoverable using the LDA attack, must be constant on the fixed-group set of traces and thus will be caught by step 3.a, and recognized by step 3.c. Additional false positives may happen due to a virtual increase in the window size due to shrinkage of the trace. These can be avoided (if needed) by tracking the indexes of the linear combinations and dropping matches with distances bigger than the window size. □

Algorithm 4 Chosen-plaintext-filtered LDA (CPF-LDA)

Inputs:
- LDA parameters (window) (*c.f.* Algorithm 2)
- Groups $[(P_1, S_1), \ldots, (P_g, S_g)]$, P_i a set of plaintexts and S_i a set of selection functions constant on P_i

Output: Selection vectors corresponding to candidate key guesses, if any

```
1: $T_\$ \leftarrow$ a set of traces on random plaintexts
2: $B \leftarrow$ indexes of non-redundant nodes from RNR($T_\$$) (c.f. Subsection 5.2)
3: for $i \in \{1, \ldots, g\}$ do
4:     $T_i \leftarrow$ traces on plaintexts from $P_i$
5:     $K_i \leftarrow$ kernel information from RNR($T_i\ |_B$) (linear relations), where $T_i\ |_B$ denotes traces
       restricted to non-redundant nodes from $B$
6:     $T_{\$,i} \leftarrow K_i(T_\$)$ (compute new traces by applying the kernel $K_i$ to $T_\$$)
7:     Apply Exact matching/LDA to traces $T_{\$,i}$ and selection vectors $S_i$
8: end for
```

The correspondence between the steps described in the text and this algorithm is as follows. Step 1 refers to the preparation of inputs for the algorithm. Step 2 is implemented by lines 1-2. The loop of Step 3 is defined by line 3. Then, step 3.a is done in lines 4-5, step 3.b is done in line 6, step 3.c is performed in line 7.

Complexity The cost of the CPF-LDA phase 1 (locating points of interest) is equal to $g + 1$ times the cost of RNR, which is $O(\mathcal{W}^\omega)$ per window of size $\mathcal{W}$ and g selection vector groups. The cost of phase 2 (matching selection functions) depends on the number of points of interest and is equal to $O(|\mathcal{K}_i|\mathcal{W})$ for matching selection vector group $\mathcal{K}_i$ per one point of interest, using the exact matching method (this assumes that points of interest are sparsely placed). The total complexity of CPF-LDA can thus be estimated by $O((N)\mathcal{W}^{\omega-1}g + |\mathcal{K}|\tau N)$ for a window of size $\mathcal{W}$, window step $O(\mathcal{W})$, N-node traces, selection vector set $\mathcal{K} = \bigcup_i \mathcal{K}_i$, and τN points of interest. In our experiments, we observed values of τ on the magnitude of 3 - 5%. Compared to the optimized LDA complexity from Subsection 5.1, equal to $O((N)\mathcal{W}^{\omega-1} + |\mathcal{K}|(N))$, the optimization leads to the factor τ improvement, assuming that the second term dominates (which is the case of a large number of selection functions). We remark that this is a rough estimate based on practical considerations and reasonable assumptions about the attacked implementation. In principle, the factor g in the first term can be removed by implementing the nesting grouping optimization (see Remark 3). An example breakdown of an application of CPF-LDA and CPF-FLDA is described in Table 5.

6.2 Chosen-plaintext-filtered FLDA

The idea of applying the CPF optimization to FLDA is similar to the LDA case. For this, we need a filtered version of RNR, which reports non-redundant nodes and kernel information *per each filtered* window. It is a direct application of filtering to RNR.

Definition 5 (FRNR)**.** *Filtered RNR* (FRNR) is a combination of general filtering (Algorithm 1) and RNR (Subsection 5.2). It takes as input the window size and a set of traces, and it outputs a list of triplets (filter position, indexes of non-redundant nodes in the window that has been filtered, kernel of the redundant window).

Proposition 3. *FRNR can be computed in time $N\mathcal{W}^\omega$ for N nodes and window $\mathcal{W}$.*

We are now ready to describe the CPF-FLDA procedure. We assume that all trace sets are processed through RNR first so that simple linear redundancies are excluded from the output of FRNR (but the filter nodes are chosen from the complete list of nodes, as in the compatibility remark from Subsection 5.2). We apply FRNR to the random trace set and focus on the filtered-non-redundant nodes. Note that we only need to store information about windows containing redundant nodes. In the next step, for each selection group, we apply FRNR to the corresponding restricted set of traces and now record the kernel information. This information describes filter positions coupled with linear combinations of nodes that are constant on the current selection group. Finally, we apply this kernel (together with filtering) to the random set of traces and apply exact matching/LDA to the resulting vectors, targeting selection vectors from the current group. Note that, due to filtering, the size of the produced columns would vary in some range. Therefore, it is not convenient to write down the resulting vectors in a trace file; we apply the desired testing method to computed vectors on the fly. The procedure is summarized in Algorithm 5.

Complexity The cost of the CPF-FLDA phase 1 (locating points of interest) is equal to $g + 1$ times the cost of FRNR, which is $O(\mathcal{W}^\omega)$ per window of size $\mathcal{W}$ and g selection vector groups (however, compared to CPF-LDA, the FLDA is forced to use sliding window step 1). The cost of phase 2 (matching selection functions) again depends on the number of points of interest and is equal to $O(|\mathcal{K}_i|\mathcal{W})$ for matching selection vectors group $\mathcal{K}_i$ per one point of interest, using the exact matching method. The total complexity of CPF-FLDA can thus be estimated by $O((N)\mathcal{W}^\omega g + \mathcal{K}\mathcal{W}\tau(N))$ for a window of size $\mathcal{W}$, N-node traces, selection vectors $\mathcal{K} = \bigcup_i \mathcal{K}_i$, and $\tau(N)$ points of interest. In our experiments, we observed values of τ on the magnitude of $0.1 - 0.3\%$ per group (as a fraction of the original number of nodes in the circuit; the total number of filtered positions to be considered by plain FLDA is bigger by a factor $\mathcal{W}$). An example breakdown of an application of CPF-LDA and CPF-FLDA is described in Table 5.

6.3 Combining CPF with other attacks

The CPF technique can be combined with other white-box attacks, directly or with some additional work, achieving its main goal: reducing the cost of considering a large number of selection vectors.

CPF-HDDA is a natural extension of CPF-LDA. However, it is not directly clear how to implement the preprocessing step, which should exclude higher-degree redundancies of

Algorithm 5 Chosen-plaintext-filtered FLDA (CPF-FLDA)

Inputs:

- Groups $[(P_1, S_1), \ldots, (P_g, S_g)]$, P_i a set of plaintexts and S_i a set of selection functions constant on P_i
- FLDA parameters (window)

Output: Selection vectors corresponding to candidate key guesses, if any

```
 1: $T_\$ \leftarrow$ a set of traces on random plaintexts
 2: $I_\$ \leftarrow$ pairs (filter position, list of non-redundant vectors in the window) from FRNR($T_\$$)
 3:         Note: sufficient to store pairs only for windows with redundancies
 4: for $i \in \{1, \ldots, g\}$ do
 5:     $T_i \leftarrow$ traces on plaintexts from $P_i$
 6:     $R_i \leftarrow$ pairs (filter position, window kernel) from FRNR($T_i |_{I_\$}$), where FRNR is applied to
        the traces $T_i$ restricted to nodes from $I_\$$ depending on the filter position
 7:     for $(f, k) \in R_i$ do
 8:         Filter traces $T_\$$ at position $f$ and apply the kernel matrix $k$ to the window
 9:         Apply exact matching/LDA to the result, using selection functions from $S_i$
10:     end for
11: end for
```

Remark: In FRNR, filter nodes should be iterated over the original trace (including linearly redundant nodes), while window nodes should be selected only from non-redundant ones (to reduce time and to separate targets for CPF-LDA and CPF-FLDA).

Table 5: Evolution of the number of nodes and time costs for steps of CPF-LDA and CPF-FLDA, on the example of 5-round AES implementation protected by $\mathrm{SEL}_{2,5}$ from `wboxkit` / CHES 2022 white-box tutorial, targeting all 2^{16} selection vectors per byte. The RNR window size is set to 250 and the FRNR window size is set to 100. CPF-FLDA successfully recovers the selected key bytes. Standard FLDA with all the selection vectors requires more than 33 hours for the same result (1 key byte recovery).

Step	Time	#nodes	Perc.
Initial circuit		1300k	100%
Transpose and RNR on random traces	2.3min		
↳ Redundant nodes		717k	55%
↳ Non-redundant nodes		584k	45%
↳ Transpose and RNR on group traces (1/16 groups)	2.3min		
↳ Redundant nodes		52k	4%
| ↳ CPF-LDA (1/16 groups)	2min		
↳ Non-redundant nodes		532k	41%
↳ FRNR on random traces	36min		
↳ Non-redundant (filter position, node) pairs		42M	99%
↳ FRNR on group traces (1/16 groups)	33min		
↳ Redundant filter positions		3009	0.23%
↳ CPF-FLDA (1/16 groups)	5 min		

the random trace set from higher-degree redundancies of the group trace sets. This does not prevent the application but may return a larger amount of points of interest.

CPF-(HO)DCA performs correlation of node vectors / their combinations with a constant selection vector inside each group. This only requires to compute the weight of the vector inside the chosen trace set. Afterward, the located points of interest can be fully checked for correlation against respective selection vectors.

CPF-WBLPN specializes the LPN problem into the low-weight codeword problem when the target selection function is constantly zero. This renders the basic PooledGauss algorithm [EKM17, CU23] invalid since the selected square submatrix has to be invertible for the algorithm to work but the target zero vector implies non-full rank. This can be solved by applying low-weight-codeword formulations of LPN algorithms (for example, [CC98]), or by only considering selection functions with value 1.

7 Higher-Order Filtered Attacks

We show in Subsection 4.1 that we can nullify one of the monomials of the selection function using filtering. The higher-order filtering version of an attack follows the same idea, but we would like to filter $\mathcal{O}$ *node vectors* at once, *e.g.* having indexes where these $\mathcal{O}$ vectors are all equal to zero. Therefore, given a window of $\mathcal{W}$ elements, we will go through all the $\binom{\mathcal{W}}{\leq\mathcal{O}}$ the combinations of $\leq \mathcal{O}$ vectors, and filter them at the same time before performing the attack $\mathcal{A}$ which is depicted in Algorithm 6.

Algorithm 6 Higher-Order filtering with an attack $\mathcal{A}$ (HOF-$\mathcal{A}$)

Inputs:
- A window containing $\mathcal{W}$ *node vectors*
- A list containing the *selection vectors*
- An attack $\mathcal{A}$ to apply after filtering the nodes
- An order $\mathcal{O} < \mathcal{W}$ of filtering

Output: Selection vectors corresponding to candidate key guesses, if any

```
1: for each order $1 \leq o < \mathcal{O}$ do
2:     for each combination $c$ of $o$ vectors in the window do
3:         Filter the trace indexes where all the vectors of $c$ are equal to zero
4:         Apply the attack $\mathcal{A}$ on this subset of traces (for the window and the selection vectors)
5:     end for
6: end for
```

Theorem 3. *Given a trace-based attack with time complexity $O(\mathcal{C}_\mathcal{W})$ for a window of size $\mathcal{W}$, the higher-order filtered version of this attack can be performed with time complexity $O(\mathcal{W}^\mathcal{O}\mathcal{C}_\mathcal{W})$.*

Proof. Filtering the node vectors has a negligible cost compared to the attack. However, we are repeating this attack $\binom{\mathcal{W}}{\leq\mathcal{O}}$ times on a window where $\mathcal{O}$ elements have been removed. Therefore, if the attack $\mathcal{A}$ has a complexity $\mathcal{C}_\mathcal{W}$, we conclude that its higher-order filtered version (HOF-$\mathcal{A}$) has time complexity $O(\binom{\mathcal{W}}{\leq\mathcal{O}}\mathcal{C}_\mathcal{W}) = O(\mathcal{W}^\mathcal{O}\mathcal{C}_\mathcal{W})$, using Proposition 1. □

Theorem 4. *Given a trace-based attack on a window of size $\mathcal{W}$ that has trace complexity $O(\mathcal{C}_\mathcal{W})$, performing the higher-order filtered version of this attack has trace complexity $O(2^\mathcal{O}\mathcal{C}_\mathcal{W})$.*

Proof. In the higher-order filtering, we filter at most $\mathcal{O}$ vectors at once $\binom{\mathcal{W}}{\mathcal{O}}$ times, which multiplies the number of indexes $\in \{0,1\}$ that we need to filter. With the law of large numbers, the ratio of zeroes of the vectors will converge towards one-half. Therefore, to filter $\mathcal{O}$ vectors at once has a trace complexity of $2^\mathcal{O}$. With a window size big enough, the minimum amount of filtered zeroes among the $\binom{\mathcal{W}}{\mathcal{O}}$ of $\mathcal{O}$-filtering will also converge towards $2^{\mathcal{O}}$. $\square$

Filtering and correlation-based attacks Correlation-based attacks have two main limits: the number of linear shares, and the *noise rate* of a masking scheme[CU23] close to one-half. For the former, as explained in Subsection 3.1, it is impossible to correlate one *node vector* to one *selection vector* if there are linear shares (*c.f.* Subsection 3.3). For the latter, the more the noise is close to one-half, the more traces you need to distinguish the correlation of a correct *node vector* with *selection vectors* to random *node vectors.*

Filtering can help in both of these cases as $\text{HOF}_{\mathcal{O}}$ can nullify up to $\mathcal{O}$ linear shares, or $\mathcal{O}$ non-linear monomials (or a mix of the two). By reducing the number of linear shares, we, therefore, need a lower degree version of HODCA since its order must match the number of linear shares of the attacked implementation (*e.g.* for ℓ linear shares, $\text{HOF}_{\mathcal{O}}$-HODCA_o can break it if $\mathcal{O} + o \geq \ell$). Similarly, by nullifying non-linear monomials of the decoding function, we can reduce the noise and therefore reduce the number of required traces to perform HODCA, which directly impacts its time complexity. Similarly, WBLPN could benefit greatly from filtering, as [CU23] shows that reducing the noise rate drastically improves its time complexity.

Filtering and LDA-based attacks The degree d of HDDA should match the highest degree of the non-linear monomials of the decoding function of a masking scheme. To reduce it by one, we need to filter every monomial of maximum degree. This was especially effective against the SEL masking scheme as its decoding function only has one monomial of maximum degree. We can also observe that the time complexity of performing $\text{HOF}_{\mathcal{O}}$ ($O(\mathcal{W}^{\mathcal{O}})$) is better than HDDA_d's ($O(\mathcal{W}^{d\omega} + |\mathcal{K}|\mathcal{W}^d)$, *c.f.* Subsection 3.3), as well as for their trace complexities. Therefore, it is worth performing filtering while there are few highest-degree monomials in the decoding function.

Possible countermeasure As explained in Section 3, the ISW masking scheme is easily broken by LDA, BU by DCA, and now SEL by FLDA (*c.f.* Section 4). As of now, the only solution remaining to force the usage of slow attacks is to combine masking schemes, *e.g.* by applying one to an implementation and then applying a second one to the result. If we apply the ISW_ℓ masking scheme with ℓ linear shares and thereafter the $\text{SEL}_{1,d}$ masking scheme with one linear share and a monomial of degree d, we will obtain the following decoding polynomial (*c.f.* Definition 1):

$$P = (x_{1,1} \oplus x_{1,2} \cdot x_{1,3} \cdots x_{1,d+1}) \oplus (x_{2,1} \oplus x_{2,2} \cdot x_{2,3} \cdots x_{2,d+1}) \oplus \cdots \oplus (x_{\ell,1} \oplus x_{\ell,2} \cdot x_{\ell,3} \cdots x_{\ell,d+1})$$

This combined masking scheme has a theoretical degree d, which would force HDDA to be degree d while having ℓ linear shares, which would oblige HODCA to be order ℓ. Moreover, using Matsui's Pilling up Lemma [Mat94], the non-linear part of $\text{SEL}_{1,d} \circ \text{ISW}_\ell$ has a noise rate $\tau = \frac{1}{2}\left(1 - \left(1 - \frac{1}{2^{d-1}}\right)^{\ell}\right)$, which may prevent WBLPN and slow down HODCA. Lastly, there exist ℓ independent monomials of maximum degree, which forces Higher-Order Filtered LDA to be of order ℓ.

However, we do not have proof of the security of the combination of two (or more) masking schemes, thus it is difficult to affirm that combining masking schemes results in a secured implementation. Another big open problem is even defining gadgets for the $\text{SEL}_{1,d}$ scheme for $d \geq 4$. Thus, our work emphasizes the crucial importance of future research on these problems.

8 Conclusion

In this paper, we introduced a new filtering methodology that, when coupled with the Linear Decoding Analysis attack (FLDA), breaks the SEL masking scheme with any parameters in quartic time per window; which is a major improvement to the previous approaches that were exponential-time in the parameters of SEL. In practice, FLDA is the fastest attack against the SEL masking scheme, which we further improved with multiple optimizations such as redundant node removal (RNR), the underlying linear algebra of LDA, and chosen-plaintext filtering (CPF). These techniques allow for the significantly reduced time needed to assess white-box solutions.

This raises the need for new countermeasures preventing existing attacks and their new possible filtered versions. One way to achieve it would be to directly compose the cubic SEL- (or the quadratic BU-) with ISW-masking, though it is more expensive and there is no proof of security of composed masking schemes in the current literature. Another alternative is the dummy shuffling, which imposes a strong structure on the implementation and is susceptible to fault attacks.

Acknowledgments We would like to thank the anonymous reviewers and the shepherd for their useful comments and suggestions which greatly enhanced the paper's clarity. This work was supported by the Luxembourg National Research Fund's (FNR) and the German Research Foundation's (DFG) joint project APLICA (C19/IS/13641232).

References

[ABMT18] Estuardo Alpirez Bock, Chris Brzuska, Wil Michiels, and Alexander Treff. On the ineffectiveness of internal encodings - revisiting the DCA attack on white-box cryptography. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 103–120. Springer, Heidelberg, July 2018. doi:10.1007/978-3-319-93387-0_6. 1, 14

[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. doi:10.1007/978-3-662-53140-2_11. 1, 4, 5

[BRVW19] Andrey Bogdanov, Matthieu Rivain, Philip S. Vejre, and Junwei Wang. Higher-order DCA against standard side-channel countermeasures. In Ilia Polian and Marc Stöttinger, editors, *COSADE 2019*, volume 11421 of *LNCS*, pages 118–141. Springer, Heidelberg, April 2019. doi:10.1007/978-3-030-16350-1_8. 2, 3, 6, 10, 11

[BU18] Alex Biryukov and Aleksei Udovenko. Attacks and countermeasures for white-box designs. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, pages 373–402. Springer, Heidelberg, December 2018. doi:10.1007/978-3-030-03329-3_13. 1, 2, 3, 4, 6, 10, 11, 13

[BU21] Alex Biryukov and Aleksei Udovenko. Dummy shuffling against algebraic attacks in white-box implementations. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 219–248. Springer, Heidelberg, October 2021. doi:10.1007/978-3-030-77886-6_8. 2, 13, 14

[CC98] A. Canteaut and F. Chabaud. A new algorithm for finding minimum-weight words in a linear code: application to McEliece's cryptosystem and to narrow-sense BCH codes of length 511. *IEEE Transactions on Information Theory*, 44(1):367–378, 1998. doi:10.1109/18.651067. 20

[CEJv02] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. A white-box DES implementation for DRM applications. In *Digital Rights Management Workshop*, volume 2696 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2002. doi:10.1007/978-3-540-44993-5_1. 1, 5

[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. doi:10.1007/3-540-36492-7_17. 1, 5

[CU23] Alex Charlès and Aleksei Udovenko. LPN-based attacks in the white-box setting. *IACR TCHES*, 2023(4):318–343, 2023. https://tches.iacr.org/index.php/TCHES/article/view/11168. doi:10.46586/tches.v2023.i4.318-343. 2, 3, 5, 7, 10, 20, 21

[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. doi:10.1007/978-3-662-43414-7_13. 1

[DR98] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. AES submission. See also http://csrc.nist.gov/archive/aes/rijndael/, 1998. 4

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard.* Information Security and Cryptography. Springer-Verlag Berlin Heidelberg, 2002. doi:10.1007/978-3-662-04722-4. 4

[EKM17] Andre Esser, Robert Kübler, and Alexander May. LPN decoded. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 486–514. Springer, Heidelberg, August 2017. doi:10.1007/978-3-319-63715-0_17. 20

[GPRW20] Louis Goubin, Pascal Paillier, Matthieu Rivain, and Junwei Wang. How to reveal the secrets of an obscure white-box implementation. *Journal of Cryptographic Engineering*, 10(1):49–66, April 2020. doi:10.1007/s13389-019-00207-5. 2, 3, 5, 6, 11

[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. doi:10.13154/tches.v2020.i3.454-482. 4, 6, 9, 15

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Heidelberg, August 2003. doi:10.1007/978-3-540-45146-4_27. 2, 5, 10, 13

[Kar11] Mohamed Karroumi. Protecting white-box AES with dual ciphers. In Kyung-Hyune Rhee and DaeHun Nyang, editors, *Information Security and Cryptology - ICISC 2010*, pages 278–291, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-24209-0_19. 4, 14

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, Heidelberg, August 1999. doi:10.1007/3-540-48405-1_25. 5

[Mat94] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 386–397. Springer, Heidelberg, May 1994. doi:10.1007/3-540-48285-7_33. 21

[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. doi:10.13154/tches.v2019.i2.225-255. 1

[Sag23] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 10.2)*, 2023. https://www.sagemath.org. 13

[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. doi:10.46586/tches.v2021.i2.61-105. 2, 6, 10, 13

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, Aug 1969. doi:10.1007/BF02165411. 3

[TGCX23] Yufeng Tang, Zheng Gong, Jinhai Chen, and Nanjiang Xie. Higher-order DCA attacks on white-box implementations with masking and shuffling countermeasures. *IACR TCHES*, 2023(1):369–400, 2023. doi:10.46586/tches.v2023.i1.369-400. 4, 6, 9, 15

[TGLZ23] Yufeng Tang, Zheng Gong, Bin Li, and Liangju Zhao. Revisiting the computation analysis against internal encodings in white-box implementations. *IACR TCHES*, 2023(4):493–522, 2023. doi:10.46586/tches.v2023.i4.493-522. 14