IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–61. DOI:XXXXXXXX

UP TO 50% OFF: Efficient Implementation of Polynomial Masking

Jorge Andresen[1], Paula Arnold[1], Sebastian Berndt[2,1], Thomas Eisenbarth[1], Sebastian Faust[3], Marc Gourjon[4], Eric Landthaler[1], Elena Micheli[3], Maximilian Orlt[5,3], Pajam Pauls[1], Kathrin Wirschem[3] and Liang Zhao[3]

[1] Institute for IT Security, University of Luebeck, Luebeck,
{jorge.andresen,eric.landthaler}@student.uni-luebeck.de,
{p.arnold,thomas.eisenbarth,p.pauls}@uni-luebeck.de
[2] Department of Electrical Engineering and Computer Science, Technische Hochschule Lübeck, Luebeck, sebastian.berndt@th-luebeck.de
[3] Technical University of Darmstadt, Darmstadt,
{sebastian.faust,elena.micheli,kathrin.wirschem,liang.zhao}@tu-darmstadt.de
[4] Max Planck Institute for Security and Privacy, Bochum, marc.gourjon@mpi-sp.org
[5] UCLouvain, Louvain-la-Neuve, Belgium maximilian.orlt@uclouvain.be

Abstract. While passive probing attacks and active fault attacks have been studied for multiple decades, research has only started to consider combined attacks that use both probes and faults relatively recently. During this period, polynomial masking became a promising, provably secure countermeasure to protect cryptographic computations against such combined attacks. Unlike other countermeasures, such as duplicated additive masking, polynomial masking can be implemented using a linear number of shares, as shown by Berndt et al. at CRYPTO '23. Based upon this fact, Arnold et al. noted at CHES '24 that polynomial masking is particularly well-suited for parallel computation. This characteristic is especially effective in scenarios involving multiple circuits with identical structures, such as the 16 SBoxes in AES. Just recently, Faust et al. showed at CHES '25 that one can also incorporate the technique of packed secret sharing into these masking schemes, given that the state-of-the-art polynomial masking scheme is secure against combined attacks.

In this work, we present provably secure advancements regarding this state-of-the-art scheme in both computational and randomness efficiency, reducing the randomness complexity by up to 50% and the computational complexity even more by going from a quadratic term to a linear one for many parameters. Moreover, we present the first implementation of a polynomial masking scheme against combined attacks along with an extensive experimental evaluation for a wide range of parameters and configurations as well as a statistical leakage detection to evaluate the security of the implementation on an Arm Cortex-M processor. Our implementation is publicly available to encourage further research in practical combined resilience.

Keywords: Polynomial Masking · Leakage and Fault Resilience · TVLA

1 Introduction

Polynomial secret sharing (or Shamir's secret sharing) is a well-established technique first presented by Shamir [Sha79]. This method has found wide applications in the field of multiparty computation [BGW88], but has also been used to develop implementations secure against physical attacks [PR11, CPR12, GSF14] as an alternative to the widely-used additive sharing. When focusing on purely passive side-channel attacks, additive masking

Licensed under Creative Commons License CC-BY 4.0.

schemes often outperform their polynomial counterparts due to the smaller number of needed shares. More concretely, if an attacker is able to obtain t temporary values (so-called *probes*), additive schemes require only $t+1$ shares, while polynomial schemes relying on an *honest majority* require $2t+1$ shares. However, if we also consider attacks that, in addition to the probes, can also actively modify e values (via so-called *faults*), this situation changes dramatically: due to the lack of redundancy, schemes relying on additive masking are often quite vulnerable even to simple additive fault attacks.
Combined attacks often allow an (t,e)-adversary to choose up to t wires to probe and e wires to fault with respect to a predetermined fault function. More precisely, the notion of combined security presented in [SFES18, DN20, FRSG22, FGM+23] allows an adversary to first inject up to e faults before it obtains the values of up to t chosen wires (*active-then-passive approach*, cf. [FRSG22]). In [BFG+24], this idea is extended to random probing and random faults. The sequence of works [BEF+23, ABEO24, FOWZ25] uses a slightly different attacker, who can adaptively choose the faults (within a predetermined fault set) corresponding to its already probed values. Thus, an adversary might learn some information from the probes before deciding to fault a specific value, and vice versa. To defend against such *combined attacks*, additive masking schemes also usually employ rather expensive solutions such as MACs [RMB+18, MAN+19] or the duplication of the circuit [DN20, FRSG22] to detect the bounded number of injected faults. Adding MACs in an efficient manner has been proven to be quite hard [TNN24] and, even if successful, gives a large overhead, especially in smaller fields. While the duplication of the circuit allows the reuse of the existing additive masking schemes, it leads to a quadratic number (i.e., $\Omega(e \cdot t)$) of shares and requires various complex *error detection* mechanisms. As typical circuits consist of a large number of linear and affine operations, this large number of shares can become problematic even for moderately small circuits and, additionally, circuits containing such duplicated values are vulnerable to horizontal attacks [BCPZ16].

In contrast, as observed by Seker et al. [SFES18], masking schemes using polynomial secret sharing easily allow for error detection, due to their relationship to the well-studied *Reed-Solomon* codes [RS60]. However, the presented solution still used a multiplication procedure from multiparty computation, which required an honest-majority assumption of $2t+e+1$ shares, leading to a rather high number of shares and the security analysis of some gadgets was found to be insufficient [BEF+23]. Relatively recently, Berndt et al. [BEF+23] observed that one can replace this multiplication procedure by a more efficient one, called **LaOla**, in the context of combined attacks that allowed the use of a minimal number of $t+e+1$ shares. The computational complexity of this approach was reduced nearly by half by Arnold et al. [ABEO24], who presented a solution that allowed to embed two secrets into a single polynomial, which in turn allowed to combine two parallel multiplications into a single, slightly more expensive, multiplication. Building upon this observation, Faust et al. [FOWZ25] showed that one could also adapt the multiparty computation technique called *packed secret sharing* to the **LaOla** multiplication. Intuitively, polynomial secret sharing embeds the secret into the support point 0, i.e., it encodes the secret value v into a polynomial p such that $p(0) = v$. In packed secret sharing, one encodes multiple secrets $v_0, v_1, \ldots, v_{k-1}$ into a polynomial p at different support points $u_0, \ldots, u_{k-1}$ such that $p(u_i) = v_i$. This allows for combining k parallel multiplications (such as those occurring in block ciphers) into a single, slightly more expensive multiplication, which drastically reduces both the computation complexity as well as the randomness requirement of the scheme. Just recently, Arnold et al. [ABE+25] showed an adaption of these algorithms to restrict the information obtainable from faulted shares.

From a purely theoretical perspective, masking schemes using polynomial secret sharing are thus optimally equipped against combined attackers: They have a minimal number of shares [BEF+23] and allow for a large number of optimizations to reduce their resource consumption. However, while the works of Arnold et al. [ABEO24], Faust et al. [FOWZ25],

and Arnold et al. [ABE+25] contain estimations on the number of needed field operations, no concrete implementation of polynomial secret sharing using the **LaOla** approach of Berndt et al. [BEF+23] has been studied. Hence, no implementation of this is available, leading to a complete lack of real-world performance numbers, which might be influenced by practically important matters such as register pressure or randomness latency. Furthermore, while all of these works contain security proofs using well-established security notions, it is not clear how much *hardening* is needed for such an implementation to pass a real-world test vector leakage assessment (TVLA).

1.1 Contribution

This work presents the first implementation of a polynomial-masked circuit designed to counter combined fault and leakage attacks. Building on the work of Faust et al. at CHES 2025 [FOWZ25] and Arnold et al. [ABE+25], our implementation incorporates both conceptual and practical advancements.

Conceptually, we introduce improved gadgets that significantly reduce randomness requirements and computational costs. A major bottleneck in previous constructions was the exhaustive generation of random zero-encodings, required for re-randomizing internal computations without impacting their overall functionality. Namely, let $F_0, \ldots, F_{n-1}$ be any e-fault- and t-probe-resilient encoding with k embedded secrets and $n = t+e+k$ shares, and let $E_0, \ldots, E_{n-1}$ denote a uniform random zero-encoding, i.e., all embedded secrets are zeros. We can re-randomize our sharing by share-wise adding such zero-encodings, i.e., compute $G_i = F_i + E_i$. This results in a uniform random encoding for the same secrets, as $\mathsf{unmask}\,(F_0, \cdots, F_{n-1}) = \mathsf{unmask}\,(G_0, \ldots, G_{n-1})$. While this re-randomization step is crucial to ensure probing resilience, it introduces significant overhead in terms of both computation and randomness. For this reason, we present a new gadget to generate such zero-encodings much more efficiently, while preserving all necessary security requirements. From an implementation perspective, we present new optimization techniques, such as parallelizable task execution, and incorporate these into our implementation. This leads to a significantly reduced number of clock cycles, enabling faster execution of the computations.

Using our new optimizations, the final outcome is a provably secure implementation that achieves a 50% reduction in randomness cost compared to the current state-of-the-art in polynomial masking. In addition to its theoretical security guarantees, the leakage resilience of the implemented code was empirically validated against up to 5 000 000 traces, demonstrating its robustness in practice. The overall outcome is depicted in Figure 1, and detailed in the following four paragraphs.

(1) Reduced Computational Complexity. Our optimization techniques reduce the overall computational complexity cost of the countermeasures for an unprotected circuit C significantly, depending on the concrete choice of parameters t, e, and k. In particular, we improve the computational complexity to generate the CPU-intensive zero-encodings from $\mathcal{O}((t+e+k)\cdot t)$ to $\mathcal{O}((e+k)\cdot t)$. Hence, for the typical setting, where $t \gg \max\{e,k\}$, we reduce the overhead from a quadratic one to a linear one. Traditionally, zero-encodings are generated by fixing a polynomial embedding k secrets, followed by the computation of $n = t+e+k$ shares, requiring $\mathcal{O}(t)$ calculations per share. We avoid this inconvenience by assigning the first t shares as uniformly random elements and calculating only the remaining $e+k$ shares. Thereby, we significantly reduce the computational overhead while maintaining share-wise generation. This refinement is discussed in detail in Section 3.1.

(2) Improved Parallel Computation. Our improved algorithm for generating random zero-encodings achieves a computational complexity of $\mathcal{O}((e+k)\cdot t)$. However, a bottleneck remains: the individual complexities of the shares range from $\mathcal{O}(t)$ to $\mathcal{O}(1)$, resulting

in an unbalanced latency of each share. This is especially troublesome in parallelized software implementations and hardware implementations, as this imbalance increases the critical path of the implementation significantly. To address this, we enhance the gadget by allowing parallel computation and evenly distributing the computational load among the different shares. We transform the sequential share-wise operations into balanced parallelizable tasks, and thus reduce the maximum complexity of certain shares from $\mathcal{O}(t)$ to $\mathcal{O}(\frac{(e+k)}{n} \cdot t)$ without altering the overall complexity $\mathcal{O}((e+k) \cdot t)$. While the total computational cost remains unchanged, this optimization reduces the maximum per-share computation. Both improvements are described in detail in Section 3.2.

(3) Reduced Randomness Complexity. Another bottleneck besides the computational complexity of the underlying operations is the randomness complexity of the algorithm, as the production of (pseudo)random values can take a relatively large number of cycles. We present a new optimization technique that significantly reduces the randomness cost in most combined-resilient compilers based on polynomial masking. Specifically, our constructions achieve a reduction of up to 50% in randomness complexity, as detailed in Section 4. While previous countermeasures always randomized the internal computation with uniform random zero-encodings, we found that this is an overly stringent requirement. Relaxing this property leads to more efficient constructions that remain provably secure. We demonstrate the applicability of this idea to both **BGW**-based and **LaOla**-based multiplication and refresh gadgets, extending the benefits to various resilient compilers while maintaining combined security, as detailed in Section 5.

(4) Evaluated Implementation. We provide a comprehensive implementation of all the optimizations and a masked SBox for AES in Section 6. This implementation shows that the asymptotic improvements of our new gadgets have already significantly reduced both the randomness consumption and the cycle count for relatively small parameters. In the parameter range of $1 \leq t \leq 3$ and $0 \leq e \leq 2$, our improved zero-encoding gadgets reduce the cycle count needed by a factor of up to 6.50×. As this is a bottleneck of the implementation, we directly reduce the cycle count of a complete multiplication by a factor of up to 2.05×, leading to a significant improvement in the SBox calculations. Furthermore, we expect this factor to grow for larger parameter sizes and more optimized high-level components. In addition, we validate the physical leakage resilience of our gadget through extensive physical testing, confirming its security under realistic conditions. Similarly, we perform extensive simulated fault injection campaigns to estimate the success rate of set-zero and additive fault attacks. Our implementation and experiments are publicly available at https://github.com/leakage-resilience/discounted-poly-masking.

1.2 Related Work

While probing security and fault resilience have been studied for several decades, efficient protections against combined attacks have only been developed in recent years. In the literature, two main approaches have been proposed to achieve combined resilience: duplicated (Boolean) masking [DN20, FRSG22, RFSG22, FGM+23] and polynomial masking [BEF+23, ABEO24, FOWZ25, ABE+25].

Duplicated masking uses leakage-resilient additive masking schemes to first generate $\mathcal{O}(t)$ shares and then duplicate each share $\mathcal{O}(e)$ times. All computations are performed on these $\mathcal{O}(e)$ sharings, enabling fault detection by comparing the individual outputs of the computations. In summary, this method requires $\mathcal{O}(e \cdot t)$ shares to achieve combined resilience and does not support encoding multiple secrets into a single sharing. As discussed by Berndt et al. [BEF+23], this large number of shares is especially problematic for the linear and affine gadgets. However, as highlighted in [FOWZ25], the approach

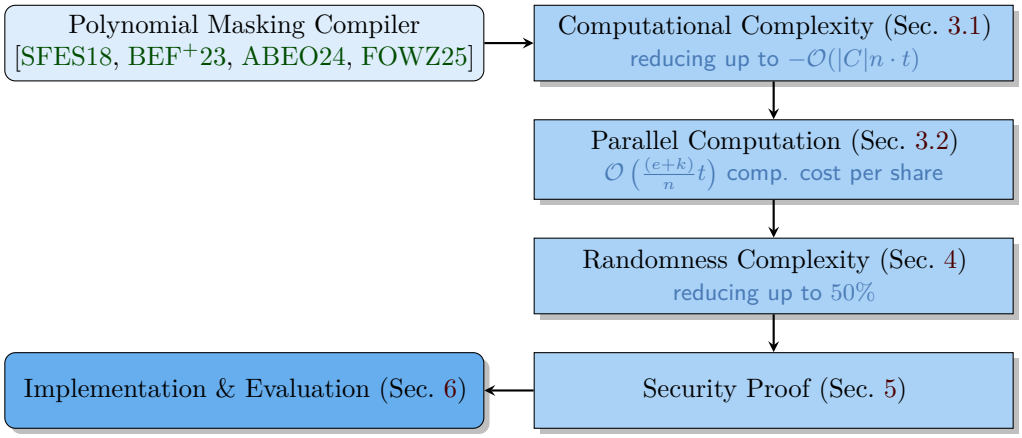


Figure 1: Contribution overview of this work

of [FRSG22, FGM+23] still remains quite efficient for non-parallelized computations and computations in binary fields.

Alternatively, polynomial masking inherently provides resilience against combined attacks by combining Reed-Solomon codes [RS60] (for error detection) and Shamir's secret sharing [Sha79] (for leakage resilience akin to additive masking). This approach requires only $\mathcal{O}(t+e)$ shares to achieve the same security level as duplicated Boolean masking, which requires $\mathcal{O}(e \cdot t)$ shares to be resilient against e faults and t probes. Moreover, leveraging the concept of packed secret sharing [FY92], Faust et al. [FOWZ25] demonstrated that computations can be efficiently parallelized by embedding $k \in \mathbb{N}$ secrets into a single sharing. They analyzed the Boolean circuit complexity and showed that (parallelized) polynomial masking [BEF+23, ABEO24, FOWZ25] outperforms duplicated Boolean masking [FRSG22, FGM+23] already for very small parameter sets such as $t = e = 1$, due to the intermediate error correction steps required in duplicated masking. Regarding the randomness complexity, Faust et al. [FOWZ25] observed that duplicated Boolean masking outperforms polynomial masking gadgets for $t, e < 5$.

Recently, Carlet et al. [CDGT24] introduced a very efficient construction for parallel computations using packed secret sharing while leveraging the fast Fourier transform (FFT). However, their analysis is limited to standalone leakage attacks and standalone random fault attacks, providing no security claims for combined attacks or faults that do not lead to random values. Even worse, during any FFT-based computation, a single fault can propagate across all output shares, making fault detection impossible.

In contrast to the work of Carlet et al. [CDGT24], Belaïd et al. [BFG+24] propose more general models for combined security including random probes and a more general random faults. However, their focus is primarily on developing these models for verification tools, and provably secure countermeasures are still missing. Arnold et al. [ABE+25] generalized the security notions of Berndt et al. [BEF+23] and presented new gadgets that significantly reduced the attack surface of an attacker using such random faults.

While Feldtkeller et al. [FRSG24] implemented duplicated masking schemes, our work is the first to implement algorithms using polynomial masking as a countermeasure against combined attacks –an essential step towards bridging the gap between theoretical advantages and practical application.

We emphasize that the definitions of combined security are still diverse: Schemes using additive masking typically require an upper bound on the number of faulty outputs to detect the faults using MACs or duplication. In contrast, schemes using polynomial masking typically require a lower bound on the degree of the polynomial that describes the sharing to detect faults.

2 Notation

Let $n \in \mathbb{N}_{\geq 2}$ represent the number of shares on which we operate, and let $\mathbb{F}$ be a finite field with at least $n+1$ elements. If not explicitly mentioned otherwise, we set $n = d + e + 1$, where d is the degree of our underlying polynomial and e is the number of faults we support. Moreover, we denote the number of probes we support by t.

The Hadamard (element-wise) product of two vectors or matrices is denoted by $*$. Furthermore, we define the Hamming weight $\mathsf{weight}(\boldsymbol{v})$ of a vector $\boldsymbol{v} \in \mathbb{F}^n$ to be the number of non-zero entries in $\boldsymbol{v}$. For a matrix $A \in \mathbb{F}^{m \times n}$, $A_{i,j}$ (or $A[i,j]$) refers to the j-th element in the i-th row. $A_{:,j}$ (or $A[:,j]$) refers to the elements in the j-th column of matrix A and $A_{i,:}$ or $(A[i,:])$ refers to the elements in the i-th row of matrix A. The identity matrix is denoted as $\mathbf{I}$ and the transpose of M is M^T. If $\boldsymbol{v} \coloneqq (v_0, \ldots, v_{n-1})^T \in \mathbb{F}^n$ is a vector and $I \subseteq \{0, \ldots, n-1\}$ is a subset of its indices, we define $\boldsymbol{v}_I \coloneqq (v_i)_{i \in I}$. Besides, we write $\boldsymbol{v}_{0:-1}$ (or $\boldsymbol{v}[0:-1]$) to denote a sub-vector containing the same first $n-1$ elements as $\boldsymbol{v}$ except for the last element in $\boldsymbol{v}$, and $\boldsymbol{v}_i$ (or $\boldsymbol{v}[i]$) to denote the i-th element in $\boldsymbol{v}$. We write $\boldsymbol{v}_{-1}$ (or $\boldsymbol{v}[-1]$) to denote the last element in $\boldsymbol{v}$. The set $\{0, \ldots, n-1\}$ is abbreviated as $[n]$, and the set $\{n_1, \ldots, n_2 - 1\}$ is abbreviated as $[n_1, n_2]$.

For values $x_0, \ldots, x_{n-1} \in \mathbb{F}$, the Vandermonde matrix with n rows and $d+1$ columns (or $n \times (d+1)$ matrix for simplicity) is written as $\mathsf{Van}_{n \times (d+1)}(x_0, \ldots, x_{n-1})$ and the entry in row $i \in [n]$ and column $j \in [d+1]$ is x_i^j. If the matrix is square (i.e., $n = d+1$), we simply write $\mathsf{Van}(x_0, \ldots, x_{n-1})$. The inverse of its square version is denoted by $\mathsf{Van}^{-1}_{(d+1)\times(d+1)}(x_0, \ldots, x_{n-1})$. We present a polynomial $p(X) = \sum_{i=0}^{d} p_i X^i$ using lowercase letters and refer to its coefficients as $p_0, \ldots, p_d$. Shares corresponding to the polynomial $p(X)$ are written as $(P_i)_{i \in [n]}$, i.e., $P_i = P(x_i)$.

Let $\mathcal{D}$ and $\mathcal{D}'$ be two probability distributions. We write $\mathcal{D} \equiv \mathcal{D}'$ to indicate that these two distributions are equivalent. We write $(y_0, \ldots, y_{n-1}) \leftarrow_{\$} \mathbb{F}^n$ to denote that each $y_i \in \mathbb{F}$, $i \in [n]$, is chosen independently and uniformly at random. Consider the random variables $X_0, X_1, \ldots, X_{n-1} \leftarrow_{\$} \mathbb{F}$. These random variables are said to be *k-wise independent* if for any subset $I \subseteq [n]$ with $|I| \leq k$, we have $\Pr\left[\bigwedge_{i \in I} X_i = x_i\right] = \prod_{i \in I} \Pr[X_i = x_i]$. Moreover, we call random variables *k-wise uniformly independent*, if for any subset $I \subseteq [n]$ with $|I| \leq k$ it holds $(X_i)_{i \in I}$ is indistinguishable from uniformly distributed. A probability distribution $\mathcal{D}$ is *perfectly simulatable* from a set S, if there exists a simulator SIM such that $\mathsf{SIM}(S)$ outputs values with the same distribution as $\mathcal{D}$. Formally, for any possible x in the domain of $\mathcal{D}$, it holds true that $\Pr[\mathsf{SIM}(S) = x] = \Pr[\mathcal{D} = x]$, denoted by $\mathsf{SIM}(S) \equiv \mathcal{D}$.

2.1 Attacker Models

In this work, we improve the computational and randomness complexity of zero-encodings. As an example, we use our construction to improve the gadgets presented in the sequence of works [BEF+23, ABEO24, FOWZ25, ABE+25]. We also consider their security model, consisting of passive and active adversaries. Passive adversaries observe the input-output behavior of a cryptographic implementation and gain access to certain intermediate computational states. Active adversaries, on the other hand, can inject faults into internal states, including input and output values of cryptographic gadgets. This can lead to unintended computations and reveal secret information. We consider adaptive adversaries, which can perform combined attacks, i.e., they can simultaneously probe and fault the intermediate values of the implementation using information from previous leakage.

Circuits. We represent calculations as a circuit C with respect to a finite field $(\mathbb{F}, +, \cdot)$, using a directed acyclic graph G. The nodes in G are labeled as one of the following: input gates, output gates, random gates, addition gates, multiplication gates, or constant transformation gates. Considering a circuit C as such a graph, a computation begins by

assigning the inputs to the input gates. Once all the parents of a node have obtained a value, a gate's value is calculated by applying the underlying operation (e.g., addition or multiplication) to these values. We continue gate by gate. Random gates do not have any parent nodes and output a uniformly random element in $\mathbb{F}$.

A circuit can *abort* the computation by returning the abort signal $\perp$. The output distribution of a circuit C with inputs $x_0, x_1, \ldots$ is denoted as $\mathsf{C}(x_0, x_1, \ldots)$. For a circuit C with inputs $x_0, x_1, \ldots$ and fixed randomness R, we write $\mathsf{C}^R(x_0, x_1, \ldots)$.

Compiler. A common approach to prevent the above attacks is to use a compiler that transforms a vulnerable circuit C into a more secure version C' that can resist these attacks. The resulting circuit C' is referred to as a fault- and leakage-resilient circuit. The main challenge in designing these circuits is to prevent the leakage of sensitive information through the intermediate values generated during computation. The transformed circuit C' operates on encoded values, which can be constructed using methods like Boolean secret sharing, polynomial secret sharing, or other encoding techniques. The choice of encoding depends on the specific type of attack the circuit is meant to resist. Additionally, each gate in the original circuit C is replaced by a gadget in C'. A gadget is a sub-circuit with the same functionality as the original gate in C, but operates on encoded inputs and outputs.

In this work, we build gadgets with respect to packed secret sharing [FY92], which generalizes polynomial masking. More details are explained in Section 2.2.

Passive Attacks. A passive *t-probing* adversary $\mathcal{A}$ can run the circuit C and arbitrarily probe up to t intermediate wires $(w_i)_{i \in I}$ of C, where $|I| \leq t$.
Let $b \leftarrow_{\$} \{0,1\}$ be a random bit that determines which input is fed into the circuit. The adversary selects two inputs, x_0 and x_1. A circuit C with input x_b is called *perfectly secure* against a *t-probing* adversary $\mathcal{A}$ if the probability that the adversary $\mathcal{A}$ is able to distinguish whether the input x_b is x_0 or x_1, equals $\frac{1}{2}$.

A well known notion of probing secure gadgets is *t-Non-Interference* (t-NI) (cf. [BBD+15, BBD+16]). This definition ensures perfect security. Intuitively, NI guarantees that any information an adversary $\mathcal{A}$ can gain from probing t arbitrary intermediate values can equally be obtained by probing t input values instead.

Definition 1 (NI [BBD+15, BBD+16])**.** A gadget G is t-non-interfering (t-NI) if for any set of t_{int} intermediate variables and any subset $\mathcal{O}$ of output indices with $t_{\text{int}} + |\mathcal{O}| \leq t$, there exists a subset of input indices I with $|I| \leq t_{\text{int}} + |\mathcal{O}|$ such that the distribution of the t_{int} intermediate variables and the output variables in $\mathcal{O}$ is perfectly simulatable using inputs with indices from I.

However, since t-NI gadgets are not necessarily composable (cf. [BBD+16]), Barthe et al. [BBD+16] defined a stronger notion, t-SNI, to obtain composability.

Definition 2 (SNI [BBD+16])**.** A circuit G is t-strong non-interfering (t-SNI) if for any set of t_{int} intermediate variables and any subset $\mathcal{O}$ of output indices with $t_{\text{int}} + |\mathcal{O}| \leq t$, there exists a subset of input indices I with $|I| \leq t_{\text{int}}$ such that the distribution of the t_{int} intermediate variables and the output variables in $\mathcal{O}$ is perfectly simulatable with inputs with indices from I.

The main difference between NI and SNI is that SNI differentiates between intermediate and output probes, ensuring that only the intermediate probes influence the number of input elements required by the simulator.

Active Attacks. In this work, we focus on two types of fault sets: the set of additive faults $\mathcal{F}^{+} := \{\xi : \xi(x) = x + a \text{ for all } a \in \mathbb{F}\}$ and the set of wire-independent faults $\mathcal{F}^{\mathsf{ind}}$,

where the wire with value x can be arbitrarily but independently faulted to value x'. Note that $\mathcal{F}^+ \subset \mathcal{F}^{\mathsf{ind}}$.

Berndt et al. [BEF+23] defined the notion of fault robustness, which guarantees that any (non-trivial) fault can be detected. Intuitively, the notion consists of three parts: *correctness* of the considered gadget; a *linear part*, which surveys all faults that result in at most one output fault; and a *non-linear part*, which takes faults into account that result in more than one output fault.

The definitions of Berndt et al. [BEF+23] have been generalized in [ABE+25] to obtain stronger security guarantees and to fix a bug that was observed in the general security proofs regarding the fault robustness in [BEF+23].

Definition 3 (ϵ-indistinguishable)**.** A distribution D over the sample-space Ω is ϵ-*indistinguishable* iff $\Delta(D, U) \leq \epsilon$, where $\Delta(\mathcal{D}_1, \mathcal{D}_2) = \max_{X \subset \Omega} |\Pr_{\mathcal{D}_1}(X) - \Pr_{\mathcal{D}_2}(X)|$ and U is the uniform distribution on Ω.

Definition 4 ((e, ϵ)–fault robustness [ABE+25])**.** A gadget G with one output sharing and up to two input sharings is (e, ϵ)-fault-robust with respect to $\mathcal{F}$, if for any couple of valid (polynomial) sharings $\boldsymbol{F}$ and $\boldsymbol{H}$, then the output $\boldsymbol{Y} \leftarrow G^R(\boldsymbol{F}, \boldsymbol{H})$, $R \leftarrow\!\!\$\, U$, is also valid. For any couple of fault vectors $\boldsymbol{\xi}^F, \boldsymbol{\xi}^H$ and for any $T \in \mathcal{A}(\mathcal{F})$ with $|T| \leq e$, and resulting $\boldsymbol{Y} + \boldsymbol{\omega} + \boldsymbol{\omega'} \leftarrow T[G^R]\left(\boldsymbol{F} + \boldsymbol{\xi}^F, \boldsymbol{H} + \boldsymbol{\xi}^H\right)$ it holds that there are numbers $s_1 \in \mathbb{N}$ and $s_2 \in \mathbb{N}$ with $s_1 + s_2 \leq |T|$ such that:

i) $\mathsf{weight}(\boldsymbol{\omega}) \leq s_1$ *(linear part)*
ii) and $\boldsymbol{\omega'}$ is either equal to $\boldsymbol{\xi}^F + \boldsymbol{\xi}^H$ or $\boldsymbol{\omega'}$ is equal to $\overline{\boldsymbol{\omega}}'$, where $\overline{\boldsymbol{\omega}}'$ is produced by the following random experiment: Let $p_{\overline{\omega}'} \in \mathbb{F}_q[X]$ be a polynomial with coefficients $p_i, i \in [n]$ is chosen such that at least $e+1-s_2$ of the coefficients $p_{d+1}, \ldots, p_{n-1}$ are chosen independently of each other and from a distribution D_ϵ that is ϵ-indistinguishable to the uniform distribution. Besides, $\boldsymbol{\omega'}$ is independent of T when $R \leftarrow\!\!\$\, U$. *(non-linear part)*

If $\epsilon = 0$, we also write e–fault robustness instead of (e, ϵ)–fault robustness. We call D_ϵ the *non-linear fault distribution.*

Combined Attacks. Berndt et al. [BEF+23] showed that it is not sufficient to examine probing resilience and fault resilience separately to obtain combined attacks. This is the reason why they introduced the notion of fault-resilient (S)NI, which can be seen as a bridge between probing resilience and fault resilience to obtain combined resilience.

Definition 5 (Fault-resilient SNI [BEF+23, Def. 8])**.** A gadget G is d-fault-resilient (strong-) non-interfering (d-fr(S)NI) with respect to a fault set $\mathcal{F}$ if $T[G]$ is d-(S)NI for any fault attack $T \in \mathcal{A}(\mathcal{F})$.

Please note that Berndt et al. [BEF+23] do not only provide fr(S)NI, but also a slightly weaker version, which we restate in the following.

Definition 6 (Weak fault-resilient NI [BEF+23, Def. 6])**.** A gadget G is d-weak-fault-resilient non-interfering (d-wfrNI) with respect to $\mathcal{F}$ if every set of d' probes in $T[G]$ can be (perfectly) simulated with $d' + |T|$ shares of each input sharing for any fault attack $T \in \mathcal{A}(\mathcal{F})$ with $|T| + d' \leq d$.

Similarly to weak fault-resilient NI, we consider a weaker version of fault-resilient SNI:

Definition 7 (Weak fault-resilient SNI [BEF+23, Def. 7])**.** A gadget G is d-weak-fault-resilient non-interfering (d-wfrSNI) with respect to $\mathcal{F}$ if every set of d_1 internal probes and d_2 output probes in $T[G]$ can be (perfectly) simulated with $d_1 + \epsilon_1$ shares of each input sharing for any fault attack $T \in \mathcal{A}(\mathcal{F})$ with ϵ_1 internal faults and ϵ_2 output faults such that $d_1 + d_2 + \epsilon_1 + \epsilon_2 \leq d$.

As shown in [ABE+25], one can combine SNI (NI) and generalized fault-invariance to obtain wfrSNI (wfrNI), which is the reason why we state their generalized fault invariance in the following. Note that eager sampling refers to the process of drawing all random values at the beginning of an algorithm, in contrast to the traditional lazy sampling, where values are generated when accessed.

Definition 8 (Fault-invariance [BEF+23, ABE+25])**.** Let $R = R_0, \ldots, R_m$ denote uniformly i.i.d. eagerly sampled randomness and $m, \ell \in \mathbb{N}$. Let p denote an intermediate value (probe) in C^R and let f^R describe the way the circuit C computes p. Moreover, let p' denote the corresponding intermediate value (probe) on the same wire in $T[\mathsf{C}^R]$, and let f'^R describe the way the circuit $T[\mathsf{C}^R]$ computes p'.

A circuit $\mathsf{C}(x_0, \ldots, x_{\ell-1})$ is *generalized-fault-invariant* with respect to a fault set $\mathcal{F}$ if for any $T \in \mathcal{A}(\mathcal{F})$ and for any probe $p' = f'^{(R_0,\ldots,R_{m-1})}(x_0, \ldots, x_{\ell-1})$ in the faulted circuit $T[\mathsf{C}^R]$, there exist fault functions $\xi, \xi_0, \ldots, \xi_{\ell-1}, \tilde{\xi}_0, \ldots, \tilde{\xi}_{m-1} \in \mathcal{F}$ such that it holds

$$f'^{(R_0,\ldots,R_{m-1})}(x_0, \ldots, x_{\ell-1}) = \xi(f^{\tilde{\xi}_0(R_0),\ldots\tilde{\xi}_{m-1}(R_{m-1})}(\xi_0(x_0), \ldots, \xi_{\ell-1}(x_{\ell-1})))$$

for any inputs $(x_0, \ldots, x_{\ell-1})$ and randomness $R_0, \ldots, R_{m-1}$.

Berndt et al. [BEF+23] considered these notions of fault-resilient (S)NI, due to their composition property summarized in the following lemma.

Lemma 1 (Composition of frSNI [BEF+23, Thm. 1])**.** *The composition rules for* (S)NI *also apply to* d-fr(S)NI *and* d-wfr(S)NI.

2.2 Background

In this paper, we focus on packed secret sharing [FY92], which is a generalization of Shamir's secret sharing [Sha79]. The following section summarizes both Shamir's and the packed secret sharing.

Shamir's Secret Sharing. In Shamir's secret sharing scheme [Sha79], a secret v_0 is encoded by constructing a polynomial $p(X)$ of degree d, where the constant term $p_0 = v_0$, and the remaining coefficients $p_1, \ldots, p_d$ are chosen uniformly at random from $\mathbb{F}^d$. The resulting polynomial is of the form $p(X) = p_0 + p_1 X + \cdots + p_d X^d$. The secret shares $F_0, \ldots, F_{n-1}$ of v_0 are generated by evaluating $p(X)$ at n predetermined and pairwise distinct non-zero points $x_0, \ldots, x_{n-1} \neq 0$. Denoting $(p_0, p_1, \ldots, p_d)^T$ by $\boldsymbol{p}$, we obtain $(F_0, \ldots, F_{n-1})^T \coloneqq \mathsf{mask}(\boldsymbol{c}) = \mathsf{Van}_{n\times(d+1)}(x_0, \ldots, x_{n-1}) \cdot \boldsymbol{p}$. Similarly, we can unmask, i.e., reconstruct $\boldsymbol{p}$ from the shares $F_0, \ldots, F_{n-1}$ by computing $\boldsymbol{p} = \mathsf{unmask}(F_0, \ldots, F_d) \coloneqq \mathsf{Van}(x_0, \ldots, x_d)^{-1} \cdot (F_0, \ldots, F_d)^T$. When aiming for error detection, we can unmask with the help of all the shares $F_i, i \in [n]$ and polynomial interpolation (cf. [Sha79, ABEO24]).

Similarly, as Arnold et al. [ABEO24] describe, two secrets, v_0 and v_d, can be embedded in a polynomial by placing them in the lowest and highest coefficients. However, [LD04] found out that using a middle coefficient to embed a secret risks revealing this value, which makes it non-trivial to extend the idea of Arnold et al. [ABEO24]. Hence, embedding $k > 2$ secrets requires an adaptation of the secret sharing scheme.

Packed Secret Sharing. Packed secret sharing [FY92] is a generalization of Shamir's secret sharing. Instead of considering only one secret v_0, we take k ($\leq d$) secrets $v_0, \ldots, v_{k-1}$ into account. Consider $d+1$ pairwise distinct, non-zero evaluation points $u_0, \ldots, u_d$. We construct a polynomial $p(X) = p_0 + p_1 X + \ldots + p_d X^d$ of degree d by setting $p(u_0) = v_0, \ldots, p(u_{k-1}) = v_{k-1}$. Moreover, we sample $d+1-k$ elements $v_k, \ldots, v_d$ uniformly at random and set $p(u_k) = v_k, \ldots, p(u_d) = v_d$. This determines the polynomial $p(X)$ completely. We call $\boldsymbol{v} = (v_0, \ldots, v_{k-1}, v_k, \ldots, v_d)^T$ with $\mathsf{Van}^{-1}(u_0, \ldots, u_d) \cdot (p_0, \ldots, p_d)^T = \boldsymbol{v}$. Now,

we consider more pairwise distinct, non-zero support points $x_0, \ldots, x_{n-1}$, and obtain the desired shares $(F_0, \ldots, F_{n-1})$ by calculating $(F_0, \ldots, F_{n-1})^T = \mathsf{Van}_{n\times(d+1)}(x_0, \ldots, x_{n-1}) \cdot (p_0, \ldots, p_d)^T$. To simplify notation, we call $U = \mathsf{Van}_{(d+1)\times(d+1)}(u_0, \ldots, u_d)$ and, additionally, $V = \mathsf{Van}_{n\times(d+1)}(x_0, \ldots, x_{n-1})$. With $M \coloneqq V \cdot U^{-1}$, we have $\mathsf{mask}(\boldsymbol{v}) \coloneqq (F_0, \ldots F_{n-1})^T = M \cdot \boldsymbol{v}$. We can then reconstruct $\boldsymbol{v}$ by computing, $\mathsf{unmask}(F_0, \ldots F_{n-1}) \coloneqq M^{-1} \cdot (F_0, \ldots F_{n-1})^T$. For further details on these calculations, we refer to Supp. B and [FOWZ25]. Note that the security of packed secret sharing relies strongly on the following lemma from [FY92].

Lemma 2 ([FY92]). *For a random degree d packed secret sharing of a vector $\boldsymbol{v} = (v_0, ..., v_{k-1}, r_k, ..., r_d)$, any $d + 1 - k$ shares are independent of the secret $(v_0, ..., v_{k-1})$.*

Furthermore, packed secret sharing guarantees that share-wise added (or multiplied) shares of two packed secret sharings embed the added (or multiplied) secrets at the same support points as the input shares [FY92]. In more detail, packed secret sharing is a linear homomorphism, meaning that for all $d \geq k - 1$ and $\boldsymbol{v}, \boldsymbol{v'} \in \mathbb{F}^d$, $\mathsf{mask}(\boldsymbol{v} + \boldsymbol{v'}) = \mathsf{mask}(\boldsymbol{v}) + \mathsf{mask}(\boldsymbol{v'})$. Furthermore, packed secret sharing satisfies multiplicativity: for all $d \geq k - 1$ and $2d < n$, and for all $\boldsymbol{v}, \boldsymbol{v'} \in \mathbb{F}^d$, $\mathsf{mask}(\boldsymbol{v} * \boldsymbol{v'}) = \mathsf{mask}(\boldsymbol{v}) * \mathsf{mask}(\boldsymbol{v'})$, where $*$ denotes the share-wise multiplication operation between two vectors.

3 Improving Computational Complexity of Zero-Encodings

A fundamental primitive used in polynomial-masked circuits is an algorithm that generates zero-encodings. Zero-encodings are sharings $(E_i)_{i\in[n]}$ that embed the secret vector $(v_0 = 0, \ldots, v_{k-1} = 0, v_k, \ldots v_d)$, where $v_j \leftarrow_{\$} \mathbb{F}$, $j \in [k, d + 1]$, are any elements in $\mathbb{F}$. The core idea is that such a primitive allows for internal re-randomization of the computation without affecting its correctness. For example, any encoding $(F_i)_{i\in[n]}$ can be re-randomized by securely adding a zero-encoding $(E_i)_{i\in[n]}$, such that the final output $(F_i + E_i)_{i\in[n]}$ remains the encoding of the same secret as $(F_i)_{i\in[n]}$, but with freshly generated randomness. This feature is often used in security proofs to prevent the accumulation of leaked values. This section explores the efficient and secure generation of zero-encodings. In particular, we introduce a method in Section 3.1 to improve the overall computational complexity of the zero-encoding gadget **ZEnc** from [FOWZ25]. Then in Section 3.2, we modify the construction to enable a highly parallelizable implementation. These improvements are presented with respect to packed secret sharing [FY92] to embed multiple secrets simultaneously in a single encoding.

3.1 Reducing Computational Complexity of Zero-Encodings

In [BEF+23, FOWZ25], the zero-encoding gadget **ZEnc** was constructed by first choosing the secret vector $\boldsymbol{v}$ of the underlying polynomial $p(X)$ and afterward computing the shares $E_0, \ldots, E_{n-1}$ using polynomial $p(X)$. Specifically, a zero-encoding $\boldsymbol{E} = (E_0, \ldots, E_{n-1})^T$ embeds a secret vector $\boldsymbol{v} = (v_0 = 0, \ldots, v_{k-1} = 0, v_k, \ldots, v_d)$, where $(v_k, \ldots, v_d) \leftarrow_{\$} \mathbb{F}^{d+1-k}$. The shares are generated as $\boldsymbol{E} = \mathsf{mask}(v_0, \ldots, v_d) = M \cdot \boldsymbol{v}$ such that it holds for each E_i that $E_i = p(x_i)$ with pairwise distinct support points $x_0, \ldots, x_{n-1}$. Since each share is computed individually with $E_i = \sum_{j=\{k,\ldots,d\}} M_{i,j} v_j$, the overall computation of the gadget results in a computational complexity of $\mathcal{O}(n \cdot (d + 1 - k))$. Assuming $n = d + 1 + e$, this equals a computational complexity of $\mathcal{O}((t + e + k) \cdot t)$, since Faust et al. [FOWZ25] proved that they support up to $t = d + 1 - k$ probes.

We can reduce the computational complexity of the zero-encoding gadget **ZEnc** from [BEF+23, FOWZ25] as follows. Instead of choosing the $d + 1 - k$ *values* in $\boldsymbol{v}$, i.e., $v_k, \ldots, v_d$, uniformly at random, this time, we directly set the $d + 1 - k$ output *shares*,

i.e., $p(x_0), \ldots, p(x_{d-k})$, to $d+1-k$ uniformly random values. This approach avoids the computation of the output shares, $p(x_0), \ldots, p(x_{d-k})$. Together with $v_0 = 0, \ldots, v_{k-1} = 0$, this method fixes $d+1$ pre-determined values, thereby fixing the underlying degree-d polynomial $p(X)$.

In order to obtain a total of n output shares, we now need to compute the remaining $n-(d+1-k)$ shares, i.e., $p(x_{d+1-k}), \ldots, p(x_{n-1})$. As described in Section 2.2, this can be done using the Vandermonde matrix and its inverse for polynomial evaluation and interpolation. We denote by $A'_{(n+k-d-1)\times(d+1)}$ the matrix $\mathsf{Van}_{(n+k-d-1)\times(d+1)}(x_{d+1-k}, \ldots, x_{n-1}) \cdot \mathsf{Van}^{-1}_{(d+1)\times(d+1)}(u_0, \ldots, u_{k-1}, x_0, \ldots, x_{d-k})$ and get

$$(p(x_{d+1-k}), \ldots, p(x_{n-1}))^T = A'_{(n+k-d-1)\times(d+1)} \cdot (p(u_0), \ldots, p(u_{k-1}), p(x_0), \ldots, p(x_{d-k}))^T.$$

Note that we can use matrix $A'_{(n+k-d-1)\times(d+1)}$ to mask any secrets $(v_0, \ldots, v_k)$. Namely, set $v_0 = p(u_0), \ldots v_{k-1} = p(u_{k-1})$ and choose $p(x_i) \leftarrow_{\$} \mathbb{F}$ for $i \in [d+1-k]$. If the $p(x_i), i \in [d+1-k, n]$ are calculated via Equation 1, then $F_i = p(x_i)$ for all $i \in [n]$ yields a valid sharing of $(v_0, \ldots, v_{k-1})$. A detailed description and proofs are provided in the full version.

However, since we aim to focus on zero-encodings, the secrets $p(u_0), \ldots, p(u_{k-1})$ are all zeros. As a result, the computation of masking k zeros can be further simplified by removing the first k columns of $A'_{(n+k-d-1)\times(d+1)}$. This results in a smaller matrix $A_{(n+k-d-1)\times(d+1-k)}$, expressed as $(p(x_{d+1-k}), \ldots, p(x_{n-1}))^T = A_{(n+k-d-1)\times(d+1-k)} \cdot (p(x_0), \ldots, p(x_{d-k}))^T$, where the $(n+k-d-1)\times(d+1-k)$ matrix $A_{(n+k-d-1)\times(d+1-k)}$ is composed of elements $a_{i,j+k}$ from the above matrix $A'_{(n+k-d-1)\times(d+1)}$ for $i \in [n+k-d-1]$ and $j \in [d+1-k]$.

Let $\boldsymbol{r} = (r_0, \ldots, r_{d-k}) \leftarrow_{\$} \mathbb{F}^{d+1-k}$ and set $p(x_0), \ldots, p(x_{d-k})$ to be the elements of $\boldsymbol{r}$, i.e., $p(x_i) = r_i$. To compute a zero-encoding, we can now map the random values to themselves by using the identity matrix $I_{[d+1-k]\times[d+1-k]}$ for the shares $F_0, \ldots, F_{d-k}$ and $A_{(n+k-d-1)\times(d+1-k)}$ to obtain the remaining corresponding shares:

$$(F_0, \ldots, F_{n-1})^T = \begin{pmatrix} I_{[d+1-k]\times[d+1-k]} \\ A_{(n+k-d-1)\times(d+1-k)} \end{pmatrix} \cdot (r_0, \ldots, r_{d-k})^T =: \tilde{A} \cdot \boldsymbol{r}. \tag{1}$$

More details on this construction are presented in Supp. B.

To show that the matrix $\tilde{A}$ is indeed a generator matrix for zero-encodings, we prove that the set of all valid zero-encodings with respect to k secrets is a sub-vector space of $\mathbb{F}^n$.

Lemma 3. *Let $\boldsymbol{Z}$ denote the set of all valid zero-encodings with respect to k underlying secrets embedded in polynomials of degree d with coefficients in $\mathbb{F}$, while considering n shares. Then $\boldsymbol{Z}$ is a sub-vector space of $\mathbb{F}^n$.*

Proof sketch. This result directly follows from the linear homomorphism property and multiplicativity of packed secret sharing [FY92]. Namely, share-wise added (resp. multiplied) shares of two packed secret sharings embed the added (resp. multiplied) secrets at the same support points as the input shares. In Lemma 19 in App. A, we show that the linear combination of two zero-encodings is again a zero-encoding. As we consider vectors of size n containing elements from the field $\mathbb{F}$, this concludes the proof. □

Now, we can show that the columns of matrix $\tilde{A}$ form a basis of this sub-vector space $\boldsymbol{Z}$. In other words, multiplying the generator matrix $\tilde{A}$ with any uniformly distributed vector $\boldsymbol{r} = (r_0, \ldots, r_{d-k})^T \leftarrow_{\$} \mathbb{F}^{d+1-k}$ results in a uniformly distributed zero-encoding $(F_0, \ldots, F_{n-1})$.

Theorem 1. *Let $\boldsymbol{Z}$ denote the set of all valid zero-encodings with respect to k secrets, n shares and polynomials of degree d with coefficients in $\mathbb{F}$. Moreover, $\tilde{A}$ denotes the matrix defined in Equation 1. Then the columns of $\tilde{A}$ form a basis of $\boldsymbol{Z}$.*

Proof. We choose some of the output shares, i.e., the output shares $p(x_0), \ldots, p(x_{d-k})$ uniformly at random. These uniformly random output shares $p(x_0), \ldots, p(x_{d-k})$ and the secrets $p(u_0) = 0, \ldots, p(u_{k-1}) = 0$ describe a uniform random polynomial $p(x)$. As the generator matrix $\tilde{A}$ simply computes the function values of this uniform random polynomial p for $F_0 = p(x_0), \ldots, F_{d-k} = p(x_{d-k})$ and $F_{d+1-k} = p(x_{d+1-k}), \ldots, F_{n-1} = p(x_{n-1})$ (cf. Eq. 1) the resulting shares are $d+1-k$-wise independent and thus yield an uniformly random zero-encoding (cf. [FY92]). This implies that $\boldsymbol{Z}$ is spanned by the columns of $\tilde{A}$. Moreover, the columns of the matrix $\tilde{A}$ are linearly independent, as the upper $(d+1-k) \times (d+1-k)$ sub-matrix of $\tilde{A}$ is the identity. Since $d+1$ elements are needed to fix a degree d polynomial, the sub-vector space $\boldsymbol{Z}$ has dimension $d+1$. Hence, the columns of $\tilde{A}$ form (one of) the smallest set that spans $\boldsymbol{Z}$ and thus are a basis of $\boldsymbol{Z}$. □

Using Theorem 1, we can show that each column of $\tilde{A}$ is a zero-encoding.

Corollary 1. *Consider $\tilde{A}$ as described above (cf. Equation 1). Each column $i \in [d+1-k]$ of $\tilde{A}$, $\tilde{A}_{:,i}$, is a zero-encoding, i.e., each $\tilde{A}_{:,i}$ embeds $\boldsymbol{v} = (0, \ldots, 0, v_k, \ldots, v_d)$ with some values $v_k, \ldots, v_d \in \mathbb{F}$ and support points $u_0, \ldots, u_d$.*

Proof. This follows from Theorem 1 with $r_j = 0$ for $j \in [d-k+1]/\{i\}$ and $r_i = 1$. □

In this construction, the first $d+1-k$ output shares, i.e., $p(x_0), \ldots, p(x_{d-k})$, are chosen uniformly at random. Thus, only the last $e+k$ shares need to be computed with respect to the previously fixed coefficients, where $n = d+1+e$ and $d = t+k-1$. We denote our optimized gadget for zero-encoding by opt-**ZEnc** (cf. Alg. 1 for the case where $\mathsf{Offset} = 0$).

We thus reduce the computational complexity from $\mathcal{O}((t+e+k) \cdot t)$ to $\mathcal{O}((e+k) \cdot t)$.

3.2 Improvement through Parallelizing Zero-Encodings

In Section 3.1, we discussed how to reduce the *overall* computational complexity of zero-encodings constructed using the **ZEnc** gadget presented in [BEF+23, FOWZ25]. We can further enable a better parallelization of the computation of n output shares by distributing the total computational workload *equally* across the computation of each share. Even though the overall complexity remains the same, by parallelizing the calculation of the shares, we can reduce the *actual* computational complexity. In terms of circuit design, we reduce the depth of a circuit without altering its total size.

The idea is that since all valid zero-encodings are spanned by the columns of $\tilde{A}$, we can use $\tilde{A}$ to construct another basis that balances the workload across the computation of each of the shares. We use the fact that as any multiplication with zero results in zero, all zero multiplications do not need to be computed. Similarly, additions involving zeros can be skipped, as the matrix $\tilde{A}$ is public. To balance the computational workload, we aim to distribute the zeros of matrix $\tilde{A}$ (cf. Equation 1) such that the number of zeros in each row is as equal as possible while requiring the columns to still form a basis of $\boldsymbol{Z}$. Even though this does not bring a performance improvement for sequential software implementations, parallel software implementations can be sped up massively. Furthermore, and more importantly, this redistribution of the computations allows for a much shorter critical path in hardware implementations of our countermeasure.

3.2.1 Equally Distributing Zero Entries in Matrix $\tilde{A}$

Recall that in Equation 1, we generate a zero-encoding $(F_0, \ldots, F_{n-1})^T$ by multiplying the matrix $\tilde{A}$ with a vector of $d+1-k$ random values $r_0, \ldots, r_{d-k}$. Due to the structure of the matrix $\tilde{A}$, the first $d+1-k$ output shares $F_0, \ldots, F_{d-k}$ are *directly* set to the $d+1-k$ random values $r_0, \ldots, r_{d-k}$. Hence, we only need to compute the remaining $n+k-d-1$ output shares $F_{d+1-k}, \ldots, F_{n-1}$. Note that while the upper part of the

matrix is the identity matrix, the lower part, $A_{(n+k-d-1)\times(d+1-k)}$, only contains non-zero elements (for the proof, refer to Lemma 20 in App. A). Hence, the first output shares are cheap to compute, while the latter are much more expensive. To balance the computation more evenly between different shares (to reduce, e.g., the length of the critical path in a hardware implementation), we transform the matrix $\tilde{A}$ into a more balanced matrix $\tilde{A}_{\mathsf{opt}}$. In the following, we provide an intuition of how to transform $\tilde{A}$ into $\tilde{A}_{\mathsf{opt}}$ via Algorithm $\mathsf{Algo}_{\mathsf{MatrixTransformation}}$ (cf. Alg. 14), for a more concrete example refer to Supp. C.1.

To distribute the computational cost *equally* during the computation of all n output shares, all rows should have a similar number of zeros. Because the bottom part of $\tilde{A}$ contains no zero elements, we start with creating zeros there. The idea, as illustrated in Figure 2, is to combine two columns such that the result will contain a new zero at a specific position in the lower half: Consider the elements of the two columns at this position, denoted in the figure by a for the left column and b for the right one. Now, by subtracting the a-th multiple of the right column from the b-th multiple of the left column, the resulting column will have a zero element at this position.

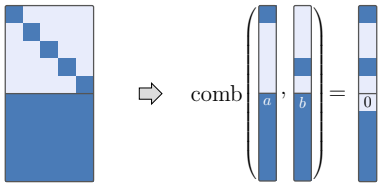


Figure 2: Given the matrix on the left, where the lighter color indicates a zero element and the darker color indicates a non-zero element, we can combine two columns to create a zero element in the lower half.

Creating multiple zeros in the bottom half can be achieved by repeating this step multiple times using more columns. To be precise, to create m many zeros, $m+1$ columns are needed. First, the columns are combined in a pairwise manner to set the first position to zero, then the resulting columns can be combined to create another zero element and so on, until the desired number of zeros is reached. This is illustrated in Figure 3. Following Lemma 3, the computed column is again a zero-encoding. Note that since the top part of the matrix is the identity matrix, every combination will result in one less zero element in the top part of the column. We will also refer to this as *shifting* the zeros from the top to the bottom part. Since the bottom part of the matrix contains no zero elements, this provides an upper bound on the number of zeros that can effectively be redistributed using linear combinations.

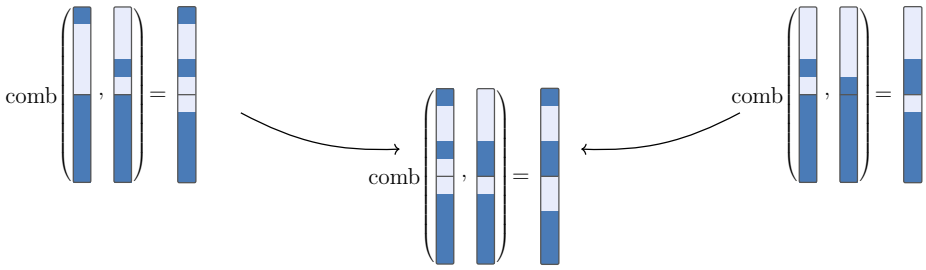


Figure 3: Three columns can be used to create two zeros in the resulting column, this is accompanied by two non-zero elements being created in the upper half of the result.

Repeating the above computation multiple times, the optimized matrix $\tilde{A}_{\mathsf{opt}}$ is then made up of the resulting columns. To achieve the goal of distributing the zeros, for each

computation, a different set of columns is used and the zeros are created at different positions. The exact parameters are dependent on the number $\mathsf{Row}_{\mathsf{Zeros}}$ of zeros each row should (approximately) have. If we use the i-th column of $\tilde{A}$ to compute a new column, the i-th element of the result will be non-zero. Hence, to achieve $\mathsf{Row}_{\mathsf{Zeros}}$ in the i-th row of $\tilde{A}_{\mathsf{opt}}$, the i-th column must not be used at least $\mathsf{Row}_{\mathsf{Zeros}}$ times. To solve this, we choose to use all but the $(i+1)$-th to the $(i+\mathsf{Row}_{\mathsf{Zeros}})$-th column to compute the optimized i-th column. This results in the cascading shape of the top part as depicted in Figure 4. To create equally distributed zeros in the bottom of the matrix, we circle through the positions where the zero element should be created and stop once we reach the desired number of zeros, producing a similar cascade in the bottom of the matrix.

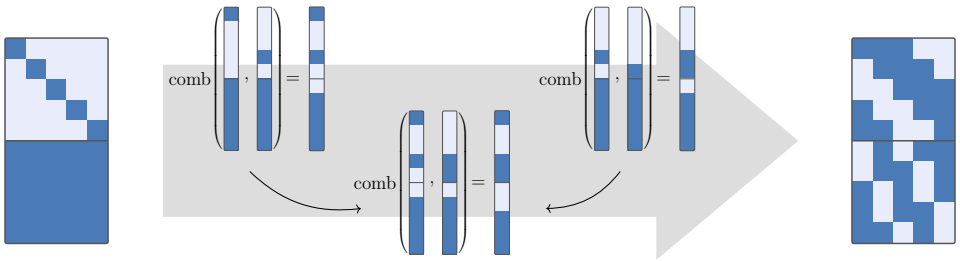


Figure 4: Visualization of $\mathsf{Algo}_{\mathsf{MatrixTransformation}}$ transforming $\tilde{A}$ on the left into $\tilde{A}_{\mathsf{opt}}$ on the right through multiple steps of computing linear combinations for each column.

Note that since all columns of $\tilde{A}_{\mathsf{opt}}$ are computed separately of each other, depending on the used support points, the resulting matrix of these steps might not be a basis of $\boldsymbol{Z}$. Hence, our algorithm has to include a check for this case and stops earlier if this case occurs. For a further discussion on this, refer to Supp. C.4. The concrete algorithm $\mathsf{Algo}_{\mathsf{MatrixTransformation}}$ is presented in Supp. C as well as all proofs including proving that the algorithm indeed balances the number of zeros per row.

3.2.2 Optimized Zero-Encoding Gadget

The resulting gadget for zero-encodings $\mathsf{opt\text{-}ZEnc}_n^{d,\mathsf{Offset}}$ is outlined in Alg. 1, where the optimizations described in Section 3.1 are highlighted in **purple**. It already includes the optimizations explained below in Section 4 (highlighted in **blue**). For now and whenever Offset is not explicitly specified for $\mathsf{opt\text{-}ZEnc}_n^{d,\mathsf{Offset}}$, we assume $\mathsf{Offset} = 0$. Please note that the computation of matrix $\tilde{A}_{\mathsf{opt}}$ (or $\tilde{A}$) in Alg. 1 depends on the degree d, number of shares n, and the number of embedded secrets k of the zero encodings we wish to generate. Besides, matrix $\tilde{A}_{\mathsf{opt}}$ (or $\tilde{A}$) can be pre-computed to improve the performance of Alg. 1.

The following lemma directly follows from the commutativity of addition and the linearity of constant multiplication.

Lemma 4. *Gadget* $\mathsf{opt\text{-}ZEnc}$ *(cf. Alg. 1) is generalized-fault-invariant with respect to* $\mathcal{F}^+$.

To guarantee fault resilience, it is sufficient to show that every change to the output of gadget $\mathsf{opt\text{-}ZEnc}$ introduced by s internal faults can be achieved by s faults on the output.

In order to simplify the proofs for e–fault robustness of the complete multiplication, we consider e–fault robustness of our gadget $\mathsf{opt\text{-}ZEnc}$ in the following Lemma.

Lemma 5. *The gadget* $\mathsf{opt\text{-}ZEnc}$ *(cf. Algorithm 1) is e-fault-robust with respect to* $\mathcal{F}^+$.

Proof. We consider the gadget step by step to show that for any additive faults injected during the gadget $\mathsf{opt\text{-}ZEnc}$, Definition 4 is fulfilled. Importantly, since $\mathsf{opt\text{-}ZEnc}$ does

Algorithm 1 opt-**ZEnc**$_n^{d,\mathsf{Offset}}$ for $n = d + e + 1$ and $k \leq d$

Output: Randomized sharing $(G_i)_{i\in[n]}$ of zeros $\boldsymbol{v}_{\mathsf{zero}} = (0_0, \ldots, 0_{k-1}, r_k, \ldots, r_d)$ where $r_k, \ldots, r_d \leftarrow_{\$} \mathbb{F}$

1 initialize $(G_i)_{i\in[n]}$
2 $G_0, \ldots, G_{\mathsf{Offset}-1} \leftarrow 0$
3 $(G_{\mathsf{Offset}}, \ldots, G_{d-k}) \leftarrow_{\$} \mathbb{F}^{d+1-(k+\mathsf{Offset})}$
4 **for** $i \in \{d+1-k, \ldots, n-1\}$ **do**
5 **for** $j \in \{\mathsf{Offset}, \ldots, d-k\}$ **do**
6 $G_i \leftarrow G_i + G_j \cdot \tilde{A}_{\mathsf{opt}}[i,j]$
7 **return** $(G_i)_{i\in[n]}$

Algorithm 2 **ZEnc**$_n^d$ for $n = d+e+1$ and $k \leq d$ [FOWZ25, Algorithm 3]

Output: Randomized sharing $(G_i)_{i\in[n]}$ of zeros $\boldsymbol{v}_{\mathsf{zero}} = (0_0, \ldots, 0_{k-1}, r_k, \ldots, r_d)$ where $r_k, \ldots, r_d \leftarrow_{\$} \mathbb{F}$

1 initialize $(G_i)_{i\in[n]}$
2 $(r_k, \ldots, r_d) \leftarrow_{\$} \mathbb{F}^{d+1-k}$
3 **for** $i \in \{0, \ldots, n-1\}$ **do**
4 **for** $j \in \{k, \ldots, d\}$ **do**
5 $G_i \leftarrow G_i + r_j M_{j,i}$
6 **return** $(G_i)_{i\in[n]}$

not have any inputs, we set $\boldsymbol{\omega}' = \mathbf{0}$. Any fault injected in share G_i in line 2 only affects output share G_i, as the elements $G_i, i \in [0, \mathsf{Offset}]$ are not used within any calculation of the gadget. Furthermore, a fault injected to line 3 of the gadget does not change the underlying distribution of the $G_i, i \in [\mathsf{Offset}, d-k+1]$, since $G_i + \xi_i$ remains uniformly distributed for any uniformly distributed element G_i, and wire-independent additive fault $\xi_i \in \mathcal{F}^+$. We call these ineffective faults randomness faults. Any fault performed to line 6, i.e., the calculation of $G_i \leftarrow G_i + G_j \cdot \tilde{A}_{\mathsf{opt}}[i,j]$ only affects output share G_i. Last but not least, any fault directly injected to an output share only affects this output share. Let s_1 denote the number of internally injected faults. By assumption, $s_1 \leq e$. Thus, setting $\boldsymbol{\omega}$ equal to the vector of internally injected faults except for the randomness faults, we obtain $\mathsf{weight}(\boldsymbol{\omega}) \in [0, s_1]$. This finishes the proof. □

4 Reducing Randomness Complexity of Refresh

An encoding can be refreshed by adding zero-encodings to it. However, prior work [BEF+23] showed that using one zero-encoding gadget **ZEnc** (cf. Alg. 2) on its own is not sufficient to generate a zero-encoding for an SNI refresh. Instead, one needs to generate $d+1-k$ different zero-encodings with **ZEnc** and add them sequentially to the input encoding of a refresh gadget. We call the gadget that adds these $d+1-k$ independently generated outputs of **ZEnc** a *strong* zero-encoding gadget, **sZEnc** (cf. Alg. 4). Prior work, e.g., [BEF+23], constructed an SNI refresh gadget **Refresh** (cf. Alg. 6) using this strong zero-encoding generator. However, this **sZEnc** requires a large amount of independent uniformly random values, which is one of the major overheads of those private circuits. To overcome this, we present a method to reduce the randomness complexity of **Refresh** (and all other gadgets used in Section 5) by 50% without affecting the combined resilience. We begin by explaining the modifications made to the zero-encoding gadgets **ZEnc** and **sZEnc**, denoted as opt-**ZEnc** (cf. Alg. 1) and opt-**sZEnc** (cf. Alg. 3), before continuing with the application to gadget **Refresh**. The details of **ZEnc** and **sZEnc** (e.g., the construction of matrix M and security proofs) can be found in [FOWZ25]. In the following updated algorithms, all changes are highlighted in **blue**.

The original gadget **ZEnc** outputs zero-encodings using $d+1-k$ random values, and gadget **sZEnc** calls **ZEnc** $d+1-k$ times, using $(d+1-k)^2$ random values in total. In our optimized gadget opt-**ZEnc**, the amount of randomness used within is reduced depending on the time of the call within opt-**sZEnc**, meaning that $j \in [d+1-k]$ fewer random values are needed for the j-th call of opt-**ZEnc**. In total, opt-**sZEnc** thus reduces

randomness usage by 50%, i.e., to $(d+1-k)^2/2$. The following part explains the high-level idea for randomness reduction in opt-**sZEnc**.

The gadget **sZEnc** (cf. Alg. 4) is the share-wise sum of $d+1-k$ zero-encodings. Thus, using the representation of zero-encodings as presented in Section 3, we can express the algorithm of **sZEnc** as the following matrix multiplication

$$\begin{pmatrix} Y_0 \\ \vdots \\ Y_{n-1} \end{pmatrix} \coloneqq R \cdot 1_{d+1-k} = \begin{pmatrix} r_0^{(0)} & \dots & r_0^{(d-k)} \\ \vdots & \ddots & \vdots \\ r_{d-k}^{(0)} & \dots & r_{d-k}^{(d-k)} \\ \sum_{i=0}^{d-k} \tilde{A}_{d+1-k,i} r_i^{(0)} & \dots & \sum_{i=0}^{d-k} \tilde{A}_{d+1-k,i} r_i^{(d-k)} \\ \vdots & \ddots & \vdots \\ \sum_{i=0}^{d-k} \tilde{A}_{n-1,i} r_i^{(0)} & \dots & \sum_{i=0}^{d-k} \tilde{A}_{n-1,i} r_i^{(d-k)} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix},$$

where $r_i^{(j)}$ is the i-th random value generated in the j-th call within **sZEnc** to **ZEnc**.

Now, we reduce the randomness used in gadget **sZEnc** by using gadget opt-**sZEnc** (cf. Alg. 3) and changing the matrix R to a lower triangular matrix R' as

$$R' \coloneqq \begin{pmatrix} r_0^{(0)} & 0 & \dots & 0 \\ r_1^{(0)} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ r_{d-k}^{(0)} & \dots & \dots & r_{d-k}^{(d-k)} \\ \sum_{i=0}^{d-k} \tilde{A}_{d+1-k,i} r_i^{(0)} & \dots & \dots & \tilde{A}_{d+1-k,d-k} r_{d-k}^{(d-k)} \\ \vdots & \ddots & \ddots & \vdots \\ \sum_{i=0}^{d-k} \tilde{A}_{n-1,i} r_i^{(0)} & \dots & \dots & \tilde{A}_{n-1,d-k} r_{d-k}^{(d-k)} \end{pmatrix}. \tag{2}$$

Following [BEF+23], we assume all elements of a column are revealed when probing one element of this column of R'. Then, intuitively, for t ($\leq d+1-k$) probes, this reveals *at most* the first t elements of $R' \cdot 1_{d+1-k}$, while the remaining elements are still re-randomized and stay $(d+1-k-t)$-wise independent.

Algorithm 3 opt-$\mathbf{sZEnc}_n^d$ for $n = d+e+1$ and $k \leq d$

Output: Randomized sharing $(Y_i)_{i\in[n]}$ of zeros $\boldsymbol{v}_{\mathsf{zero}} = (0_0, \dots, 0_{k-1}, r_k, \dots, r_d)$ where $r_k, \dots, r_d \leftarrow_{\$} \mathbb{F}$

initialize $(Y_i)_{i\in[n]}$
for $j \in \{0, \dots, d-k\}$ **do**
  $\left(G_i^{(j)}\right)_{i\in[n]} \leftarrow$ opt-$\mathbf{ZEnc}_n^{d,\mathsf{Offset}\leftarrow j}$
  $(Y_i)_{i\in[n]} \leftarrow (Y_i)_{i\in[n]} + \left(G_i^{(j)}\right)_{i\in[n]}$
return $(Y_i)_{i\in[n]}$

Algorithm 4 $\mathbf{sZEnc}_n^d$ for $n = d+e+1$ and $k \leq d$ [FOWZ25, Algorithm 4]

Output: Randomized sharing $(y_i)_{i\in[n]}$ of zeros $\boldsymbol{v}_{\mathsf{zero}} = (0_0, \dots, 0_{k-1}, r_k, \dots, r_d)$ where $r_k, \dots, r_d \leftarrow_{\$} \mathbb{F}$

initialize $(Y_i)_{i\in[n]}$
for $j \in \{0, \dots, d-k\}$ **do**
  $(G_i)_{i\in[n]} \leftarrow \mathbf{ZEnc}_n^d$
  $(Y_i)_{i\in[n]} \leftarrow (Y_i)_{i\in[n]} + (G_i)_{i\in[n]}$
return $(Y_i)_{i\in[n]}$

Note that the correctness of the gadget opt-**sZEnc** follows directly from the fact that $\tilde{A}$ is a basis of the set of all zero-encodings (cf. Theorem 1), and by the correctness of share-wise addition. In the following, we give the updated security proofs of gadget opt-**sZEnc**.

Lemma 6 (Probing resilience of opt-sZEnc_n^d)**.** *For any probing set P with $|P| = d' \leq d+1-k$ probes, it holds that for $(Y_i)_{i\in[n]} \leftarrow$ opt-sZEnc_n^d, there exists a subset $I \subset [n]$ with $|I| = n - d'$ such that the values $(Y_i)_{i\in I}$ are $(d+1-k-d')$-wise uniformly independent conditioned on the probes. More formally, denote with $\mathcal{P}$ the distribution of the d' probed values in opt-sZEnc_n^d, and with $\mathcal{Y}$ the joint distribution of any $(d+1-k-d')$-tuple of distinct outputs Y_i such that $i \in I$. Then for any $\mathbf{y} \in \mathbb{F}^{d+1-k-d'}$ and any $\mathbf{p} \in \mathbb{F}^{d'}$ such that $\Pr[\mathcal{P} = \mathbf{p}] \neq 0$, it holds $\Pr[\mathcal{Y} = \mathbf{y} | \mathcal{P} = \mathbf{p}] = \frac{1}{|\mathbb{F}|^{d+1-k-d'}}$.*

Proof. We begin by constructing the set I^c and then set $I = [n] \setminus I^c$. We initialize $I^c \leftarrow \{\emptyset\}$ and include indices based on the probes in P. To this end, we consider two groups of probes:

- Group K: $K = \bigcup_{l\in[d+1-k]} K^l$, where K^l consists of indices of leaked elements during the computation of opt-$\mathsf{ZEnc}_n^{d,l}$ (cf. line 3 of Alg. 1), i.e., internal probes during the computation of $\left(G_i^{(l)}\right)_{i\in[n]} \leftarrow$ opt-$\mathsf{ZEnc}_n^{d,l}$ and probes of $\left(G_i^{(l)}\right)_{i\in[n]}$. For each probe in K^l, we include l in I^c.
- Group I: All probes of $Y_i^{(l)}, i \in [n], l \in [d+1-k]$, namely the set $I = \bigcup_{l\in[d+1-k]} I^{(l)}$, where $I^{(l)} = \{I_0^{(l)}, \ldots, I_{n-1}^{(l)}\}$ with $I_i^{(l)} \coloneqq Y_i^{(l)} = \sum_{m=0}^{l} G_i^{(m)}$, and $0 \leq l \leq d-k$ is the l-th loop iteration of opt-sZEnc_n^d in line 4. For each probe of $Y_i^{(l)}$, we include i in I^c.

Notably, the i-th output Y_i equals $Y_i^{(l)}$ for every l such that $\min\{i, d-k\} \leq l \leq d-k$, and can be rewritten as $Y_i = Y_i^{(i)} = Y_i^{(i^*-1)} + G_i^{(i^*)} + \sum_{j=i^*+1}^{i} G_i^j$ for every $i^* \leq i$. Let $I = (i_0, \ldots, i_{n-d'})$ such that $i_0 < \ldots < i_{n-d'}$. Using the above with $i^* = i_0$, we can rewrite $(Y_{i_0}, \ldots, Y_{i_{n-d'}})$ as follows:

$$\left(Y_{i_0}^{(i_0-1)}, Y_{i_1}^{(i_0-1)} + \sum_{j=i_0+1}^{i_1} G_i^j, \ldots, Y_{i_{n-d'}}^{(i_0-1)} + \sum_{j=i_0+1}^{i_{n-d'}} G_i^j \right) \tag{3}$$

$$+ (G_{i_0}^{(i_0)}, G_{i_1}^{(i_0)}, \ldots, G_{i_{n-d'}}^{(i_0)}). \tag{4}$$

To conclude the proof, we need to prove that $(Y_{i_0}, Y_{i_1}, \ldots, Y_{i_{n-d'}})$ is $(d+1-k-d')$-wise independent conditioned on the probes. This holds if we prove that the tuple in line 4 is $(d+1-k-d')$-wise independent and independent of the tuple in line 3 conditioned on the probes. First, we observe that the tuple in line 4 is independent of the one in line 3, which is computed ignoring the i_0-th opt-ZEnc execution. Moreover, the i_0-th execution of sZEnc produces $(d-k+1-i_0)$-wise uniformly independent values. We observe that $i_0 \in I$ implies that the probes are not internal to the i_0-th execution of opt-ZEnc, and that no value $(Y_i^{(l)})_{l\in[d-k+1]}$ is probed. Further, i_0 being the smallest of a $(d-k-d')$-tuple within $[0, d-k]$ implies that $i_0 \in [0, (d-k)-(d-k-d')] = [0, d']$, and that at least i_0 probes are either on the first i_0 executions of opt-ZEnc, or on the computation of the first i_0 output shares $(Y_i)_{i<i_0}$. This implies that at least i_0 probes are on variables independent of $(G_j^{(i_0)})_{j\in[n]}$, and at most $d'-i_0$ probes can be used to recompute $(G_j^{(i_0)})_{j\neq i_0}$. Conditioned on $d'-i_0$ such probes, $(G_{i_0}^{(i_0)}, G_{i_1}^{(i_0)}, \ldots, G_{i_{d-k-d'}}^{(i_0)})$ stay $(d-k+1-i_0-(d'-i_0)) = (d-k+1)$-wise uniformly independent. □

The following lemma derives directly from the commutativity of addition and the linearity of constant multiplication.

Lemma 7 (Generalized-fault-invariance)**.** *Gadget* opt-sZEnc *(cf. Alg. 3) is generalized-fault-invariant with respect to $\mathcal{F}^+$.*

Lemma 8 (Adapted from [BEF+23, Thm. 19])**.** *Every change to the output distribution of gadget* opt-sZEnc_n^d *(cf. Alg. 3) introduced by s internal additive faults can be achieved by s direct faults on the output.*

Proof. This statement directly follows from Lemma 5, as opt-**sZEnc**$_n^d$ consists of $d+1-k$ executions of opt-**ZEnc** together with share-wise addition. Thus, the internal faults can be pushed to the output due to the associativity of the addition. □

We use the optimizations introduced in Section 3 and this section to improve the complexity of the original **Refresh** gadget (cf. Alg. 6). We adapt the original **Refresh** gadget (cf. Alg. 6) presented by Berndt et al. in [BEF+23] to get our improved gadget opt-**Refresh** (cf. Alg. 5). The correctness of the algorithm follows directly from the correctness of opt-**sZEnc** and the linearity of share-wise addition.

Algorithm 5 opt-**Refresh** for $n = d+e+1$ and $k \leq d$

Input : *Degree-d* shares of $\boldsymbol{v} = (v_0, \ldots, v_d)$ as $(F_i)_{i\in[n]}$
Output : Randomized degree-d shares of $\boldsymbol{v}$ as $(Q_i)_{i\in[n]}$

1 $(Y_i)_{i\in[n]} \leftarrow$ opt-**sZEnc**$_n^d$
2 $(Q_i)_{i\in[n]} \leftarrow (F_i)_{i\in[n]} + (Y_i)_{i\in[n]}$
3 **return** $(Q_i)_{i\in[n]}$

Algorithm 6 **Refresh** for $n = d+e+1$ and $k \leq d$ [FOWZ25, Algorithm 6]

Input : *Degree-d* shares of $\boldsymbol{v} = (v_0, \ldots, v_d)$ as $(F_i)_{i\in[n]}$
Output : Randomized degree-d shares of $\boldsymbol{v}$ as $(Q_i)_{i\in[n]}$

1 $(Y_i)_{i\in[n]} \leftarrow$ **sZEnc**$_n^d$
2 $(Q_i)_{i\in[n]} \leftarrow (F_i)_{i\in[n]} + (Y_i)_{i\in[n]}$
3 **return** $(Q_i)_{i\in[n]}$

Theorem 2. *Gadget* opt-**Refresh** *(cf. Alg. 5) is* $(d+1-k)$-frSNI*, w.r.t* $\mathcal{F}^+$.

Proof. Note that due to [ABE+25], it is sufficient to show that opt-**Refresh** is generalized-fault-invariant, SNI, and weak simulator-invariant. Because of the commutativity of addition and linearity of constant multiplication, opt-**Refresh** is generalized-fault-invariant, it is left to show that the algorithm is a $d+1-k$-SNI refresh and the strong-simulator invariance for the corresponding simulator. To prove $(d+1-k)$-SNI, we must show that every set of $d+1-k = t_{\mathsf{int}} + t_{\mathsf{out}}$ probes can be simulated from t_{int} values from each input sharing, where t_{int} is the number of internal probes and t_{out} is the number of output probes.

We structure the proof as follows: First, we partition the variables into groups. Next, we assign each group the input variables used for the simulation, using the set I constructed in Lemma 6. Then, we describe the simulator. To conclude, we show that the simulation is perfect. We start by considering the following groups.

Group 1. Any *internal* value of the computation of $(Y_i)_{i\in[n]} \leftarrow$ opt-**sZEnc**.
Group 2. Any value $(Y_i)_{i\in[n]}$.
Group 3. The input shares $(F_i)_{i\in[n]}$.
Group 4. The output shares $(Q_i)_{i\in[n]}$.

Notably, group 4 contains only output variables, while groups 1, 2, and 3 only include internal variables.

We now define a set P that we will use to construct I from Lemma 6. We initialize P as the set containing all the probes in Groups 1 and 2. Moreover, for every input share probe F_i, we include Y_i in P. P has t_{int} elements, and Lemma 6 guarantees the existence of a set I of $n - t_{\mathsf{int}}$ elements which are $(d+1-k-t_{\mathsf{int}})$-uniformly independent.

The simulator S to be constructed takes as input $(F_i)_{i\in I^c}$, where $I^c = [n] \setminus I$. Notably, $|I^c| = t_{\mathsf{int}}$, which is required from an SNI simulator. The simulator now works as follows:

Preprocessing. Run a complete computation of opt-**sZEnc** and store the resulting values.
Probes in groups 1 and 2. Outputs the corresponding values computed in the preprocessing phase.
Probes in group 3. Return the corresponding input F_i.[1]

[1] By construction of P and I, if F_i is probed then i is in I^c.

Probes in group 4. Output a random value $Q_i \leftarrow\!\!\$\ \mathbb{F}$ if $i \in I$, or $Q_i = Y_i + F_i$ otherwise, where Y_i is computed in the preprocessing phase.

To conclude the proof of SNI, we show that the simulation is perfect. We observe that S simulates the probes in groups 1, 2, and 3 exactly as in a real execution of the gadget. The same holds for the probes in group 4, such that $i \in I^c$. Therefore, it remains to argue about the probes in group 4 such that $i \in I$. By Lemma 6, we know that the tuple $(Y_i)_{i\in I}$ is $(d+1-k-t_{\mathsf{int}})$-wise uniformly independent conditioned on the probes in P. Since the circuit computes $(Q_i)_{i\in I}$ by summing $(Y_i)_{i\in I}$ with the independent tuple $(F_i)_{i\in I}$, $(Q_i)_{i\in I}$ stays $(d+1-k-t_{\mathsf{int}})$-wise uniformly independent conditioned on the probes in P. Since the number of probes of $(Q_i)_{i\in I}$ is t_{out}, and $t_{\mathsf{int}} + t_{\mathsf{out}} = d+1-k$, this proves that any tuple of such probes is uniformly distributed in $\mathbb{F}^{t_{\mathsf{out}}}$. This proves perfect simulation of the probes in group 4, and proves SNI.

Finally, we need to show $\mathcal{F}^+$-weak simulator invariance (cf. [ABE+25]). In order to do so, for any tuple of probes $p_0, \ldots, p_{d'-k}$, we define a new simulator $S' = (S'_0, \ldots, S_{d-k})$ and d functions $\tilde{R}_0, \ldots, \tilde{R}_{d-1}$ such that S' is a valid $(d+1-k)$-SNI simulator and meets conditions 1, 2, and 3, of the definition of weak-simulation invariance. In short, we define $S_i'^{\tilde{\mathbf{R}}}(\mathbf{a})$ as the identity on $\tilde{\mathbf{R}}$, and $\tilde{R}_i(\mathbf{R}, \mathbf{F})$ describes exactly how the gadget computes the probes: if p_i is a probe in group 1 or 2, let $\tilde{R}_i(\mathbf{F}, \mathbf{R}) = f_i^{\mathbf{R}}(\cdot)$,[2] if p_i is a probe in group 3, set $\tilde{R}_i(\mathbf{F}, \mathbf{R}) = F_i$, and if p_i is a probe in group 4, set $\tilde{R}_i(\mathbf{F}, \mathbf{R}) = F_i + Y_i(\mathbf{R})$, where $Y_i(\mathbf{R})$ describes the way opt-**sZEnc** computes Y_i from the random values $\mathbf{R}$. By inspection, S'_i computes the probes exactly as the real gadget, and therefore satisfies the second condition of weak fault-invariance. Lemma 6 guarantees that any $(d+1-k-t_{\mathsf{int}})$-tuple $(Y_i(\mathbf{R}))_{i\in I}$ is uniformly distributed and independent of the probes within opt-**sZEnc**. This implies that any $(d+1-k-t_{\mathsf{int}})$-tuple $\tilde{R}_i(\mathbf{F}, \mathbf{R}) = (Y_i(\mathbf{R}) + F_i)_{i\in I}$ is uniformly distributed, independent of the t_{int} probes within opt-**sZEnc**, and of the inputs $(F_i)_{i\in[n]}$. We observe that the probes in group 1 and2 are input-independent, and the probes in group 3 and those in group 4 such that $i \in I^c$ only depend on the values input to the simulator. Moreover, each $\tilde{R}_i$ is linear in its inputs, and therefore constant additive shifts have no impact on their joint independence. This proves conditions 1 and 2, as well as the validity of S' as an SNI simulator. This concludes the proof. □

Theorem 3. *Gadget* opt-**Refresh** *(cf. Alg. 5) is e-fault-robust with respect to* $\mathcal{F}^+$.

Proof. Let $\boldsymbol{F} + \boldsymbol{\xi}^F$ denote the input, where $\boldsymbol{F}$ corresponds to a valid encoding and $\boldsymbol{\xi}^F$ refers to any fault vector in $\mathcal{F}^+$. In Lemma 8, we show that any change to the output of opt-**sZEnc** caused by s internal faults can be achieved by s direct faults on the output. The final output of opt-**Refresh** only share-wise adds the input parameters to the output of opt-**sZEnc**, which is a linear and share-wise operation. Thus using the notation of Definition 4, we set $\boldsymbol{\omega}$ to the internal faults of opt-**sZEnc** and faults within line 2 of the algorithm. Let s_1 denote the number of internal faults. It directly follows that $\mathsf{weight}(\boldsymbol{\omega}) \in [0, s_1]$ as desired. Moreover, it holds true that $\boldsymbol{\omega}' = \boldsymbol{\xi}^F$ due to the linearity of the addition. Together with the properties of the underlying polynomials, this implies e–fault robustness with respect to $\mathcal{F}^+$. □

5 Secure Compiler

In this section, we show how to use the optimizations presented in Section 3 and Section 4 to improve the complexity of both **BGW**-like multiplications and **LaOla**-like multiplications [BEF+23, FOWZ25]. Throughout this section, let $f(X)$ be a polynomial of degree d embedding the secrets $\boldsymbol{v}^{(F)} = (v_0^{(F)}, \ldots, v_d^{(F)})$, and $g(X)$ be a polynomial of degree d with respect to the secrets $\boldsymbol{v}^{(G)} = (v_0^{(G)}, \ldots, v_d^{(G)})$. As discussed in Section 2.2, the support

[2] Notably, the probes in this group are computable independently of the input.

points of the secrets are denoted as $u_0, \ldots, u_d$, i.e., we have $f(u_i) = v_i^{(F)}$ and $g(u_i) = v_i^{(G)}$ for each $i \in [d+1]$. The corresponding shares are denoted by $\boldsymbol{F} = (F_0, \ldots, F_{n-1})$ and $\boldsymbol{G} = (G_0, \ldots, G_{n-1})$, respectively. In [ABE+25], the authors use $\mathsf{Comp}\hat{A}^{(d_{\mathsf{in}}, d_{\mathsf{out}}, n)}$ (cf. Alg. 10) to construct a matrix $\hat{A}$. Through the multiplication with $\hat{A}$, $G_i = \sum_{j\in[n]} \hat{A}_{i,j} F_j$, we transform the sharing $(F_i)_{i\in[n]}$ of degree d_{in}, and get a new sharing $(G_i)_{i\in[n]}$ which has degree d_{out}. More importantly, matrix $\hat{A}$ guarantees that once sharing $(F_i)_{i\in[n]}$ is faulted, not only all the low degree ($\leq d_{\mathsf{out}}$) coefficients of $(G_i)_{i\in[n]}$ will be re-randomized, but also all the high order ($> d_{\mathsf{out}}$) coefficients will be re-randomized. Through this re-randomization technique, two objectives are achieved. First, it ensures *fault propagation*: once faults are injected, such that *some* high-order coefficients $(F_i)_{i\in[n]}$ become non-zero, then with high probability, *all* high-order coefficients of $(G_i)_{i\in[n]}$ will also be non-zero. Second, it limits the attacker's influence on the high-order coefficients, since these coefficients are random values and independent from the injected faults.

5.1 BGW-Like Multiplication

The original **BGW**-based multiplication was introduced by **B**en-Or, **G**oldwasser, and **W**igderson [BGW88]. We begin this section by adapting the BGW-based multiplication gadget of Faust et al. [FOWZ25] with respect to our opt-**ZEnc** gadget (cf. Alg. 1). Subsequently, we show that the resulting multiplication gadget maintains the same combined resilience as the original gadget.

In a **BGW**-like multiplication, a share-wise multiplication is first performed, i.e., we compute $F_i \cdot G_i$ for all $i \in [n]$. However, as the resulting product of the shares corresponds to a degree-$2d$ polynomial $f(X) \cdot g(X)$, we need to reduce the degree of the resulting polynomial to d. In order to guarantee that $f(X) \cdot g(X)$ correctly embeds the multiplication result of the secrets (i.e., $v_0^{(F)} \cdot v_0^{(G)}, \ldots, v_d^{(F)} \cdot v_d^{(G)}$), and satisfies the combined resilience requirement, we thus need at least $n \geq 2d + e + 1$ shares. We focus here on the concrete BGW-like multiplication gadget **BGW** (cf. Alg. 7), which is an adapted version from [ABE+25]. The modifications we made are highlighted in **blue**. For further details on the construction of the matrix $\hat{A}$, which consists of a degree-reduction matrix $\hat{\lambda}^{\mathsf{DegRed}}$ together with a re-randomization matrix please refer to Supp. B.1, [FOWZ25] and [ABE+25]. We first show that this modification still gives a correct multiplication result.

Algorithm 7 $\mathbf{BGW}_{\hat{A}\leftarrow\mathsf{Comp}\hat{A}^{(2d,d,n)}}$ for $n = 2d + e + 1$, $k \leq d$

Input : Two *degree-d* shares of secret vectors $\boldsymbol{v}^{(F)}$ and $\boldsymbol{v}^{(G)}$ as $(F_i)_{i\in[n]}$ and $(G_i)_{i\in[n]}$

Output : *Degree-d* shares of $v_0^{(F)} \cdot v_0^{(G)}, \ldots, v_d^{(F)} \cdot v_d^{(G)}$ as $(H_i)_{i\in[n]}$

initialize $(H_i)_{i\in[n]}$
$(Q_i)_{i\in[n]} \leftarrow \mathbf{SWMult}\left((F_i)_{i\in[n]}, (G_i)_{i\in[n]}\right)$
for $j \in [n]$ **do**
  if $j < d + 1 - k$ **then**
    $\left(Y_i^{(j)}\right)_{i\in[n]} \leftarrow \text{opt-}\mathbf{ZEnc}_n^{d-j}$
  else
    $\left(Y_i^{(j)}\right)_{i\in[n]} \leftarrow \text{opt-}\mathbf{ZEnc}_n^{d}$
for $i \in \{0, \ldots, n-1\}$ **do**
  for $j \in \{0, \ldots, n-1\}$ **do**
    $H_{i,j} \leftarrow \hat{A}_{i,j} \cdot Q_j + Y_i^{(j)}$
    $H_i \leftarrow H_i + H_{i,j}$
return $(H_i)_{i\in[n]}$

Lemma 9 (Correctness of gadget $\mathbf{BGW}_{\hat{A}\leftarrow\mathsf{Comp}\hat{A}^{(2d,d,n)}}$ (cf. Alg. 7))**.** *If $n \geq 2d+1$, the gadget $\mathbf{BGW}_{\hat{A}\leftarrow\mathsf{Comp}\hat{A}^{(2d,d,n)}}$ on input of two degree-d shares of secret vectors $\boldsymbol{v}^{(F)}$ and $\boldsymbol{v}^{(G)}$ as $(F_i)_{i\in[n]}$ and $(G_i)_{i\in[n]}$ outputs degree-d shares of secret vector $(v_0^{(F)}\cdot v_0^{(G)},\ldots,v_d^{(F)}\cdot v_d^{(G)})$.*

Proof. In Line 5, zero-encodings of degree $(d,...,k)$ are generated. By correctness of $\mathsf{opt\text{-}ZEnc}$, the sharing $(Y_i^{(j)})_{i\in[n]}$ encodes k zeros and $d+1-k,\ldots,1$ random values. Line 7 of the gadget $\mathbf{BGW}_{\hat{A}\leftarrow\mathsf{Comp}\hat{A}^{(2d,d,n)}}$ (cf. Alg. 7) constructs $n-(d-k+1)$ zero-encodings with respect to a degree d polynomial and $\mathsf{Offset}\leftarrow 0$. The share-wise multiplication operation $\mathbf{SWMult}$ gives us a degree-$2d$ sharing of the secrets $v_0^{(F)}\cdot v_0^{(G)},\ldots,v_{k-1}^{(F)}\cdot v_{k-1}^{(G)}$. Finally, on line 10, 11, 12, after inserting matrix $\hat{A}_{i,j}$ and multiplying it with the share-wise multiplication shares $(Q_i)_{i\in[n]}$, we get

$$\begin{aligned}H_i &= \sum_{j=0}^{n-1}\hat{A}_{i,j}\cdot Q_j + Y_i^{(j)} = \sum_{j=0}^{n-1}((\lambda_{i,j}^{(2d\to d)})+\sum_{k=d_{\mathsf{in}}+1}^{n-1}\hat{R}_i^{(k)}\cdot V_{k,j}^{-1})\cdot Q_j + Y_i^{(j)}\\ &= \sum_{j=0}^{n-1}(\lambda_{i,j}^{(2d\to d)}\cdot Q_j + Y_i^{(j)})+\sum_{k=2d+1}^{n-1}\hat{R}_i^{(k)}\cdot f_k,\end{aligned}$$

where $(f_k)_{k\in[2d+1,n-1]}=0$ when no faults are injected to shares $(Q_i)_{i\in[n]}$. Furthermore, we obtain the following coefficients $(h_u)_{u\in[d+1]}$

$$\begin{aligned}h_u &= \sum_{i=0}^{n-1}H_i\cdot V_{u,i}^{-1} = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(\lambda_{i,j}^{(2d\to d)}\cdot Q_j\cdot V_{u,i}^{-1}) + \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(Y_i^{(j)}\cdot V_{u,i}^{-1})\\ &= \sum_{i=0}^{n-1}(\sum_{j=0}^{n-1}(\lambda_{i,j}^{(2d\to d)}\cdot Q_j\cdot V_{u,i}^{-1})) + 0 = v_u^{(F)}\cdot v_u^{(G)},\end{aligned}$$

where matrix $\lambda^{(2d\to d)}$ reduces the degree of the sharing $(Q_i)_{i\in[n]}$ from $2d$ to d and V^{-1} is the inverse Vandermonde matrix reconstructing the underlying secrets $v_u^{(F)}\cdot v_u^{(G)}$ for $u\in[d+1]$. Furthermore, as $(Y_i^{(j)})_{i\in[n]}$ is a zero-encoding, the underlying secret is 0. □

Now, we show that these modifications not only guarantee security against passive probing attacks but also in the presence of active faults.

Lemma 10. $\mathbf{BGW}_{\hat{A}\leftarrow\mathsf{Comp}\hat{A}^{(2d,d,n)}}$ *(cf. Alg. 7) is $(d+1-k)-\mathsf{SNI}$ with respect to $\mathcal{F}^+$.*

Proof. We consider the following five groups of probes: Group 1 contains all intermediate probes during $\mathbf{SWMult}$ and probes of F_i or G_i. In the next group, Group 2, all probes of $(Y_i^{(j)})$ and the probes during an evaluation of $(Y_i^{(j)})_{i\in[n]}\leftarrow\mathsf{opt\text{-}ZEnc}$ are described. Group 3 consists of all probes of $H_{i,j}$ for $i\in[n],j\in[n]$. In Group 4, we consider the internal probes $H_i^{(j)}$, $j\in[n-1]$. We denote these probes as $G_4=\bigcup_{j\in[n]}G_4^{(j)}$ where $G_4^{(j)}=\cup_{i\in[n]}G_4^{(j,i)}$ and $G_4^{(j,i)}$ describes a probe in $H_i^{(j)}$ (if $j<n-1$). The fifth group contains all output probes, i.e., probes of $H_i^{(n-1)}$.

Constructing input sets: We build the input sets I_F, I_G and I_R for F_j, G_j and the probed randomness of the zero-encodings $(Y_i^{(j)})_{i\in[n]}$, respectively. In more detail, if $j\in I_F$, we assume the simulator obtains F_j. We start by $I_F=I_G=I_R=\emptyset$ and proceed group-wise as follows: For any probe of $Q_i\leftarrow F_i\cdot G_i$, F_i or G_i (namely the probes of Group 1), set $I_F\leftarrow I_F\cup\{i\}$, and $I_G\leftarrow I_G\cup\{i\}$. For any probe in Group 2, i.e., probes of $(Y_i^{(j)})$ or $(Y_i^{(j)})\leftarrow\mathsf{opt\text{-}ZEnc}$, set $I_F\leftarrow I_F\cup\{j\}$, $I_G\leftarrow I_G\cup\{j\}$ and $I_R\leftarrow I_R\cup\{j\}$. For a probe $H_{i,j}$ for $i\in[n],j\in[n]$, set $I_F\leftarrow I_F\cup\{j\}$, $I_G\leftarrow I_G\cup\{j\}$ and $I_R\leftarrow I_R\cup\{j\}$. For any

probe in $G_4^{(j)}$, we do the following. For all probes $G_4^{(j,i)} \in G_4^{(j)}$: if $j \notin I_R$: $I_F \leftarrow I_F \cup \{j\}$, $I_G \leftarrow I_G \cup \{j\}$ and $I_R \leftarrow I_R \cup \{j\}$. Otherwise: let $j^* = \max\{\ell \in [j] \mid \ell \notin I_R\}$. Then, set $I_F \leftarrow I_F \cup \{j^*\}$, $I_G \leftarrow I_G \cup \{j^*\}$ and $I_R \leftarrow I_R \cup \{j^*\}$. We do not add any element to I_F, I_G or I_R for probes in Group 5. Note that for any probe in groups 1 to 4, we add at most one element per probe into our sets. Let $t \leq d+1-k$ denote the total number of internal probes of our adversary. Then, $|I_F| \leq t$, $|I_G| \leq t$ and $|I_R| \leq t$.

Simulating probes: Now, we simulate all probes with the help of inputs of $(F_i)_{i\in[n]}$ and $(G_i)_{i\in[n]}$ with indices from the sets I_F, and I_G, respectively. We proceed group by group.

- **G1** For all probes of F_i, G_i or Q_i we have $i \in I_F$ by construction. Hence, we know F_i and G_i and can compute $Q_i = F_i \cdot G_i$.
- **G2** For any element $j \in I_R$, we assume that all $(Y_i^{(j)})_{i\in[n]}$ are leaked. The values $(Y_i^{(j)})_{i\in[n]} \leftarrow$ opt-**ZEnc**, $j \in [n]$ or the outputs of this evaluation can be simulated by calculating them according to opt-$\mathbf{ZEnc}_n^d$ for $j \in [d+1-k, n]$ and opt-$\mathbf{ZEnc}_n^j$ for $j \in [d+1-k]$. This implies the simulation of all probes in Group 2.
- **G3** If $j \in [n]$ is a probe in Group 3, $j \in I_F$, $j \in I_G$ and $j \in I_R$. Hence, F_j and G_j are known. Moreover, the elements $(Y_i^{(j)})_{i\in[n]}$ were calculated in step G2. Altogether, we can perfectly simulate $H_{i,j} \leftarrow \hat{A}_{i,j} \cdot F_j \cdot G_j + Y_i^j$, since $\hat{A}_{i,j} \leftarrow \mathsf{Comp}\hat{A}(2d, d, n)$ can be calculated independent of any inputs, since $2d, d, n$ are assumed to be fixed public parameters. Moreover, $\hat{A}$ is calculated out of public parameters (namely $\lambda_{i,j}^{(2d\to d)}$) and elements that are chosen independently of each other and uniformly at random (cf. [ABE+25]).
- **G4** We denote the value of H_i in step j by $H_i^{(j)}$ and show its simulation by induction. The probes in Group 4 are ordered according to the values $j \in [n]$, such that $j_0 \leq \cdots \leq j_m$.

  Induction basis: We consider j_0. Recall that if j_0 is a probe in Group 4, $j_0 \in I_F$, $j_0 \in I_G$ and $j_0 \in I_R$. Hence, F_{j_0} and G_{j_0} are known. Moreover, the elements $(Y_i^{(j)})_{i\in[n]}$ were calculated in step G2. Hence, if $H_{i,j}$ has not been simulated yet, we can compute it as $H_{i,j} = \hat{A}_{i,j_0} \cdot F_{j_0} \cdot G_{j_0} + Y_i^{j_0}$. Furthermore, we know that $H_i^{(j_0)} = \sum_{j=0}^{j_0} \hat{A}_{i,j} \cdot F_j \cdot G_j + Y_i^j$. We distinguish two cases:

  a) If it exists an element $j \in [j_0]$ with $j \notin I_R$, we choose $H_i^{(j_0)}$ uniformly random. Let $j_\ell = \min\{j \in [j_0] | j \notin I_R\}$ and $j'_\ell = \max\{j \in [j_0] | j \notin I_R\}$. If the element $j'_\ell \notin I_R$ is in $[d+1-k, n]$, then the elements $(Y_i^{(j_\ell)})_{i\in[n]}$ are $d+1-k$-wise uniformly independent due to the construction by opt-**ZEnc**. Hence, by the probing threshold of $d+1-k$, choosing $Y_i^{(j_\ell)}$ uniformly random simulates $Y_i^{(j_\ell)}$. As $H_i^{(j_0)} = \sum_{j=0}^{j_0} \hat{A}_{i,j} \cdot F_j \cdot G_j + Y_i^j$, the element $Y_i^{j_\ell}$ re-randomizes $H_i^{(j_0)}$ and thus, $H_i^{(j_0)}$ is indistinguishable from uniformly random. If $j'_\ell \in [d+1-k]$, there already exist probes. Now, considering j_ℓ, we know $(Y_i^{(j_\ell)})_{i\in[n]}$ is at least $(d+1-k-j_\ell)$-wise independent and any $(d+1-k-j_\ell)$ tuple of elements is uniformly distributed by Lemma 2. By definition of j_ℓ, at least ℓ elements have already been probed. Due the probing threshold of $d+1-k$, and the $(d+1-k-j_\ell)$-wise uniform independence of $(Y_i^{(j_\ell)})_{i\in[n]}$, choosing the element $Y_i^{(j_\ell)}$ uniformly random re-randomizes $H_i^{(j_0)}$. It follows that $H_i^{(j_0)}$ is indistinguishable from chosen uniformly random.

  b) Else, $j \in I_R$ for all $j \in [j_0]$. Then, by construction of the sets I_F and I_G, we know that $\forall j \in [j_0]$ it holds true that $j \in I_F$ and $j \in I_G$. Hence, $H_i^{(j_0)}$ can be perfectly simulated by computing $H_i^{(j_0)} = \sum_{j=0}^{j_0} \hat{A}_{i,j} \cdot F_j \cdot G_j + Y_i^j$.

  Induction step: $j_{w-1} \to j_w$

  Similar to the induction basis, we consider the following two cases:

a) If there exists an element $j \in [j_{w-1}, j_w]$ with $j \notin I_R$, we choose $H_i^{(j)}$ uniformly random. Again, set set $j_\ell = \min\{j \in [j_{w-1}, j_w] : j \notin I_R\}$ and $j'_\ell = \max\{j \in [j_{w-1}, j_w] : j \notin I_R\}$. If $j'_\ell \in [d+1-k, n]$, we know that $(Y_i^{(j_\ell)})_{i\in[n]}$ is at least $d+1-k$-wise independent and any $d+1-k$ tuple is uniformly distributed. Due to the probing threshold of $d+1-k$, $(Y_i^{(j_\ell)})_{i\in[n]}$ is indistinguishable from uniform, and hence $H_i^{(j_w)}$ is indistinguishable from uniformly distributed.
If $j'_\ell \in [d+1-k]$, we consider j_ℓ. We know that $(Y_i^{(j_\ell)})_{i\in[n]}$ is at least $d+1-k-j_\ell$-wise uniformly independent. Note that exists no probe of $(Y_i^{(j_\ell)})_{i\in[n]}$, G_{j_ℓ} and F_{j_ℓ} (as otherwise $j \in I_R$). Moreover, for by definition of the input set for group 4, we know that if $H_i^{(j_w)}$ had been probed for ω elements $i \in [n]$, at least $j_w - \omega, \ldots, j_w \in I_R \cap I_F \cap I_G$. Thus, the $(Y_i^{(j_\ell)})_{i\in[n]}$ are at least $(d+1-k) - j_w + \omega$-wise uniformly independent and by the probing threshold of $d+1-k$, the element $Y_i^{(j_\ell)}$ is indistinguishable from uniform for the adversary and re-randomizes $H_i^{(j)}$. Hence, $H_i^{(j)}$ is indistinguishable from uniformly distributed for the adversary.

b) Else, we have $j \in I_R$ for all $j \in [j_{w-1}, j_w]$. By construction of the sets I_F and I_G, we know that $\forall j \in [j_{w-1}, j_w]$ also $j \in I_F$ and $j \in I_G$. Hence $H_i^{(j_w)}$ can be perfectly simulated by computing $H_i^{(j_w)} = H_i^{(j_{w-1})} + \sum_{j=j_{w-1}+1}^{j_w} \sum_{j=0}^{j_0} \hat{A}_{i,j} \cdot F_j \cdot G_j + Y_i^j$, since $\hat{A}_{i,j}$ can be perfectly simulated, as they only depend on public parameters, as well as elements that are chosen uniformly at random and independent of each other for all $i \in [n]$ (cf. [ABE+25]).

G5 To simulate the output probes, we proceed as follows. For all $i : G_5^{(i)} \in G_5$, we perform the following procedure: If $n-1 \notin I_R$, then we choose the element(s) $H_i^{(n-1)}$ uniformly random. This perfectly simulates the element, due to the probing threshold of $d+1-k$ and $d+1-k$-wise uniform independence of $(Y_i^{(n-1)})_{i\in[n]}$.
Otherwise, $n-1 \in I_R$ due to probes of previous groups. Let $j^* = \max\{j \in [n-1] | j \in I_R\} \cap \{-1\}$. If $j^* = -1$, we choose $H_i^{(n-1)}$ uniformly random, since in $H_i^{(n-1)} = \sum_{j=0}^{n-2} H_{i,j} + H_{i,n-1} = \sum_{j=0}^{n-2} (\hat{A}_{i,j} Q_j + Y_i^{(j)}) + H_{i,n-1}$. Now, $(Y_i^{(0)})_{i\in[n]}$ are at least $d+1-k$-wise uniformly independent. Again, by the probing threshold of $d+1-k$, we know that $H_i^{(n-1)}$ is indistinguishable from uniform.
Now, $H_i^{(n-1)} = H_i^{(j^*)} + \sum_{j=j^*+1}^{n-2} (\hat{A}_{j,i} Q_j + Y_i^{(j)}) + H_{i,n-1}$, where $H_i^{(j^*)}$ is perfectly simulated in G4 and $H_{i,n-1} = \hat{A}_{n-1,i} \cdot F_{n-1} \cdot G_{n-1} + Y_i^{n-1}$ can be calculated, since $n-1 \in I_R$ and hence by construction, $n-1 \in I_F \cap I_G$ and $\hat{A}_{n-1,i}$ is only dependent on public parameters as well as uniform random elements.
If $j \in I_R$ for all $j \in [j^*, n-1]$, we can calculate $H_i^{(n-1)}$ directly, since all necessary elements $H_i^{(j^*)}, H_{i,n-1}$ and $\hat{A}_{j,i}, Q_j, Y_i^{(j)}$ are known to the simulator. Else, there is $j \in [j^*, n-1]$ with $j \notin I_R$. Let $j'_\ell = \max\{j \in [j^*, n-1] : j \notin I_R$. If $j'_\ell \geq d+1-k$, the underlying zero-encodings $(Y_i^{(j'_\ell)})_{i\in[n]}$ are $d+1-k$-wise uniformly independent. Hence, by the probing threshold, choosing $H_i^{(n-1)}$ uniformly random perfectly simulates this element. If $j'_\ell < d+1-k$, we know that $(H_i^{(j'_\ell)})_{i\in[n]}$ is at least $d+1-k+|P_{j'_\ell}|$-wise uniformly independent, where $P_{j'_\ell}$ is the index set consisting of all probes done until the j'_ℓ-th iteration. (Details about this assumption are provided hereafter.) Due to the threshold of $d+1-k$, we can choose the element $H_i^{(n-1)}$ uniformly at random.

Independence: It is left to show that $(H_i^{(j'_\ell)})_{i\in[n]}$ is at least $d+1-k+|P_{j'_\ell}|$-wise uniformly independent, if there is no probe of $(Y_i^{j'_\ell})_{i\in[n]}$. We prove this by induction. Let P_j denote the set of indices for $i \in [n]$ that are probed in iteration j, such that $P_0 \subset P_1 \subset \cdots \subset P_{n-1}$.

Induction basis $j = 0$: Since $(Y_i^{(0)})_{i\in[n]}$ are $d+1-k$-wise uniformly independent and independent from $(F_i)_{i\in[n]}$, $(F_i)_{i\in[n]}$, also $(H_i^{(0)})_{i\in[n]} = \hat{A}_{i,0}F_0 \cdot G_0 + Y_i^{(0)}$ is $d+1-k$-wise uniformly independent as well, if $|P_0| = 0$. Otherwise, for any probe in $H_i^{(0)}$, the elements $(H_i^{(0)})_{i\in[n]}$ remain $d+1-k-|P_0|$-wise uniformly independent.

Induction step $j-1 \to j$: Recall that $H_i^{(j)} = H_i^{(j-1)} + (\hat{A}_{i,j}F_iG_i + Y_i^{(j)})$. If there is no probe in $H_i^{(j)}$, then $H_i^{(j)}$ is at least $d+1-k-|P_j| = d+1-k-|P_{j-1}|$-wise uniformly independent by induction hypothesis. Otherwise, there is at least one probe in $H_i^{(j)}$, i.e., $|P_j \setminus P_{j-1}| \neq 0$. Then, $(H_i^{(j)})_{i\in[n]}$ is at least m-wise uniformly independent, where $m = \max\{d+1-k-(j+1)-|P_j \setminus P_{j-1}|, d+1-k-(P_{j-1} \cup |P_j \setminus P_{j-1}|$ and thus $m \geq d+1-k-(|P_{j-1}| \cup |P_j \setminus P_{j-1}|) = d+1-k-|P_j|$.

Hence, we proved that gadget $\mathbf{BGW}_{\hat{A}\leftarrow\mathsf{Comp}\hat{A}^{(2d,d,n)}}$ (cf. Alg. 7) is $(d+1-k)-\mathsf{SNI}$. □

Lemma 11. *The gadget* $\mathbf{BGW}_{\hat{A}\leftarrow\mathsf{Comp}\hat{A}^{(2d,d,n)}}$ *(cf. Alg. 7) is fault-resilient* SNI *(i.e.,* $(d+1-k)-\mathsf{frSNI}$*) with respect to* $\mathcal{F}^+$*.*

Proof. In Lemma 10, we have proved that gadget $\mathbf{BGW}_{\hat{A}\leftarrow\mathsf{Comp}\hat{A}^{(2d,d,n)}}$ (cf. Alg. 7) satisfies d-SNI by constructing simulator S. Now, we will prove that $\mathbf{BGW}_{\hat{A}\leftarrow\mathsf{Comp}\hat{A}^{(2d,d,n)}}$ satisfies SNI even when the corresponding circuit is faulted. To achieve this, we use the concept of generalized-fault-invariance (cf. Def. 8). The frSNI proof consists of three parts. First, we show that $\mathbf{BGW}_{\hat{A}\leftarrow\mathsf{Comp}\hat{A}^{(2d,d,n)}}$ (cf. Alg. 7) satisfies the generalized-fault-invariance (cf. Lemma 3), i.e., all the additive faults injected into gadget $\mathbf{BGW}_{\hat{A}\leftarrow\mathsf{Comp}\hat{A}^{(2d,d,n)}}$ (cf. Alg. 7) or its sub-gadgets (e.g., **SWMult** (cf. Alg. 16), $\mathsf{opt}\text{-}\mathbf{ZEnc}_n^{d,\mathbf{Offset}}$ (cf. Alg. 1, and $\mathsf{Comp}\hat{A}^{(2d,d,n)}$ (cf. Alg. 10)) can be *pushed* to the circuit inputs $\mathbf{x}, \mathbf{y}$, or the outputs $\mathbf{z}$ or the internal randomness $\mathbf{R}$. Here we refer to the (possibly faulted) inputs $(F_i)_{i\in[n]}$ and $(G_i)_{i\in[n]}$ as $\mathbf{x}$ and $\mathbf{y}$, and the (possibly faulted) outputs $(H_i)_{i\in[n]}$ as $\mathbf{z}$.

Then, we construct d functions $\tilde{R}_i \colon \mathbb{F}^{2n} \times \mathbb{F}^m \to \mathbb{F}^n$ that take the circuit inputs $\mathbf{x}$, and randomness $\mathbf{R} = (R_0, \ldots, R_{m-1})$ as inputs. We denote $\tilde{\mathbf{R}} = (\tilde{R}_0, \ldots, \tilde{R}_{d-1})$. Recall that in Lemma 10, we construct a d-SNI simulator S which takes the internal randomness of the circuit $\mathsf{C}^{\mathbf{R}}$ and less than d inputs from each input sharing to simulate probes $\mathbf{p} = (p_0, \ldots, p_{d-1})$. However, because of the generalized-fault-invariance, we may not only push the faults to the inputs $\mathbf{x}$ or outputs $\mathbf{y}$, but also to the internal randomness $\mathbf{R}$ of the circuit. Hence, to simulate the probes $\mathbf{p}' = (p'_0, \ldots, p'_{d-1})$ in a faulted circuit $T[\mathsf{C}^{\mathbf{R}}]$, the SNI simulator S' needs the faulted internal randomness. This is where $\tilde{\mathbf{R}}$ is used, as it describes how the internal randomness $\mathbf{R}$ is affected after being pushed with faults.

Finally, we construct a d-SNI simulator S' that simulates $\leq d$ probes $\mathbf{p}'$ from the faulty circuit $T[\mathsf{C}^{\mathbf{R}}]$, i.e., $\Pr_{\mathbf{R}\leftarrow\$\,\mathbb{F}^m}(f'^{\mathbf{R}}(\mathbf{x},\mathbf{y}) = \mathbf{p}') = \Pr_{\tilde{\mathbf{R}}\leftarrow\$\,\mathbb{F}^{\tilde{m}}}(S'^{\tilde{\mathbf{R}}}(\mathbf{a},\mathbf{b}) = \mathbf{p}')$, using function $\tilde{\mathbf{R}}$, where f' is the function describing how $\mathbf{p}'$ is computed using input $\mathbf{x}, \mathbf{y}$ and randomness $\mathbf{R}$ in the faulted circuit $T[\mathsf{C}^{\mathbf{R}}]$, and $\mathbf{a}, \mathbf{b}$ are the selection of values (at most one per input sharing) that simulator S' takes as inputs for the simulation of probes $\mathbf{p}' = (p'_0, \ldots, p'_{d-1})$.

Generalized-Fault-Invariance. In the following, we consider all possible fault injections within p' and define the resulting fault functions in $\mathcal{F}^+$. We only consider faults that have an impact on the calculation of p' In more detail, let $|T|$ denote the number of faults inserted into $\mathsf{C}^{\mathbf{R}}$. We define $\tilde{T} = \min_{|\overline{T}|}\{\overline{T} \in \mathcal{A}(\mathcal{F}) | \overline{p} = p'\}$, where the intermediate value (probe) $\overline{p}$ of $\overline{T}[\mathsf{C}^{\mathbf{R}}]$ denotes a value on the same wire as p' in $T[\mathsf{C}^{\mathbf{R}}]$. We consider $\tilde{T}$ in the following. Note that this happens without loss of generality, as any other faults do not have an impact on probe p', by definition of $\tilde{T}$. We study these faults in $\tilde{T}$ (i.e., the faults in T that affect probe p') one after another and shift them to the input, output, or randomness as follows:

Group 1: Faults a injected into input shares $(F_i)_{i\in[n]}$, $(G_i)_{i\in[n]}$. In this case, the faults can be directly propagated to the corresponding inputs, since $(F_i)_{i\in[n]}$, $(G_i)_{i\in[n]}$ are the inputs. We define fault function $\xi^{(F_i)}(x) = x + a$ and $\xi^{(G_i)}(x) = x + a$.

Group 2: Faults a injected into line 1, 10, 11, 12, i.e., faults inserted within the initialization of $(H_i)_{i\in[n]}$, or during the calculation of $H_{i,j} \leftarrow \hat{A}_{i,j} \cdot Q_j + Y_i^{(j)}$ and $H_i \leftarrow H_i + H_{i,j}$. These faults can be propagated to the outputs as $H_i = \sum_{j\in[n]} \hat{A}_{i,j} \cdot Q_j + Y_i^{(j)}$. We define the corresponding fault function as $\xi^{(H_i)}(x) = x + a$. Besides, as proved in [ABE+25], gadget $(\hat{A}_{i,j})_{i,j\in[n]} \leftarrow \mathsf{Comp}\hat{A}^{(2d,d,n)}$ (cf. Alg. 10) is generalized-fault-invariant, i.e., we can push the faults injected during the computation of $\mathsf{Comp}\hat{A}^{(d_{\mathsf{in}},d_{\mathsf{out}},n)}$ to its internal randomness or outputs (i.e., $(\hat{A}_{i,j})_{i,j\in[n]}$). Basically, this comes from the fact that

$$\begin{aligned}\hat{A}_{i,j} + a &= \lambda_{i,j}^{(2d\to d)} + \sum_{k=2d+1}^{n-1} \hat{R}_j^{(k)} \cdot V_{k,i}^{-1} + a \\ &= \lambda_{i,j}^{(2d\to d)} + \sum_{k=2d+1}^{n-2} (\hat{R}_j^{(k)} \cdot V_{k,i}^{-1}) + \sum_{m=2d+1}^{n-2} r_m^{(n-1)} \cdot x_i^j \cdot V_{n-2,i}^{-1} + (r_{n-1}^{(n-1)} + \frac{a}{x_i^j \cdot V_{n-1,i}^{-1}}) x_i^j \cdot V_{n-1,i}^{-1},\end{aligned}$$

where $r_j^{(k)}$ are the internal randomness of gadget $\mathsf{Comp}\hat{A}^{(2d,d,n)}$ (cf. Alg. 10), and matrix V^{-1} is the inverse Vandermonde matrix. Hence, we can propagate such a fault a to the inputs (i.e., $r_m^{(n)}$) of $\mathsf{Comp}\hat{A}^{(2d,d,n)}$ and set $\tilde{\xi}(r)^{(\hat{A}_{i,j})} = r + \frac{a}{x_i^j \cdot V_{n,i}^{-1}}$.

Group 3: Faults a injected into line 5, 7, i.e., during the calculations of opt-**ZEnc**. Since we have proved that opt-**ZEnc** is generalized-fault-invariant, all the faults without the calculation of opt-**ZEnc** can be shifted to its inputs or outputs. Moreover, zero-encodings $Y_i^{(j)}$ linearly goes into the output H_i. Hence, any faults a injected to $Y_i^{(j)}$ can be written as $H_i + a$. For these faults, we define $\xi_{\mathsf{out}}^{(Y_i^{(j)})}(x) = x + a$.

Group 4: Faults a injected into line 2, either into the inputs $(F_i)_{i\in[n]}$, $(G_i)_{i\in[n]}$ or to the share-wise multiplication result $(Q_i)_{i\in[n]}$. We focus on the case where faults a are injected to Q_i. In this case the faults $a \cdot \hat{A}_{i,j}$ propagate linearly to the output H_i. We define $\tilde{\xi}^{(Q_i)}(x) = x + a \cdot \hat{A}_{i,j}$.

Since all the above fault functions are additive fault functions, we can use the fault functions (that are relevant for calculating p') defined as described above to define the overall fault functions on the randomness, as well as the inputs and the outputs.

Compute $\tilde{\mathbf{R}}$ for SNI Simulation in Faulted Circuit. Next, we aim to show that for the SNI simulator $S_0, \ldots, S_{d-1}$ constructed in Lemma 10, there exist d functions $\tilde{R}_i \colon \mathbb{F}^{2n} \times \mathbb{F}^m \to \mathbb{F}^n$ of the circuit inputs $\mathbf{x}, \mathbf{y}$ and randomness $\mathbf{R} = (R_0, \ldots, R_{m-1})$ with

1. for every randomness tuple $\mathbf{R} \in \mathbb{F}^m$, tuple of functions in $\mathcal{F}^+$, $\xi^{(0)}, \ldots, \xi^{(d-1)} \colon \mathbb{F}^n \to \mathbb{F}^n$ and $\xi^{R,(0)}, \ldots, \xi^{R,(d-1)} \colon \mathbb{F}^m \to \mathbb{F}^m$, the joint distribution of
$$\tilde{R}_0(\xi^{(0)}(\mathbf{x}, \mathbf{y}), \xi^{R,(0)}(\mathbf{R})), \ldots, \tilde{R}_{d-1}(\xi^{(d-1)}(\mathbf{x}, \mathbf{y}), \xi^{R,(d-1)}(\mathbf{R})) \tag{5}$$
is independent of at least $n - d$ input variables for $\mathbf{R} \leftarrow\!\!\$\ \mathbb{F}^m$.
2. for every $i \in [d]$, for every input tuple $\mathbf{x}, \mathbf{y} \in \mathbb{F}^n$, randomness tuple $\mathbf{R} \in \mathbb{F}^m$, and probe p_i within G, $f_i^{\mathbf{R}}(\mathbf{x}, \mathbf{y}) = S_i^{\tilde{R}_i(\mathbf{x},\mathbf{y},\mathbf{R})}(\mathbf{a}, \mathbf{b})$, where f_i is the function describing how probe p_i is computed using input $\mathbf{x}, \mathbf{y}$ and randomness $\mathbf{R}$, S_i is the sub simulator of $S = (S_0, \ldots, S_{d-1})$ defined by d-SNI (cf. Lemma 10), and $\mathbf{a} = (a_0, a_1, \ldots), \mathbf{b} = (b_0, b_1, \ldots)$ is the selection of values (at most one per input sharing) that S_i takes as input for the simulation of probe p_i.
3. The input variables from which the joint random variable in Equation 5 is independent include the ones that $S_0, \ldots, S_{d-1}$ do not take as input.

The construction of the simulator S contains three different sources of randomness: the randomness used within the calculation of $(Y_i^{(j)})_{i\in[n]} \leftarrow$ opt-**ZEnc**, $\forall j \in [n]$, the randomness used to construct the matrix $(\hat{A}_{i,j})_{i,j\in[n]}$, and the random choice of choosing $H_i^{(j)}$ uniformly at random or not (cf. Lemma 10). Recall that any simulator $S_i^{\tilde{R}_i(\mathbf{x},\mathbf{y},\mathbf{R})}(\mathbf{a}, \mathbf{b})$

simulates only one probe, p_i. We construct the function $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R})$ corresponding to the SNI simulator $S_i^{\tilde{R}_i(\mathbf{x},\mathbf{y},\mathbf{R})}(\mathbf{a}, \mathbf{b})$ as follows:

Group 1 In this group, we consider probes into input shares $(F_i)_{i\in[n]}$, $(G_i)_{i\in[n]}$. Since faults can be pushed into the input shares, $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R}) \coloneqq \mathbf{R}$, which is independent of $\mathbf{x}, \mathbf{y}$.

Group 2 In this group, we consider probes in (lines 1, 10, 11, 12), i.e., probes of $(Q_i)_{i\in[n]}$, $\hat{A}_{i,j}$, $\hat{A}_{i,j} \cdot Q_j$, $Y_i^{(j)}$, $H_{i,j}$, and $H_i^{(j)}$ (i.e., the value of H_i in step j, line 11).

- Probe $(Q_i)_{i\in[n]}$: we set $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R}) \coloneqq (\mathbf{R}, \mathbf{a}, \mathbf{b})$, and $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R})$ is independent from $\mathbf{x} \setminus \mathbf{a}$ and $\mathbf{y} \setminus \mathbf{b}$ for $\mathbf{R} \leftarrow\!\!\$\ \mathbb{F}$.
- Probe $\hat{A}_{i,j}$: we set $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R}) \coloneqq \lambda_{i,j}^{(2d\to d)} + \sum_{k=2d+1}^{n-1} \hat{R}_j^{(k)} \cdot V_{k,i}^{-1}$, and $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R})$ is independent from inputs $\mathbf{x}, \mathbf{y}$.
- Probe $\hat{A}_{i,j} \cdot Q_j$: By construction of S, we have $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R}) \coloneqq (\lambda_{i,j}^{(2d\to d)} + \sum_{k=2d+1}^{n-1} \hat{R}_j^{(k)} \cdot V_{k,i}^{-1}) \cdot F_i \cdot G_i$. And $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R})$ is independent from $\mathbf{x} \setminus \mathbf{a}$ and $\mathbf{y} \setminus \mathbf{b}$ for $\mathbf{R} \leftarrow\!\!\$\ \mathbb{F}$.
- Probe $Y_i^{(j)}$: In this case, we set $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R})$ to be the calculation of the probed element. Such a function $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R})$ exists due to the fact that opt-**ZEnc** is generalized-fault-invariant, and $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R})$ is independent from inputs $\mathbf{x}, \mathbf{y}$.
- $H_{i,j}$: In this case, we set $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R}) \coloneqq \hat{A}_{i,j} \cdot F_j \cdot G_j + Y_i^j$, and $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R})$ is independent from $\mathbf{x} \setminus \mathbf{a}$ and $\mathbf{y} \setminus \mathbf{b}$ for $\mathbf{R} \leftarrow\!\!\$\ \mathbb{F}$.
- $H_i^{(j)}$: We set $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R})$ exactly to simulate $H_i^{(j)}$ (cf. Lemma 10), which guarantees that either $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R})$ is independent from $\mathbf{x} \setminus \mathbf{a}$ and $\mathbf{y} \setminus \mathbf{b}$ for $\mathbf{R} \leftarrow\!\!\$\ \mathbb{F}$, or $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R})$ is independent from inputs $\mathbf{x}, \mathbf{y}$. Probes in $Y_{i,j}$ and probes during the computation of the i-th use of **ZEnc** are simulated according to **ZEnc**.

Group 3 In this group, we consider the probes in lines 5 and 7. If p_i is a probe in Group 3, we set $\tilde{R}_i(\mathbf{x}, \mathbf{R})$ to be the calculation of the probed element. Such a function $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R})$ exists due to the assumption that opt-**ZEnc** is generalized-fault-invariant. Moreover, $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R})$ is independent of $\mathbf{x}$ and $\mathbf{y}$, since opt-**ZEnc** does not take any inputs.

Group 4 In this group, we consider probes in line 2. As faults can be pushed into the inputs or outputs, we set $\tilde{R}_i(\mathbf{x}, \mathbf{y}, \mathbf{R}) \coloneqq \mathbf{R}$, which is independent of $\mathbf{x}, \mathbf{y}$.

Then, we set $f_i^{\mathbf{R}} = \tilde{R}_i$. Furthermore, we know that each simulator $S_i^{\tilde{R}_i(\mathbf{x},\mathbf{y},\mathbf{R})}$ takes at most one per sharing; hence, the input variables from which the joint random variable in Equation 5 is independent include the ones that $S_0, \ldots, S_{d-1}$ do not take as input.

Construct frSNI Simulator S' To prove d-fr(S)NI, we want to construct a simulator S' such that, for every $d' \leq d$ tuple of probes in the faulted circuit $\mathbf{p}' = (p_0, \ldots, p_{d'-1})$,

$$\Pr_{\mathbf{R} \leftarrow\!\!\$\ \mathbb{F}^m}(f'^{\mathbf{R}}(\mathbf{x}, \mathbf{y}) = \mathbf{p}') = \Pr_{\tilde{\mathbf{R}} \leftarrow\!\!\$\ \mathbb{F}^{\tilde{m}}}(S'^{\tilde{\mathbf{R}}}(\mathbf{a}) = \mathbf{p}'), \tag{6}$$

where f' describes the way the faulted circuit computes the probes corresponding to $\mathbf{p}'$, $\mathbf{x} = (x_1, \ldots, x_{n-1})$ and $\mathbf{y} = (y_1, \ldots, y_{n-1})$ are the circuit inputs, $\mathbf{R} = (R_0, \ldots, R_{m-1})$ are the random values input to the circuit, $a_0, \ldots$ are the inputs used by the simulator S', and $\tilde{\mathbf{R}} = (\tilde{R}_0, \ldots, \tilde{R}_{\tilde{m}-1})$ are the random values input to the simulator. To achieve (S)NI, we require the simulator S' to take at most as many input values as the number of (internal) probes. We can rewrite the left-hand side of Equation 6 by making the vector components explicit, yielding

$$\Pr_{\mathbf{R} \leftarrow\!\!\$\ \mathbb{F}^m}(f'^{\mathbf{R}}(\mathbf{x}, \mathbf{y}) = \mathbf{p}') = \Pr_{\mathbf{R} \leftarrow\!\!\$\ \mathbb{F}^m}(f_0'^{\mathbf{R}}(\mathbf{x}, \mathbf{y}) = p_0', \ldots, f_{d'-1}'^{\mathbf{R}}(\mathbf{x}, \mathbf{y}) = p_{d'-1}'). \tag{7}$$

Since our circuit is $\mathcal{F}^+$-fault-invariant, we know that for every $i \in [d']$, $f_i'^{\mathbf{R}}(\mathbf{x}, \mathbf{y}) = \xi^{(i)}(f_i^{\xi_0^{R,(i)}(R_0),\ldots,\xi_{m-1}^{R,(i)}(R_{m-1})}(\xi_0^{(i)}(x_0, y_0), \ldots, \xi_{n-1}^{(i)}(x_{n-1}, y_{n-1})))$, where the functions $\xi^{(\cdot)}$ are elements of $\mathcal{F}^+$. Therefore, we can rewrite the right-hand side of Equation 7 as $\Pr_{\mathbf{R} \leftarrow\!\!\$\ \mathbb{F}^m}(f_0'^{\mathbf{R}}(\mathbf{x}, \mathbf{y}) = p_0', \ldots, f_{d'-1}'^{\mathbf{R}}(\mathbf{x}, \mathbf{y}) = p_{d'-1}') = \Pr_{\mathbf{R} \leftarrow\!\!\$\ \mathbb{F}^m}(\xi^{(j)}(f_j^{\xi^{R,(j)}(\mathbf{R})}(\mathbf{x}'^{(h)}, \mathbf{y}'^{(j)})) = p_j' \mid \forall j \in [d'-1])$ where we use the notation $\mathbf{x}'^{(i)}, \mathbf{y}'^{(i)}$ and $\xi^{R,(i)}(\mathbf{R})$ for the corresponding

terms $(\xi_0^{(i)}(x_0, y_0), \ldots, \xi_{n-1}^{(i)}(x_{n-1}, y_{n-1}))$ and $(\xi_0^{R,(i)}(R_0), \ldots, \xi_{m-1}^{R,(i)}(R_{m-1}))$ respectively for every $i \in [n]$. Due to the d-(S)NI property, for every chosen probe p_i there exists a simulator S_i that, for every input tuple, can simulate one probe given just one share per input sharing. Next, we can rewrite $\Pr_{\mathbf{R} \leftarrow\$ \mathbb{F}^m}(\xi^{(j)}(f_j^{\xi^{R,(j)}(\mathbf{R})}(\mathbf{x}'^{(h)}, \mathbf{y}'^{(j)})) = p'_j \mid \forall j \in [d'-1]) = \Pr_{\mathbf{R} \leftarrow\$ \mathbb{F}^m}(\xi^{(j)}(S_j^{\tilde{R}_j(\mathbf{x}'^{(j)}, \xi^{R,(j)}(\mathbf{R}))}(\tilde{\mathbf{a}}^{(j)}, \tilde{\mathbf{b}}^{(j)})) = p'_j \mid \forall j \in [d'-1])$, We now define the simulator S' for the faulted circuit as the one sampling $\mathbf{R} \leftarrow\$ \mathbb{F}^m$ and computing $(p'_0, \ldots, p_{d'-1})$ as $\xi^{(j)}(S_j^{\tilde{R}_j(\mathbf{x}'^{(j)}, \xi^{R,(j)}(\mathbf{R}))}(\tilde{\mathbf{a}}^{(j)}, \tilde{\mathbf{b}}^{(j)})) = p'_j$ for all $j \in [d'-1]$. We know that $\tilde{\mathbf{a}}^{(0)}, \ldots, \tilde{\mathbf{a}}^{(d'-1)}$ (and $\tilde{\mathbf{b}}^{(0)}, \ldots, \tilde{\mathbf{b}}^{(d'-1)}$) contain at most one share per input each, therefore all together correspond to at most d' shares per input sharing. Moreover, item 1 and 3 guarantee that $(\tilde{R}_0(\mathbf{x}'^{(0)}, \mathbf{y}'^{(0)}, \xi^{R,(0)}(\mathbf{R})), \ldots, \tilde{R}_{d'-1}(\mathbf{x}'^{(d'-1)}, \mathbf{y}'^{(d'-1)}, \xi^{R,(d'-1)}(\mathbf{R})))$ depends on at most d' input variables, which are taken as input from S' as $\tilde{\mathbf{a}}^{(0)}, \ldots, \tilde{\mathbf{a}}^{(d'-1)}$ and $\tilde{\mathbf{b}}^{(0)}, \ldots, \tilde{\mathbf{b}}^{(d'-1)}$. This makes S' a valid simulator, and concludes the proof. □

Finally, we prove the fault-robustness of the gadget, i.e., the fact that faults will propagate after being introduced.

Lemma 12. *The gadget* $\mathsf{BGW}_{\hat{A} \leftarrow \mathsf{Comp}\hat{A}^{(2d,d,n)}}$ *(cf. Alg. 7) is* (e, p)*-fault-robust with respect to* $\mathcal{F}^+$*, where* $p \leq \sum_{\ell=0}^{s_2} \binom{n}{e-\ell} \frac{1}{|\mathbb{F}|^e}$.

Proof. Let $\boldsymbol{F} = \boldsymbol{F}^{\text{unf}} + \boldsymbol{\xi}^F$ and $\boldsymbol{G} = \boldsymbol{G}^{\text{unf}} + \boldsymbol{\xi}^G$ denote our (possibly faulted) input sharings, where $\boldsymbol{F}^{\text{unf}}, \boldsymbol{G}^{\text{unf}}$ denote valid encodings and $\boldsymbol{\xi}^F, \boldsymbol{\xi}^G$ refer to fault vectors in $\mathcal{F}^+$. We analyze the gadget step by step.

SWMult. Faults injected in **SWMult** only affect one share at a time, as we calculate $Q_i \leftarrow F_i \cdot G_i$ for all $i \in [n]$ individually. However, since one fault in $\boldsymbol{Q}$ affects all output shares $\boldsymbol{H}$, we need to consider the faults in more detail. Namely, we calculate the probability that $(Q_i)_{i\in[n]} = (F_i \cdot G_i + \xi_i^Q)_{i\in[n]}$ results in a sharing which is not equal to $(F_i^{\text{unf}} \cdot G_i^{\text{unf}})_{i\in[n]}$ but of degree up to $2d$. If $\boldsymbol{F}$ and $\boldsymbol{G}$ are valid sharings, but $\exists i \in [n] : \xi_i^Q \neq 0$ and $\exists in \in [n] : Q_i \neq F_i \cdot G_i$, then $\boldsymbol{Q}$ does not correspond to a polynomial of degree less then or equal to $2d$, since $\mathsf{weight}(\boldsymbol{\xi}^Q) \leq e$ by assumption and by Fundamental Lemma it follows that $\boldsymbol{\xi}^Q$ corresponds to a polynomial of degree at least $n - e$ which is greater than $2d$ by definition of $n \geq 2d + 1 + e$. Hence, the degree of $q(X)$ is greater than or equal to $2d$. If at least one of the sharings $\boldsymbol{F}$ or $\boldsymbol{G}$ is not valid, we aim to calculate the probability that an adversary can set the faults in a way such that $\boldsymbol{Q}$ results in a polynomial of degree up to $2d$. In order to do so, recall that we consider $\mathcal{F}^+$ with wire-independent faults. Thus, an adversary can add a value to a wire, which can be chosen arbitrarily, but needs to be independent of the actual value on this wire. Moreover, as discussed in Section 3, a polynomial of degree $2d$ can be built by choosing $2d+1$ shares arbitrarily, and calculating the remaining $n - (2d + 1) \geq e$ shares correspondingly. Keeping this in mind, we can bound the probability that $\mathbf{SWMult}(\boldsymbol{F}, \boldsymbol{G})$ corresponds to a degree-$2d$ polynomial by considering the number all possible subsets of size e of $[n]$, $\binom{n}{e}$, and multiplying it with the probability that a vector of size e is correct by chance. Due to the wire-independent faults, the probability is at best $\frac{1}{|\mathbb{F}|}$. Similarly, we calculate the probability of $\boldsymbol{Q}$ resulting in a sharing with respect to a polynomial of degree up to $2d$ after an additional injection of $s_2 \leq e$ faults. All together, we can bound the probability of $\boldsymbol{Q}$ corresponding to a polynomial of degree up to $2d$ by $p \leq \sum_{\ell=0}^{s_2} \binom{n}{e-\ell} \frac{1}{|\mathbb{F}|^e}$. We consider further propagation of these faults later.

Lines 5, 7. We continue with faults injected in lines 5 and 7 or during the computation of $(Y_i^{(j)})_{i\in[n]} \leftarrow$ opt-**ZEnc**. In Lemma 5, we showed that the gadget opt-**ZEnc** is e-fault-robust with respect to $\mathcal{F}$. In more detail, we showed that $\boldsymbol{Y}^{(j)} + \boldsymbol{\omega}^{(j)} + \boldsymbol{\omega}'^{(j)} \leftarrow T[\text{opt-}\mathbf{ZEnc}]$ where $\boldsymbol{\omega}'^{(j)} = \mathbf{0}$ and $\mathsf{weight}(\boldsymbol{\omega})^{(j)} \in [0, s^{\text{opt-}\mathbf{ZEnc}(j)}]$, where $s^{\text{opt-}\mathbf{ZEnc}(j)}$ denotes the number of injected faults during the calculation (and to the output of) the j-th opt-**ZEnc**. Moreover,

one fault injected into one share $Y_i^{(j)}$ affects at most one output share.
Lines 10, 11, 12. Faults in lines $10, 11, 12$ affect at most one output share. We denote the corresponding fault vector as $\boldsymbol{\xi}^H$ and the number of injected faults as s^H. Notably, $\mathsf{weight}(\boldsymbol{\xi})^H \leq s^H$.
Fault Propagation of $\hat{A}$ and $\boldsymbol{Q}$. Finally, we consider the fault propagation during the calculation of $\hat{A} \leftarrow \mathsf{Comp}\hat{A}(2d, d, n)$ and the propagation of faults within $\boldsymbol{Q}$.
We begin by taking a deeper look into faults injected into $\hat{A}$. In [ABE+25] it is shown that faults injected to $\hat{A} \leftarrow \mathsf{Comp}\hat{A}(2d, d, n)$ have one of the following three properties: A fault (i) only affects the underlying randomness, but does not change the distribution of the elements. Since $\mathsf{Comp}\hat{A}$ only has inputs that are considered as public parameters, these faults are ineffective, or (ii) affects $\hat{A}$ in at most one component, i.e., $\hat{A}_{i,j}$, or (iii) affects one row of $\hat{A}$, i.e., $\hat{A}_{i,:}$. Let $s^{\hat{A}}$ denote the number of faults injected during the computation of $\hat{A} \leftarrow \mathsf{Comp}\hat{A}(2d, d, n)$ and the number of faults injected into the output $\hat{A}$. Due to the above, we can write $\hat{A} + \xi^{\hat{A}} \leftarrow T[\mathsf{Comp}\hat{A}(2d, d, n)]$. Altogether we set $\boldsymbol{\omega} \coloneqq \sum_{j=0}^{n-1} \boldsymbol{\omega}^{\mathsf{opt\text{-}ZEnc}(j)} + \boldsymbol{\xi}^H + (\xi_{i,j}^{\hat{A}} Q_j)_{i\in[n]}^T$ and it follows that $\mathsf{weight}(\boldsymbol{\omega}) \in [0, s_1]$, where $s_1 = \sum_{j=0}^{n-1} s^{\mathsf{opt\text{-}ZEnc}(j)} + s^H + s^{\hat{A}}$. This finishes the linear part.
Now, consider the propagation of faults injected within the calculation of Q, and to Q. Recall that the faulted sharing $(Q_i)_{i\in[n]} \neq (F_i \cdot G_i)_{i\in[n]}$ corresponds to a polynomial of degree up to $2d$ with probability p. In this case, due to the correctness of $\hat{A}$, the faults are not detectable anymore. Otherwise, $\boldsymbol{Q}$ corresponds to a polynomial of degree $> 2d$. We set $\mathcal{S} = \{0\} \cup [d+1, n]$ By definition of $\hat{A}$ it holds true that

$$h_m = \sum_{i=0}^{n-1}\Big(\sum_{j=0}^{n-1} \lambda_{i,j}^{(2d\to d)} Q_j\Big) V_{m,i}^{-1} + \underbrace{\sum_{i=0}^{n-1}\Big(\sum_{j=0}^{n-1}\sum_{k=2d+1}^{n-1}\sum_{\ell\in\mathcal{S}} r_l^{(k)} x_i^{\ell} V_{k,j}^{-1} Q_j V_{m,i}^{-1}\Big)}_{\star)} + \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} Y_i^{(j)} V^{-1}$$

where $\star) = \sum_{k=2d+1}^{n-1} r_m^{(k)} \cdot q_k$ and q_k denotes the k-th coefficient of $\boldsymbol{Q}$. Let s_2 denote the number of linear faults inserted after line 10. Since $r_m^{(k)}$ is uniformly i.i.d. sampled for all $k \in [2d+1, m], m \in \mathcal{S}$, the polynomial $h(X)$ has at least $e - s_2$ coefficients with indices from $d+1, \ldots, n-1$ that are uniformly distributed and independent of each other.

Thus, all together, if $\boldsymbol{\xi}^F \neq 0$ or $\boldsymbol{\xi}^G \neq 0$, it exists a vector $\boldsymbol{\omega'}$ such that $H + \boldsymbol{\omega} + \boldsymbol{\omega'}$ is the output of the faulted gadget $\mathbf{BGW}_{\hat{A}\leftarrow\mathsf{Comp}\hat{A}(2d,d,n)}$, where $\boldsymbol{\omega'}$ is indistinguishable to a sharing that corresponds to a polynomial $c(X)$ with e coefficients out of $c_{d+1}, \ldots, c_{n-1}$ are chosen uniformly at random and independent of each other except with probability p. Hence, the resulting distribution is p-indistinguishable, finishing the proof. □

Note here, that the probability $\sum_{\ell=0}^{s_2} \binom{n}{e-\ell} \frac{1}{|\mathbb{F}|^e}$ can roughly be upper bounded by $n \cdot (n/|\mathbb{F}|)^e$. For sufficiently small $n \ll |\mathbb{F}|$, this is an exponentially small bound.

5.2 LaOla-Like Multiplication

In this section, we consider the multiplication gadget initially introduced by Berndt et al. [BEF+23] and refined very recently by [ABE+25]. Their core idea can be outlined as follows: Instead of first share-wise multiplying the shares F_i and G_i for $i \in [n]$, which results in a degree $2d$ polynomial $f(X) \cdot g(X)$, and then reducing the degree of polynomial $f(X) \cdot g(X)$ back to d (cf. Sec. 5.1), Berndt et al. [BEF+23] *securely* reduce the degree of the degree-d polynomials $f(X)$ and $g(X)$ to $d/2$ *before* performing share-wise multiplications. Using such a pre-degree reduction technique, they are able to keep the number of necessary shares at an optimal $n = d + e + 1$ (instead of setting $n = 2d + e + 1$ as the BGW-like multiplication discussed in Sec. 5.1).

More specifically, they construct two polynomials $f'(X)$ and $f''(X)$, each of degree $\geq d$, such that $f(u_i) = f'(u_i) + f''(u_i) = v_i^{(F)}$ for $i \in [k]$, and $f(X)$ has degree $d/2$. Similarly, for $g(X)$, they construct two degree $\geq d$ polynomials $g'(X), g''(X)$, such that $g(u_i) = g'(u_i) + g''(u_i) = v_i^{(G)}$ for $i \in [k]$, and $g(X)$ has degree $d/2$. Furthermore, the product of the secrets satisfies the following:

$$\begin{aligned} v_i^{(F)} \cdot v_i^{(G)} &= f(u_i) \cdot g(u_i) = (f'(u_i) + f''(u_i)) \cdot (g'(u_i) + g''(u_i)) \\ &= f'(u_i) \cdot g'(u_i) + f''(u_i) \cdot g'(u_i) + f'(u_i) \cdot g''(u_i) + f''(u_i) \cdot g''(u_i). \end{aligned}$$

This shows that we can perform share-wise multiplications of $f'(u_i) \cdot g'(u_i)$, $f''(u_i) \cdot g'(u_i)$, $f'(u_i) \cdot g''(u_i)$ and $f''(u_i) \cdot g''(u_i)$. By adding these results together, we obtain a sharing of $v_0^{(F)} \cdot v_0^{(G)}, \ldots, v_{k-1}^{(F)} \cdot v_{k-1}^{(G)}$. Since the sum of polynomials, i.e., $f'(X)+f''(X)$, $g'(X)+g''(X)$ are of degree $d/2$, the resulting polynomials after multiplication will have degree d, because the share-wise addition operation will not increase the degree.
We can summarize this approach using the following algorithms. Please note that the underlying algorithms for share-wise addition **SWAdd** (cf. Alg. 15) and share-wise multiplication **SWMult** (cf. Alg. 16) are identical to those presented by Faust et al. [FOWZ25], as they do not involve any zero-encodings. We recall them in Supp. D for completeness.

SplitRed. In the following, we consider the gadget **SplitRed** (cf. Alg. 8), which takes shares of a degree-d polynomial as inputs and splits this polynomial into two polynomials, each of degree $\geq d$, while preserving the underlying secrets. We highlight the modifications made to gadget **SplitRed** (cf. Alg. 8) compared to [ABE+25] in **blue**. To allow for a wider range of parameters, we also slightly adapted the algorithm to work for odd values of n by appropriately rounding up or down certain instances of $n/2$. Note that the values $\hat{A}_{i,j}$, $i, j \in [n]$ correspond to a matrix, which *unmask* the shares $(F_i)_{i\in[n]}$ that correspond to a degree-d polynomial, and then *re-mask* the embedded secrets into a polynomial of degree-$d/2$, while re-randomizing the high order coefficients $d/2+1, \ldots, n-1$ in case the input sharing is of degree higher than $d+1$. Recall that both the *unmask* and *re-mask* operations correspond to matrix multiplications. Hence, we can pre-calculate the corresponding matrix $\hat{\lambda}$ and execute the *unmask* and *re-mask* in one step to make sure that an adversary does not gain intermediate information about the underlying secrets during unmasking. A detailed discussion on the factors $\hat{A}_{i,j}$, $i, j \in [n]$ is provided in [ABE+25].

Please note that replacing **ZEnc** with opt-**ZEnc** does not affect the security proofs, as they rely solely on the independence of the elements. Both **ZEnc** and opt-**ZEnc** produce a zero-encoding. Specifically, both output n shares that are $(d+1-k)$-wise independent and correspond to a polynomial with embedded secrets $(0_0, \ldots, 0_{k-1})$. However, since we aim to apply our construction to reduce randomness as well, it is necessary to revisit all the security proofs.

Lemma 13. *The modified gadget* $\textbf{SplitRed}_{\mathsf{Comp}\hat{A}(d,\frac{d}{2},n)}$ *(cf. Alg. 8) is correct. Specifically,* $f'(u_m) + f''(u_m) = f(u_m)$ *for* $m \in [k]$*, assuming no faults are induced.*

Proof sketch. The proof is similar to the one presented in [ABE+25], as we only changed the way to create the zero-encodings, and outputs of opt-**ZEnc** are zero-encodings (see full version for details). Formally, one can see that F'_i contains the terms $\hat{G}_i^j$ and F''_i contains $-\hat{G}_i^j$. Using this, the ℓ-th coefficients f'_ℓ and f''_ℓ of the underlying polynomials contain the terms $\hat{G}_i^j \cdot V_{\ell,i}^{-1}$, resp. $-\hat{G}_i^j \cdot V_{\ell,i}^{-1}$, which cancel out in the addition. Finally, $\hat{A}$ is applied, guaranteeing the correct output degree by correctness of $\mathsf{Comp}\hat{A}(d, \frac{d}{2}, n)$, since no input is faulted by assumption. □

Lemma 14. *The modified gadget* $\textbf{SplitRed}_{\mathsf{Comp}\hat{A}(d,\frac{d}{2},n)}$ *(cf. Alg. 8) is* $(d+1-k)$-frNI *with respect to* $\mathcal{F}^+$*.*

Algorithm 8 $\mathbf{SplitRed}_{\hat{A}\leftarrow\mathsf{Comp}\hat{A}(d,\frac{d}{2},n)}$ for $n = d+e+1$ and $k \le \lfloor \frac{d}{2} \rfloor$

Input : *Degree-d* shares of $\boldsymbol{v}^{(F)} = \left(v_0^{(F)}, \ldots, v_d^{(F)}\right)$ as $(F_i)_{i\in[n]}$

Output : *Degree-d* shares of $\boldsymbol{v}' = (v'_0, \ldots, v'_d)$ as $(F'_i)_{i\in[n]}$ and *degree-d* shares of $\boldsymbol{v}'' = (v''_0, \ldots, v''_d)$ as $(F''_i)_{i\in[n]}$ such that the coefficients $f_j = f'_j + f''_j$ for $j \in \left[\lfloor \frac{d}{2} \rfloor + 1\right]$ and $v_i = v'_i + v''_i$ for $i \in [n]$

initialize $(F'_i)_{i\in[n]}$, $(F''_i)_{i\in[n]}$
for $j \in \lceil n/2 \rceil$ **do**
  if $j < (d+1-k)$ **then**
    $\left(\hat{G}_i^j\right)_{i\in[n]} \leftarrow$ opt-$\mathbf{ZEnc}_n^{d-j}$
  else
    $\left(\hat{G}_i^j\right)_{i\in[n]} \leftarrow$ opt-$\mathbf{ZEnc}_n^{d}$
for $j \in \lfloor n/2 \rfloor$ **do**
  if $j < \frac{d}{2} - k + 1$ **then**
    $\left(\tilde{G}_i^j\right)_{i\in[n]} \leftarrow$ opt-$\mathbf{ZEnc}_n^{\frac{d}{2}-j}$
  else
    $\left(\tilde{G}_i^j\right)_{i\in[n]} \leftarrow$ opt-$\mathbf{ZEnc}_n^{\frac{d}{2}}$
  $\left(G_i^j\right)_{i\in[n]} \leftarrow \left(\tilde{G}_i^j\right)_{i\in[n]} + \left(\hat{G}_i^j\right)_{i\in[n]}$
if $n \bmod 2 = 1$ **then**
  $(G^{\lfloor n/2 \rfloor - 1})_{i\in[n]} \leftarrow (G^{\lfloor n/2 \rfloor - 1})_{i\in[n]} + (\hat{G}^{\lceil n/2 \rceil - 1})_{i\in[n]}$
for $j \in \lfloor n/2 \rfloor$ **do**
  for $i \in [n]$ **do**
    $\mathcal{F}'^j_i \leftarrow \hat{A}_{i,j} \cdot F_j$
for $j \in \lfloor n/2 \rfloor$ **do**
  $\left(\mathcal{F}_i^j\right)_{i\in[n]} \leftarrow \left(\mathcal{F}'^j_i\right)_{i\in[n]} + \left(G_i^j\right)_{i\in[n]}$
for $j \in \lfloor n/2 \rfloor$ **do**
  $(F'_i)_{i\in[n]} \leftarrow (F'_i)_{i\in[n]} + \left(\mathcal{F}_i^j\right)_{i\in[n]}$
for $j \in \lceil n/2 \rceil$ **do**
  if $j < \frac{d}{2} - k + 1$ **then**
    $\left(\tilde{G}_i^{j+\lfloor n/2 \rfloor}\right)_{i\in[n]} \leftarrow$ opt-$\mathbf{ZEnc}_n^{\frac{d}{2}-j}$
  else
    $\left(\tilde{G}_i^{j+\lfloor n/2 \rfloor}\right)_{i\in[n]} \leftarrow$ opt-$\mathbf{ZEnc}_n^{\frac{d}{2}}$
  $\left(G_i^{j+\lfloor n/2 \rfloor}\right)_{i\in[n]} \leftarrow \left(\tilde{G}_i^{j+\lfloor n/2 \rfloor}\right)_{i\in[n]} - \left(\hat{G}_i^j\right)_{i\in[n]}$
for $j \in \lceil n/2 \rceil$ **do**
  for $i \in [n]$ **do**
    $\mathcal{F}'^{j+\lfloor n/2 \rfloor}_i \leftarrow \hat{A}_{i,j+\lfloor n/2 \rfloor} \cdot F_{j+\lfloor n/2 \rfloor}$
for $j \in \lceil n/2 \rceil$ **do**
  $\left(\mathcal{F}_i^{j+\lfloor n/2 \rfloor}\right)_{i\in[n]} \leftarrow \left(\mathcal{F}'^{j+\lfloor n/2 \rfloor}_i\right)_{i\in[n]} + \left(G_i^{j+\lfloor n/2 \rfloor}\right)_{i\in[n]}$
for $j \in \lceil n/2 \rceil$ **do**
  $(F''_i)_{i\in[n]} \leftarrow (F''_i)_{i\in[n]} + \left(\mathcal{F}_i^{j+\lfloor n/2 \rfloor}\right)_{i\in[n]}$
return $(F'_i)_{i\in[n]}$, $(F''_i)_{i\in[n]}$

Proof sketch. The claim follows from [ABE+25], as our version of $\mathbf{SplitRed}_{\mathsf{Comp}\hat{A}(d,\frac{d}{2},n)}$ only adapts the used zero-encodings. However, since the zero-encodings do not obtain any input, they are independent of the secret. Moreover, in the corresponding proof in [ABE+25] (and [FOWZ25, Algorithm 7], respectively), if one element within the calculation of the zero-encoding is probed, the simulator reveals the complete zero-encoding. Now, computing the zero-encoding with respect to the shares, as in opt-**ZEnc**, instead of with respect to the coefficients, as done in **ZEnc**, does not change the zero-encoding itself, as shown in Section 3. Our second modification, which involves using less randomness, requires some additional adjustments. However, the reasoning is equivalent to the one presented in Lemma 6: for all probed elements $(\hat{G}_i^j)_{i\in[n]}$ or $(\tilde{G}_i^j)_{i\in[n]}$, the adversary has fewer probes remaining, limiting his ability to extract information. A detailed proof is provided in Supp. B. □

Share-Wise Computations. Using **SplitRed**, we can calculate degree $\geq d$ shares of $\boldsymbol{v}' = (v'_0, \ldots, v'_d)$ as $(F'_i)_{i\in[n]}$ and degree $\geq d$ shares of $\boldsymbol{v}'' = (v''_0, \ldots, v''_d)$ as $(F''_i)_{i\in[n]}$ such that $f_j = f'_j + f''_j$ for $j \in [\frac{d}{2}+1]$. Recall that we want to obtain a multiplication of two sharings $(F_i)_{i\in[n]}$ and $(G_i)_{i\in[n]}$. Now, we can use **SplitRed** for both shares, and obtain $(F'_i)_{i\in[n]}$, $(F''_i)_{i\in[n]}$, $(G'_i)_{i\in[n]}$ and $(G''_i)_{i\in[n]}$. We now share-wise multiply all four combinations of $(F'_i)_{i\in[n]} \cdot (G'_i)_{i\in[n]}$, $(F'_i)_{i\in[n]} \cdot (G''_i)_{i\in[n]}$, $(F''_i)_{i\in[n]} \cdot (G'_i)_{i\in[n]}$ and $(F''_i)_{i\in[n]} \cdot (G''_i)_{i\in[n]}$ using the standard share-wise multiplication. Hence, the only thing left to do is to securely add the four vectors to each other. This is exactly what we do in the following Algorithm 9.

Algorithm 9 SWComp for $n = d + e + 1$ and $k \leq d$ [FOWZ25, Algorithm 8]

Input : 4 degree $> d$ sharings of $\boldsymbol{v}^{(j)}$ as $\left(H_i^{(j)}\right)_{i\in[n]}$, with $j \in [4]$

Output : *Degree-d* sharing $(Q_i)_{i\in[n]}$ of $\boldsymbol{v}^{(0)} + \boldsymbol{v}^{(1)} + \boldsymbol{v}^{(2)} + \boldsymbol{v}^{(3)}$

1 $(Q_i)_{i\in[n]} \leftarrow$ opt-**sZEnc**$_n^d$

2 $(Q_i)_{i\in[n]} \leftarrow \left(\left(\left(\left((Q_i)_{i\in[n]} + \left(H_i^{(0)}\right)_{i\in[n]}\right) + \left(H_i^{(1)}\right)_{i\in[n]}\right) + \left(H_i^{(2)}\right)_{i\in[n]}\right) + \left(H_i^{(3)}\right)_{i\in[n]}\right)$

3 **return** $(Q_i)_{i\in[n]}$

The correctness of Algorithm 9 follows directly from the correctness of share-wise addition (cf. [FOWZ25, Algorithm 1]) and the fact that opt-**sZEnc** is a zero-encoding, as shown in Section 4. Furthermore, the fault-invariance of Algorithm 9 follows directly from the fault-invariance of opt-**sZEnc** and the fact that we only use share-wise addition afterwards.

LaOla Multiplication. Now we have all the necessary components to restate the multiplication gadget from [FOWZ25, BEF+23], incorporating our modifications. Note that the multiplication Algorithm is identical to the one presented in [FOWZ25, Algorithm 9]. However, we modified the underlying algorithms **SplitRed** and **SWComp**, and thus need to prove that our changes do not compromise the security of the gadget.

Altogether, Algorithm 17 takes two sharings, $(F_i)_{i\in[n]}$ and $(G_i)_{i\in[n]}$ as inputs. Then, **SplitRed** is used for both, to obtain $(F'_i)_{i\in[n]}$, $(F''_i)_{i\in[n]}$, $(G'_i)_{i\in[n]}$ and $(G''_i)_{i\in[n]}$. Afterwards, all four combinations of $(F'_i)_{i\in[n]} \cdot (G'_i)_{i\in[n]}$, $(F'_i)_{i\in[n]} \cdot (G''_i)_{i\in[n]}$, $(F''_i)_{i\in[n]} \cdot (G'_i)_{i\in[n]}$ and $(F''_i)_{i\in[n]} \cdot (G''_i)_{i\in[n]}$ are share-wise multiplied. As the last step, the resulting sharings are share-wise added via **SWComp**. The output of **SWComp** equals the final output of multiplication. A detailed description of this algorithm is recalled in Supp. D.1.

Lemma 15. *The multiplication gadget* $\mathbf{Mult}_{\mathbf{k-Input}}$ *is* $(d+1-k)$-frSNI *with respect to* $\mathcal{F}^+$ *and e-fault-robust with respect to* $\mathcal{F}^{\mathsf{ind}}$.

Proof sketch. The algorithm $\mathbf{Mult_{k-Input}}$ consists of three subroutines, **SplitRed**, **SWMult** and **SWComp**. Since e-fault-robustness of these subroutines has been shown in [ABE+25], their proof can be used directly. Namely, [ABE+25] state the following:

Assume (at least one call of) the algorithm $\mathbf{ZEnc}_n^m$ outputs n shares with respect to a polynomial whose coefficients $c_i, i \in [1, m+1]$ are indistinguishable from chosen independently and uniformly at random from $\mathbb{F}$ as well as independent of the injected faults. Furthermore, assume the algorithm **sZEnc** outputs a sharing with respect to a polynomial whose coefficients $c_i, i \in [1, d+1]$ are chosen independently and uniformly at random from $\mathbb{F}$. Moreover, assume that **ZEnc** and **sZEnc** are e-fault-robust with respect to fault set $\mathcal{F}^+$. Then, the **SplitRed** algorithm is e-fault-robust with respect to $\mathcal{F}^+$.

Our algorithms only differ in the substitution from **ZEnc** to opt-**ZEnc**, the substitution from **sZEnc** to opt-**sZEnc**, as well as the consideration of k secrets instead of one. However, as shown in [FOWZ25], we can lift the properties from one to k secrets. Hence, it is sufficient to show that our replacements, opt-**ZEnc** and opt-**sZEnc**, satisfy the properties that are required for the corresponding zero-encodings **ZEnc** and **sZEnc** to finish the proof. Now, opt-**ZEnc** as well as opt-**sZEnc** are e-fault-robust as shown in Lemma 5 and 8. Moreover, opt-**sZEnc** outputs a sharing with respect to a polynomial whose coefficients are chosen independently and uniformly at random from $\mathbb{F}$ by the correctness of the gadget. Finally, by the correctness of opt-**ZEnc**, the call of opt-$\mathbf{ZEnc}_n^{\frac{d}{2}}$ results in a sharing with respect to coefficients $c_1, \ldots, c_{\frac{d}{2}}$ indistinguishable from uniformly random and independent of each other. Moreover, as shown in the proof of Lemma 5, the output sharing of opt-**ZEnc** corresponds to a polynomial whose coefficients $p_1, \ldots, p_{\frac{d}{2}}$ remain indistinguishable from uniformly distributed independent of the faults. Moreover, $\mathbf{Mult_{k-Input}}$ is $(d+1-k)$-frSNI as opt-**ZEnc** satisfies the same properties as **ZEnc** required within the proof of t-frSNI in [ABE+25]. As we only exchange opt-**ZEnc** to **ZEnc**, the statement follows directly. □

6 Evaluation

We evaluate the impact of our zero-encodings on the **LaOla** multiplication and an AES-Sbox in practice. Performance benchmarks in terms of cycle and randomness count for a wide range of parameters and configurations are given. The security is evaluated independently of the proofs, by a physical side-channel evaluation and simulated fault injections.

6.1 Implementation

Our C implementation is generic in $d \geq 1, e \geq 0$ and we always consider the case that $d = t$, where t is the masking order. The implementation allows to switch between the original zero-encodings and our improved ones, as well as hardened operations for Arm Cortex-M4. Besides, we provide scripts to generate support points for arbitrary fields, optionally supporting efficient squaring via the Frobenius automorphism (FA). Field multiplication uses exp-log tables [Bel23] for efficient, constant-time multiplication. Inverses in GF(2^8) are computed by a sequence of squares and multiplications that is optimal in the number of multiplications [DK09, Figure 1]. The affine transformation of the AES-Sbox is as in [PR11, Algorithm 3]. One SBox invocation requires 4 multiplications and 14 squarings, using either FA or **LaOla**. For odd d, the first **SplitRed** in **LaOla** is invoked for $\lceil \frac{d}{2} \rceil$, the latter for $\lfloor \frac{d}{2} \rfloor$, returning $\left((G_i)_{i \in [n]}, (0)_{i \in [n]} \right)$ for $\lfloor \frac{d}{2} \rfloor = 0$.

Tables must be placed in SRAM to work around FLASH caches to ensure constant-time execution for qualitative measurements of execution time and power consumption. The implementation is moderately optimized manually and using the compiler options `-O2 -funroll-loops -fpeel-loops` with GCC `12.2`. Cryptographically secure randomness is pre-sampled using a PRNG based on a hardware-accelerated AES. Note that pre-sampling

Table 1: Number of cycles and random bytes required for the execution of the masked Sbox and its subcomponents on Cortex-M4 with pre-sampled randomness. Cells contain performance values for opt-**ZEnc** (bold, left) and **ZEnc** (regular, right). For odd d, the implementation invokes **SplitRed** and **ZEnc** with both $\lfloor \frac{d}{2} \rfloor$ and $\lceil \frac{d}{2} \rceil$ equally often, hence, the average performance is displayed.

	$e=0$						$e=1$						$e=2$					
Component	$d=1$		$d=2$		$d=3$		$d=1$		$d=2$		$d=3$		$d=1$		$d=2$		$d=3$	
`sbox w/ Frobenius`	**2 212**	2 287	**8 907**	17k	**18k**	34k	**8 190**	9 499	**19k**	29k	**37k**	62k	**17k**	18k	**34k**	46k	**54k**	83k
#randombytes	**8**	8	**52**	64	**84**	108	**36**	36	**76**	88	**136**	176	**68**	68	**140**	152	**184**	224
`sbox w/o Frobenius`	**6 740**	7 065	**35k**	71k	**73k**	148k	**31k**	36k	**80k**	122k	**157k**	268k	**70k**	72k	**143k**	199k	**232k**	365k
#randombytes	**36**	36	**234**	288	**378**	486	**162**	162	**342**	396	**612**	792	**306**	306	**630**	684	**828**	1 008
`laola_mult`	**356**	374	**1 905**	3 898	**4 034**	8 154	**1 660**	1 961	**4 404**	6 746	**8 677**	15k	**3 828**	3 961	**7 909**	11k	**13k**	20k
#randombytes	**2**	2	**13**	16	**21**	27	**9**	9	**19**	22	**34**	44	**17**	17	**35**	38	**46**	56
`split_red`	**172**	182	**716**	1 392	**1 578**	2 812	**552**	654	**1 272**	2 001	**2 904**	4 994	**994**	1 020	**2 153**	3 220	**3 638**	6 121
#randombytes	**1**	1	**5**	6	**7.5**	9	**2**	2	**5**	6	**10**	13.5	**2**	2	**8**	9	**10**	13.5
`sZenc`	**33**	41	**94**	391	**348**	2 146	**41**	71	**196**	515	**498**	2 441	**101**	88	**295**	594	**730**	3 008
#randombytes	**1**	1	**3**	4	**6**	9	**1**	1	**3**	4	**6**	9	**1**	1	**3**	4	**6**	9
`zEnc`$(d,0)$	**27**	42	**50**	307	**105**	683	**35**	77	**109**	395	**162**	797	**98**	92	**144**	485	**247**	924
#randombytes	**1**	1	**2**	2	**3**	3	**1**	1	**2**	2	**3**	3	**1**	1	**2**	2	**3**	3
`zEnc`$(\frac{d}{2},0)$	**20**	26	**35**	88	**97**	247	**24**	46	**94**	108	**132**	310	**56**	52	**121**	128	**158**	356
#randombytes	**0.5**	0.5	**1**	1	**1.5**	1.5	**0.5**	0.5	**1**	1	**1.5**	1.5	**0.5**	0.5	**1**	1	**1.5**	1.5

random numbers allows for a clearer vulnerability analysis, as it allows measuring the performance of the actual gadgets instead of the PRNG.

Our implementation will be open-sourced for further research and to serve as a helpful starting point in the significant engineering challenge to implement not only the main algorithms but also the underlying generation of various sub-components. Furthermore, it can serve as an interesting starting point for researching fault and combined resilience given the algorithmic resilience to such attacks for $e \geq 1$.

6.2 Performance Benchmark

To understand the general applicability of polynomial masking using the **LaOla** approach as well as the individual improvements of our newly designed gadgets, we analyze the performance of our implementations using a wide range of parameters and configurations. More concretely, we measure and analyze the impact on cycle count and randomness consumption of the different settings for Frobenius squaring and the used zero-encodings (ours vs. [ABE+25]) for parameters $d \in [1, 3]$, $e \in [0, 2]$.

The cycle count is repeatedly measured by reading the SysTick timers and constant-time operation is carefully checked. For the number of random bytes consumed, the number of byte-reads to the buffer holding the pre-sampled randomness is counted.

Discussion. The results of the performance measurements are shown in Table 1. The upper part of the table compares the implementation of the complete AES-Sbox, while the lower part addresses the involved sub-components. For each setting (d, e) and each component, the upper row contains the cycle count while the lower row contains the number of consumed random bytes. For each setting of d, the column contains two performance values, the left corresponding to the implementation's performance with the improved opt-**ZEnc** gadget presented in this work, whereas the right value is for the same implementation using the **ZEnc** gadget of [BEF+23].

FA allows squaring an encoded value locally in linear time without consuming randomness instead of executing an expensive masked multiplication. Thus, the Sbox using FA squaring is, across all settings, more than 3× faster and requires 4.5× less randomness.

For $d = 1$, our improvements have no impact on randomness consumption, but slightly decrease the cycle count (3.5%–13.8%) due to the improved computation opt-**ZEnc** (Section 3). For $d > 1$ with FA, the usage of opt-**ZEnc** improves performance by 1.39×–2.03×, with an average of 1.71×, with larger increases when disregarding additional shares for fault resilience and increasing d. When disabling FA, performance increases follow a

similar pattern, ranging from 1.37×–1.92×, averaging 1.65×. These performance increases are attributed to significantly faster invocations of opt-**ZEnc** and opt-**sZEnc** at higher orders. Here, our more efficient calculation of opt-**sZEnc** results in performance increases of 2.01×–6.17× with an average of 4.00×. These are largely due to the improvements in opt-**ZEnc**, which range from 3.37×–6.50×, increasing with both the provided offset to the randomness reduction and d. As these results show, improvements are more significant when looking at the improved sub-components, indicating that the impact on the higher-level components would be larger if their implementation with regard to basic operations, loads, and stores would be optimized even more. We conclude that our optimizations provide large benefits for the improved gadgets themselves, specifically with increasing d, and thus already offer a modest increase for higher-level components.

The impact of e is as follows. Considering a fixed d, adding a share for fault resilience increases the cycle count for the Sbox between 2.13×–3.70×, with an average of 2.92× when FA and opt-**ZEnc** are used. Compared to the values for **ZEnc**, the numbers are similar, reaching from 1.91×–4.15× increase in cycle counts with an average of 3.03×.

Regarding randomness consumption, the most affected component is opt-**sZEnc**, which at higher orders reduces randomness consumption by 25–33% compared to **sZEnc**. Additionally, calls to opt-**ZEnc** with a reduced degree also contribute to randomness improvements. This also leads to a randomness reduction of 8–23% in the Sbox for $d > 1$, with an average of 17% both with and without the use of FA.

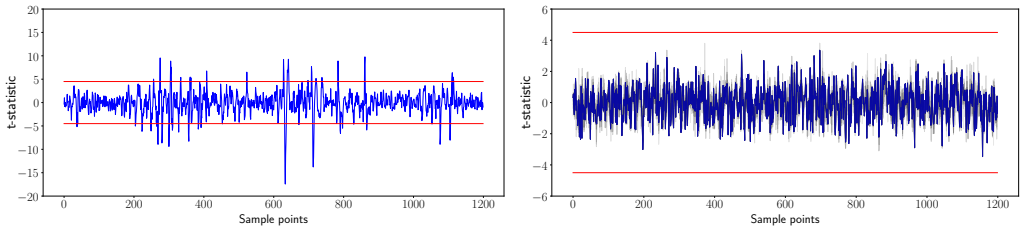
We remark that even more optimizations are possible. For example, one could adapt a form of byte-slicing and position the i-th shares of 4 independently encoded secrets in a single 4-byte register to enable a single instruction, multiple data (SIMD) execution of 4 Sboxes in parallel, with slight changes required for the operations involving table lookups.

There is a multitude of efficient software masked implementations [SS16, BFG+17, GR17, CB23]. For example, Cui and Balash [CB23] implement a first-order Boolean masked AES that requires 54 809 clock cycles through instruction set extension, accelerating field multiplications, and block-level parallelization that performs multiple operations of one AES encryption in parallel; Schwabe and Stoffelen [SS16] demonstrated a highly optimized first-order Boolean masked AES that requires 4 300 cycles, using techniques such as T_Tables, Vectorpermute, and Bitslicing. Although our approach is more resource intensive, we stress that these implementations only provide security against passive attackers. In contrast, our solution offers security against much stronger, combined attackers.

6.3 Leakage Assessment

We perform statistical leakage detection to assess the side-channel resilience of our first-order implementation of **LaOla** for the parameter set of $d = 1, e = 0, n = 2$.

The power consumption of the processor executing the implementation is captured using a ChipWhisperer CW308 UFO with an STM32F415RGT6 target and a Husky as an oscilloscope. The microcontroller is clocked at 7.37 MHz while the power consumption is captured using a 12-bit analog-to-digital converter (ADC) at 29.48 MHz Bandwidth, corresponding to 4 measurement samples per clock cycle. The built-in amplifier is set to 29 dB to make use of the full 12-bit range of the ADC. To analyze our power traces for leakage, we use a first-order t-test-based Test Vector Leakage Assessment (TVLA) [GJJ+11]. This requires capturing and comparing the statistical moments of two equally sized sets of measurements (traces for short), one invoked on a fixed secret value (i.i.d. encoded for each trace) while the other set is invoked on inputs chosen uniformly at random. The TVLA threshold for rejecting the null-hypothesis is set to ± 4.5, representing an alpha certainty of $\alpha = 0.0001$. If this threshold is exceeded, it is assumed possible to distinguish between the two sets of measurements, which indicates the presence of exploitable leakage. We refer the reader to [CDG+13, SM16] for more details on TVLA.



(a) 1 000 traces, randomness zeroed (b) 5 000 000 traces, presampled randomness

Figure 5: Results of TVLA assessment for the **LaOla** multiplication (a) 1st order without randomness, (b) 1st order with randomness. The x-axis represents sample point indices. Saturation increases with trace amount, final result in blue.

First, we validate the quality of our setup by performing a t-test on our implementation with randomness turned off, meaning all random values are set to zero. In this setting, a qualitative setup is expected to detect leakage already with a few traces. This is the case for our setup, as leakage is clearly visible when measuring only 1 000 traces, as shown in Figure 5a. Each trace consists of 1 200 sample points.

For the security analysis of the **LaOla** gadget, we capture 5 000 000 traces in total, using the same code and measurement settings but with randomness pre-sampled by the PRNG. As seen in Figure 5b, the implementation is not showing leakage when provided with strong randomness. Furthermore, the entire Sbox also passes TVLA at 1 000 000 traces (cf. Supp. F.1). Hence, we confirm that resilience against first-order side-channel analysis (SCA) attacks, as stated by the theoretical proofs, indeed holds.

Preventing Microarchitectual Leakage. Our initial C implementation of the algorithms showed microarchitectural leakage, introduced by compiler optimizations. We describe how these were mitigated by hardening using Assembly code at few, selective locations. The majority of observed leakages was due to 32-bit memory accesses for reading consecutive single-byte shares simultaneously in one cycle. To prevent this, we treated shares as 4-byte words instead, as suggested in [BGG+21]. This ensures that memory accesses do not combine shares. Another source of leakage arose from compiler introduced dual load/store instructions (`ldrd`/`strd`) on shares, causing observable leakage in field additions and multiplications. We mitigated them by implementing these field operations using inline Assembly. The last source of leakage was due to residue in registers causing transition leakages in the subsequent execution. To prevent this, only the field multiplication was augmented by `mov reg, #0` instructions to clear the used registers after computation. Interestingly, we did not need to add mitigation against leakage across instructions, e.g. consecutive memory accesses or operations, which is often needed for other masking schemes. The absence of weaknesses from these leakages allowed us to implement largely in C, instead of writing the whole code in Assembly, which is a significant effort.

6.4 Fault Robustness

We evaluate the ability of our optimized multiplication to withstand fault attacks. We conduct fault simulations and compare the output of two QEMU emulations of our implementation. Both receive the same inputs and pre-sampled randomness, but one is faulty. A fault is *effective* if the returned encoded values differ. A fault is *detected* iff one of the e higher-order coefficients f_{d+1} to f_{n-1} is non-zero. We adapt our implementation to cover two kinds of faults: a *set-zero fault* replaces the output of a field operation by 0, while an *additive fault* adds a uniformly i.i.d. value to the field operations' output. *Set-value faults*, i.e., setting the output to an adversary-controlled value, are covered in

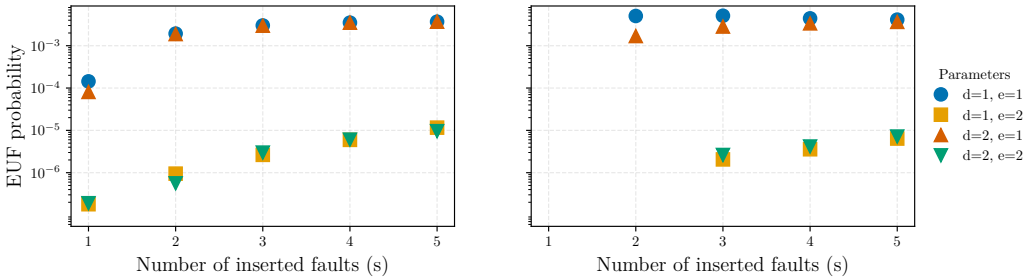


Figure 6: Estimated probability $\hat{p}$ of effective but undetected faults (EUFs) over 10^8 fault simulations with $s > 0$ faults for additive random faults (left) and set-zero faults (right). No EUFs occur for d, e with $e > 2$ in all experiments.

our evaluation by additive faults. There is a large body of existing work on tools for simulating fault injection, see [BLK23]. However, these tools cannot distinguish between operations of the implemented algorithm and control-flow, hence causing unrelated side-effects, e.g., hard-faults. As described in Section 2.1, our countermeasures protect against faults in circuit operations. To precisely evaluate the security claims w.r.t. the stated class of attacks, we simulate the injection of faults on input shares and elementary field operations. We do not fault outputs directly but rather exploit that in the analyzed gadgets every output share is the result of an operation within the gadget, for which we do simulate faults. Hence, any set or add fault on an output is covered by a corresponding fault on the operation computing this output. Control-flow faults are not considered in our countermeasures, threat model, and simulations, but can be mitigated easily: as all loops in the executable are already unrolled and no data-dependent branches occur, the attack surface for control-flow faults is minimal. Protection against control flow faults on remaining branches could be achieved by inlining the few remaining functions.

Let $\vec{x}$ denote the inputs for a simulation run, i.e., the encoded secrets, the pre-sampled randomness, the field operations to fault, denoted *fault locations*, and the injected values in case of additive faults. The inputs $\vec{x}_i$ for the i^{th} run are sampled independently and uniformly. Each simulation $s(\vec{x}_i)$ returns 1 for effective but undetected faults (EUFs) and 0 otherwise. To estimate the probability for EUFs we choose the unbiased estimator $\hat{p} = \frac{1}{n}\Sigma_{i=1}^{n} s(\vec{x}_i)$ for n simulation runs. By sampling inputs independently and uniformly, we ensure that its statistical bias is zero. We perform $n = 10^8$ simulations with i.i.d. inputs for each protection order d, e and $s > 0$ injected faults. By the central limit theorem, the unbiased estimator converges to the true probability for EUFs. However, a very high number of simulations n is required due to the rare event of EUFs for $e > 1$, as seen in Supp. F.2. Remark that exhaustive enumeration is infeasible due to the large sample space, which is why we perform randomized experiments to estimate the probability of EUFs. Therefore, our estimated probability $\hat{p}$ corresponds to an empirical lower bound on the success probability for a fault attack. The estimated probabilities are in line with the theoretically proven upper bounds in the earlier parts of the paper as they are consistently lower. This approach is sufficient for initial insights on the practical fault resilience but also highlights that more advanced methods, e.g., formal verification, is unavoidable for protection with $e > 1$, due to the rare event of EUFs.

Simulation Results. The resulting probabilities for EUFs are visualized in Figure 6 and absolute values are provided in Supp. F.2. We emphasize that polynomial masking ensures security against faults with non-zero success probability for $s \leq e$. For $s = 1, e \leq 2$, at least 71% of the simulations result in effective faults, respectively 91% for $s > 1$. The fault resilience is not significantly impacted by the side-channel protection order d, which

confirms the ability to adapt the countermeasures' security parameters independently. For $s \leq e$, *all* of the effective set-zero faults are detected. The resilience against additive faults increases significantly by approx. 3 orders of magnitude across any number of injected faults, e.g., from 0.014% for $e = s = 1$ to 0.000095% for $e = s = 2$. Interestingly, the protection against $s > e$ faults, i.e., beyond the protection order, is quite strong as consistently 99.999% of the effective faults, even for $e = 2, s = 5$, are detected. Further, we perform the same simulations for protections with $e > 2$ and while 66.4% at $s = 1$ to 99.56% at $s = 5$ of the simulations result in effective faults, none of them remain undetected. Additionally, we report that even for 10^{10} set-zero fault simulations at $d = 1, e = 2, s = 5$ not a single EUF is observed, indicating that statistical analyses, even with k-means clustering, are not worthwhile. More sophisticated approaches, e.g., symbolic methods and systematic fault injection based on the circuit-structure, are needed.

The results, especially the strong resilience even for $s > e$, suggest that our construction can protect against strong attacks like EMFI, affecting many operations at once. Further, the low probabilities for EUFs strongly indicate that simple randomized fault attacks are very unlikely to succeed for implementations protected with $e = 2$ and not worth-while for $e > 2$ even when $s = 5$ faults are injected simultaneously.

Supplementary Material

A Conclusion

Combined attackers that perform both passive side-channel attacks as well as active fault attacks are among the strongest and most dangerous physical attackers against implementations. While polynomial masking has been proposed as a countermeasure against such attackers, it has not been practically evaluated yet. In this paper, we first improve all known gadgets using polynomial sharing both with regard to randomness complexity and with regard to computational complexity. We rigorously prove that our optimizations do not weaken the security of the presented schemes and estimate the resource improvements with regard to the different parameters t, e, and k. Secondly, we present the first implementation of a polynomial masking scheme secure against such combined attacks along with an extensive experimental evaluation for a wide range of parameters and configurations. To determine the practical security of the implementation, we perform a statistical leakage detection on an Arm Cortex-M processor and also stochastically evaluate the security against faults using more than 10 billion runs. We thus have taken the first steps to make polynomial masking a practically usable countermeasure against these attacks. To encourage further research in practical combined resilience, our implementation is publicly available.

B Construction of Matrix A

Matrix M. Recall the definition of matrix M in Paragraph 2.2. Note that while the definition is similar to the one in [FOWZ25], the authors define the Vandermonde matrix as a transpose of the Vandermonde matrix we defined in Section 2: For their matrix M', it holds that $M' = (U^T)^{-1} \cdot V^T = (VU^{-1})^T = M^T$.

Constructing Matrix A. We construct the elements of matrix A by calculating $a_{i,j} \coloneqq \sum_{l=0}^{d} M_{i,l}^{-1} \cdot M_{l,j}$, where $M = U^{-1} \cdot V$. Recall that we want to have the following correlations:

$$\begin{aligned} \mathsf{mask}\,(\boldsymbol{v}) &= M \cdot \boldsymbol{v} = \boldsymbol{F}^T \\ \mathsf{unmask}\,(\boldsymbol{F}) &= M^{-1} \cdot \boldsymbol{F}, \end{aligned}$$

where $\boldsymbol{F} = (F_0, \ldots, F_{n-1})^T$ are the vector of n output shares.

For a polynomial $p(X)$ with coefficients $p_0, \ldots, p_d$ evaluated at support points $x_0, \ldots, x_n$ we can write

$$\begin{pmatrix} p(x_0) \\ \vdots \\ p(x_{n-1}) \end{pmatrix} = \mathsf{Van}_{n\times(d+1)}(x_0, \ldots, x_{n-1}) \cdot \begin{pmatrix} p_0 \\ \vdots \\ p_d \end{pmatrix}.$$

This translates to our case as follows:

$$\begin{pmatrix} p(u_0) \\ \vdots \\ p(u_k) \\ p(x_0) \\ \vdots \\ p(x_{n-1}) \end{pmatrix} = \mathsf{Van}_{(n+k)\times(d+1)}(u_0, \ldots, u_k, x_0, \ldots, x_{n-1}) \cdot \begin{pmatrix} p_0 \\ \vdots \\ p_d \end{pmatrix}.$$

Since square Vandermonde matrices are invertible, we can write

$$\begin{pmatrix} p_0 \\ \vdots \\ p_d \end{pmatrix} = \mathsf{Van}^{-1}\left(u_0, \ldots, u_{k-1}, x_0, \ldots, x_{d-k}\right) \cdot \begin{pmatrix} p(u_0) \\ \vdots \\ p(u_k) \\ p(x_0) \\ \vdots \\ p(x_{d-k}) \end{pmatrix}.$$

Hence, in our case, as we aim to construct the shares $F_0, \ldots, F_{n-1}$ as evaluation of our polynomial $p(X)$ at the points $x_0, \ldots, x_{n-1}$, we can choose $\tilde{A}$ as

$$\tilde{A} = \begin{pmatrix} \mathbf{I}_{(d+1-k)\times(d+1-k)} \\ A \end{pmatrix},$$

where A' is defined by

$$A' \coloneqq \mathsf{Van}_{(n+k-d-1)\times(d+1)}\left(x_{d+1-k}, \ldots, x_{n-1}\right) \cdot \mathsf{Van}^{-1}_{(d+1)\times(d+1)}\left(u_0, \ldots, u_{k-1}, x_0, \ldots, x_{d-k}\right), \tag{8}$$

then we have

$$\begin{aligned}
\begin{pmatrix} F_{d+1-k} \\ \vdots \\ F_{n-1} \end{pmatrix} =& \mathsf{Van}_{(n+k-d-1)\times(d+1)}\left(x_{d+1-k}, \ldots x_{n-1}\right) \cdot \begin{pmatrix} p_0 \\ \vdots \\ p_d \end{pmatrix} \\
=& \mathsf{Van}_{(n+k-d-1)\times(d+1)}\left(x_{d+1-k}, \ldots x_{n-1}\right) \\
& \cdot \mathsf{Van}^{-1}_{(d+1)\times(d+1)}\left(u_0, \ldots, u_{k-1}, x_0, \ldots, x_{d-k}\right) \\
& \cdot \begin{pmatrix} p(u_0) \\ \vdots \\ p(u_k) \\ p(x_0) \\ \vdots \\ p(x_{d-k}) \end{pmatrix} \\
=& \mathsf{Van}_{(n+k-d-1)\times(d+1)}\left(x_{d+1-k}, \ldots x_{n-1}\right) \\
& \cdot \mathsf{Van}^{-1}_{(d+1)\times(d+1)}\left(u_0, \ldots, u_{k-1}, x_0, \ldots, x_{d-k}\right) \\
& \cdot \left(v_0, \ldots, v_{k-1}, r_0, \ldots, r_{d-k}\right)^T \\
=& A' \cdot \left(v_0, \ldots, v_{k-1}, r_0, \ldots, r_{d-k}\right)^T.
\end{aligned}$$

Since $v_0 = 0, \ldots, v_{k-1} = 0$, we can construct matrix A by removing the first k columns of matrix A'. Finally, we can see that

$$\begin{aligned}
\left(F_0, \ldots, F_{n-1}\right)^T &= \begin{pmatrix} I_{(d+1-k)\times(d+1-k)} \\ A \end{pmatrix} \cdot \left(r_0, \ldots, r_{d-k}\right)^T && (9) \\
&\coloneqq \tilde{A} \cdot \boldsymbol{r}. && (10)
\end{aligned}$$

B.1 Matrix for DegreeReduction

In this section, we build the matrix $\left(\lambda_{i,j}^{\mathsf{DegRed}}\right)_{i\in[n],j\in[d+1]}$ for degree reduction. We assume to have two sets of pairwise distinct support points, $(u_0, \ldots, u_{2d})$ and $(x_0, \ldots, x_{n-1})$. Please

note that we assume to have these additional d support points for the secrets, $u_{d+1}, \ldots, u_{2d}$, as the underlying polynomial has degree $2d$ due to the share-wise multiplication that is performed before the degree reduction in the **BGW**-based multiplication. As the masking function $\mathsf{mask}(\cdot)$ presented in Section 2.2 is assumed to have $\ell+1$ inputs when creating shares with respect to a polynomial of degree ℓ, this assumption simplifies the proof. If one only wants to use k support points for the secrets and mask according to the matrix $A'_{(n+k-d-1)\times(d+1)}$ (cf. Equation 1), the following calculation has to be adapted according to Section B.3.

Our matrix for degree reduction is constructed to perform *unmasking* followed by *masking*, which can be done in one step. Thus, we can construct the matrix similar to the construction in [FOWZ25].

Recall that

$$\begin{aligned}\mathsf{unmask}(F_0, \ldots, F_{n-1}) =& M^{-1} \cdot (F_0, \ldots, F_n)^T && (11)\\ =& \mathsf{Van}_{(2d+1)\times(2d+1)}(u_0, \ldots, u_{2d}) \cdot \mathsf{Van}^{-1}_{(2d+1)\times n}(x_0, \ldots, x_{n-1}) && (12)\\ & \cdot (F_0, \ldots, F_n)^T,\end{aligned}$$

where $\mathsf{Van}^{-1}_{(2d+1)\times n}(x_0, \ldots, x_{n-1})$ is defined as the top $(2d+1)\times n$ sub-matrix of $\mathsf{Van}^{-1}_{n\times n}(x_0, \ldots, x_{n-1})$. More construction details are presented in [FOWZ25, Supp. A.2]. Now, we aim to reconstruct a sharing with respect to a degree-d polynomial. Thus, we consider

$$\mathsf{mask}(\boldsymbol{v}) = \mathsf{Van}_{n\times(d+1)}(x_0, \ldots, x_n) \cdot \mathsf{Van}^{-1}_{(d+1)\times(d+1)}(u_0, \ldots, u_d) \cdot (v_0, \ldots, v_d)^T.$$

We pad the matrices with zeros to be able to define:

$$\begin{aligned}\left(\lambda^{\mathsf{DegRed}}_{i,j}\right)_{i\in[n],j\in[n]} :=& \mathsf{Van}_{n\times n}(x_0, \ldots, x_{n-1}) \cdot \mathsf{Van}^{-1}_{n\times n}(u_0, \ldots, u_d)\\ & \cdot \mathsf{Van}_{(2d+1)\times(2d+1)}(u_0, \ldots, u_{2d}) \cdot \mathsf{Van}^{-1}_{(2d+1)\times n}(x_0, \ldots, x_{n-1})\end{aligned}$$

B.2 Matrix for Degree Reduction and Coefficients Re-Randomization

[ABE+25] defines the matrix $\hat{A}_{i,j}$ (cf. Alg. 10), aiming to decrease the degree of F_i and re-randomize *all* coefficients of the polynomial, whenever one fault is present. We denote the degree *before* the degree reduction by d_{in} and the degree we aim to have after the degree reduction by d_{out}. Matrix $\left(\lambda^{(d_{\mathsf{in}}\to d_{\mathsf{out}})}_{i,j}\right)_{i,j\in[n]^2}$ will reduce the degree of polynomial sharing from degree d_{in} to d_{out} using similar idea as matrix $\lambda^{\mathsf{DegRed}}_{i,j}$.

B.3 Masking and Unmasking using Matrix $A'_{(n+k-d-1)\times(d+1)}$

Faust et al. [FOWZ25] use $n+d+1$ support points to *mask* and *unmask* their secrets. First, they fix a polynomial at support points $u_0, \ldots, u_d$ and afterwards evaluate it at the n support points $x_0, \ldots, x_{n-1}$ to obtain their shares. Namely, one can calculate the shares $p(x_0), \ldots, p(x_{n-1})$ via the following equation

$$\begin{pmatrix} p(x_0) \\ \vdots \\ p(x_{n-1}) \end{pmatrix} := \mathsf{Van}(x_0, \ldots, x_{n-1}) \cdot \mathsf{Van}(u_0, \ldots, u_d)^{-1} \cdot (v_0, \ldots, v_d). \quad (13)$$

We will show how to use $k+n$ support points to mask and unmask. As described in Section 3.1, we only need $k+n$ support points to construct a matrix $A'_{(n+k-d-1)\times(d+1)}$, which

Algorithm 10 $\mathsf{Comp}\hat{A}^{(d_{\mathsf{in}},d_{\mathsf{out}},n)}$

Input : Parameters $d_{\mathsf{in}}, d_{\mathsf{out}}, n \in \mathbb{N}$.
Output : Matrix $\hat{A}$.

for $k \in [d_{\mathsf{in}}+1, n]$ **do**
  initialize $\left(\hat{R}_i^{(k)}\right)_{i\in[n]}$ as $\mathbf{0}_n$
  for $j \in [d_{\mathsf{out}}+1, n] \cup \{0\}$ **do**
    $r_j^{(k)} \leftarrow_{\$} \mathbb{F}$
    for $i \in [n]$ **do**
      $\hat{R}_i^{(k)} \leftarrow \hat{R}_i^{(k)} + r_j^{(k)} \cdot x_i^j$
for $i \in [n]$ **do**
  for $j \in [n]$ **do**
    $\bar{R}_{i,j} \leftarrow \sum_{k=d_{\mathsf{in}}+1}^{n-1} \hat{R}_i^{(k)} \cdot V_{k,j}^{-1}$
$\left(\hat{A}_{i,j}\right)_{i,j\in[n]} \leftarrow \left(\lambda_{i,j}^{(d_{\mathsf{in}}\to d_{\mathsf{out}})}\right)_{i,j\in[n]} + \left(\bar{R}_{i,j}\right)_{i,j\in[n]}$
return $\left(\hat{A}_{i,j}\right)_{i,j\in[n]}$

can be further optimized to obtain zero-encodings. However, this matrix $A'_{(n+k-d-1)\times(d+1)}$ can be used to mask and unmask any secret.

Recall that instead of choosing the $d+1-k$ *values* $v_k, \ldots, v_d$, uniformly at random, this time, we directly set the $d+1-k$ output *shares*, $p(x_0), \ldots, p(x_{d-k})$, to $d+1-k$ uniformly random values. This approach not only avoids the computation of the output shares, $p(x_0), \ldots, p(x_{d-k})$, but also makes the evaluation of the support points $u_k, \ldots, u_d$ completely unnecessary. Together with $v_0, \ldots, v_{k-1}$, this method fixes $d+1$ pre-determined values, thereby fixing the underlying degree-d polynomial $p(X)$.

In order to obtain a total of n output shares, we now need to compute the remaining $n-(d+1-k)$ shares, i.e., $p(x_{d+1-k}), \ldots, p(x_{n-1})$. This can be achieved using the Vandermonde matrix and its inverse for evaluation and interpolation, as described in Section 2.2. We defined

$$\begin{aligned} A'_{(n+k-d-1)\times(d+1)} \coloneqq & \mathsf{Van}_{(n+k-d-1)\times(d+1)}\left(x_{d+1-k}, \ldots, x_{n-1}\right) \\ & \cdot \mathsf{Van}^{-1}_{(d+1)\times(d+1)}\left(u_0, \ldots, u_{k-1}, x_0, \ldots, x_{d-k}\right) \end{aligned}$$

and obtain

$$\begin{pmatrix} p(x_{d+1-k}) \\ \vdots \\ p(x_{n-1}) \end{pmatrix} = A'_{(n+k-d-1)\times(d+1)} \cdot \begin{pmatrix} p(u_0) \\ \vdots \\ p(u_{k-1}) \\ p(x_0) \\ \vdots \\ p(x_{d-k}) \end{pmatrix}. \tag{14}$$

Lemma 16. *Let $u_0, \ldots, u_d$ and $x_0, \ldots, x_{n-1}$ denote pairwise distinct support points. Assume that we fix $p(u_0) \ldots p(u_{k-1})$ to be our secrets $v_0, \ldots, v_{k-1}$ and choose the elements $p(x_0) \ldots, p(x_{d-k})$ uniformly random from $\mathbb{F}$. If we compute $p(x_{d+1-k}), \ldots, p(x_{n-1})$ according to Equation 14, then $p(x_0), \ldots p(x_{n-1})$ yields a valid sharing of the secret $(v_0, \ldots, v_d)$, where $v_i, i \in [k, d+1]$ are uniformly distributed.*

Proof. Note that this lemma can be proven similarly to Lemma 1. We choose some of the output shares, $p(x_0) \ldots, p(x_{d-k})$ uniformly at random. These shares, together

with the secrets $p(u_0) \dots p(u_{k-1})$, describe a uniformly random polynomial $p(X)$ with $p(u_0) = v_0 \dots p(u_{k-1}) = v_{k-1}$ as secrets. The matrix $A'_{(n+k-d-1)\times(d+1)}$ simply computes the function values of this polynomial at points $x_{d+1-k}, \dots, x_{n-1}$. We set the shares F_i to be the evaluations of the polynomial $p(X)$ at points x_i for all $i \in [n]$, i.e., $F_i = p(x_i)$. These shares are $d+1-k$-wise independent (cf. [FY92]) and embed the secrets $v_0, \dots, v_{k-1}$ at support points $u_0, \dots, u_{k-1}$. Thus, $(F_i)_{i\in[n]}$ is a valid encoding of the secrets $v_0, \dots, v_{k-1}$. □

C Matrix Transformation

C.1 Example

In this section, we explain $\mathsf{Algo}_{\mathsf{MatrixTransformation}}$ using a concrete example with parameters $n = 10$, $d = 7$, and $k = 3$. We consider the finite field $\mathsf{GF}(2^8)$ with modulus $m(x) = x^8 + x^4 + x^3 + x + 1$, which is often used for AES. Within this field, addition (equivalent to bit-wise XOR operation) and multiplication are performed on polynomials. For readability, we use an integer representation of the field elements. In our example, we use the support points $(u_0, \dots, u_8) = (212, 247, 7, 151, 56, 41, 187, 244)$ and $(x_0, \dots, x_9) = (213, 107, 208, 176, 29, 141, 17, 28, 171, 6)$. For these support points, we compute matrix $\tilde{A}$ with the help of Equation 1:

$$\tilde{A} = \left(\begin{array}{ccccc} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ \hline 75 & 245 & 40 & 50 & 226 \\ 176 & 184 & 223 & 44 & 216 \\ 56 & 64 & 159 & 174 & 252 \\ 139 & 117 & 9 & 171 & 41 \\ 180 & 187 & 108 & 246 & 93 \end{array}\right). \tag{15}$$

Note that the top part (above the dividing line) of matrix $\tilde{A}$ consists of $(d+1-k)^2 - (d+1-k) = 20$ zeros. The bottom part contains $(n-(d+1-k)) \cdot (d+1-k) = 25$ non-zero elements. Our goal is to construct a matrix $\tilde{A}_{\mathsf{opt}}$ that redistributes the zeros in the top part evenly across all n rows. Hence, $\tilde{A}_{\mathsf{opt}}$ should have (approximately) $\mathsf{Row}_{\mathsf{Zeros}} = \lfloor \frac{(d+1-k)^2-(d+1-k)}{n} \rfloor = 2$ zeros in each of the $n = 10$ row.

As we will see below, the resulting matrix is

$$\tilde{A}_{\mathsf{opt}} = \left(\begin{array}{ccccc} 17 & 231 & 73 & 0 & 0 \\ 0 & 208 & 180 & 250 & 0 \\ 0 & 0 & 13 & 240 & 255 \\ 73 & 0 & 0 & 132 & 137 \\ 146 & 128 & 0 & 0 & 40 \\ \hline 0 & 27 & 0 & 11 & 180 \\ 0 & 128 & 94 & 0 & 163 \\ 64 & 0 & 194 & 0 & 180 \\ 183 & 0 & 21 & 253 & 0 \\ 157 & 118 & 0 & 55 & 0 \end{array}\right). \tag{16}$$

Before we can start, we first introduce some variables: $\mathsf{Col}_{\mathsf{TopZeros}}$ describes the number of zeros that $\tilde{A}_{\mathsf{opt}}$ should have in the top half of each column. $\mathsf{Col}_{\mathsf{BottomZeros}}$ is the number of zeros that the bottom half of each column should have. As our algorithm shifts zeros from the top to the bottom part of the matrix, this means that $\mathsf{Col}_{\mathsf{BottomZeros}} =$

$(d-k) - \mathsf{Col}_{\mathsf{TopZeros}}$. The algorithm $\mathsf{Algo}_{\mathsf{MatrixTransformation}}$ (cf. Alg. 14) consists of three sub-algorithms: $\mathsf{Algo}_{\mathsf{BottomZeros}}$, $\mathsf{Algo}_{\mathsf{HelperColumns}}$, and $\mathsf{Algo}_{\mathsf{ZeroShift}}$. We proceed by explaining these three sub-algorithms step by step.

Positions of the Zero Elements. $\mathsf{Algo}_{\mathsf{BottomZeros}}$ (cf. Alg. 11) determines the positions in the bottom of the matrix where the zero elements should be created, processing one column at a time. Here, $\mathsf{Index}_{\mathsf{TargetCol}}$ denotes the *index of the target column* of $\tilde{A}_{\mathsf{opt}}$ to be generated. The variable $\mathsf{Col}_{\mathsf{BottomZeros}}$ specifies the *number of zeros to be shifted* to the bottom part of the target column.

Consider the 0-th column $\tilde{A}[:,0]$ of $\tilde{A}$ as the target column. Following the idea of the overall algorithm, the first $\mathsf{Col}_{\mathsf{TopZeros}}$ zero elements (highlighted in blue) below the diagonal will not be changed. The subsequent $\mathsf{Col}_{\mathsf{BottomZeros}}$ zero elements (highlighted in red) will be shifted to the bottom part of the target column $\tilde{A}_{\mathsf{opt}}[:,0]$, specifically to the positions output by $\mathsf{Algo}_{\mathsf{BottomZeros}}$.

$$\tilde{A}[:,0] = \begin{pmatrix} \mathbf{1} \\ 0 \\ 0 \\ 0 \\ 0 \\ \hline 75 \\ 176 \\ 56 \\ 139 \\ 180 \end{pmatrix} \Rightarrow \tilde{A}_{\mathsf{opt}}[:,0] = \begin{pmatrix} \mathbf{17} \\ 0 \\ 0 \\ 73 \\ 146 \\ \hline 0 \\ 0 \\ 64 \\ 183 \\ 157 \end{pmatrix}.$$

Hence, we have now identified which zeros should remain in the top part and can shift the other zeros to the positions in the bottom part. By construction, all upper parts of the columns have $\mathsf{Col}_{\mathsf{TopZeros}}$ zero elements. Since they are all positioned relative to the diagonal, this also means that each of the upper rows has $\mathsf{Col}_{\mathsf{TopZeros}} = \mathsf{Row}_{\mathsf{Zeros}} = 2$ many zeros. In the bottom part of $\tilde{A}_{\mathsf{opt}}$, the shifted zeros of each column are distributed *uniformly* across the rows. To be precise, $\mathsf{Col}_{\mathsf{BottomZeros}}$ many zeros will be created per column and distributed using this cascading shape. Thus, $\tilde{A}_{\mathsf{opt}}$ will contain $\mathsf{Col}_{\mathsf{BottomZeros}} \cdot (d+1-k) = (d-k-\mathsf{Col}_{\mathsf{TopZeros}}) \cdot (d+1-k)$ many zeros, which is easily shown to be greater or equal to the number of rows in the bottom part $n-d-1+k$ times $\mathsf{Row}_{\mathsf{Zeros}}$ (in the concrete algorithm, we will stop once enough zeros are created and the bottom of $\tilde{A}_{\mathsf{opt}}$ will hence always only contain $(n-d-1+k) \cdot \mathsf{Row}_{\mathsf{Zeros}}$ zeros). It follows that all rows have $\mathsf{Row}_{\mathsf{Zeros}}$ zeros.

$$\tilde{A} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ \hline 75 & 245 & 40 & 50 & 226 \\ 176 & 184 & 223 & 44 & 216 \\ 56 & 64 & 159 & 174 & 252 \\ 139 & 117 & 9 & 171 & 41 \\ 180 & 187 & 108 & 246 & 93 \end{pmatrix} \Rightarrow \tilde{A}_{\mathsf{opt}} = \begin{pmatrix} 17 & 231 & 73 & 0 & 0 \\ 0 & 208 & 180 & 250 & 0 \\ 0 & 0 & 13 & 240 & 255 \\ 73 & 0 & 0 & 132 & 137 \\ 146 & 128 & 0 & 0 & 40 \\ \hline 0 & 27 & 0 & 11 & 180 \\ 0 & 128 & 94 & 0 & 163 \\ 64 & 0 & 194 & 0 & 180 \\ 183 & 0 & 21 & 253 & 0 \\ 157 & 118 & 0 & 55 & 0 \end{pmatrix}.$$

Index of the Helper Columns. We have determined the target positions for the zeros to be shifted, but the actual shifting operation still needs to be performed. Depending on the parameters, it is possible that multiple zeros need to be shifted into the same row. In such cases, additional helper columns $\mathsf{HelperCols}$ are required to facilitate the process. $\mathsf{Algo}_{\mathsf{HelperColumns}}$ (cf. Alg. 12) determines the indices of the $\mathsf{HelperCols}$ needed to shift the

zeros in the target column $\mathsf{TargetCol}$. For example, consider the 0-th column $\tilde{A}[:,0]$ as a starting point. To shift the zeros (highlighted in red) at position 3 and 4 to the bottom part of the column, we need the 3-th column $\tilde{A}[:,3]$ and 4-th column $\tilde{A}[:,4]$ as $\mathsf{HelperCols}$, as, due to the top part of $\tilde{A}$ being the identity matrix, they each contain the one element at position 3 and 4, respectively:

$$\tilde{A}[:,0] = \begin{pmatrix} \mathbf{1} \\ 0 \\ 0 \\ 0 \\ 0 \\ \hline 75 \\ 176 \\ 56 \\ 139 \\ 180 \end{pmatrix}, \tilde{A}[:,3] = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \mathbf{1} \\ 0 \\ \hline 50 \\ 44 \\ 174 \\ 171 \\ 246 \end{pmatrix}, \tilde{A}[:,4] = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \mathbf{1} \\ \hline 226 \\ 216 \\ 252 \\ 41 \\ 93 \end{pmatrix}.$$

By applying a linear combination of $\tilde{A}[:,0]$, $\tilde{A}[:,3]$, and $\tilde{A}[:,4]$, the zero elements in the top part of the target column can now be shifted to its bottom part.

Zero Element Shifting. To obtain $\tilde{A}_{\mathsf{opt}}$, we need to shift two zero elements per column using $\mathsf{Algo}_{\mathsf{ZeroShift}}$. Take the 0-th ($i = 0$) column $\tilde{A}[:,0]$ as our target column. Since we want to shift the last two zeros in the top part of $\tilde{A}[:,0]$ to the bottom, we choose the 3-rd and 4-th ($j = 3, 4$) columns of $\tilde{A}$ as the helper columns. Starting with column $\tilde{A}[:,3]$, we can create a new intermediate column $\tilde{A}_{\mathsf{tmp1}}$ which contains one additional non-zero element in the top part and one additional zero element in the bottom part, compared to the i-th ($i = 0$) column of $\tilde{A}$, by performing the following linear combination,

$$50 \cdot \tilde{A}[:,0] - 75 \cdot \tilde{A}[:,3] = 50 \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ \hline 75 \\ 176 \\ 56 \\ 139 \\ 180 \end{pmatrix} - 75 \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ \hline 50 \\ 44 \\ 174 \\ 171 \\ 246 \end{pmatrix} = \begin{pmatrix} 50 \\ 0 \\ 0 \\ -75 \cdot 1 = 75 \\ 0 \\ \hline 50 \cdot 75 - 75 \cdot 50 = 0 \\ 50 \cdot 176 - 75 \cdot 44 = 133 \\ 50 \cdot 56 - 75 \cdot 174 = 74 \\ 50 \cdot 139 - 75 \cdot 171 = 31 \\ 50 \cdot 180 - 75 \cdot 246 = 142 \end{pmatrix} = \tilde{A}_{\mathsf{tmp1}}.$$

This guarantees that we shift one zero element to the bottom part of $\tilde{A}_{\mathsf{tmp1}}$. It is left to shift the second zero. This process involves two steps. First, we create an intermediate column $\tilde{A}_{\mathsf{tmp2}}$ which has one zero element in the bottom part by combining the 3-rd column $\tilde{A}[:,3]$ and the 4-th column $\tilde{A}[:,4]$ of $\tilde{A}$ as follows:

$$226 \cdot \tilde{A}[:,3] - 50 \cdot \tilde{A}[:,4] = 226 \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ \hline 50 \\ 44 \\ 174 \\ 171 \\ 246 \end{pmatrix} - 50 \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ \hline 226 \\ 216 \\ 252 \\ 41 \\ 93 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 226 \\ 50 \\ \hline 0 \\ 113 \\ 27 \\ 247 \\ 22 \end{pmatrix} = \tilde{A}_{\mathsf{tmp2}}.$$

Then, by combining columns $\tilde{A}_{\mathsf{tmp1}}$ and $\tilde{A}_{\mathsf{tmp2}}$, we can shift the second zero element by

computing as follows and obtain $\tilde{A}_{\mathsf{opt}}[:,0]$,

$$\tilde{A}_{\mathsf{opt}}[:,0] = 113 \cdot \tilde{A}_{\mathsf{tmp1}} - 133 \cdot \tilde{A}_{\mathsf{tmp2}} = 113 \cdot \left(\begin{array}{c} 50 \\ 0 \\ 0 \\ 75 \\ 0 \\ \hline 0 \\ 133 \\ 74 \\ 31 \\ 142 \end{array}\right) - 133 \cdot \left(\begin{array}{c} 0 \\ 0 \\ 0 \\ 226 \\ 50 \\ \hline 0 \\ 113 \\ 27 \\ 247 \\ 22 \end{array}\right) = \left(\begin{array}{c} 17 \\ 0 \\ 0 \\ 73 \\ 146 \\ \hline 0 \\ 0 \\ 64 \\ 183 \\ 157 \end{array}\right).$$

By applying the above *recursive* linear combination approach to all the columns of matrix $\tilde{A}$, we construct all the columns of the desired matrix $\tilde{A}_{\mathsf{opt}}$, ensuring that the zeros are *equally* distributed across each row.

C.2 General Algorithm for Matrix Transformation

Generally, $\mathsf{Algo}_{\mathsf{MatrixTransformation}}$ (cf. Alg. 14) computes each of the $d+1-k$ target columns (indexed by $\mathsf{Index}_{\mathsf{TargetCol}}$) of $\tilde{A}_{\mathsf{opt}}$ separately using the columns from $\tilde{A}$. The parameter $\mathsf{Col}_{\mathsf{TopZeros}}$ denotes the number of zeros in the top part of each column of $\tilde{A}_{\mathsf{opt}}$, and $\mathsf{Col}_{\mathsf{BottomZeros}}$ is the number of zeros in the bottom part of each column of $\tilde{A}_{\mathsf{opt}}$ after shifting the zeros. Due to Lemma 20, we know that the bottom part of $\tilde{A}$ does not consist of any zeros. Hence, there are in total $(d+1-k)^2 - (d+1-k)$ zeros in $\tilde{A}$ and, after the zero shifting, we distribute at least $\mathsf{Row}_{\mathsf{Zeros}} = \lfloor \frac{(d+1-k)^2-(d+1-k)}{n} \rfloor$ zeros to each of the n rows. In the top part of the matrix, the algorithm keeps the first $\mathsf{Row}_{\mathsf{Zeros}}$ zeros below the diagonal, this implies that $\mathsf{Col}_{\mathsf{TopZeros}} = \mathsf{Row}_{\mathsf{Zeros}}$. As all other zeros of the upper part are shifted to the part below, we have $\mathsf{Col}_{\mathsf{BottomZeros}} = d - k - \mathsf{Col}_{\mathsf{TopZeros}}$ since there are $d-k$ zeros in each column of $\tilde{A}$.

$\mathsf{Algo}_{\mathsf{MatrixTransformation}}$ (cf. Alg. 14) is structured in a similar way to the example above. On line 1, we first initialize $\tilde{A}_{\mathsf{opt}}$ with $\tilde{A}$. During each iteration of the for-loop, we first create a matrix $\tilde{A}_{\mathsf{tmp}}$ to store the intermediate value, which will be used in the computation of $\mathsf{Algo}_{\mathsf{ZeroShift}}$ (cf. Alg. 13). Then, we use $\mathsf{Algo}_{\mathsf{BottomZeros}}$ (cf. Alg. 11) to compute the location of the zeros to be shifted to the bottom and $\mathsf{Algo}_{\mathsf{HelperColumns}}$ (cf. Alg. 12) to compute the index of the $\mathsf{HelperCols}$ from $\tilde{A}$ that will be involved during the computation of the $\mathsf{TargetCol}$ of $\tilde{A}_{\mathsf{opt}}$. On line 7, we shift the zeros on column $\tilde{A}[:, \mathsf{Index}_{\mathsf{TargetCol}}]$ using the recursive approach discussed above and get the new transformed column $\mathsf{TargetCol}$. After $d+1-k$ iterations, we computed all the columns of $\tilde{A}_{\mathsf{opt}}$.

Let the matrix $\tilde{A}$ and the parameters $\mathsf{Row}_{\mathsf{Zeros}}$, $\mathsf{Col}_{\mathsf{BottomZeros}}$, and $\mathsf{Col}_{\mathsf{TopZeros}}$ be as defined above, more explicitly,

$$\tilde{A} = \begin{pmatrix} I_{(d+1-k)\times(d+1-k)} \\ A_{(n+k-d-1)\times(d+1-k)} \end{pmatrix}$$

$$= \left(\begin{array}{cccccccc} \mathbf{1_0} & 0 & \dots & 0 & \dots & 0 & 0 \\ 0 & \mathbf{1_1} & \dots & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \dots & 0 & 0 \\ 0 & 0 & \dots & \mathbf{1_i} & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & \dots & \mathbf{1_{d-k-1}} & 0 \\ 0 & 0 & \dots & 0 & \dots & 0 & \mathbf{1_{d-k}} \\ \hline a_{0,0} & a_{0,1} & \dots & a_{0,i} & \dots & a_{0,d-k-1} & a_{0,d-k} \\ a_{1,0} & a_{1,1} & \dots & a_{1,i} & \dots & a_{1,d-k-1} & a_{1,d-k} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ a_{e+k-1,0} & a_{e+k-1,1} & \dots & a_{e+k-1,i} & \dots & a_{e+k-1,d-k-1} & a_{e+k-1,d-k} \end{array}\right).$$

To generate the i-th column of $\tilde{A}_{\mathsf{opt}}$, we keep the first $\mathsf{Col}_{\mathsf{TopZeros}}$ elements of the i-th column of $\tilde{A}$ below the diagonal to value zero (marked in blue) and shift the next $\mathsf{Col}_{\mathsf{BottomZeros}}$ zero elements to the bottom sub-column (marked in red), i.e.,

$$\tilde{A}_{\mathsf{opt}}[:, i] = \left(\begin{array}{c} \vdots \\ b_{\mathsf{Col}_{\mathsf{BottomZeros}}-1} \\ 0 \\ \vdots \\ 0 \\ \mathbf{1_i} \\ 0_0 \\ \vdots \\ 0_{\mathsf{Col}_{\mathsf{TopZeros}}-1} \\ b_0 \\ \vdots \\ \hline 0'_{0,i} \\ \vdots \\ 0'_{\mathsf{Col}_{\mathsf{BottomZeros}}-1,i} \\ c'_{\mathsf{Col}_{\mathsf{TopZeros}},i} \\ \vdots \\ c'_{e+k-1,i} \end{array}\right),$$

where b_j denote the elements in the i-th column of $\tilde{A}_{\mathsf{opt}}$, which we aim to shift to non-zero elements.To shift the zero elements at position $b_0, \dots, b_{\mathsf{Col}_{\mathsf{BottomZeros}}-1}$ to the bottom part, we need columns with indices $i+1+\mathsf{Col}_{\mathsf{TopZeros}}, \dots, i+1+\mathsf{Col}_{\mathsf{TopZeros}}+\mathsf{Col}_{\mathsf{BottomZeros}}$ mod $(d+1-k)$ of $\tilde{A}$, as given by $\mathsf{Algo}_{\mathsf{HelperColumns}}$ (cf. Alg. 12).

After computing the positions $\mathsf{Pos}_{\mathsf{BottomZeros}}$ of the zero elements to be shifted ($\mathsf{Algo}_{\mathsf{BottomZeros}}$ in Alg. 11) and the indices $\mathsf{Index}_{\mathsf{HelperCols}}$ of the $\mathsf{HelperCols}$ ($\mathsf{Algo}_{\mathsf{HelperColumns}}$ in Alg. 12), we can use $\mathsf{Algo}_{\mathsf{ZeroShift}}$ (cf. Alg. 13) to shift the zeros in the $\mathsf{TargetCol}$ recursively. Specifically, $\mathsf{Algo}_{\mathsf{ZeroShift}}$ shifts the first zero element to the bottom using one $\mathsf{TargetCol}$ and $\mathsf{Col}_{\mathsf{BottomZeros}}$ $\mathsf{HelperCols}$ and outputs $\mathsf{Col}_{\mathsf{BottomZeros}}$ transformed columns that all have zero elements at the same position. Then, these transformed columns shift the second zero elements to the same position and output $\mathsf{Col}_{\mathsf{BottomZeros}} - 1$ transformed columns. After $\mathsf{Col}_{\mathsf{BottomZeros}}$ iterations, we can get a transformed column that has zero elements at all positions in $\mathsf{Pos}_{\mathsf{BottomZeros}}$.

The matrix $\tilde{A}_{\mathsf{opt}}$ generated by $\mathsf{Algo}_{\mathsf{MatrixTransformation}}$ (cf. Alg. 14) can also be interpreted as a transformation of $\tilde{A}$, i.e., $\tilde{A}_{\mathsf{opt}} = \tilde{A} \cdot T_{\mathsf{Transformation}}$. Note that all the columns of $\tilde{A}_{\mathsf{opt}}$ are directly computed from the columns of $\tilde{A}$ rather than from the previously updated $\tilde{A}_{\mathsf{previous}}$ (on line 12). Consequently, the rank of $\tilde{A}_{\mathsf{opt}}$ and $T_{\mathsf{Transformation}}$ may be lower than that of $\tilde{A}$, where the rank equality check (on line 10) fails. When such a rank-decreasing event occurs (e.g., during the l-th iteration, where $\mathsf{Index}_{\mathsf{TargetCol}} == (l-1)$), we output the previously generated $\tilde{A}_{\mathsf{previous}}$ (line 12). In this case, only the zero elements in the first l columns of $\tilde{A}_{\mathsf{opt}}$ are shifted, while the remaining $d-k-l$ columns are identical to that of $\tilde{A}$. This is discussed further in Supp. C.4. Additionally, note that the algorithm will already stop once enough zeros are shifted, meaning our goal of at least $\mathsf{Row}_{\mathsf{Zeros}}$ zeros per row is reached. Hence, it might not be necessary to transform all columns. In that case, the remaining columns will directly equal the ones of $\tilde{A}$.

Algorithm 11 $\mathsf{Algo}_{\mathsf{BottomZeros}}(\mathsf{Index}_{\mathsf{TargetCol}}, \mathsf{Col}_{\mathsf{BottomZeros}})$

Input : The index $\mathsf{Index}_{\mathsf{TargetCol}}$ of the currently considered target column. The number $\mathsf{Col}_{\mathsf{BottomZeros}}$ of the zero elements to be shifted to the bottom of each column.

Output : $\mathsf{Pos}_{\mathsf{BottomZeros}}$ containing the target positions where we wish to shift zeros inside TargetCol.

1 **for** $i \in \{0, \ldots, \mathsf{Col}_{\mathsf{BottomZeros}} - 1\}$ **do**

2 $\quad \mathsf{Pos}_{\mathsf{BottomZeros}}[i] \leftarrow ((\mathsf{Index}_{\mathsf{TargetCol}} \cdot \mathsf{Col}_{\mathsf{BottomZeros}} + i) \mod e + k) + d + 1 - k$

3 **return** $\mathsf{Pos}_{\mathsf{BottomZeros}}$

Algorithm 12 $\mathsf{Algo}_{\mathsf{HelperColumns}}(\mathsf{Index}_{\mathsf{TargetCol}}, \mathsf{Col}_{\mathsf{TopZeros}})$

Input : The index $\mathsf{Index}_{\mathsf{TargetCol}}$ of the currently considered target column. The number $\mathsf{Col}_{\mathsf{TopZeros}}$ of zero elements to be maintained in the top part of each column.

Output : $\mathsf{Index}_{\mathsf{HelperCols}}$ containing the indices of the columns of HelperCols within the matrix.

1 **for** $i \in \{0, \ldots, d - k - \mathsf{Col}_{\mathsf{TopZeros}} - 1\}$ **do**

2 $\quad \mathsf{Index}_{\mathsf{HelperCols}}[i] \leftarrow ((\mathsf{Index}_{\mathsf{TargetCol}} + i + 1 + \mathsf{Col}_{\mathsf{TopZeros}}) \mod d + 1 - k)$

3 **return** $\mathsf{Index}_{\mathsf{HelperCols}}$

C.3 Proving the Correctness of Algorithm $\mathsf{Algo}_{\mathsf{MatrixTransformation}}$

We can show that the columns of the output matrix of the algorithm are a basis of the set of all zero-encodings.

Lemma 17. *Let the columns of* $\tilde{A} = \begin{pmatrix} I \\ A \end{pmatrix}$ *form a basis of the vector space* $\boldsymbol{Z}$ *consisting of all zero-encodings. Then, on input* $\tilde{A}$*, Algorithm* $\mathsf{Algo}_{\mathsf{MatrixTransformation}}$ *outputs a matrix* $\tilde{A}_{\mathsf{opt}}$ *and the columns of* $\tilde{A}_{\mathsf{opt}}$ *represent a basis of* $\boldsymbol{Z}$*.*

Proof. Algorithm $\mathsf{Algo}_{\mathsf{MatrixTransformation}}$ computes the columns for $\tilde{A}_{\mathsf{opt}}$ by applying linear combinations on the columns of $\tilde{A}$. By Lemma 3, we know that a linear combination of zero-encodings is also a zero-encoding. Thus, the columns of the output matrix $\tilde{A}_{\mathsf{opt}}$ consist of zero-encodings. Moreover, Algorithm $\mathsf{Algo}_{\mathsf{MatrixTransformation}}$ ensures that the ranks of $\tilde{A}$ and $\tilde{A}_{\mathsf{opt}}$ are equal. When representing the algorithm as a transformation from $\tilde{A}$ to $\tilde{A}_{\mathsf{opt}}$, i.e., $\tilde{A}_{\mathsf{opt}} = \tilde{A} \cdot T_{\mathsf{Transformation}}$, this implies that $T_{\mathsf{Transformation}}$ has full rank, meaning it is invertible. Hence, since the columns of $\tilde{A}$ represent a basis, the columns of $\tilde{A}_{\mathsf{opt}}$ do as well. □

It is left to prove that the algorithm indeed balances the number of zeros per row.

Algorithm 13 $\mathsf{Algo}_{\mathsf{ZeroShift}}\left(\tilde{A}, \tilde{A}_{\mathsf{tmp}}, \mathsf{Index}_{\mathsf{TargetCol}}, \mathsf{Pos}_{\mathsf{BottomZeros}}, \mathsf{Index}_{\mathsf{HelperCols}}\right)$

Input : An $n \times (d+1-k)$ matrix $\tilde{A} = \begin{pmatrix} I \\ A_{(e+k)\times(d+1-k)} \end{pmatrix}$. An $n \times (d+1-k)$ matrix $\tilde{A}_{\mathsf{tmp}}$ storing the intermediate result. $\mathsf{Index}_{\mathsf{TargetCol}}$ is the index of the currently considered target column. $\mathsf{Index}_{\mathsf{HelperCols}}$ containing the indices of the columns of HelperCols within the matrix.

Output : Column OutputCol transformed from TargetCol with zeros at positions $\mathsf{Pos}_{\mathsf{BottomZeros}}$, i.e., $\mathsf{OutputCol}[i] = 0$, $i \in \mathsf{Pos}_{\mathsf{BottomZeros}}$.

```
// Count the total number of zeros to be shifted
```

$\mathsf{ZerosToShift} \leftarrow \mathsf{size}\left(\mathsf{Pos}_{\mathsf{BottomZeros}}\right)$
if $\mathsf{ZerosToShift} = 1$ **then**
  $\mathsf{TargetCol}_{\mathsf{tmp}} \leftarrow \tilde{A}_{\mathsf{tmp}}[:, \mathsf{Index}_{\mathsf{TargetCol}}]$
  $\mathsf{HelperCol}_{\mathsf{tmp}} \leftarrow \tilde{A}_{\mathsf{tmp}}[:, \mathsf{Index}_{\mathsf{HelperCols}}[0]]$
  if $\mathsf{TargetCol}_{\mathsf{tmp}}[\mathsf{Pos}_{\mathsf{BottomZeros}}[0]] = 0$ **then**
    return $\mathsf{TargetCol}_{\mathsf{tmp}}$
  $\mathsf{TargetCol}_{\mathsf{tmp}} \leftarrow \mathsf{TargetCol}_{\mathsf{tmp}} \cdot \mathsf{HelperCol}_{\mathsf{tmp}}[\mathsf{Pos}_{\mathsf{BottomZeros}}[0]] - \mathsf{HelperCol}_{\mathsf{tmp}} \cdot \mathsf{TargetCol}_{\mathsf{tmp}}[\mathsf{Pos}_{\mathsf{BottomZeros}}[0]]$
  return $\mathsf{TargetCol}_{\mathsf{tmp}}$
else
  $\mathsf{TargetCol}_{\mathsf{tmp1}} \leftarrow \mathsf{Algo}_{\mathsf{ZeroShift}}\left(\tilde{A}, \tilde{A}_{\mathsf{tmp}}, \mathsf{Index}_{\mathsf{TargetCol}}, \mathsf{Pos}_{\mathsf{BottomZeros}}[0:-1], \mathsf{Index}_{\mathsf{HelperCols}}[0:-1]\right)$
  $\mathsf{TargetCol}_{\mathsf{tmp2}} \leftarrow \mathsf{Algo}_{\mathsf{ZeroShift}}\left(\tilde{A}, \tilde{A}_{\mathsf{tmp}}, \mathsf{Index}_{\mathsf{HelperCols}}[0], \mathsf{Pos}_{\mathsf{BottomZeros}}[0:-1], \mathsf{Index}_{\mathsf{HelperCols}}[1:]\right)$
  $\tilde{A}_{\mathsf{tmp}}[:, \mathsf{Index}_{\mathsf{TargetCol}}] \leftarrow \mathsf{TargetCol}_{\mathsf{tmp1}}$
  $\tilde{A}_{\mathsf{tmp}}[:, \mathsf{Index}_{\mathsf{HelperCols}}[0]] \leftarrow \mathsf{TargetCol}_{\mathsf{tmp2}}$
  $\mathsf{TargetCol}_{\mathsf{tmp3}} \leftarrow \mathsf{Algo}_{\mathsf{ZeroShift}}\left(\tilde{A}, \tilde{A}_{\mathsf{tmp}}, \mathsf{Index}_{\mathsf{TargetCol}}, \mathsf{Pos}_{\mathsf{BottomZeros}}[-1], \mathsf{Index}_{\mathsf{HelperCols}}[0]\right)$
  return $\mathsf{TargetCol}_{\mathsf{tmp3}}$

Algorithm 14 $\mathsf{Algo}_{\mathsf{MatrixTransformation}}\left(\mathsf{Col}_{\mathsf{BottomZeros}}, \mathsf{Col}_{\mathsf{TopZeros}}\right)$

Input : The number $\mathsf{Col}_{\mathsf{BottomZeros}}$ of the zero elements to be shifted to the bottom of each column. The number $\mathsf{Col}_{\mathsf{TopZeros}}$ of zero elements to be maintained in the top part of each column.

Output : The optimized matrix $\tilde{A}_{\mathsf{opt}}$.

$\tilde{A}_{\mathsf{opt}} \leftarrow \tilde{A}$, where $\tilde{A}$ is generated according to Equation 1
$\tilde{A}_{\mathsf{previous}} \leftarrow \tilde{A}_{\mathsf{opt}}$
for $\mathsf{Index}_{\mathsf{TargetCol}} \in \{0, \ldots, d-k\}$ **do**
  $\tilde{A}_{\mathsf{tmp}} \leftarrow \tilde{A}$
  $\mathsf{Pos}_{\mathsf{BottomZeros}} \leftarrow \mathsf{Algo}_{\mathsf{BottomZeros}}\left(\mathsf{Index}_{\mathsf{TargetCol}}, \mathsf{Col}_{\mathsf{BottomZeros}}\right)$
  $\mathsf{Index}_{\mathsf{HelperCols}} \leftarrow \mathsf{Algo}_{\mathsf{HelperColumns}}\left(\mathsf{Index}_{\mathsf{TargetCol}}, \mathsf{Col}_{\mathsf{TopZeros}}\right)$
  $\mathsf{TargetCol} \leftarrow \mathsf{Algo}_{\mathsf{ZeroShift}}\left(\tilde{A}, \tilde{A}_{\mathsf{tmp}}, \mathsf{Index}_{\mathsf{TargetCol}}, \mathsf{Pos}_{\mathsf{BottomZeros}}, \mathsf{Index}_{\mathsf{HelperCols}}\right)$
  $\tilde{A}_{\mathsf{opt}}[:, \mathsf{Index}_{\mathsf{TargetCol}}] \leftarrow \mathsf{TargetCol}$
  // Compute $T_{\mathsf{Transformation}}$, where $\tilde{A}_{\mathsf{opt}} = \tilde{A}_{\mathsf{previous}} \cdot T_{\mathsf{Transformation}}$
  $T_{\mathsf{Transformation}} \leftarrow \mathsf{Solve_Right}\left(\tilde{A}_{\mathsf{previous}}, \tilde{A}_{\mathsf{opt}}\right)$
  // Check if $T_{\mathsf{Transformation}}$ has full rank, and if the number of zeros in each row is greater than $\mathsf{Row}_{\mathsf{Zeros}}$
  if $\mathsf{Rank}\left(\tilde{A}\right) \neq \mathsf{Rank}\left(T_{\mathsf{Transformation}}\right)$ **or** *number of zeros in each row* $> \mathsf{Row}_{\mathsf{Zeros}}$ **then**
    return $\tilde{A}_{\mathsf{previous}}$, $\mathsf{Error}_{\mathsf{EarlyStop}}$
  $\tilde{A}_{\mathsf{previous}} \leftarrow \tilde{A}_{\mathsf{opt}}$
return $\tilde{A}_{\mathsf{opt}}$

Lemma 18. *Let z denote the number of zeros of the input matrix $\tilde{A}$. Then, if Algorithm 14 does not output the early-stop error $\mathsf{Error}_{\mathsf{EarlyStop}}$, the output matrix $\tilde{A}_{\mathsf{opt}}$ has at least $\mathsf{Row}_{\mathsf{Zeros}} = \lfloor \frac{z}{n} \rfloor$ zeros in each row.*

Proof. From algorithms $\mathsf{Algo}_{\mathsf{BottomZeros}}$ (cf. Alg. 11) and $\mathsf{Algo}_{\mathsf{HelperColumns}}$ (cf. Alg. 12) and the structure of $\tilde{A}$, we can verify that matrix $\tilde{A}_{\mathsf{opt}}$ has at least $\mathsf{Row}_{\mathsf{Zeros}} = \lfloor \frac{(d+1-k)^2-(d+1-k)}{n} \rfloor$ zeros in each row. For the top part, positions of zeros that are kept unchanged are in the zig-zag form, guaranteeing that each row in the top part has at least $\mathsf{Row}_{\mathsf{Zeros}}$ zeros. The bottom part has at least $\mathsf{Row}_{\mathsf{Zeros}}$ in each row because we uniformly shift the zeros from the top part to each row. □

C.4 Rank of $\tilde{A}_{\mathsf{opt}}$.

Algorithm $\mathsf{Algo}_{\mathsf{MatrixTransformation}}$ (cf. Alg. 14) can be represented as a transformation from $\tilde{A}$ to $\tilde{A}_{\mathsf{opt}}$, i.e., $\tilde{A}_{\mathsf{opt}} = \tilde{A} \cdot T_{\mathsf{Transformation}}$ with respect to matrix $T_{\mathsf{Transformation}}$. Since we compute all the columns of $\tilde{A}_{\mathsf{opt}}$ directly from the columns of $\tilde{A}$ rather than from the previously updated matrix $\tilde{A}_{\mathsf{previous}}$, the column operations are far more complex than the elementary column operations. While using only elementary operations would have guaranteed that the rank of $\tilde{A}_{\mathsf{opt}}$ remains the same as $\tilde{A}$, our combinations allow that the rank of $\tilde{A}_{\mathsf{opt}}$ and $T_{\mathsf{Transformation}}$ may be lower than the one of $\tilde{A}$. In other words, $T_{\mathsf{Transformation}}$ might be a non-bijective function. Whether or not this is the case depends on the support points used as well as their permutation.

To circumvent this issue, we insert a rank check in our algorithm (cf. line 10 in Alg. 14). If such a rank-decreasing event occurs (e.g., during the l-th iteration, where $\mathsf{Index}_{\mathsf{TargetCol}} == (l-1)$), we output the previously generated $\tilde{A}_{\mathsf{previous}}$. In this case, the zero elements in the first l columns of $\tilde{A}_{\mathsf{opt}}$ are shifted, while the remaining $d-k-l$ columns are identical to the ones of $\tilde{A}$.

Considering the field $\mathsf{GF}(2^8)$ and a modulus of $m(x) = x^8 + x^4 + x^3 + x + 1$, we evaluated this algorithm for two different scenarios. First, we chose the support points uniformly random but pairwise distinct from $\mathsf{GF}(2^8)$. For each parameter set (n, d, k) where $n \in [30], d \in [18], k \in [17]$, we performed 10 000 trials of sampling the support points and computing the optimized matrix and found that the probability p of such a rank-decreasing event happening is rather low with $p < 0.01$.

Second, we chose Frobenius support points. When considering AES, [PR11, ABEO24, FOWZ25] demonstrated that using Frobenius support points can accelerate certain exponentiation operations. Faust et al. provided a detailed explanation of constructing a combined secure exponentiation gadget with respect to packed secret sharing using Frobenius support points in [FOWZ25, Section 5]. Please note that instead of considering two pairwise disjoint sets of Frobenius support points, one for the secrets and one for the shares, we can equivalently consider one large pairwise disjoint set of Frobenius support points including both the support points for the secret as well as the ones for the shares. The only change in this case is that all the support points need to be considered for the permutation afterwards. For further details on the exponentiation and permutation gadget, we refer to [FOWZ25]. We sample support points set of size $d+1+n$ using the algorithm from [PR11, Appendix D]. We were able to find that for every combination of parameters (n, d, k) where $n \in [30], d \in [18], k \in [17]$, there exist at least one tuple of sets of Frobenius support points such that, when in the right order, the output $\tilde{A}_{\mathsf{opt}}$ of $\mathsf{Algo}_{\mathsf{MatrixTransformation}}$ has the same rank as $\tilde{A}$.

D Further Information on Gadgets

D.1 Share-Wise Gadgets

Algorithm 15 SWAdd for $n = d + e + 1$, and $k \leq d$ [FOWZ25, Algorithm 1]

Input : *Degree-d* shares of $\boldsymbol{v}^{(F)} = \left(v_0^{(F)}, \ldots, v_d^{(F)}\right)$ as $(F_i)_{i\in[n]}$ and degree-d shares of $\boldsymbol{v}^{(G)} = \left(v_0^{(G)}, \ldots, v_d^{(G)}\right)$ as $(G_i)_{i\in[n]}$

Output : *Degree-d* shares of $\boldsymbol{v}^{(F)} + \boldsymbol{v}^{(G)} = \left(v_0^{(F)} + v_0^{(G)}, \ldots, v_d^{(F)} + v_d^{(G)}\right)$ as $(Q_i)_{i\in[n]}$

for $i \in \{0, \ldots, n-1\}$ **do**
  $Q_i \leftarrow F_i + G_i$
return $(Q_i)_{i\in[n]}$

Algorithm 16 SWMult for $n = d + e + 1$ [FOWZ25, Algorithm 2]

Input : *Degree-d* shares of $\boldsymbol{v}^{(F)} = \left(v_0^{(F)}, \ldots, v_d^{(F)}\right)$ as $(F_i)_{i\in[n]}$ and degree-d shares of $\boldsymbol{v}^{(G)} = \left(v_0^{(G)}, \ldots, v_d^{(G)}\right)$ as $(G_i)_{i\in[n]}$

Output : *Degree-2d* shares of $\boldsymbol{v}^{(F)} \cdot \boldsymbol{v}^{(G)} = \left(v_0^{(F)} \cdot v_0^{(G)}, \ldots, v_d^{(F)} \cdot v_d^{(G)}\right)$ as $(Q_i)_{i\in[n]}$

for $i \in \{0, \ldots, n-1\}$ **do**
  $Q_i \leftarrow F_i \cdot G_i$
return $(Q_i)_{i\in[n]}$

D.2 Multiplication Gadget

Algorithm 17 Mult$_{\mathsf{k-Input}}$ for $n = d + e + 1$ and $k < \frac{d}{2}$ [FOWZ25, Algorithm 9]

Input : *Degree-d* shares of $\boldsymbol{v}^{(F)} = \left(v_0^{(F)}, \ldots, v_d^{(F)}\right)$ as $(F_i)_{i\in[n]}$ and degree-d shares of $\boldsymbol{v}^{(G)} = \left(v_0^{(G)}, \ldots, v_d^{(G)}\right)$ as $(G_i)_{i\in[n]}$

Output : *Degree-d* shares of $\left(v_0^{(F)} \cdot v_0^{(G)}, \ldots, v_{k-1}^{(F)} \cdot v_{k-1}^{(G)}\right)$ as $(Q_i)_{i\in[n]}$

$\left((F'_i)_{i\in[n]}, (F''_i)_{i\in[n]}\right) \leftarrow \mathbf{SplitRed}_{\hat{A}\leftarrow \mathsf{Comp}\hat{A}(d,\frac{d}{2},n)}\left((F_i)_{i\in[n]}\right)$
$\left((G'_i)_{i\in[n]}, (G''_i)_{i\in[n]}\right) \leftarrow \mathbf{SplitRed}_{\hat{A}\leftarrow \mathsf{Comp}\hat{A}(d,\frac{d}{2},n)}\left((G_i)_{i\in[n]}\right)$
$\left(H_i^{(0)}\right)_{i\in[n]} \leftarrow \mathbf{SWMult}\left((F'_i)_{i\in[n]}, (G'_i)_{i\in[n]}\right)$
$\left(H_i^{(1)}\right)_{i\in[n]} \leftarrow \mathbf{SWMult}\left((F'_i)_{i\in[n]}, (G''_i)_{i\in[n]}\right)$
$\left(H_i^{(2)}\right)_{i\in[n]} \leftarrow \mathbf{SWMult}\left((F''_i)_{i\in[n]}, (G'_i)_{i\in[n]}\right)$
$\left(H_i^{(3)}\right)_{i\in[n]} \leftarrow \mathbf{SWMult}\left((F''_i)_{i\in[n]}, (G''_i)_{i\in[n]}\right)$
$(Q_i)_{i\in[n]} \leftarrow \mathbf{SWComp}\left(\left(H_i^{(0)}\right)_{i\in[n]}, \left(H_i^{(1)}\right)_{i\in[n]}, \left(H_i^{(2)}\right)_{i\in[n]}, \left(H_i^{(3)}\right)_{i\in[n]}\right)$
return $(Q_i)_{i\in[n]}$

E Other Observations: FFT

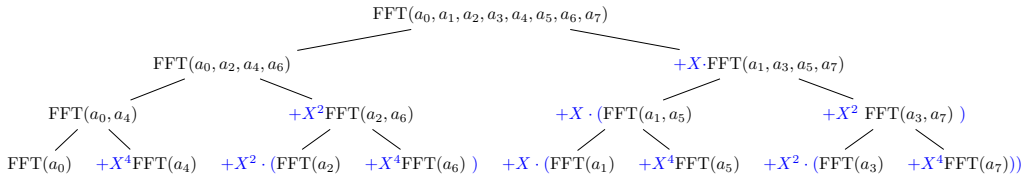
We illustrate the challenges of using Shamir's secret sharing (or packed secret sharing) with the FFT algorithm (Cooley-Tukey, 1965) when aiming for fault resilience with an example. We begin by recalling the FFT algorithm. Consider the FFT algorithm applied to the coefficients $(a_0, \ldots, a_7)$ with respect to the standard basis $(1, X, \ldots, X^d)$. FFT employs a

divide-and-conquer technique to optimize computations. The divide part can be described as follows. Specifically, the polynomial $p(X) = a_0 + a_1X + a_2X^2 + \cdots + a_7X^7$ can be rewritten as $p(X) = \left(a_0 + a_2X^2 + a_4X^4 + a_6X^6\right) + X \cdot \left(a_1 + a_3X^2 + a_5X^4 + a_7X^6\right) = p_{\text{even}}(X^2) + X \cdot p_{\text{odd}}(X^2)$. This transformation divides the original polynomial into two polynomials with respect to X^2, one containing only the even coefficients and the other only the odd ones.
We can see how this construction helps when considering the conquer part. The evaluation points of the polynomial are the n-th root of unity ω and its powers ω^i for $i \in [n]$. A key property of the n-th root of unity ω is, that $\omega^n = \omega^0 = 1$. Recall that powers ω are periodic with period n, since the n-th roots of unity can be visualized as points on the complex unit circle, spaced $2\pi/n$ radians apart. Thus, multiplying by ω corresponds to a clockwise rotation by this angle, and multiplying with ω^n rotates by 2π radians, i.e., $\omega^j = \omega^{j+n}$. Hence, all together, if n is even, say $n = 2 \cdot w$, we obtain all n of the evaluations by plugging in the w-th roots of unity into p_{even} and p_{odd}. Assume $n = 2^m$. Then, we can reduce the problem of evaluating one polynomial of degree $n-1$, $p(X)$, at the $n-$th roots of unity to that of evaluating two degree $n/2 - 1$ polynomials p' and p_{odd} at the $n/2-$th roots of unity. Consider first p_{even}, we only need to evaluate half $n/2 - 1$ roots of unity, since $p_{\text{even}}(X^{2\cdot(j)}) = p_{\text{even}}(X^{2\cdot(j)+n}) = p_{\text{even}}(X^{2\cdot(j+n/2)})$. Thus, reconstructing Similarly, we can argue for the odd elements p_{odd}. Hence, continuing this approach, we end up in a polynomial of degree 0, which is easy to evaluate, since it is a constant.

All together, evaluating a polynomial of degree n at n distinct points, namely the n-th roots of unity, has computational complexity $\mathcal{O}(n \cdot \log_2(n))$.

However, this also results in one error being propagated through the elements. Even if we assume the lowest layer to be hard-coded, an adversary can fault two elements in the second-last layer.

We can summarize the FFT algorithm in the following figure:

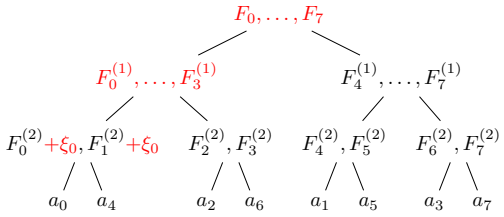


Let $F_0, \ldots, F_{n-1}$ denote the n resulting shares of the FFT algorithm. We can calculate the n resulting shares $F_0, \ldots, F_{n-1}$ with the help of the lower nodes. For elements Starting counting the layers from top, Layer 0, to bottom, Layer $\log_2(n) = m$, we can calculate the elements of layer i, $0 \leq i \leq m$, as

$$F_j^{(i)} = F_j^{(i+1)} + \omega^j \cdot$$
$$F_{j+1}^{(i)} = F_j^{(i+1)} + \omega^{j+n/2^j} \cdot$$

For example, starting from the bottom layer, we calculate $F_0^{(2)} = a_0 + \omega^0 \cdot a_4$ and $F_1^{(2)} = a_0 + \omega^{0+n/4} \cdot a_4$. Similarly, we compute $F_0^{(1)} = F_1^{(2)} + \omega^0 \cdot F_2^{(2)}$ and $F_1^{(1)} = F_1^{(2)} + \omega^{0+n/4} \cdot F_2^{(2)}$, continuing until the top layer.

The issue when inserting two faults $\xi_0 = \xi_1 \in \mathcal{F}^+$ is that with these two inserted faults, an adversary can generate a valid encoding of $a_0 + \xi_0$ by inserting two faults in the second last layer. Following the formula above, the error propagates to all shares $F_i, i \in [n]$ as depicted in red in the following figure:



We can generalize this idea. Let e denote the number of possible faults, d the degree of the polynomial, and $n = d + e + 1$ the number of shares. Moreover, we assume that $n = 2^m$. Our counterexample shows that an adversary can use any node of the FFT tree with less than or equal to e elements to create a valid encoding of a different secret. Counting the layers from top (Layer 0) to Bottom (Layer $\log(n) = m$), note that Layer j, with $0 \leq j \leq m$, contains 2^j nodes, each of which contains $n/2^j$ elements. As already mentioned, an adversary can use our attack as long as $e \geq n/2^j$. Thus, considering $\mathbb{F}_q$, where $q >> n$, we obtain

$$\begin{aligned} &\iff 2^j \geq n/e \\ &\iff 2^j \geq d+1/e + 1 \\ &\iff j > \log(d+1) - \log(e) \end{aligned}$$

This means that the attack can be used for all j, such that $j > \log(d+1) - \log(e)$. Hence, when aiming to use FFT, while still being secure against combined attacks, further countermeasures are needed, especially in case e and d are in a similar range. If $\log(e) \approx \log(d)$, the adversary could probe nearly any layer $j \neq 0$ to create a new valid encoding.

In case $d >> e$, such that $\log(d) >> \log(e)$, FFT could be used, but it needs further countermeasures against this attack for the higher layers. Note that we can assume that the FFT tree only has $\log(d)$ layers instead of $\log(n)$, since FFT is a deterministic algorithm and all but d leafs are zero. Hence, one can *hardcode* the bottom layer $\log(d)$ without losing the improvement of FFT.

However, using the additional countermeasures, i.e., duplicating or using a matrix multiplication to construct the layers $j > \log(d+1) - \log(e)$. However, doing so, we increase the complexity a lot. Thus, further improvements would be needed.

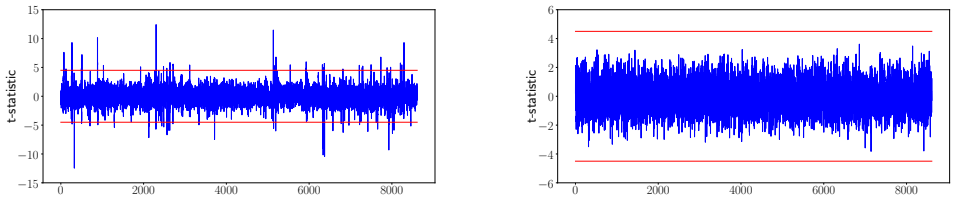
F Additional Experiments

F.1 Leakage Assessment

In addition to performing TVLA for the **LaOla** multiplication, we also evaluate the security of our Sbox implementation using **LaOla**. Similarly to Section 6.3, we perform a fixed-vs-random t-test to assess leakage. As seen in Fig. 7a, leakage is detectable using 1 000 traces with randomness disabled. When running TVLA on the Sbox with 1 000 000 traces using pre-sampled randomness, no leakage is detectable (see Fig. 7b).

F.2 Fault Injection Simulation

In Table 3, we present the results of our simulated fault injection experiments on the **LaOla** multiplication with 100 000 000 runs per parameter set. Furthermore, we report the results for set-zero faults on $d = 1, e = 3, s = 5$ with 10^{10} runs in Table 2.



(a) 1 000 traces, randomness zeroed (b) 1 000 000 traces, presampled randomness

Figure 7: Results of TVLA assessment for the Sbox (a) 1st order without randomness, (b) 1st order with randomness. The x-axis represents sample point indices.

Table 2: Set-zero fault injection for 10^{10} experiments at $d = 1, e = 3, s = 5$.

e	s	Detected	EUFs
3	5	9 928 707 864	0

A Zero-Encodings

Lemma 3 states that the zero-encodings with respect to k secrets and n shares embedded in polynomials of degree up to d in $\mathbb{F}$ are a sub-vector space of $\mathbb{F}^n$. The proof uses the following lemma showing that linear combinations of zero-encodings are zero-encodings.

Lemma 19. *Let $\boldsymbol{F} = (F_0, \ldots, F_{n-1})^T$ and $\boldsymbol{F'} = (F'_0, \ldots, F'_{n-1})^T$ be two zero-encodings. Let $a, b \in \mathbb{F}$ be arbitrary values in field $\mathbb{F}$. Then, $\boldsymbol{F''} = a \cdot \boldsymbol{F} + b \cdot \boldsymbol{F'} = (a \cdot F_0 + b \cdot F'_0, \ldots, a \cdot F_{n-1} + b \cdot F'_{n-1})$ is a zero-encoding as well.*

Proof. Assume $(F_0, \ldots, F_{n-1})$ is an encoding of $\boldsymbol{v} = (v_0 = 0, \ldots, v_{k-1} = 0, v_k, \ldots, v_d)$ and $(F'_0, \ldots, F'_{n-1})$ is an encoding of $\boldsymbol{v'} = (v'_0 = 0, \ldots, v'_{k-1} = 0, v'_k, \ldots, v'_d)$. To prove that $\boldsymbol{F''}$ is a valid zero-encoding, we first show that the elements of $\boldsymbol{F''}$ are evaluations of a polynomial p with respect to evaluation points $(x_0, \ldots, x_{n-1})$. Then, we show that the first k underlying secrets $p(u_0), \ldots, p(u_{k-1})$ of p are equal to zeros and the remaining $d + 1 - k$ embedded secrets $p(u_k), \ldots, p(u_d)$ are uniform random values. Recall that share-wise addition or multiplication with a constant value (except for value zero) does not change the degree of the underlying polynomial staying up to d [FY92]. Since $\boldsymbol{F''}$ is a linear-combination of $(F_0, \ldots, F_{n-1})$ and $(F'_0, \ldots, F'_{n-1})$, which are shares with respect to a polynomial of degree up to d, $\boldsymbol{F''}$ is also a polynomial of degree at most d. For $l \in [d+1]$, we have $v''_l = \sum_{i=0}^{d} M^{-1}_{l,i} \cdot (a \cdot F_i + b \cdot F'_i) = \sum_{i=0}^{d} (a \cdot F_i \cdot M^{-1}_{i,l} + b \cdot F'_i \cdot M^{-1}_{i,l}) = a \cdot v_l + b \cdot v'_l$, where the matrix M^{-1} is the unmasking matrix (cf. Sec. 2.2). Note that since $v_0, \ldots, v_{k-1} = 0$ and $v'_0, \ldots, v'_{k-1} = 0$, then $v''_0, \ldots, v''_{k-1} = 0$. Hence, linear combinations of zero-encodings are zero-encodings. □

Lemma 20. *Every element of $A_{(n+k-d-1)\times(d+1-k)}$ is non-zero.*

Proof. Let us denote the pairwise distinct support points by $\tilde{x}_1, \ldots, \tilde{x}_\mu$ and $\hat{x}_1, \ldots, \hat{x}_\nu$. The elements of the Vandermonde matrix are thus $\mathsf{Van}_{\mu\times\nu}(\tilde{x}_1, \ldots, \tilde{x}_\mu)_{i,j} = \tilde{x}_i^{j-1}$ where $i \in \{1, \ldots, \mu\}, j \in \{1, \ldots, \nu\}$. The elements of the inverse Vandermonde matrix are $\mathsf{Van}^{-1}_{\nu\times\nu}(\hat{x}_1, \ldots, \hat{x}_\nu)_{i,j} = (-1)^{i+j} S_{n-i,j} P_\nu$, where $l = j$ or $o = j$, $P_\nu = \left(\prod_{l<o}^{\nu} (\hat{x}_o - \hat{x}_l)\right)^{-1}$, and $i, j \in \{1, \ldots, \nu\}$ [Raw19]. For the variables $\hat{x}_1, \ldots, \hat{x}_\nu$, $S_{q,j}$ refers to the elementary symmetric polynomials without $\hat{x}_j$, i.e., $S_q := S_q(\hat{x}_1, \ldots, \hat{x}_\nu) = \sum_{1 \le i_1 \le \cdots \le i_q \le \nu}^{\nu} \hat{x}_{i_1} \hat{x}_{i_2} \ldots \hat{x}_{i_q}$, for $1 \le q \le \nu$, $S_0 := S_0(\hat{x}_1, \ldots, \hat{x}_\nu) = 1$, and $S_q = 0$ for $q \notin \{0, \ldots, \nu\}$ and finally $S_{q,j} := S_q(\hat{x}_1, \ldots, \hat{x}_{j-1}, \hat{x}_{j+1}, \ldots, \hat{x}_\nu)$ for $1 \le q \le \nu - 1$ and $1 \le j \le \nu$.

Table 3: Results of simulated fault injection on **LaOla** for 10^8 experiments across various d, e, s displaying the number of detected and effective but undetected faults (EUFs).

		Additive random				**Set-zero**			
e	s	$d=1$		$d=2$		$d=1$		$d=2$	
		Detected	EUFs	Detected	EUFs	Detected	EUFs	Detected	EUFs
1	1	83 190 169	14 385	78 445 172	8 094	79 488 142	0	76 221 093	0
1	2	96 921 865	193 467	95 137 419	188 600	94 763 847	504 018	94 049 869	169 288
1	3	99 148 100	301 037	98 671 402	297 546	98 234 108	510 804	98 264 791	283 857
1	4	99 513 350	350 983	99 412 720	350 287	99 165 346	443 233	99 280 310	337 304
1	5	99 586 033	369 509	99 568 189	369 313	99 445 025	413 348	99 524 591	365 623
2	1	75 487 264	18	70 824 161	19	73 385 368	0	66 999 876	0
2	2	93 903 520	95	91 444 003	56	92 709 437	0	89 024 701	0
2	3	98 445 418	264	97 475 680	298	97 931 510	206	96 322 655	262
2	4	99 573 450	597	99 244 868	603	99 367 938	358	98 749 702	415
2	5	99 869 072	1 161	99 771 121	957	99 789 818	641	99 572 227	714
3	1	69 788 964	0	66 481 226	0	63 761 164	0	62 086 533	0
3	2	90 779 205	0	88 706 597	0	86 736 506	0	85 478 139	0
3	3	97 156 244	0	96 182 745	0	95 093 894	0	94 467 018	0
3	4	99 089 976	0	98 694 517	0	98 149 339	0	97 913 978	0
3	5	99 699 214	0	99 560 713	0	99 287 005	0	99 211 775	0

Let us now compute the (i,j)-th element of the matrix $\sum_{q=1}^{\nu} \mathsf{Van}_{\mu\times\nu}(\tilde{x}_1,\ldots,\tilde{x}_\mu)_{i,q} \cdot \mathsf{Van}^{-1}_{\nu\times\nu}(\hat{x}_1,\ldots,\hat{x}_\nu)_{q,j} = \sum_{q=1}^{\nu} \tilde{x}_i^{q-1}\cdot(-1)^{q+j}S_{n-q,j}\cdot P_\nu = \sum_{q=1}^{\nu} \tilde{x}_i^{q-1}\cdot(-1)^{q+j}S_{n-q,j}\cdot P_\nu$. By replacing ν with $\nu' = \nu-1$ and writing $S_{n-q,j}$ as $\hat{S}_{\nu-q} := S_{\nu-q}(\hat{x}_1,\ldots,\hat{x}_{j-1},\hat{x}_{j+1},\ldots,\hat{x}_\nu)$, we get $\sum_{q=0}^{\nu'} \tilde{x}_i^q\cdot(-1)^{q+j+1}\hat{S}_{\nu-q}\cdot P_\nu = (-1)^{\nu'+1+j}\sum_{q=0}^{\nu'} \tilde{x}_i^q\cdot(-1)^{\nu-q}\hat{S}_{\nu-q}\cdot P_\nu$ Now, since $\hat{S}_{\nu-q}$ are elementary symmetric polynomials, we can rewrite the numerator and obtain $(-1)^{\nu'+1+j}\prod_{q=1,q\neq j}^{\nu'}(\tilde{x}_i-\hat{x}_q)\cdot P_\nu$. Since all support points are pairwise distinct and since there are no non-trivial zero divisors in fields, it follows that all elements of the product matrix are non-zero. As the computation of $A_{(n+k-d-1)\times(d+1-k)}$ is clearly a special case of this, the lemma follows. □

B Proofs for Section 5 (Secure Compiler)

Lemma 14. *The modified gadget* $\mathsf{SplitRed}_{\mathsf{Comp}\hat{A}(d,\frac{d}{2},n)}$ *(cf. Alg. 8) is* $(d+1-k)$-frNI *with respect to* $\mathcal{F}^+$.

Proof. This proof is similar to the one presented in [FOWZ25, Algorithm 7]. We consider three different sets of probes:

G1 The probes of Group 1, K, contain any probes during the evaluation of any opt-**ZEnc**, and its outputs. Namely, $K = (\bigcup_{l\in\lceil\frac{n}{2}\rceil}\hat{K}^l)\cup(\bigcup_{l\in\lfloor\frac{n}{2}\rfloor}\tilde{K}^l)\cup(\bigcup_{l\in\lceil\frac{n}{2}\rceil}\tilde{K}'^l)\cup(\bigcup_{l\in[n]}K^l)$, where $\hat{K}^l$ denotes the probes in the executions and the outputs of $(\hat{G}_i^l)_{i\in[n]} \leftarrow \text{opt-}\mathbf{ZEnc}_n^{d-j}, l < d+1-k$ and $(\hat{G}_i^l)_{i\in[n]} \leftarrow \text{opt-}\mathbf{ZEnc}_n^{d-j}, d+1-k \le l < \lceil\frac{n}{2}\rceil$. $\tilde{K}^l$ corresponds to probes of $(\tilde{G}_i^l)_{i\in[n]} \leftarrow \text{opt-}\mathbf{ZEnc}_n^{\frac{d}{2}-l}, l < \frac{d}{2}+1-k$ and $(\tilde{G}_i^l)_{i\in[n]} \leftarrow \text{opt-}\mathbf{ZEnc}_n^{\frac{d}{2}}, \frac{d}{2}+1-k \le l < \lfloor\frac{n}{2}\rfloor$. Similarly, $\tilde{K}'^l$ describes probes in $(\tilde{G}_i^{l+\lfloor\frac{n}{2}\rfloor})_{i\in[n]} \leftarrow \text{opt-}\mathbf{ZEnc}_n^{\frac{d}{2}-j}, l < \frac{d}{2}+1-k$ and $(\hat{G}_i^{l+\lfloor\frac{n}{2}\rfloor})_{i\in[n]} \leftarrow \text{opt-}\mathbf{ZEnc}_n^{\frac{d}{2}-l}, d+1-k \le l < \lceil\frac{n}{2}\rceil$. K^l denotes the probes of $G_i^l \leftarrow \tilde{G}_i^l + \hat{G}_i^l$ if $l < \lfloor\frac{n}{2}\rfloor$ and $G_i^l \leftarrow \tilde{G}_i^{j+\lfloor\frac{n}{2}\rfloor} - \hat{G}_i^l$ for $\lfloor\frac{n}{2}\rfloor \le l < n$.

G2 The second group of probes consists of two parts: Group 2.A contains probes of $\mathcal{F}_i'^j \leftarrow \hat{A}_{i,j}\cdot F_j$ for $i\in[n]$ and $j\in\lceil\frac{n}{2}\rceil$, as well as probes of $\mathcal{F}_i^j \leftarrow \mathcal{F}_i'^j + G_i^j$ for $j\in\lceil\frac{n}{2}\rceil$

for $j \in \lceil \frac{n}{2} \rceil$ and $i \in [n]$. Similar to Group 2.A, Group 2.B contains probes of $\mathcal{F}_i'^{j+\lfloor \frac{n}{2} \rfloor} \leftarrow \hat{A}_{i,j+\lfloor \frac{n}{2} \rfloor} \cdot F_{j+\lfloor \frac{n}{2} \rfloor}$ for $j \in \lceil \frac{n}{2} \rceil$ and $i \in [n]$, and probes of $\mathcal{F}_i^{j+\lfloor \frac{n}{2} \rfloor} \leftarrow \mathcal{F}_i'^{j+\lfloor \frac{n}{2} \rfloor} + G_i^{j+\lfloor \frac{n}{2} \rfloor}$ for $j \in \lceil \frac{n}{2} \rceil$ and $i \in [n]$.

G3 Our last group of probes consists of two parts: Part 3.A contains probes of $F_i'^j \leftarrow F_i'^{j-1} + \mathcal{F}_i^j$ for $j \in \lceil \frac{n}{2} \rceil$ and $i \in [n]$. Please note that in this case $F_i'^j$ denotes the j-th iteration of the assignment to F_i'. Furthermore, 3.B consider probes of $F_i''^j \leftarrow F_i''^{j-1} + \mathcal{F}_i^{j+\lfloor \frac{n}{2} \rfloor}$ for $j \in \lceil \frac{n}{2} \rceil, n \in [n]$. Again, the power of j denotes that we consider the j-th iteration of the assignment.

Note that G3 contains the output probes. However, since we only aim to prove NI, we do not need to consider these probes separately.

Constructing input sets: Now, we create the set of input indices for the simulator. Please note that only the set I_F contains indices for input variables $(F_i)_{i\in[n]}$. The set I_R is constructed to simplify the proof and contains superscripts denoting which (additional) zero-encodings need to be constructed. We initialize the sets $I_F = \emptyset = I_R$ and insert elements as follows:

G1 For any probe in $\tilde{K}^l$, we set $I_R \leftarrow I_R \cup \{l\}$ and $I_F \leftarrow I_F \cup \{l\}$. For any probe in K'^l,we assign $I_R \leftarrow I_R \cup \{j + \lfloor \frac{n}{2} \rfloor\}$ and $I_F \leftarrow I_F \cup \{j + \lfloor \frac{n}{2} \rfloor\}$. Furthermore, for any probe in K^l, we set $I_R \leftarrow I_R \cup \{l\}$ and $I_F \leftarrow I_F \cup \{l\}$. For any probe in $\hat{K}^l$, if there are more than $\frac{d}{2} + 1 - k$ probes in $K^l, l < \lfloor \frac{n}{2} \rfloor$, Group2.A or Group3.A, and a probe in $\hat{K}^l$ but not in $\tilde{K}^l, \tilde{K}'^l$ then $I_R \leftarrow I_R \cup \{j\}$ and $I_F \leftarrow I_F \cup \{j\}$. If there are more than $\frac{d}{2} + 1 - k$ probes in $K^l, \lfloor \frac{n}{2} \rfloor \leq l < n$, Group2.B or Group3.B, and a probe in $\hat{K}^l$ but not in $\tilde{K}^l, \tilde{K}'^l$ then $I_R \leftarrow I_R \cup \{j + \lfloor \frac{n}{2} \rfloor\}$ and $I_F \leftarrow I_F \cup \{j + \lfloor \frac{n}{2} \rfloor\}$. Note that this means that I_F obtains at most one element for a probe in $\hat{K}^l$.

G2 For any probe with respect to l in Group 2.A, we set $I_R \leftarrow I_R \cup \{l\}$ and $I_F \leftarrow I_F \cup \{l\}$. For a probe with respect to element l in Group 2.B, we include $l + \lfloor \frac{n}{2} \rfloor$, namely $I_R \leftarrow I_R \cup \{l + \lfloor \frac{n}{2} \rfloor\}$ and $I_F \leftarrow I_F \cup \{l + \lfloor \frac{n}{2} \rfloor\}$.

G3 For a probe with respect to l in Group3.A, we set $I_R \leftarrow I_R \cup \{l\}$ and $I_F \leftarrow I_F \cup \{l\}$. For a probe with respect to l in Group3.B, we put $l + \lfloor \frac{n}{2} \rfloor$ into the input sets, hence $I_R \leftarrow I_R \cup \{l + \lfloor \frac{n}{2} \rfloor\}$ and $I_F \leftarrow I_F \cup \{l + \lfloor \frac{n}{2} \rfloor\}$.

Note that we insert at most one element per probe into I_F. Hence, if t denotes the number of probes, we obtain $|I_F| \leq t$.

Simulating Probes: We now show how to simulate the probes with respect to inputs with indices from I_F. As before, we proceed group by group.

G1 For any element $l \in I_R$, which includes the probes in G1, we simulate all elements $i \in [n]$ corresponding to the value l. Namely, for a probe in $\hat{K}^l$, we simulate $(\hat{G}_i^l)_{i\in[n]}$ by using opt-**ZEnc** directly.Note that this is possible, since opt-**ZEnc** is independent of the input $(F_i)_{i\in[n]}$. Similarly, for a probe in $\tilde{K}^l$, or $\tilde{K}'^l$, we follow opt-**ZEnc** as denoted above to obtain $(\tilde{G}_i^l)_{i\in[n]}$ for $l \in [n]$. If $l < \lfloor \frac{n}{2} \rfloor$ we simulate $G_i^l = \tilde{G}_i^l + \hat{G}_i^l$ by additionally computing the distribution of $\hat{G}_i^l$, for $i \in [n]$. Otherwise, we simulate $G_i^l = \tilde{G}_i^l - \hat{G}_i^{l-\lfloor \frac{n}{2} \rfloor}$ by additionally computing the distribution of $\hat{G}_i^{l-\lfloor \frac{n}{2} \rfloor}$ for $i \in [n]$. It is left to simulate G_i^l for the remaining elements $l \in I_R$ that have not been simulated yet. Note that this includes the probes in K^l. If $l < \lfloor \frac{n}{2} \rfloor$, probes in K^l are simulated as follows. If $(\tilde{G}_i^l)_{i\in[n]}$ or $(\hat{G}_i^l)_{i\in[n]}$ are already simulated, use these elements, otherwise simulate the elements as described above. Then compute $(G_i^l)_{i\in[n]} = (\tilde{G}_i^l)_{i\in[n]} + (\hat{G}_i^l)_{i\in[n]}$. Otherwise $l \geq \lfloor \frac{n}{2} \rfloor$. In this case, probes in K^l are simulated as follows. If $(\tilde{G}_i^{l+\lfloor \frac{n}{2} \rfloor})_{i\in[n]}$ or $(\hat{G}_i^l)_{i\in[n]}$ are already simulated, use these elements. Otherwise, simulate them as described above. Then compute $(G_i^l)_{i\in[n]} = (\tilde{G}_i^l)_{i\in[n]} - (\hat{G}_i^l)_{i\in[n]}$. This perfectly simulates all probes in K.

G2 For any probe of $\mathcal{F}_i'^l$ we can directly set $\mathcal{F}_i'^l = \hat{A}_{i,l} \cdot F_l$, since $\hat{A}_{i,l}, i \in [n]$ can be simulated perfectly as it does not take any inputs and is calculated out of public parameters

and elements that are chosen independent of each other and uniformly at random (cf. [ABE+25]) and $l \in I_F$ by construction of the index set. If $\mathcal{F}_i^l$ is probed, then $l \in I_R$ and $l \in I_F$. The element G_i^l was already simulated in G1, since $l \in I_R$. Hence, we use this value and calculate $\mathcal{F}_i^l = \hat{A}_{i,l} \cdot F_l + G_i^l$.

G3 We start by considering Group 3.A. Sort the probes according to $j \in \lfloor \frac{n}{2} \rfloor$, i.e., let $j_0 \leq j_1 \leq \ldots, \leq j_{m-1}$ for $m \in \mathbb{N}$ denote probes of $F_i'^{j_w}$, $w \in [m]$, i.e. the probes of Group 3.A. (Note that we assume the existence of at least one probe here, since otherwise we do not need to simulate any element $F_i'^j$.) Let $P_w \coloneqq \{j \in [j_w] | j \in (I_R \cup \hat{K}^j)\}$, for $w \in [m]$ denote the set of indices for which there was already a probe. Moreover, we denote $P = \bigcup_{w \in [m]} P_w$. We start by proving by induction that if it exists an $j \in [j_w] : j \notin P$, then $F_i^{j_w}$ is at least $d+1-k-|P_w|$-wise independent and any $d+1-k-|P_w|$ tuple of $(F_i^{j_w})_{i \in [n]}$ is uniformly distributed.

Induction basis $j_w = j_0$: Recall that $(F_i'^{j_0})_{i \in [n]} = \sum_{j=0}^{j_0} (\hat{A}_{i,j} \cdot F_j + G_i^j)_{i \in [n]}$. Let $\ell = \min\{j \in [j_0] | j \notin P_w\}$ and $\ell' = \max\{j \in [j_0] | j \notin P_w\}$. If $\ell' \geq d+1-k$, then $(G_i^\ell)_{i \in [n]}$ are $d+1-k$-wise uniformly independent and independent of $(F_i)_{i \in [n]}$ by correctness of opt-**ZEnc**. Hence, $(F_i'^{j_0})_{i \in [n]}$ is at least $d+1-k-|P_w|$-wise uniformly independent.
Otherwise, we know that $(G_i^\ell)_{i \in [n]}$ are $d+1-k-\ell$-wise uniformly independent. By definition of ℓ, $\ell \leq |P_w|$, since $\forall j \in [\ell] : j \in P_w$. Thus, $(F_i'^{j_0})_{i \in [n]}$ is at least $d+1-k-|P_w|$-wise uniformly independent.

Induction step $j_{w-1} \to j_w$: By construction, we have $(F_i'^{j_w})_{i \in [n]} = (F_i'^{j_{w-1}})_{i \in [n]} + \sum_{j=j_{w-1}}^{j_w} (\hat{A}_{i,j} \cdot F_j + G_i^j)_{i \in [n]}$. We define $\ell' = \max\{j \in [j_{w-1}, j_w] | j \notin P_w\}$. If $\ell' \geq d+1-k$, then the $(G_i^\ell)_{i \in [n]}$ are $d+1-k$-wise uniformly independent, hence $(F_i'^{j_w})_{i \in [n]}$ is $d+1-k-|P_w \setminus P_{w-1}| \geq d+1-k-|P_w|$- wise uniformly independent. Otherwise, consider $\ell = \min\{j \in [j_{w-1}, j_w] | j \notin P_w\}$. We know that $(G_i^\ell)_{i \in [n]}$ are at least $d+1-k-\ell$-wise uniformly independent. Thus, $(F_i'^{j_w})_{i \in [n]}$ is at least $\max\{d+1-k-\ell-|P_w \setminus P_{w-1}|, d+1-k-|P_w|-|P_w \setminus P_{w-1}|\}$- wise uniformly independent. Since this maximum is greater than or equal to $d+1-k-|P_w|-|P_w \setminus P_{w-1}| = d+1-k-|P_w|$, this finishes the induction.
Simulation of Group 3 starting from j_0: If $\exists j \in [j_0] : j \notin P_w$, then choose the element $F_i'^{j_0}$ uniformly random. This perfectly simulates the element due to the probing threshold of $d+1-k$ and the $d+1-k-|P_0|$-wise uniform independence of $(F_i'^{j_0})_{i \in [n]}$. Note that in case $\forall j \in [j_0] : j \in P_w$, but $\exists j \in [j_0] : j \notin I_R$, then by construction, this element corresponds to a probe in $\hat{G}_i^j$, where less than $\frac{d}{2}+1-k$ elements are probed in K^l, and the corresponding subgroups of Group 2 and Group 3 (Subgroup A, if $j \leq \lfloor \frac{n}{2} \rfloor$, and Subgroup B otherwise). In this case, $(\tilde{G}_i^l)_{i \in [n]}$ is not probed and re-randomizes the element G_i^l. Hence, choosing $F_i'^{j_0}$ uniformly random perfectly simulates the element, due to the threshold of $d+1-k$. Otherwise, if $\forall j \in [j_0] : j \in I_R$, then we calculate $(F_i'^{j_0})_{i \in [n]} = \sum_{j=0}^{j_0} (\hat{A}_{i,j} \cdot F_j + G_i^j)_{i \in [n]}$ directly, since $j \in I_R \to j \in I_F$ by construction. Hence G_i^j has already been simulated, F_j is known to the simulator, and $\hat{A}_{i,j}$ can be perfectly simulated, as it does not take any inputs and is calculated out of public parameters and elements that are chosen independently and uniformly at random (cf. [ABE+25]). Now, *we simulate the elements $j_w, w > 0$*, building upon j_{w-1} as follows. Recall that $(F_i'^{j_w})_{i \in [n]} = (F_i'^{j_{w-1}})_{i \in [n]} + \sum_{j=j_{w-1}}^{j_w} (\hat{A}_{i,j} \cdot F_j + G_i^j)_{i \in [n]}$. If $\forall j \in [j_{w-1}, j_w] : j \in I_R$, then $j \in I_F$ as well and the simulator already perfectly simulated all needed elements to calculate $F_i'^{j_w}$. Otherwise, $\exists j \in [j_{w-1}, j_w] : j \notin I_R$ and we choose $F_i'^{j_w}$ uniformly random. This perfectly simulates $F_i'^{j_w}$, as the elements $(F_i'^{j_w})_{i \in [n]}$ are at least $d+1-k-|P_w|$-wise independent, any $d+1-k-|P_w|$ tuple is indistinguishable from uniformly distributed and our probing threshold of $d+1-k$. (Note that in also simulates the element $F_i'^{j_w}$ perfectly, in case $\exists j \in [j_{w-1}, j_w] : j \notin I_R, j \in R_w$, since $\tilde{G}_i^j$ was not probed, as otherwise $j \in I_R$. Since there are not more than $\frac{d}{2}+1-k$ probes in Group 2 or Group 3 (Part A if $j < \lfloor \frac{n}{2} \rfloor$, part B if $\lfloor \frac{n}{2} \rfloor \leq j < n$), the element $G_i^l = \hat{G}_i^l \pm \tilde{G}_i^j$ is

at least 1-wise independent and indistinguishable from uniformly distributed. Altogether, this means that G_i^l is indistinguishable from uniformly distributed.)

Since $|I_F|$ obtains at most one element per probe, and all elements are perfectly simulated, this finishes the proof. □

Acknowledgements This work has been funded by the German Research Foundation (DFG) CRC 1119 CROSSING (project S7), by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE, and by the ERC Grant 101044770 (CRYPTOLAYER). Part of the funding comes by the European Union through the ERC projects 101096871 (acronym BRIDGE). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them. This work was supported by funding from the German Federal Ministry of Research, Technology and Space (BMFTR) under the projects Sec4IoMT and RILKOSAN.

References

[ABE+25] Paula Arnold, Sebastian Berndt, Thomas Eisenbarth, Sebastian Faust, Marc Gourjon, Elena Micheli, Maximilian Orlt, Pajam Pauls, Kathrin Wirschem, and Liang Zhao. Ohlala: Combined security against region faults attacks in constant protection order. Cryptology ePrint Archive, Paper 2025/1275, 2025.

[ABEO24] Paula Arnold, Sebastian Berndt, Thomas Eisenbarth, and Maximilian Orlt. Polynomial sharings on two secrets: Buy one, get one free. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(3):671–706, 2024.

[BBD+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015*, volume 9056 of *Lecture Notes in Computer Science*, pages 457–485. Springer, 2015.

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS*, pages 116–129. ACM, 2016.

[BCPZ16] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, volume 9813 of *Lecture Notes in Computer Science*, pages 23–39. Springer, 2016.

[BEF+23] Sebastian Berndt, Thomas Eisenbarth, Sebastian Faust, Marc Gourjon, Maximilian Orlt, and Okan Seker. Combined fault and leakage resilience: Composability, constructions and compiler. In *CRYPTO (3)*, volume 14083 of *Lecture Notes in Computer Science*, pages 377–409. Springer, 2023.

[Bel23] Nicolas Belleville. Accelerating exp-log based finite field multiplication. *IACR Cryptol. ePrint Arch.*, page 375, 2023.

[BFG+17] Josep Balasch, Sebastian Faust, Benedikt Gierlichs, Clara Paglialonga, and François-Xavier Standaert. Consolidating inner product masking. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 724–754. Springer, 2017.

[BFG+24] Sonia Belaïd, Jakob Feldtkeller, Tim Güneysu, Anna Guinet, Jan Richter-Brockmann, Matthieu Rivain, Pascal Sasdrich, and Abdul Rahman Taleb. Formal definition and verification for combined random fault and random probing security. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024*, volume 15490 of *Lecture Notes in Computer Science*, pages 167–200. Springer, 2024.

[BGG+21] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. Masking in fine-grained leakage models: Construction, implementation and verification. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):189–228, 2021.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *ACM STOC*, pages 1–10. ACM, 1988.

[BLK23] Yohannes B Bekele, Daniel B Limbrick, and John C Kelly. A survey of qemu-based fault injection tools & techniques for emulating physical faults. *IEEE Access*, 11:62662–62673, 2023.

[CB23] Songqiao Cui and Josep Balasch. Efficient software masking of AES through instruction set extensions. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2023, Antwerp, Belgium, April 17-19, 2023*, pages 1–6. IEEE, 2023.

[CDG+13] Jeremy Cooper, Elke Demulder, Gilbert Goodwill, Joshua Jaffe, Gary Kenworthy, Pankaj Rohatgi, et al. Test vector leakage assessment (TVLA) methodology in practice. In *International Cryptographic Module Conference*, volume 20, 2013.

[CDGT24] Claude Carlet, Abderrahman Daif, Sylvain Guilley, and Cédric Tavernier. Quasi-linear masking against SCA and FIA, with cost amortization. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(1):398–432, 2024.

[CPR12] Jean-Sébastien Coron, Emmanuel Prouff, and Thomas Roche. On the use of shamir's secret sharing against side-channel analysis. In *CARDIS*, volume 7771 of *Lecture Notes in Computer Science*, pages 77–90. Springer, 2012.

[DK09] Ivan Damgård and Marcel Keller. Secure multiparty AES (full paper). *IACR Cryptol. ePrint Arch.*, page 614, 2009.

[DN20] Siemen Dhooghe and Svetla Nikova. My gadget just cares for me - how NINA can prove security against combined attacks. In *CT-RSA*, volume 12006 of *Lecture Notes in Computer Science*, pages 35–55. Springer, 2020.

[FGM+23] Jakob Feldtkeller, Tim Güneysu, Thorben Moos, Jan Richter-Brockmann, Sayandeep Saha, Pascal Sasdrich, and François-Xavier Standaert. Combined private circuits - combined security refurbished. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS*, pages 990–1004. ACM, 2023.

[FOWZ25] Sebastian Faust, Max Orlt, Kathrin Wirschem, and Liang Zhao. All-you-can-compute: Packed secret sharing for combined resilience. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025.

[FRSG22] Jakob Feldtkeller, Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. CINI MINIS: domain isolation for fault and combined security. In *ACM CCS*, pages 1023–1036. ACM, 2022.

[FRSG24] Jakob Feldtkeller, Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. Combined threshold implementation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(4):307–334, 2024.

[FY92] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In S. Rao Kosaraju, Mike Fellows, Avi Wigderson, and John A. Ellis, editors, *ACM STOC*, pages 699–710. ACM, 1992.

[GJJ+11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, Pankaj Rohatgi, et al. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[GR17] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 567–597, 2017.

[GSF14] Vincent Grosso, François-Xavier Standaert, and Sebastian Faust. Masking vs. multiparty computation: how large is the gap for AES? *J. Cryptogr. Eng.*, 4(1):47–57, 2014.

[LD04] Chun-Pong Lai and Cunsheng Ding. Several generalizations of shamir's secret sharing scheme. *Int. J. Found. Comput. Sci.*, 15(2):445–458, 2004.

[MAN+19] Lauren De Meyer, Victor Arribas, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. M&m: Masks and macs against physical attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(1):25–50, 2019.

[PR11] Emmanuel Prouff and Thomas Roche. Higher-order glitches free implementation of the AES using secure multi-party computation protocols. In Bart Preneel and Tsuyoshi Takagi, editors, *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, volume 6917 of *Lecture Notes in Computer Science*, pages 63–78. Springer, 2011.

[Raw19] Edris Ahmad Rawashdeh. A simple method for finding the inverse matrix of Vandermonde matrix. *Matematički Vesnik*, 71(3):207–213, 2019.

[RFSG22] Jan Richter-Brockmann, Jakob Feldtkeller, Pascal Sasdrich, and Tim Güneysu. VERICA - verification of combined attacks automated formal verification of security against simultaneous information leakage and tampering. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):255–284, 2022.

[RMB+18] Oscar Reparaz, Lauren De Meyer, Begül Bilgin, Victor Arribas, Svetla Nikova, Ventzislav Nikov, and Nigel P. Smart. CAPA: the spirit of beaver against physical attacks. In *CRYPTO (1)*, volume 10991 of *Lecture Notes in Computer Science*, pages 121–151. Springer, 2018.

[RS60] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[SFES18] Okan Seker, Abraham Fernandez-Rubio, Thomas Eisenbarth, and Rainer Steinwandt. Extending glitch-free multiparty protocols to resist fault injection attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):394–430, 2018.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[SM16] Tobias Schneider and Amir Moradi. Leakage assessment methodology - extended version. *J. Cryptogr. Eng.*, 6(2):85–99, 2016.

[SS16] Peter Schwabe and Ko Stoffelen. All the AES you need on cortex-m3 and M4. In Roberto Avanzi and Howard M. Heys, editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, volume 10532 of *Lecture Notes in Computer Science*, pages 180–194. Springer, 2016.

[TNN24] Dilara Toprakhisar, Svetla Nikova, and Ventzislav Nikov. CAPABARA: A combined attack on CAPA. In *COSADE*, volume 14595 of *Lecture Notes in Computer Science*, pages 76–89. Springer, 2024.