IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–62. DOI:XXXXXXXX

All-You-Can-Compute: Packed Secret Sharing for Combined Resilience

Sebastian Faust, Maximilian Orlt, Kathrin Wirschem and Liang Zhao

Technical University of Darmstadt, Darmstadt, Germany,
{sebastian.faust,maximilian.orlt,kathrin.wirschem,liang.zhao}@tu-darmstadt.de

Abstract. Unprotected cryptographic implementations are vulnerable to implementation attacks, such as passive side-channel attacks and active fault injection attacks. Recently, countermeasures like polynomial masking and duplicated masking have been introduced to protect implementations against combined attacks that exploit leakage and faults simultaneously. While duplicated masking requires $\mathcal{O}(t \cdot e)$ shares to resist an adversary capable of probing t values and faulting e values, polynomial masking requires only $\mathcal{O}(t + e)$ shares, which is particularly beneficial for affine computation. At CHES'24, Arnold et al. showed how to further improve the efficiency of polynomial masking in the presence of combined attacks by embedding two secrets into one polynomial sharing. This essentially reduces the complexity of previous constructions by half. The authors also observed that using techniques from packed secret sharing (Grosso et al., CHES'13) cannot easily achieve combined resilience to encode an arbitrary number of secrets in one polynomial encoding. In this work, we resolve these challenges and show that it is possible to embed an arbitrary number of secrets in one encoding and propose gadgets that are secure against combined attacks. We present two constructions that are generic and significantly improve the computational and randomness complexity of existing compilers, such as the laOla compiler presented by Berndt et al. at CRYPTO'23 and its improvement by Arnold et al. For example, for an AES evaluation that protects against t probes and e faults, we improve the randomness complexity of the state-of-the-art construction when $t + e > 3$, leading to an improvement of up to a factor of 2.41.

Keywords: Polynomial Masking · Parallel Computation · Packed Secret Sharing · Leakage and Fault Resilience · Combined Attacks

1 Introduction

Since the late nineties, it is well-known that cryptographic implementations can successfully be broken by physical attacks. Such attacks exploit physical phenomena such as the power consumption [KJJ99], noise [GST14], or electromagnetic radiation [GMO01]; or allow an adversary to induce faults through clock glitches [DEG+18], voltage glitches [ZDCT13], electromagnetic pulses [DDRT12, DLM19], and laser beams [BS97, SA02]. While side-channel attacks are passive and enable the adversary to learn sensitive intermediate information about computations, fault attacks actively manipulate the inner workings of cryptographic implementations.

A widely used countermeasure against side-channel attacks is *masking*. A masking scheme uses an encoding that splits a sensitive value v into a vector $(F_0, \ldots, F_{n-1})$ of n values, called shares, such that a specific number of shares is needed to reconstruct v. Two of the most commonly used encodings are additive masking and polynomial masking. *Additive masking* generates $n - 1$ shares $F_0, \ldots, F_{n-2}$ uniformly at random from a finite field $\mathbb{F}$, and then sets F_{n-1} such that $v = \sum_{i=0}^{n-1} F_i$. Alternatively, *polynomial masking* constructs a polynomial $p(X) = v + a_1X^1 + \cdots + a_dX^d$, such that the coefficients a_i are chosen uniformly at random, and the constant term represents the secret. Then, the shares are defined as function values $F_i = p(x_i)$ with predetermined non-zero pairwise distinct support points $x_0, \ldots, x_{n-1}$. Intuitively, masked implementations compute on such encodings such that the leakage depends only on a limited number of shares. Due to the properties of the encoding, this implies that the leakage does not contain any sensitive information about the shared secrets.

Generally, additive masking is highly efficient in terms of randomness and computational complexity while protecting against a passive adversary measuring up to t $(< n)$ values in the masked implementation. However, if no further countermeasures are used, the scheme becomes vulnerable to active fault attacks [GIP+15]. For instance, an adversary can insert a single fault by adding ξ to one share F_i that leads to a valid encoding corresponding to the secret $v + \xi$. Even worse, the advantage of an attacker launching both a side-channel and fault attack can far exceed the sum of the advantages of each individual attack. The core issue is that faults can not only alter the input-output behavior of a cryptographic implementation but also compromise the effectiveness of leakage countermeasures. This has, for example, been demonstrated in studies by Berndt et al. [BEF+23] and Feldtkeller et al. [FRSG22]. Hence, it is necessary to design countermeasures that protect cryptographic implementations against *both* side-channel and fault attacks and achieve so-called *combined resilience*.

Countermeasures for Combined Resilience. While countermeasures against standalone side-channel attacks and fault attacks are well-established, protection against attacks that exploit both leakage and faults simultaneously has only recently been explored. One approach to protect against combined attacks is *duplication* of t-th order masked computation, where each sharing (and computation on it) is copied (and executed) $e + 1$ times. This allows the detection of any e faults as at least one share stays unaffected by the faults. However, achieving security against t probes and e faults requires $\mathcal{O}(t \cdot e)$ shares[1] [DN20, RFSG22], which is particularly costly for masked affine computation. Moreover, the duplication technique makes the masked computation susceptible to more powerful adversaries, e.g., exploiting horizontal attacks [BCPZ16, BEF+23].

A recent direction to address these shortcomings is to use polynomial masking [SFES18, BEF+23, ABEO24]. The idea is that if a fault is added to a share of a polynomial masked secret, the degree of the recovered polynomial increases, making the fault detectable. Therefore, one can show that polynomial masking with $\mathcal{O}(t + e)$ shares achieves the same security as additive masking with $\mathcal{O}(t \cdot e)$ shares [GIP+15, SFES18, DN20, GPS21, FRSG22, FGM+23]. While polynomial masking offers strong combined resilience, masking computation is more costly than using standard additive masking. Recently, Arnold et al. [ABEO24] showed how to improve the efficiency of polynomial masking with combined resilience by sharing two secrets with one polynomial. Technically, this is achieved by placing the secrets in the highest and lowest monomials of the polynomial. Arnold et al. [ABEO24] showed that such an approach is particularly useful for parallel computation, where the goal is to compute multiple circuits with the same structure but on different inputs (e.g., think of the AES computing the 16 S-Boxes in parallel).

Packed Secret Sharing. In the context of secure multi-party computation (MPC), Franklin and Yung [FY92] introduced the concept of packed secret sharing to securely evaluate functions that compute on many inputs. In packed secret sharing, for a random polynomial f one can store secrets at multiple support points $f(u_i)$, for u_i pairwise distinct. Damgård et al. [DIK10] showed how to use packed secret sharing for securely computing

[1]Please note that [DN20, RFSG22] use $e + 1$ exact copies of $t + 1$ shares which leads to a total of $(t + 1) \cdot (e + 1)$ shares, and all the subsequent operations are executed on these $(t + 1) \cdot (e + 1)$ shares.

arbitrary functions among n parties with quasilinear complexity. This approach inspired Grosso et al. [GSF14], who showed how to use packed-secret sharing to protect against side-channel leakage. To obtain an efficient scheme, the authors rely on a "trick" proposed by Damgård et al. [DIK10], where a "masked version" of the shared secrets is reconstructed and then re-shared. While this approach works well in the presence of side-channel leakage, an adversary can fault the temporarily reconstructed values, thereby inducing undetectable faults[2]. The main contribution of this work is to present new techniques to extend packed secret sharing based polynomial masking such that it can resist combined attacks. We provide more details on our contribution below.

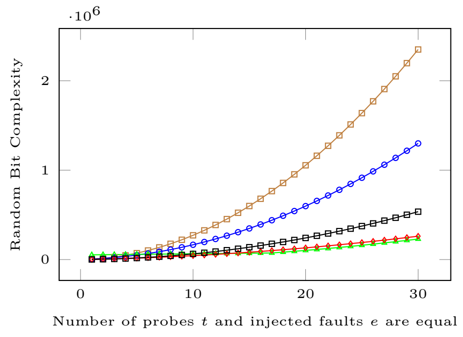
1.1 Contribution

Our overall objective is to develop compilers for parallel computation that remain secure against combined attacks, where an adversary can measure t probes and simultaneously insert e faults. To this end, we extend the work of Grosso et al. [GSF14] and demonstrate that refined linear modifications enable the use of packed secret sharing in already existing combined-resilient circuits. Further, we developed new gadgets that are particularly useful for packed secret sharing. As case studies of our main techniques, we present a detailed analysis of a linear compiler, an adaption of the well-known BGW compiler [BGW88], as well as the recently proposed LaOla compiler [BEF$^+$23]. Additionally, we introduce a novel modified Frobenius exponentiation gadget that efficiently (share-wise) computes the exponentiation of secrets on packed secret shares. Figure 1 illustrates the effectiveness of our approaches through AES encryption. The concrete numbers are given in Table 1 for $t, e \leq 8$, and a detailed analysis of these values is provided in Section G. For a better comparison with the duplicated Boolean masking [FRSG22, FGM$^+$23] we give the efficiency of all constructions in terms of Boolean circuit complexity and random bit complexity.

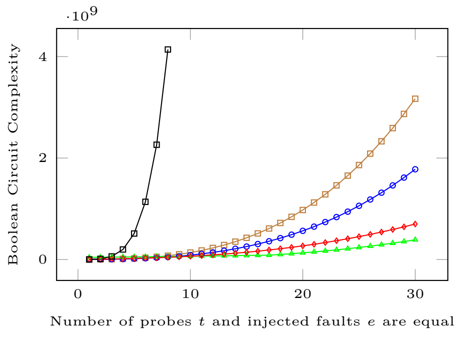
In Figure 1, we depict the random bit complexity (cf. Fig. 1a) and the Boolean circuit complexity (cf. Fig. 1b) for one round of AES-128 with respect to 16 inputs, using our BGW-based PSS and LaOla-based PSS compiler, the LaOla compiler of [BEF$^+$23], the double-sharing compiler [ABEO24], and the duplicated Boolean masking approach of [FRSG22, FGM$^+$23]. Note that the computational and randomness complexities presented in Table 1 and Figure 1 do not include the final error detection mechanism for any of the example constructions listed. Due to our fault-robustness property, all our constructions require fault detection gadgets solely immediately before decoding the final outputs. The complexity impact of the final error detection on the presented values is minimal. For instance, in the BGW-based PSS approach with $t = e = 3$ (or $t = e = 8$), the computational complexity increases only slightly from $10.431 \cdot 10^6$ to $10.717 \cdot 10^6$ (or $42.785 \cdot 10^6$ to $45.720 \cdot 10^6$), while still outperforming the duplicated Boolean Masking approach in [FRSG22, FGM$^+$23]. As the duplication requires error detection after each gadget and our construction only for the final outputs, this effect becomes even more relevant with larger circuits. A formal discussion is provided in Section 6.

We emphasize that this work operates in a finite field, which is chosen to be $\mathbb{F}_{2^8}$ for this example. Moreover, this work can resist t field element probes and detect e field element faults. In contrast, duplicate Boolean masking [FRSG22, FGM$^+$23] operates in the binary field $\mathbb{F}_2$ and can resist t glitch-extended bit-probes and correct e bit-faults. A crucial error correction component for achieving combined security in the duplicated Boolean masking [FRSG22, FGM$^+$23] is a sorting-based majority function. As no concrete sorting algorithm was chosen by [FRSG22, FGM$^+$23], we choose the insertion sort, which is known to be efficient for a small number of faults e (cf. [CLRS22, YYG11]). For further details, we refer to Supplementary Material G.1.

[2]Indeed, even one fault is sufficient to break the scheme of Grosso et al. [GSF14].



(a) Random Bit Complexity



(b) Boolean Circuit Complexity

Figure 1: Random bit complexity (a) and Boolean circuit complexity (b) for one AES round of our *BGW-based PSS* (◇) and *LaOla-based PSS* compiler (△), the original *LaOla* [BEF+23] (□) and *double-sharing* compiler [ABEO24] (○), and the *Duplicated Boolean Masking* [FRSG22, FGM+23] (□). The security parameters are t probes and e faults with $e = t$.

Table 1: Random bit complexity ($\times 10^4$) and Boolean circuit complexity ($\times 10^6$) for one AES round of our *BGW-based PSS* and *LaOla-based PSS* compiler, the original *LaOla* [BEF+23] and the *double-sharing* compiler [ABEO24], and the *Duplicated Boolean masking* [FRSG22, FGM+23] as in Figure 1. The security parameters are t probes and e faults with $e = t$ and $e, t \in \{1, \dots, 8\}$. The best complexity is marked in bold.

$t = e$	LaOla [BEF+23]		Double-Sharing [ABEO24]		LaOla-based PSS (this work)		BGW-based PSS (this work)		Dup. Boolean Mask [FRSG22, FGM+23]	
	Rand	Comp	Rand	Comp	Rand	Comp	Rand	Comp	Rand	Comp
1	0.410	**0.804**	0.589	0.941	5.304	41.981	0.245	4.018	**0.115**	1.260
2	1.331	2.634	1.2864	**2.499**	5.433	44.276	0.533	6.841	**0.346**	12.792
3	2.764	6.262	2.253	**5.241**	5.562	46.632	0.864	10.431	**0.691**	60.716
4	4.710	12.340	3.488	**9.508**	5.690	49.048	1.238	14.871	**1.152**	197.630
5	7.168	21.521	4.992	**15.643**	5.819	51.525	**1.656**	20.244	1.728	512.024
6	10.137	34.458	6.765	**23.988**	5.948	54.062	**2.117**	26.632	2.419	1137.670
7	13.619	51.804	8.806	34.888	6.077	56.660	**2.621**	**34.118**	3.226	2263.032
8	17.613	74.211	11.117	48.683	6.206	59.319	**3.168**	**42.785**	4.147	4140.661

(1) Linear Compiler. The most challenging aspect of our compiler is that packed secret sharing involves multiple secrets at pairwise distinct support points, and some computation requires securely permuting the secrets with these support points.

For example, a simple share-wise addition of two masked secrets $s_0 = f(u_0), s_1 = f(u_1), \dots, s_d = f(u_d)$ and $s'_0 = g(u_0), s'_1 = g(u_1), \dots, s'_d = g(u_d)$ using support points $u_0, u_1, \dots, u_d$ easily generates a new masking with secrets $h(u_0), h(u_1), \dots, h(u_d)$ such that it holds $h(u_i) = s_i + s'_i$, (cf. Sec. 3). However, the complexity increases when permuted operations are required, such as $h(u_i) = s_i + s'_{\pi(i)}$ for an arbitrary permutation $\pi : \{0, 1, \dots, d\} \to \{0, 1, \dots, d\}$. For this reason, we present a new combined resilient gadget that efficiently implements various operations, such as permutations for packed secret sharing in Section 3.4.

(2) BGW Approach. We address the issues highlighted in the work of Grosso et al. [GSF14]. Specifically, Section 4 demonstrates that parallel computation with packed secret sharing can be implemented without the need for intermediate reconstruction that is vulnerable to fault attacks. To avoid such vulnerabilities, we build a *BGW-based PSS compiler* by extending Franklin and Yung's work [FY92], which illustrates how the seminal BGW construction [BGW88] can be used with packed secret sharing for combined resilience.

In particular, we adapt these constructions such that we can rigorously prove their security against combined leakage and fault attacks. Further, we reduced the randomness complexity by incorporating optimizations proposed by Gennaro et al. [GRR98]. Considering one AES round, our *BGW-based PSS compiler* improves the recently optimized version of Arnold et al. [ABEO24] from $t + e \geq 2$ considering the randomness complexity and from $t + e > 13$ regarding the computational complexity. In general, our *BGW-based PSS compiler* improves the version of Arnold et al. [ABEO24] by a factor of 4.31 (resp. 1.87) regarding the randomness (resp. computational) complexity of one AES round for $t + e > 30$.

(3) LaOla Approach. Additionally, we build the *LaOla-based PSS compiler* by applying similar modifications to the recently proposed LaOla compiler and formally prove the combined resilience property of our compiler (cf. Sec. 4). While our *BGW-based PSS compiler* requires $2d + e + 1$ shares for technical reasons, our *LaOla-based PSS compiler* achieves the same combined resilience using only $d+e+1$ shares. Although this improvement entails slightly more complex constructions for nonlinear computations, it often results in greater efficiency for our methods. Notably, Figure 1 illustrates that this approach is more efficient for AES with higher security orders. For example, *our LaOla-based PSS compiler* improves our *BGW-based PSS compiler* approach by a factor of 1.95 regarding the computational complexity.

(4) Frobenius Optimization. The Frobenius property enables efficient exponentiation of masked secrets. Concretely, we have developed a share-wise operating gadget that allows for the exponentiation of all secrets within a packed secret sharing (cf. Sec. 5). Additionally, we provide a formal proof of its combined security. For example, this gadget improves the efficiency of an AES round (with security parameters $t + e > 30$) by achieving 7 squaring operations using only 7 share-wise multiplication, which improves the randomness complexity by a factor of 4.31. We emphasize that this improvement is also significant for low-order security.

1.2 Related Work

Recently, two different approaches to achieve combined resilience have been proposed: duplicated (additive) masking [DN20, FRSG22, RFSG22, FGM$^+$23] and polynomial masking [BEF$^+$23, ABEO24]. The duplicated approach can be viewed as a composition of leakage-resilient and fault-resilient encoding. It uses additive masking as a leakage-resilient encoding and then duplicates, i.e., copies, each share $\mathcal{O}(e)$ times for fault resilience. While this technique allows for relatively fast and secure bit operations, it requires at least $\mathcal{O}(t \cdot e)$ shares and lacks the properties necessary to encode multiple secrets together, as we do with polynomial masking using packed secret sharing [FY92]. However, the approach of [FRSG22, FGM$^+$23] remains very efficient for non-parallelized computations and computations in binary fields (cf. Sec. 7.3). Moreover, while Boolean masking with error detection [DN20] has shown some vulnerability to Statistical Ineffective Fault Attacks (SIFA) (cf. [RFSG22]), recent works [FRSG22, FGM$^+$23] propose combining Boolean masking with error correction to address this issue. Although this approach introduces a higher computational cost compared to error detection, it offers enhanced protection against SIFA. Our security analyses for polynomial masking circumvent this issue as we prove fault robustness (cf. Def. 3). The robustness property is specifically designed to address ineffective faults, such as those in SIFA attacks, by ensuring that any fault results in corrupted outputs. Additionally, we demonstrate that with (weak) fault-resilient SNI [BEF$^+$23], the effects of each fault can be simulated without the underlying secrets.

Following the ideas of Arnold et al. [ABEO24], we consider packed secret sharing to mask multiple secrets using only one polynomial. As mentioned in their work, they encountered issues raised by Grosso et al. [GSF14], making it challenging to protect their leakage-resilient computation on packed secret sharing against combined attacks (cf. Sec. 4). Consequently, they proposed an elegant compromise solution: instead of encoding arbitrarily many secrets together, they suggested methods for jointly encoding two secrets. The advantage of their modified encoding is that it still allows for computation that is not only standalone leakage-resilient but also combined-resilient. Specifically, they adapted the LaOla compiler presented by Berndt et al. [BEF$^+$23] to compute two identical circuits with different inputs in parallel at the cost of one. Hence, their approach almost halved the complexity of a protected AES evaluation (cf. Fig. 1). Building on the work of Arnold et al., our research resolves all the issues raised by Grosso et al. and presents the first compiler that is secure against combined attacks while using packed secret sharing. As we are now able to compute an arbitrary number of secrets in parallel, this significantly improves the complexity of the protected computation (see Fig. 1).

In addition to a series of prior works [DN20, FRSG22, RFSG22, FGM$^+$23, BEF$^+$23, ABEO24] which do consider combined resilience, the recent work of Carlet et al. [CDGT24] presented efficient constructions considering parallel computations, however [CDGT24] is only provably secure against standalone leakage attacks and standalone fault attacks. Specifically, since fault attacks can significantly compromise the effectiveness of leakage countermeasures, these constructions are not immediately provably secure against combined attacks (e.g., consider an adversary that faults some computation and observes the effect via leakage). One notable example is the work of Carlet et al. [CDGT24], who propose a masking scheme with quasi-linear complexity based on packed secret sharing and fast Fourier transform (FFT). The analysis only considers standalone attacks against leakage and fault attacks. It remains an interesting open question if the techniques (most importantly the FFT) can be used in the context of combined resilience. The main challenge is that when considering combined attacks where the adversary can simultaneously measure probes and introduce faults, we require a pre-calculated matrix for error detection (cf. Sec. 4), preventing us from using the FFT optimization. Further, their multiplication of encodings is split into two separate calculation steps. Carlet et al. pointed out in [CDGT24, Sec. 6.4 on p. 420] that two simultaneous faults could be introduced in these two steps, and it remains an open question whether these faults can be effectively detected. Table 2 and Table 3 numerically compare the constructions presented in our work with other combined resilient constructions [DN20, FRSG22, FGM$^+$23, BEF$^+$23, ABEO24]. Especially, the work [ABEO24] supports circuit evaluation in parallel for two inputs. Although [CDGT24] is not provably secure in our stronger combined model, it allows for parallel evaluation of k different inputs, similar to our work on packed secret sharing, and hence for completeness we include it in our comparison.

2 Preliminaries and Notation

Notation. By $n \in \mathbb{N}_{\geq 2}$, we denote the number of shares on which we operate. Moreover, let $\mathbb{F}$ denote some finite field with at least $n+1$ elements. We primarily work with polynomials of degree d, constructed using secrets denoted as $v_i \in \mathbb{F}$ and, when necessary,

[3]To the best of our knowledge, the only existing gadget constructions requiring linear number of shares and proved to be combined secure against adaptive adversaries are those from [BEF$^+$23, ABEO24], where the *adaptive* adversaries can use information from previous leakage to insert faults. This work adapted the gadget constructions from [BEF$^+$23, ABEO24] while maintaining the combined security property.

[4]It was shown by Berndt et al. [BEF$^+$23] that the refresh gadget from [SFES18] is not SNI.

[5]Note that Carlet et al. [CDGT24] focused on the security against *non-adaptive* adversaries that execute probing and fault injecting attacks *independently*, which is slightly different from the combined security model considered in this work and [BEF$^+$23, ABEO24]. Importantly, the multiplication and refresh gadget from [CDGT24] are not SNI.

Table 2: A comparison of the combined secure compilers[3] with regard to the masking scheme, the number of *parallelizable* inputs, the number of shares, and security properties. We adjust the parameters such that all the compilers can tolerate t probes and detect e injected faults. ✓ are protocols not susceptible to the horizontal attacks [BCPZ16] or secure against *adaptive* adversaries, ✗ are vulnerable solutions, and – are the solutions that consider non-adaptive adversaries.

Masking Scheme	Compiler	# *Parallelizable* Inputs	# Shares	*Adaptive* Adversary	Horizontal Attacks
Duplicated Boolean Masking	[DN20]	1	$(t+1)\cdot(e+1)$	–	✗
	[FRSG22, FGM+23]	1	$(t+1)\cdot(2e+1)$	–	✗
Polynomial Masking	[SFES18]	1	$2t+e+1$	–	✗
	[BEF+23]	1	$t+e+1$	✓	✓
	[ABEO24]	2	$t+e+2$	✓	✓
	[CDGT24]*	k	$2(t+k)+e-1$	–	✗
	LaOla-based PSS (this work)	$\boldsymbol{k}$	linear operation: $\boldsymbol{t+k+e}$ non-linear operation: $\boldsymbol{t+2k+e}$	✓	✓
	BGW-based PSS (this work)	$\boldsymbol{k}$	$\boldsymbol{2t+2k+e-1}$	✓	✓

Table 3: A comparison of the computational and randomness complexity of the addition, multiplication, and refresh gadgets when given k inputs, with regard to probing security t, detectable injected faults e, and $n = t + e + 1$. SNI gadgets with the best asymptotic complexity are marked in **bold**. We only keep the highest-order terms.

Compiler	Addition	Multiplication		Refresh	
	Comp	Rand	Comp	Rand	Comp
[DN20]	$\mathcal{O}(kn^2)$	$\mathcal{O}(kn^2)$	$\mathcal{O}(kn^3)$	$\mathcal{O}(kn^2)$	$\mathcal{O}(kn^3)$
[FRSG22, FGM+23]	$\mathcal{O}(kn^2)$	$\mathcal{O}(kn^2)$	$\mathcal{O}(kn^3)$	$\mathcal{O}(kn^2)$	$\mathcal{O}(kn^3)$
[SFES18]*[4]	$\mathcal{O}(kn)$	$\mathcal{O}(kn^2)$	$\mathcal{O}(kn^3)$	$\mathcal{O}(kt)$	$\mathcal{O}(kt^2+kte+kt)$
[BEF+23]	kn	$k(3t^2+2te)$	$k(6t^3+10t^2e+4te^2)$	$k(t^2+t)$	$k(2t^3+2t^2e)$
[ABEO24]	$\frac{1}{2}k(n+1)$	$\frac{1}{2}k(3t^2+2te)$	$\frac{1}{2}k(6t^3+10t^2e+4te^2)$	$\frac{1}{2}k(t^2+t)$	$\frac{1}{2}k(2t^3+2t^2e)$
[CDGT24]*[5]	kt	t	$2k^2+2kt$	kt	$kt+k^2$
LaOla-based PSS (this work)	$\boldsymbol{n+k-1}$	$\boldsymbol{3t^2+2te+tk-ke-k^2}$	$\boldsymbol{6t^3+10t^2e+4te^2+4ekt+8kt^2-2k^3-4ek^2-2e^2k}$	$\boldsymbol{t^2}$	$\boldsymbol{2t^3+2t^2e+2kt^2}$
BGW-based PSS (this work)	$n+2k+t-2$	$\boldsymbol{2t^2+te+2tk}$	$\boldsymbol{8t^3+8t^2e+2te^2+8ekt+16kt^2+8k^2t}$	$\boldsymbol{t^2}$	$4t^3+2t^2e+4kt^2$

random elements $r_i \leftarrow_\$ \mathbb{F}$, for $i \in [d+1]$. If $\boldsymbol{v} \coloneqq (v_0, \ldots, v_d) \in \mathbb{F}^{d+1}$ is a vector and $I \subseteq \{0, \ldots, d\}$ is a subset of the indices, we define $\boldsymbol{v}_I \coloneqq (v_i)_{i \in I}$. We use $*$ to denote the Hadamard (or element-wise) product between two vectors or matrices. We denote the set $\{0, \ldots, n-1\}$ by $[n]$. The Hamming weight $\mathsf{weight}(\boldsymbol{v})$ of a vector $\boldsymbol{v} \in \mathbb{F}^{d+1}$ is the number of non-zero entries of $\boldsymbol{v}$. For a matrix $A \in \mathbb{F}^{m \times n}$, $A_{i,j}$ denotes the j-th element in the i-th row. We refer to the identity matrix as $\mathbf{I}$. By writing $(y_0, \ldots, y_{n-1}) \leftarrow_\$ \mathbb{F}^n$, we mean that all y_i with $i \in [n]$ are chosen independently and uniformly at random from the field $\mathbb{F}$. Moreover, for values $x_0, \ldots, x_{n-1} \in \mathbb{F}$, we denote the Vandermonde matrix that has $d+1$ rows and n columns (or $(d+1) \times n$ matrix for simplicity) by

$$\mathsf{Van}_{(d+1)\times n}(x_0, \ldots, x_{n-1}) = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ x_0 & x_1 & x_2 & \ldots & x_{n-1} \\ x_0^2 & x_1^2 & x_2^2 & \ldots & x_{n-1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_0^d & x_1^d & x_2^d & \ldots & x_{n-1}^d \end{pmatrix}_{(d+1)\times n},$$

and its inverse by $\mathsf{Van}^{-1}_{(d+1)\times n}(x_0, \ldots, x_{n-1})$. In Section 2.2, we can use $\mathsf{Van}_{(d+1)\times n}(x_0, \ldots, x_{n-1})$ to generate n shares for a *degree-d* polynomial.

Let $\mathcal{D}$ and $\mathcal{D}'$ be two probability distributions. We write $\mathcal{D} \equiv \mathcal{D}'$ to denote the equivalence of the distributions. Let $X_0, X_1, \ldots, X_{n-1} \leftarrow_\$ \mathbb{F}$ be a series of random variables. We say that $X_0, X_1, \ldots, X_{n-1} \in \mathbb{F}$ are *k-wise independent* if $\Pr\left[\bigwedge_{i \in I} X_i = x_i\right] = \prod_{i \in I} \Pr[X_i = x_i]$ for all $I \subseteq [n]$ with $|I| \leq k$. Moreover, a probability distribution $\mathcal{D}$ is *perfectly simulatable* from a set S, if there exists a simulator SIM such that the output of $\mathsf{SIM}(S)$ has the same distribution as $\mathcal{D}$. More specifically, for any possible x in the domain of $\mathcal{D}$, it holds that $\Pr[\mathsf{SIM}(S) = x] = \Pr[\mathcal{D} = x]$ which we denote by $\mathsf{SIM}(S) \equiv \mathcal{D}$.

2.1 Attacker Models

In this work, we consider two types of adversaries: passive and active adversaries. The *passive* adversary learns the input-output behavior of the cryptographic implementation as well as some intermediate values. The *active* adversary can inject faults into the internal values (including the input and output values of the gadgets) of the implementation, which can lead to unexpected computations and expose secret information. Furthermore, we consider *adaptive* adversaries that can perform combined attacks and thus *simultaneously* probe and fault the intermediate values of the implementation using information from previous leakage.

Circuits. Computations are represented as circuit C with respect to a field $\mathbb{F}$, i.e., we consider a directed acyclic graph G where each node is labeled as input gate, output gate, random gate, addition gate, multiplication gate, or constant gate. Moreover, when considering a field $\mathbb{F}$ with characteristic $p \neq 2$, we assume the existence of a power-of-p gate that performs $x \mapsto x^p$. Picturing a circuit C as a directed acyclic graph, we start a computation by assigning the inputs to the input gates. Once all the parents of a gate have a value, we continue by calculating the value of the gate by applying the underlying operation (e.g., multiplication or addition) to these values. We proceed gate by gate. Note that random gates do not have parents and output a uniformly random element in $\mathbb{F}$.

A circuit can output the symbol $\perp \notin \mathbb{F}$ to denote that a computation aborts. We denote the output distribution of a circuit C with inputs $x_1, x_2, \ldots$ by $\mathsf{C}(x_1, x_2, \ldots)$. For a circuit C with inputs $x_1, x_2, \ldots$ and (fixed) randomness R, the output is given by $\mathsf{C}^R(x_1, x_2, \ldots)$.

Compiler. A common way to prevent the above attacks is to use a *compiler* that can transform a circuit C that may be vulnerable to attacks into a circuit C' that can resist such attacks. Note that the transformed circuits C' are called fault- and leakage-resilient circuits. When designing such a circuit C', the main challenge is to prevent the leakage of sensitive information through the intermediate values produced by C'. More precisely, the transformed circuit C' computes on encoded values that can be generated using Boolean secret sharing, polynomial secret sharing, and other similar techniques. The specific choice of the encoding schemes mainly depends on the types of attacks we aim to resist. Besides, each gate in C is transformed into a *gadget* in C'. A gadget is essentially a sub-circuit with the same functionality as the original gate in C, except that the inputs and outputs are both encoded values. In concrete circuit constructions, we are mainly interested in how to generate gadgets for addition and multiplication. In this work, we focus on constructing gadgets using packed secret sharing (PSS) [FY92], which is very efficient and beneficial in the scenario of parallel computations.

Passive Attacks. A passive *t-probing* adversary $\mathcal{A}$ can run the circuit C and arbitrarily execute standard glitch-free probe on field elements [6] for up to $|I| \leq t$ intermediate wires $(w_i)_{i \in I}$ of C. Let $b \leftarrow_{\$} \{0,1\}$ and let the adversary $\mathcal{A}$ choose two inputs, x_0 and x_1. We say that a circuit C with input x_b is *perfectly secure* against a *t-probing* adversary $\mathcal{A}$ if the probability that the adversary $\mathcal{A}$ can distinguish whether the input x_b is x_0 or x_1, equals to $\frac{1}{2}$.

[6] Note that we consider the standard glitch-free t-probing model on field elements instead of the glitch-extended t-probing model on bits presented in [FRSG22, FGM+23].

Non-interference (NI) [BBD+15, BBD+16] is a useful notion to ensure perfect security. Intuitively, NI guarantees that all information that an adversary $\mathcal{A}$ can obtain by t probes of arbitrary intermediate values can also be obtained by t probes on the inputs. In order to guarantee security, we need to choose a secret sharing scheme on the secret inputs, such that any subset of t shares does not reveal any information about the secret inputs.

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 $t_{\text{int}} + |\mathcal{O}| \le t$, there exists a subset of input indices I with $|I| \le t_{\text{int}} + |\mathcal{O}|$ such that the distribution of the t_{int} intermediate variables and the output variables in $\mathcal{O}$ is perfectly simulatable with inputs with indices from I.

As NI gadgets do not support *composability* [BBD+16], meaning that the composition of two NI gadgets is not necessarily NI anymore, we consider a stronger version of non-interference called SNI.

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 $t_{\text{int}} + |\mathcal{O}| \le t$, there exists a subset of input indices I with $|I| \le t_{\text{int}}$ 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.

Note that the difference between NI and SNI is that the set of probes is separated into internal and output probes. This means that the SNI property requires that the number of input shares on which the distribution of a given position set can depend is determined by the number of internal positions present in that state. This separation enables the composability of SNI gadgets. A detailed discussion is presented in [BBD+16].

Active Attacks. We mainly consider two types of fault sets: the set of wire independent faults $\mathcal{F}^{\mathsf{ind}}$ where the wire with value x can be arbitrarily but independently faulted to a value v, and the set of additive faults $\mathcal{F}^{+} := \{\xi : \xi(x) = x + a \text{ for all } a \in \mathbb{F}\}$. It is easy to see that $\mathcal{F}^{+} \subset \mathcal{F}^{\mathsf{ind}}$. More formally, let $w_0, \dots w_{W-1}$ denote the wire indices of a circuit C^R. A fault attack T is defined as a tuple of functions $T = (\xi_0, \dots, \xi_{W-1})$, where each function $\xi_i \in \mathcal{F}, i \in [W]$ describes how wire w_i is faulted. During a computation, an active adversary $\mathcal{A}$ with access to a set of faults $\mathcal{F}$ can change the wire value from $x_i \in \mathbb{F}$ to $\xi(x_i) := v_i$ for $\xi \in \mathcal{F}$.

The faulted circuit C^R with inputs $(x_1, \dots)$ is denoted by $T\left[\mathsf{C}^R(x_1, \dots)\right]$ and the number of faulted wires is denoted by $|T| = e$. We refer to the outputs of the unfaulted circuit C by $y \leftarrow \mathsf{C}^R(x_1, \dots)$ and the output of the faulted circuit $\mathsf{C}^{R'}(x_1, \dots)$ by $y' \leftarrow T\left[\mathsf{C}^{R'}(x_1, \dots)\right]$. A circuit C^R is called *ϵ-secure* against e-fault adversaries if $\Pr[y' \in \{y, \perp\}] \ge 1 - \epsilon$ for all $\mathcal{A}$. Intuitively, ϵ-security means the injected faults can be corrected or identified with probability $p \ge 1 - \epsilon$. Seker et al. [SFES18] proposed a technique called *infective computation*, which guarantees that the injected error is propagated during computation and the result is an error state of the circuit C. This notion was enhanced as *fault-robustness* by Berndt et al. [BEF+23]. Fault-robustness guarantees that the injected faults lead to a detectable faulted output. So far, this notion has been mainly used for Shamir's secret sharing [Sha79]. In this work, we extend it to packed secret sharing [FY92].

Note that we call a value $\boldsymbol{y} \in \mathbb{F}^n$ *valid* encoding for a specific encoding $\mathrm{Enc} : \mathbb{F} \to \mathbb{F}^n$, if there exists a number $x \in \mathbb{F}$ and some randomness r such that $\mathrm{Enc}(x; r) = \boldsymbol{y}$. Let s_{input} denote the number of faults that are injected into the gadget inputs, and s_{int} indicate the faults that are injected during the computation of the gadget. Intuitively, fault-robustness (cf. Def. 3) guarantees that for a total of $|T| = s_{\text{input}} + s_{\text{int}}$ faults, there exists $t_1 + t_2 \le |T|$ such that the *effect* on the gadget's outputs is that either $\le t_1$ outputs are deterministically affected or the outputs contain *abnormal/error* high-order coefficients of $x^{d+1}, \dots, x^{n-t_2}$ which can be detected.

Definition 3 (e-fault-robust [BEF+23, Def. 3], [ABEO24, Def. 3][7])**.** A gadget G with one output sharing and two input sharings (with respect to the polynomial sharing $[\![\cdot]\!]$) is *e-fault-robust* with respect to a fault set $\mathcal{F}$, if for any valid input sharings $(x_0, \dots, x_{n-1}) \in [\![x]\!]$ and $(x'_0, \dots, x'_{n-1}) \in [\![x']\!]$, the output $(y_0, \dots, y_{n-1}) \leftarrow G\left((x_0, \dots, x_{n-1}), (x'_0, \dots, x'_{n-1})\right)$ is also valid. Further it holds for any fault vectors $(v_0, \dots, v_{n-1}) \in [\![v]\!]$, $(v'_0, \dots, v'_{n-1}) \in [\![v']\!]$ and any $T \in \mathcal{A}(\mathcal{F})$ with $|T| \leq e$ and $(y_i + w_i + w'_i)_{i \in [n]} \leftarrow T[G]\left((x_i + v_i)_{i \in [n]}, (x'_i + v'_i)_{i \in [n]}\right)$, that there exists numbers t_1 and t_2 with $t_1 + t_2 \leq |T|$ such that

- $\mathsf{weight}((w_0, \dots, w_{n-1})) \in [0, t_1] \cup [\mathsf{weight}(\boldsymbol{v} + \boldsymbol{v}') - t_1, \mathsf{weight}(\boldsymbol{v} + \boldsymbol{v}') + t_1]$ where $\boldsymbol{v} = (v_0, \dots, v_{n-1})$ and $\boldsymbol{v}' = (v'_0, \dots, v'_{n-1})$,
- and $\boldsymbol{w}' = (w'_0, \dots, w'_{n-1})$ is the zero vector or produced by the following random experiment: A polynomial $p_{\boldsymbol{w}'} \in \mathbb{F}[x]$ is chosen such that the coefficients of $x^{d+1}, x^{d+2}, \dots, x^{n-t_2}$ are drawn uniformly at random from $\mathbb{F}$. Then, $w'_i = p_{\boldsymbol{w}'}(\alpha_i)$ for some pairwise different points $\alpha_i \in \mathbb{F} \setminus \{0\}$.

In the subsequent proof about combined resilience, we mainly consider *additive* faults launched by *non-adaptive* adversaries. However, under the *mild* conditions of fault-invariance (cf. Def. 7) and frSNI (cf. Def. 4) as required in [BEF+23, Corollary 2], we can transform our gadget that is secure against additive faults $\mathcal{F}^+$ into the gadget that is secure against the stronger *wire-independent faults* $\mathcal{F}^{\mathsf{ind}}$ launched by *adaptive* adversaries, at the cost of obtaining a slightly weaker security notion. More details are explained in Lemma 2.

Combined Attacks. We can combine passive and active adversaries as introduced by Berndt et al. [BEF+23] to obtain a (t, e)-combined adversary $\mathcal{A}$. This combined adversary can *simultaneously* probe t wires w_i and choose another e wires w'_i to fault. More specifically, we define the security as follows: First, the (t, e)-combined adversary chooses two inputs x_0 and x_1 for the circuit C^R. Then one input x_b is chosen uniformly at random by $b \leftarrow_\$ \{0, 1\}$, and the circuit C^R is evaluated on input x_b. Next, the adversary $\mathcal{A}$ can probe or fault on wires of C^R, before he is required to output a bit $b' \in \{0, 1\}$. $\mathcal{A}$ wins if $b = b'$ with probability greater than $\frac{1}{2} + \mathsf{negl}$, i.e. if he is able to guess whether x_0 or x_1 has been chosen as input. In more detail, let us denote the output of the original circuit by y_b and the output of the faulted circuit by y'_b. We say that circuit C^R is *ϵ-secure* against (t, e)-combined adversaries $\mathcal{A}$ if both $\Pr[b = b'] = \frac{1}{2}$ and $\Pr[y_{b'} \in \{y_b, \perp\}] \geq 1 - \epsilon$ hold.

We can *securely* compose $\mathsf{(S)NI}$ gadgets in the case of passive adversaries who can only perform probing attacks as shown in [BBD+16]. However, an $\mathsf{(S)NI}$ gadget does not necessarily remain $\mathsf{(S)NI}$ in the presence of faults, as demonstrated in [BEF+23]. For instance, an adversary can insert faults that cancel out the randomness of a gadget. To address this, Berndt et al. adapted the $\mathsf{(S)NI}$ property to ensure composability under combined attacks.

Definition 4 (Fault-resilient $\mathsf{(S)NI}$ [BEF+23, Def. 5])**.** A gadget G is t-fault-resilient (strong-) non-interfering (t-$\mathsf{fr(S)NI}$) with respect to a fault set $\mathcal{F}$ if $T[G]$ is t-$\mathsf{(S)NI}$ for any fault attack $T \in \mathcal{A}(\mathcal{F})$.

Sometimes, it might be sufficient to consider a slightly weaker version of the frSNI notion. We use a similar notion as before but with a bounded number of faults, and we treat the faults as probes. More details on the difference between the notions are presented in [BEF+23].

[7]Arnold et al. [ABEO24] generalized the definition of fault-robustness proposed by Berndt et al. [BEF+23]. We also consider the generalized fault-robustness here.

Definition 5 (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 $|T| + d' \leq d$.

Definition 6 (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 $d_1 + d_2 + \epsilon_1 + \epsilon_2 \leq d$.

Theorem 1 (Composition of frSNI [BEF+23, Thm. 1])**.** *The composition of two d-frSNI (or d-wfrSNI) gadgets with respect to $\mathcal{F}$ is d-frSNI (or d-wfrSNI) with respect to $\mathcal{F}$ if $\mathcal{F} \subseteq \mathcal{F}^{\mathsf{ind}}$.*

Berndt et al. [BEF+23] showed that a circuit C, which is *independently* proven to be leakage- and fault resilient, is not secure against the combined attacks. To guarantee combined security, Berndt et al. also introduced a notion called fault invariance, which allows to *push* injected faults to the inputs or the outputs of the gadgets. Intuitively, if the faults injected into the gadgets can also be obtained by faulting the inputs, then *sufficiently* secret-shared inputs (i.e. considering a high enough number of shares n) can guarantee that the combined adversary $\mathcal{A}$ learns nothing about the secret inputs. In other words, fault-invariance ensures that the information obtained from a probe is independent of any faults, as the internal values z_i of a circuit $\mathsf{C}(x_0, \ldots, x_{n-1})$ depend only on the internal randomness R and the inputs $(x_0, \ldots, x_{n-1})$. Therefore, we can represent them as functions $z_i = f^R(x_0, \ldots, x_{n-1})$.

Definition 7 (Fault-invariance [BEF+23, Def. 8])**.** A circuit $\mathsf{C}(x_0, \ldots, x_{n-1})$ is fault-invariant with respect to a fault set $\mathcal{F}$ if for any $T \in \mathcal{A}\,(\mathcal{F})$, any intermediate value f in C^R and the corresponding value f' in $T[\mathsf{C}^R]$, there are $\xi, \xi_0, \ldots, \xi_{n-1} \in \mathcal{F}$ such that it holds

$$f'^R(x_0, \ldots, x_{n-1}) = \xi(f^R(\xi_0(x_0), \ldots, \xi_{n-1}(x_{n-1})))$$

for any inputs $(x_0, \ldots, x_{n-1})$ and randomness R.

To prove that a gadget is frSNI, it suffices to show that the gadget is fault-invariant and SNI:

Lemma 1 ([BEF+23, Cor. 1])**.** *If a gadget is d-NI (d-SNI) and fault-invariant with respect to $\mathcal{F} \subset \mathcal{F}^{\mathsf{ind}}$, the gadget is d-frNI (frSNI, respectively) with respect to $\mathcal{F}$.*

Moreover, Berndt et al. showed a connection of d-$\mathsf{fr(S)NI}$ with respect to $\mathcal{F}^+$ and d-$\mathsf{wfr(S)NI}$ with respect to $\mathcal{F}^{\mathsf{ind}}$:

Lemma 2 ([BEF+23, Cor. 2])**.** *If a gadget is d-$\mathsf{fr(S)NI}$ and fault-invariant with respect to $\mathcal{F}^+$, it is adaptively d-$\mathsf{wfr(S)NI}$ with respect to $\mathcal{F}^{\mathsf{ind}}$.*

2.2 Packed Secret Sharing

The packed secret-sharing technique introduced by Franklin and Yung [FY92] is a generalization of the standard Shamir's secret sharing scheme [Sha79]. In this section, we explain step by step the differences between Shamir's secret sharing and packed secret sharing to obtain the notation and background we need for the following sections.

We begin by reconsidering Shamir's secret sharing, where we want to share one secret, say v_0, into n shares $F_0, \ldots, F_{n-1} \in \mathbb{F}$. We can do this by sampling d values $r_1, \ldots, r_d$ uniformly at random from the field $\mathbb{F}$ and constructing the polynomial with coefficients

$(v_0, r_1, \dots, r_d)$, namely $f(X) = v_0 + r_1 X^1 + \dots + r_d X^d$. Now, our secret shares are the evaluation of the polynomial at pairwise disjoint points $u_0, \dots, u_{n-1} \in \mathbb{F} \setminus \{0\}$, i.e. $F_i = f(u_i)$. We can represent this idea with the help of a Vandermonde matrix as follows. Denote $\mathsf{Van}_{(d+1)\times n}(u_0, \dots u_n)$ by U. In order to encode (or mask) a secret v_0, we evaluate our degree d polynomial $f(X)$ with coefficients $\boldsymbol{v}' = (v_0, r_1, \dots, r_d)$ at n pairwise non-zero disjoint points $u_0, \dots, u_{n-1}$,

$$\boldsymbol{v}' \cdot U = (f(u_0), \dots, f(u_{n-1})).$$

To recover all coefficients of our polynomial from $d+1$ distinct shares, we can use the inverse matrix $U^{-1} = \mathsf{Van}^{-1}_{(d+1)\times(d+1)}(u_0, \dots, u_d)$ to compute $\boldsymbol{v}' = (f(u_0), \dots, f(u_d)) \cdot U^{-1}$. From this representation, we already see that we can recover each element v'_j directly by calculating $\sum_{i=0}^{d} f(u_i) \cdot U^{-1}_{i,j}$. Consequently, the decoding (or unmasking) is well defined, i.e., $v_0 = \sum_{i=0}^{d} f(u_i) \cdot U^{-1}_{i,0}$. Similarly, as described by Arnold et al. [ABEO24], we can put two secrets, v_0 and v_d, into one polynomial. In this case, one represents the lowest and the highest coefficient of the polynomial, respectively. Additionally, Arnold et al. [ABEO24] mentioned that it is not possible to securely extend the scheme directly to $k > 2$ secrets since this would reveal the secrets in some cases. In particular, they provide an example demonstrating the insecurity of hiding the secrets in the coefficients other than the highest and lowest coefficients. Instead, if we aim to consider not only one or two secrets but k secrets in one encoding, we need to slightly adapt the masking and unmasking procedure.

In more detail, let $v_0, \dots, v_{k-1}$ be our secrets and $r_k, \dots, r_d$ be the random values chosen independently and uniformly at random from $\mathbb{F}$. We denote the vector $(v_0, \dots, v_{k-1}, r_k, \dots, r_d)$ by $\boldsymbol{v}$.

As a first step, we construct a polynomial, $f(X) = \sum_{i=0}^{d} f_i \cdot X^i$, such that the secrets are embedded at certain points of this polynomial. Specifically, instead of embedding one secret into $f(0)$, we consider k secrets in $f(u_0), \dots, f(u_{k-1})$, where $u_0, \dots, u_{k-1}$ are pairwise disjoint support points. As in the "one dimensional" case of Shamir's secret sharing, we must also consider some random numbers $r_k, \dots, r_d$ to fix the remaining coefficients if $k < d+1$. Rather than inserting the values directly, we fix the rest of the coefficients by choosing some more pairwise disjoint support points $u_k, \dots, u_d$ such that $f(u_i) = r_i$ for $i \in [k, \dots, d]$. Analogous to the case of Shamir's secret sharing, this can be represented by

$$\boldsymbol{v} \cdot U^{-1} = \boldsymbol{f},$$

where $\boldsymbol{f} = (f_0, \dots, f_d)$ and $U^{-1} = \mathsf{Van}^{-1}_{(d+1)\times(d+1)}(u_0, \dots, u_d)$.

In our second step, we insert more randomness. We can think of the following step as evaluating the polynomial $f(X)$ at random points $x_0, \dots, x_{n-1}$, which is disjoint from $u_0, \dots, u_d$, to obtain n final shares. Considering a Vandermonde matrix with respect to these pairwise disjoint points $x_0, \dots, x_{n-1}$, $V = \mathsf{Van}_{(d+1)\times n}(x_0, \dots, x_{n-1})$, we receive our shares $\boldsymbol{F} = (F_i)_{i\in[n]}$ by calculating

$$(F_0, \dots, F_{n-1}) = \boldsymbol{f} \cdot V.$$

Finally, we obtain the *packed secret sharing* scheme presented in [FY92].

To simplify the notation, we consider the masking of the values $\boldsymbol{v}$ as a matrix multiplication with $M = U^{-1} \cdot V$, i.e.,

$$\mathsf{mask}(\boldsymbol{v}) \coloneqq \boldsymbol{F} = \boldsymbol{v} \cdot M.$$

We can unmask $\boldsymbol{F}$ similar to the reconstruction phase in Shamir's secret sharing. This is possible because, in both cases, the underlying polynomial is of degree d, which allows the decoding of $\boldsymbol{v}$ using $d+1$ secret shares of $\boldsymbol{F}$. Consequently, by setting $V^{-1} =$

$\mathsf{Van}^{-1}_{(d+1)\times(d+1)}(x_0,\ldots,x_d)$ we obtain the $(d+1)\times(d+1)$-matrix $M^{-1} = V^{-1}\cdot U$, which enables us to unmask by calculating

$$\mathsf{unmask}\,(\boldsymbol{F}) \coloneqq (F_0,\ldots,F_d)\cdot M^{-1}.$$

One can easily see, that correctness still holds, as

$$F_l = \sum_{i=0}^{d} f_i \cdot V_{i,l} = \sum_{i=0}^{d}\left(\sum_{j=0}^{d} F_j \cdot V^{-1}_{j,i}\right)\cdot V_{i,l} = \sum_{j=0}^{d} F_j \sum_{i=0}^{d} \cdot V^{-1}_{j,i}\cdot V_{i,l} = F_l,$$

since $\sum_{i=0}^{d} V^{-1}_{j,i}\cdot V_{i,l} = 0\ \forall j \neq l$ and $\sum_{i=0}^{d} V^{-1}_{j,i}\cdot V_{i,l} = 1$ if $j = l$.

Note that both matrices M and M^{-1} can be pre-computed without knowing the vector $\boldsymbol{v}$. Thus, we can see the masking and unmasking of packed secret sharing as a matrix multiplication, which is similar to the masking and unmasking process of Shamir's secret sharing. As before, we can unmask each element v_j individually by computing the linear combination of $d+1$ shares through $\sum_{i=0}^{d} F_i \cdot M^{-1}_{i,j}$.

In order to guarantee the security of our scheme in the upcoming sections, we also need to keep Lemma 3 in mind. More details on Lemma 3 can be found in [FY92].

Lemma 3 ([FY92])**.** *For a random* degree-d *packed secret sharing of a vector* $\boldsymbol{v} = (v_0,\ldots,v_{k-1},r_k,\ldots,r_d)$*, any* $d-k+1$ *shares are independent of the secret* $(v_0,\ldots,v_{k-1})$.

3 Linear Compiler

Before presenting a general compiler, the goal of this section is to provide a compiler for affine computations. This approach helps us get familiar with the notation before moving on to the more complex gadgets needed for the general compiler. Therefore, we first present share-wise addition in Section 3.1, and share-wise multiplication in Section 3.2. In Section 3.3 we consider two different ways to generate zero-encodings and present a general affine transformation gadget in Section 3.4. We conclude this section by describing two methods to refresh shares in Section 3.5.

Recall that we are interested in using packed secret sharing [FY92] to construct leakage- and fault-resilient gadgets. To achieve this, we take advantage of the error-detecting capabilities of packed secret sharing. Specifically, unlike in Section 2.2, where only a subset of $d+1$ shares are required to unmask the secret vector $\boldsymbol{v}$, our approach requires all $n \geq d+e+1$ shares during the computation and reconstruction, where d denotes the degree of the polynomial and we aim to be secure against e faults and t probes, where $t < d+1-k$ and k denotes the number of embedded secrets. For simplicity, let $n = d+e+1$ denote the number of secret shares in the subsequent discussion. This is necessary due to the potential of an active adversary to fault the gadget inputs or inject faults during the computation. For further details regarding this extension, we refer to the Supplementary Material A.1. Furthermore, we consistently use the same support points $u_0,\ldots,u_d$ for all packed secret sharings to embed the secrets $\boldsymbol{v} = (v_0,\ldots,v_{k-1},r_k,\ldots,r_d)$. This approach 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]:

1. Linear Homomorphism: For all $d \geq k-1$ and $\boldsymbol{v},\boldsymbol{v'} \in \mathbb{F}^d$, $\mathsf{mask}\,(\boldsymbol{v}+\boldsymbol{v'}) = \mathsf{mask}\,(\boldsymbol{v}) + \mathsf{mask}\,(\boldsymbol{v'})$.
2. 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.

The correctness of the above properties follows directly from the underlying polynomials. In addition, unary gates (gates with only one input wire), such as addition or multiplication with a constant, can be handled easily since these operations are linear and can be computed locally on the packed secret shares. Therefore, we focus on binary gates, such as addition and multiplication gates that take two inputs.

3.1 Share-Wise Addition

Gadget **SWAdd** (cf. Alg. 1) computes a *share-wise* addition such that the output $(Q_i)_{i\in[n]}$ represents the sum of $\boldsymbol{v}$ and $\boldsymbol{v}'$. The correctness of the gadget immediately follows from the linear homomorphism property of packed secret sharing. Besides, the output $(Q_i)_{i\in[n]}$ remains a *degree-d* polynomial since the sum of two *degree-d* polynomials results in a polynomial of degree $\leq d$. Next, we analyze the security of the generalized affine gadgets.

Lemma 4 ([BEF+23, Thm. 6])**.** *Share-wise affine gadgets are $(d+1-k)$-frNI with respect to $\mathcal{F}^+$ (or wfrNI with respect to $\mathcal{F}^{\mathsf{ind}}$) and e-fault-robust with respect to $\mathcal{F}^{\mathsf{ind}}$.*

Proof. Fault-robustness with respect to wire independent faults directly follows from Definition 3, as t_1 inserted faults only deterministically affect t_1 outputs.

To prove the frNI property, it is sufficient to show that our share-wise affine gadgets are NI and fault-invariant, due to Lemma 1. Moreover, due to Lemma 2 NI and fault-invariance of the gadget directly implies wfrNI with respect to $\mathcal{F}^{\mathsf{ind}}$. It remains to prove that a share-wise affine gadget is NI and fault-invariant.

Similar to the argument presented in [BEF+23, Thm. 6], the NI property holds because the gadget performs share-wise operations and thus, all output values solely depend on input shares with the same index. Choosing packed secret sharing instead of Shamir's secret sharing does not affect this fact. For example, considering share-wise addition, $Q_i \leftarrow F_i + G_i$ only depends on the i-th share F_i and G_i for all $i \in [n]$. Since each probe requires only one share of each input sharing, t probes can be simulated by at most t shares of each input. Consequently, our share-wise gadgets are t-NI when using a polynomial of degree $d \geq t+1$. However, for privacy reasons, we only consider $t \leq d+1-k$ since exceeding this threshold would reveal information about the shares, given that our shares are only $(d+1-k)$-independent (cf. Lem. 3).

Furthermore, fault-invariance directly follows from the fact that share-wise addition is a linear homomorphism, and addition satisfies associativity. An additive fault $\xi \in \mathcal{F}^+$ adds a constant value a on a wire. Due to linearity, this addition of a can be moved to the output. Specifically, for $Q_i \leftarrow F_i + G_i$ it holds that $(F_i + G_i) + a = F_i + (G_i + a) = (F_i + a) + G_i$. Note that in the first term, the fault is moved to the output as $\xi(Q_i) = Q_i + a$. In the second and third terms, the fault is moved to the two inputs G_i and F_i as $\xi(G_i)$ and $\xi(F_i)$, respectively. This concludes the proof. □

Algorithm 1 **SWAdd** for $n = d+e+1$, and $k \leq d$

Input : *Degree-d* shares of $\boldsymbol{v} = (v_0, \ldots, v_d)$ as $(F_i)_{i\in[n]}$ and *degree-d* shares of $\boldsymbol{v}' = (v'_0, \ldots, v'_d)$ as $(G_i)_{i\in[n]}$

Output : *Degree-d* shares of $\boldsymbol{v} + \boldsymbol{v}' = (v_0 + v'_0, \ldots, v_d + v'_d)$ as $(Q_i)_{i\in[n]}$

```
1 for $i \in [n]$ do
2     $Q_i \leftarrow F_i + G_i$
3 return $(Q_i)_{i\in[n]}$
```

3.2 Share-Wise Multiplication

Multiplication is a non-linear operation and one of the most complex gadgets in our scheme. It uses the multiplicative property of packed secret sharing, and we can use the share-wise multiplication gadget **SWMult** [BEF+23, Alg. 2] to compute the product of two secret vectors $\boldsymbol{v}$, $\boldsymbol{v}'$, and get $\boldsymbol{v} * \boldsymbol{v}' = (v_i \cdot v_i')_{i\in[d]}$. However, this process increases the degree of the resulting underlying polynomial $q(X)$ from d to $2d$. Hence, to guarantee the correctness of the multiplication, we need to reduce the degree of $q(X)$ to d before the next multiplication. To the best of our knowledge, there are two types of degree-reduction. The first type of degree reduction technique (we call it BGW-based) reduces the polynomial's degree after the multiplication, which requires at least $2d + 1$ shares. In contrast, the second type (we call it LaOla-based) reduces the degree before the multiplication, which requires a smaller number of shares but may increase the overall computational complexity for non-linear operations. We compare and discuss the details of these degree reduction techniques in Section 4.

Algorithm 2 SWMult for $n = d + e + 1$

Input : *Degree-d* shares of $\boldsymbol{v} = (v_0, \dots, v_d)$ as $(F_i)_{i\in[n]}$ and *degree-d* shares of $\boldsymbol{v}' = (v_0', \dots, v_d')$ as $(G_i)_{i\in[n]}$
Output : *Degree-$2d$* shares of $\boldsymbol{v} \cdot \boldsymbol{v}' = (v_0 \cdot v_0', \dots, v_d \cdot v_d')$ as $(Q_i)_{i\in[n]}$

```
1 for $i \in [n]$ do
2     $Q_i \leftarrow F_i \cdot G_i$
3 return $(Q_i)_{i\in[n]}$
```

3.3 Zero-Encoding Gadgets

Now, we present two different methods to re-randomize shares while maintaining their functionality. We can do so by generating encodings of zeros, which can be share-wise added to shares $(F_i)_{i\in[n]}$ without changing the embedded secret. The first gadget, ZEnc_n^d, uses a straightforward approach to encode zeros. The second gadget, sZEnc_n^d, is needed to guarantee security for gadgets presented in Section 4.

ZEnc_n^d Gadget. We generate a zero encoding by masking with respect to the "secret" consisting of k zeros $0_0, \dots, 0_{k-1}$. Specifically, to generate a sharing $(g_i)_{i\in[n]}$, we first construct a vector $\boldsymbol{v}_{\mathsf{zero}} = (0_0, \dots, 0_{k-1}, r_k, \dots, r_d)$, where $(r_k, \dots, r_d) \leftarrow_\$ \mathbb{F}^{d+1-k}$ are chosen uniformly at random. Let U^{-1} denote $\mathsf{Van}^{-1}_{(d+1)\times(d+1)}(u_0, \dots, u_d)$ and V denote $\mathsf{Van}_{(d+1)\times n}(x_0, \dots, x_{n-1})$. Next, we compute

$$(g_0, \dots, g_{n-1}) = \boldsymbol{v}_{\mathsf{zero}} \cdot M, \tag{1}$$

where M denotes the matrix multiplication result of $U^{-1} \cdot V$. Note that the above matrix multiplication $\boldsymbol{v}_{\mathsf{zero}} \cdot M$ has computational complexity $\mathcal{O}(nd)$. We can reduce the complexity to $\mathcal{O}(n \cdot (d + 1 - k))$ based on the following observation: since $\boldsymbol{v}_{\mathsf{zero}}$ contains k zeros, we can omit the multiplication of the zeros in $\boldsymbol{v}_{\mathsf{zero}}$ with the elements in matrix M by *removing* the first k zeros from $\boldsymbol{v}_{\mathsf{zero}}$ and first k rows from M. Due to this observation, the correctness of our improved construction still holds. ZEnc_n^d (cf. Alg. 3) uses packed secret sharing to generate a sharing of zeros. However, the refresh gadget instantiated with ZEnc_n^d (cf. Alg. 3) does not satisfy SNI. For example, Berndt et al. provided a counterexample in [BEF+23, Appendix B.5], which can be easily adapted to packed secret sharing.

Algorithm 3 ZEnc_n^d for $n = d + e + 1$ and $k \leq d$

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

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

$\mathsf{sZEnc}_{\boldsymbol{n}}^{\boldsymbol{d}}$ Gadget. Based on ZEnc_n^d (cf. Alg. 3), we construct a gadget $\mathsf{sZEnc}_{\boldsymbol{n}}^{\boldsymbol{d}}$ (cf. Alg. 4) that basically sums $d + 1 - k$ different zero encodings generated by ZEnc_n^d and ensures the **Refresh** (cf. Alg. 6) to be SNI. The output $(y_i)_{i\in[n]}$ of $\mathsf{sZEnc}_{\boldsymbol{n}}^{\boldsymbol{d}}$ is guaranteed to be a zero encoding, since it is the sum of the zero encodings generated by ZEnc_n^d. Note that we improve the complexity of $\mathsf{sZEnc}_{\boldsymbol{n}}^{\boldsymbol{d}}$ from nd^2 to $n \cdot (d + 1 - k)^2$ by changing the number of the **for** loops (on line 2 of Alg. 4) from d (as in [BEF+23, Algo. 4]) to $d+1-k$. Before proving that our $\mathsf{sZEnc}_{\boldsymbol{n}}^{\boldsymbol{d}}$ can achieve SNI refreshing, we first prove the probing security of $\mathsf{sZEnc}_{\boldsymbol{n}}^{\boldsymbol{d}}$ in Lemma 5.

Lemma 5 (Probing resilience of $\mathsf{sZEnc}_{\boldsymbol{n}}^{\boldsymbol{d}}$ [BEF+23, Lem. 3])**.** *For any set P with $d' \leq d + 1 - k$ probes, for $(e_i)_{i\in[n]} \leftarrow \mathsf{sZEnc}_{\boldsymbol{n}}^{\boldsymbol{d}}$ there is a subset $A \subset [n]$ with $|A| = n - d'$ such that it holds*

1. *the values $(e_i)_{i\in A}$ are $(d+1-k) - d'$-wise independent, as well as independent from P and $(e_i)_{i\in[n]\setminus A}$,*
2. *P can be perfectly simulated with $(e_i)_{i\in[n]\setminus A}$.*

Proof. The argument is similar to the one provided in Lemma 3 of [BEF+23]. Let us consider the possible probes of P:

- Group K is defined as $K = \bigcup_{l\in[n]} K_l$, where K_l consists of the internal probes $\left(g_i^l\right)_{i\in[n]} \leftarrow \mathsf{ZEnc}_n^d$ or $\left(g_i^l\right)_{i\in[n]}$.
- Group J consists of all probes in the set $J = \bigcup_{i\in[n]} J_i$, with $J_i = \{y_i^{(0)}, \dots, y_i^{(d-k)}\}$, where $y_i^{(j)} = \sum_{k=0}^{j} \left(g_i^j\right)_{i\in[n]}$.

Next, we simulate all probes in P perfectly as follows. Note that $P = \tilde{K} \cup \tilde{J}$ for $\tilde{K} \subset K$ and $\tilde{J} \subset J$.

If one value in J_i is probed, we simulate all elements of J_i[8]. Moreover, we assume that for any probe in K_l, all values $\left(g_i^l\right)_{i\in[n]}$ are leaked. Note that we simulate ZEnc_n^d by choosing all $r_k, \dots, r_d$ uniformly at random and following the algorithm to calculate $\left(g_i^l\right)_{i\in[n]}$. We still have $(d+1-k) - |\tilde{K}|$ elements $\left(g_i^l\right)_{i\in[n]}$ that are not leaked and $(d+1-k)$- wise independent and independent of $\tilde{K}$. Now, we need to simulate the probes in $\tilde{J}$. If the corresponding $\left(g_i^l\right)_{i\in[n]}$ are already simulated due to $\tilde{K}$, we consider these values. Otherwise, we choose $\left(g_i^l\right)_{i\in[n]}$ uniformly random, since the probe threshold is $(d+1-k)$ and elements $\left(g_i^l\right)_{i\in[n]}$ are $(d+1-k)$-wise independent. The values that are not fixed in $\tilde{K}$ are added to y_i and thus sum up to an output that is $(d+1-k) - |C|$-wise independent, where $|C|$ is the number of probes in $\tilde{J}$. We obtain $|C| \leq d'$. □

[8]For simplicity, we use an upper-bound estimate here. The high-level idea is that we simulate more probes than the adversary can probe.

Algorithm 4 sZEnc_n^d for $n = d + e + 1$ and $k \le d$

Output : Randomized sharing of zeros $(y_i)_{i\in[n]}$.

```
1 initialize $(y_i)_{i\in[n]}$
2 for $j \in [d+1-k]$ do
3     $(g_i)_{i\in[n]} \leftarrow \mathsf{ZEnc}_n^d$(cf. Alg. 3)
4     $(y_i)_{i\in[n]} \leftarrow (y_i)_{i\in[n]} \oplus (g_i)_{i\in[n]}$
5 return $(y_i)_{i\in[n]}$
```

3.4 $(\varphi_0, \ldots, \varphi_d)$-Transformation Gadget

In this section, we present a generic gadget construction for performing various linear transformations, as illustrated in $\mathsf{Gadget_{LT}}$ (cf. Alg. 5 and Tab. 4). To enhance the generality of the linear transformation gadget $\mathsf{Gadget_{LT}}$, we allow it to accept m different groups of secret sharings $\left(F_i^{(0)}\right)_{i\in[n]}, \ldots, \left(F_i^{(m-1)}\right)_{i\in[n]}$ as inputs, merge them as needed using the function $\ell_{i,j}^{\varphi_0,\ldots,\varphi_d}$, and obtain one secret sharing $(H_i)_{i\in[n]}$ as output. Table 4 provides a selection of useful examples for choosing $\ell_{i,j}^{\varphi_0,\ldots,\varphi_d}$ for different operations.

To ensure compatibility of our construction with AES, we introduce gadgets to permute and recombine the underlying secrets. Both operations can be realized using our linear transformation gadget $\mathsf{Gadget_{LT}}$. We consider an arbitrary permutation function $\pi : \{0, 1, \ldots, d\} \to \{0, 1, \ldots, d\}$, the **Permutation** gadget takes a secret sharing $(F_i)_{i\in[n]}$ which correspond to secrets $v_0, \ldots, v_d$, and outputs a secret sharing $(F'_i)_{i\in[n]}$ which correspond to re-ordered secrets, $v_{\pi(0)}, v_{\pi(1)}, \ldots, v_{\pi(d)}$.

The recombination operation can be seen as a generalization of the permutation operation. Let $\pi_R : [d+1] \to I \times J$ be an arbitrary function, with $I = \{i | i \in [d+1]\}$, such that $|I| = d+1$ and $J = \{j | j \in [m]\}$. The **Recombination** gadget takes m sharings $\left(F_i^{(0)}\right)_{i\in[n]}, \ldots, \left(F_i^{(m-1)}\right)_{i\in[n]}$ as inputs and allows to construct a secret sharing of arbitrary recombination of the underlying secrets. Specifically, the recombination outputs a sharing corresponding to the secrets $\tilde{v}_0, \ldots, \tilde{v}_d$, where $\tilde{v}_l = v_{\pi_R(l)} = v_{i,j} = v_i^{(j)}$ denotes the i-th secret vector of the j-th input sharing for every $l \in [d]$. We believe such operation to be useful when developing other parallelizing techniques, such as paralleling the evaluation of multiple gates of the same types, as explored by Goyal et al. [GPS21, GPS22] for designing MPC protocols. Additionally, our combined resilient *BGW-based* multiplication gadget $\mathsf{Mult_{BGW}}$ (cf. Sec. 4.1), which is adapted from the multiplication algorithm (together with the degree reduction) proposed by Franklin and Yung [FY92] can be realized using our linear transformation gadget $\mathsf{Gadget_{LT}}$ with corresponding function $\ell_{i,j}^{\varphi_0,\ldots,\varphi_d}$ and matrix $\lambda^{(\mathsf{DegRed})}$. Since the secret permutation gadget is essential in the AES process, we will explain the generic gadget $\mathsf{Gadget_{LT}}$ (cf. Alg. 5) by illustrating its use for permutation. In the following section, we explain step-by-step how to construct function $\ell^{\varphi_0,\ldots,\varphi_d}(\cdot)$ for gadget **Permutation** such that it satisfies combined resilience.

Permutation Gadget Recall that we can permute a vector $\boldsymbol{v} = (v_0, \ldots, v_d)$ and obtain a permuted vector $\boldsymbol{v}'$ by multiplying $\boldsymbol{v}$ with a $(d+1) \times (d+1)$ permutation matrix $M_{\boldsymbol{v}\to\boldsymbol{v}'}$ through $\boldsymbol{v}' = \boldsymbol{v} \cdot M_{\boldsymbol{v}\to\boldsymbol{v}'}$. Next, we explain how to realize this permutation operation using the secret shares $(F_i)_{i\in[n]}$ of $\boldsymbol{v}$. The goal is to output the secret shares $(F'_i)_{i\in[n]}$ of $\boldsymbol{v}'$. Recall the fundamental properties of packed secret sharing introduced in [FY92]. Let U^{-1} denote $\mathsf{Van}^{-1}_{(d+1)\times(d+1)}(u_0, \ldots u_d)$ and V denote $\mathsf{Van}^{-1}_{(d+1)\times n}(x_0, \ldots x_{n-1})$. Namely, we generate secret shares $\boldsymbol{F} = (F_0, \ldots, F_{n-1})$ of $\boldsymbol{v}$ through the following step:

$$\boldsymbol{F} = \boldsymbol{v} \cdot M_{\boldsymbol{v} \to \boldsymbol{F}}, \tag{2}$$

where $M_{\boldsymbol{v} \to \boldsymbol{F}} = U^{-1} \cdot V$. As discussed in Supplementary Material A.1, there also exists a matrix $M_{\boldsymbol{F} \to \boldsymbol{v}}$ such that we can reconstruct secrets through $\boldsymbol{v} = \boldsymbol{F} \cdot M_{\boldsymbol{F} \to \boldsymbol{v}}$. Additionally, we can generate secret shares $\boldsymbol{F}' = \left(F'_0, \ldots, F'_{n-1}\right)$ of the permuted $\boldsymbol{v}'$ using $M_{\boldsymbol{v} \to \boldsymbol{v}'}$ through the following step:

$$\begin{aligned} \boldsymbol{F}' &= \boldsymbol{v} \cdot M_{\boldsymbol{v} \to \boldsymbol{v}'} \cdot U^{-1} \cdot V & (3)\\ &= \boldsymbol{F} \cdot M_{\boldsymbol{F} \to \boldsymbol{v}} \cdot M_{\boldsymbol{v} \to \boldsymbol{v}'} \cdot M_{\boldsymbol{v} \to \boldsymbol{F}} & (4)\\ &= \boldsymbol{F} \cdot M_{\boldsymbol{F} \to \boldsymbol{F}'}, & (5) \end{aligned}$$

where the details on constructing matrix $M_{\boldsymbol{F} \to \boldsymbol{F}'}$ are provided in Supplementary Material A.4. In a nutshell, the multiplication of $\boldsymbol{F}$ and $M_{\boldsymbol{F} \to \boldsymbol{F}'}$ is equivalent to the following processes: reconstructing the secret $\boldsymbol{v}$, generating the permutation $\boldsymbol{v}'$ of the recovered $\boldsymbol{v}$, and re-sharing of $\boldsymbol{v}'$ as $\boldsymbol{F}'$. For simplicity, we denote matrix $M_{\boldsymbol{F} \to \boldsymbol{F}'}$ by M in the following discussion.

It is easy to verify that by setting $\lambda^{(\mathsf{P})}_{j,i} = M_{i,j}$, $\mathsf{Gadget_{LT}}$ outputs $(F'_i)_{[n]}$ that encode secrets $\boldsymbol{v}'$, when given inputs $(F_i)_{[n]}$ that encodes secrets $\boldsymbol{v}$. Similarly, by constructing specific matrices M for the function $\ell^{\varphi_0,\ldots,\varphi_d}_{i,j}$, $\mathsf{Gadget_{LT}}$ can realize various linear transformation operations, such as recombination, multiplication with degree reduction, and generation of zero-encodings. Furthermore, to realize combined resilience, we need to set $\lambda^{(\mathsf{P})}_{j,i} = M_{i,j} + E_{i,j}$, where $\lambda_{i,j}$ are the elements of the matrix $\mathsf{Van}_{(d+1)\times n}\left(x_0, \ldots, x_{n-1}\right)$ and the error propagation (or preserving) term $E_{i,j}$ [SFES18] is defined as follows:

$$E_{i,j} = \begin{cases} \lambda_{i,n-j-1} & \text{if } 0 \leq j < e \\ 0 & \text{if } e \leq j \leq n-1 \end{cases}. \tag{6}$$

Intuitively, the error propagation term $E_{i,j}$ would preserve the error coefficients (i.e., the coefficients of the degree $> d$ terms in the polynomial corresponding to faulty shares) such that the faulted inputs $(F_i + \Delta_i)_{i\in[n]}$ can be detected.

Next, we prove the correctness and combined resilience property of our generic gadget $\mathsf{Gadget_{LT}}$.

Algorithm 5 $\mathsf{Gadget_{LT}}$ for $n = d + e + 1$, $d \leq k$, and $m \leq k$

Input : m groups of *degree-d* shares of secret vectors $\boldsymbol{v}^{(0)}, \ldots, \boldsymbol{v}^{(m-1)}$ as $\left(F^{(0)}_i\right)_{i\in[n]}, \ldots, \left(F^{(m-1)}_i\right)_{i\in[n]}$

Output : *Degree-d* shares of *arbitrary* chosen secrets $\tilde{\boldsymbol{v}} = \left(v^{(0)}_{l_0}, \ldots, v^{(d)}_{l_d}\right)$ as $(H_i)_{i\in[n]}$, where $(l_0, \ldots, l_d) \in \{0, \ldots, m-1\}^{d+1}$ and the chosen secrets can be computed through $v^{(0)}_{l_0} = \varphi_0\left(v^{(0)}_0, \ldots, v^{(m-1)}_0\right), \ldots, v^{(d)}_{l_d} = \varphi_d\left(v^{(0)}_d, \ldots, v^{(m-1)}_d\right)$.

```
1 initialize $(H_i)_{i\in[n]}$
2 for $i \in [n]$ do
3     $\left(\tilde{H}_{i,0}, \ldots, \tilde{H}_{i,n-1}\right) \leftarrow \mathsf{ZEnc}^d_n$(cf. Alg. 3)
4     for $j \in [n]$ do
5         $\tilde{H}_{i,j} \leftarrow \tilde{H}_{i,j} + \ell^{\varphi_0,\ldots,\varphi_d}_{i,j}\left(F^{(0)}_i, \ldots, F^{(m-1)}_i\right)$
6         $H_j \leftarrow H_j + \tilde{H}_{i,j}$
7 return $(H_i)_{i\in[n]}$
```

Table 4: Instantiations of different functions $\ell_{i,j}^{\varphi_0,\ldots,\varphi_d}$ in Algorithm 5. The input sharings are $F_i^{(0)}, \ldots, F_i^{(m-1)}$ that embed the secret vectors $\boldsymbol{v}^{(0)}, \ldots, \boldsymbol{v}^{(m-1)}$. Note that it is possible that function $\ell_{i,j}^{\varphi_0,\ldots,\varphi_d}\left(F_i^{(0)}, \ldots, F_i^{(m-1)}\right)$ only uses one or two inputs, instead of m inputs. The details about how to construct the constant terms $\lambda_{j,i}^{(\mathsf{P})}$, $\lambda_{j+ln,i}^{(\mathsf{P})}$, $\lambda_{j+ln,i}^{(\mathsf{R})}$, for different operations are provided in Supplementary Material A.

Gadget	$\ell_{i,j}^{\varphi_0,\ldots,\varphi_d}\left(F_i^{(0)}, \ldots, F_i^{(m-1)}\right)$	Functionality
Refresh$_{\mathsf{LT}}$	$\mathbf{I}_{i,j} \cdot F_i$	Re-randomizing the encoding of $\boldsymbol{v}$
Permutation	$\lambda_{j,i}^{(\mathsf{P})} \cdot F_i$	$v_{\pi(0)}, v_{\pi(1)}, \ldots, v_{\pi(d)}$
Recombination	$\sum_{l=0}^{m-1} \lambda_{j+ln,i}^{(\mathsf{R})} \cdot F_i^{(l)}$	$\tilde{v}_0, \ldots, \tilde{v}_{m-1}$, where $\tilde{v}_l = v_{\pi_r(l)} = v_i^{(j)}, j \in [m]$ for $i \in [d+1]$
Mult$_{\mathsf{BGW}}$ [9]	$\lambda_{j,i}^{(\mathsf{DegRed})} \cdot F_i^{(0)} \cdot F_i^{(1)}$	Element-wise multiplying the secrets in $\boldsymbol{v}^{(0)}$, $\boldsymbol{v}^{(1)}$ to get $\boldsymbol{v}^{(0)} * \boldsymbol{v}^{(1)} = \left(v_0^{(0)} \cdot v_0^{(1)}, \ldots, v_d^{(0)} \cdot v_d^{(1)}\right)$ and reducing the degree of output shares

Lemma 6. *Algorithm 5 is correct for all instantiations of* $\ell_{i,j}^{\varphi_0,\ldots,\varphi_d}(\cdot, \ldots, \cdot)$ *shown in Table 4. When considering* **Mult$_{\mathsf{BGW}}$**, *it is necessary to have* $n > 2d$ *shares and polynomials of degree* d.

Proof (sketch). We aim to show that the functionality as denoted in Table 4 is correct when inserting $\ell_{i,j}^{\varphi_0,\ldots,\varphi_d}$ as specified in Table 4 into Algorithm 5. The core idea for the proof is the following observation: For one specific j, over the iteration of i, we obtain

$$H_j = \sum_{i=0}^{n-1} \ell_{i,j}^{\varphi_0,\ldots,\varphi_d}\left(F_i^{(0)}, \ldots, \mathbb{F}_i^{(m-1)}\right) + \left(0, \ldots, 0, \sum_{i=0}^{n-1} r_{i,k}, \ldots, \sum_{i=0}^{n-1} r_{i,(n-1)}\right) \cdot M_j,$$

where M_j denotes the j-th column of M. To prove correctness, we need to focus on the first term of this equation, as the latter term is the j-th share of the sum of n zero encodings. Supplementary Material A details how to carefully choose the elements $\ell_{i,j}^{\varphi_0,\ldots,\varphi_d}$. Note that $n > 2d$ shares are required in case we consider **Mult$_{\mathsf{BGW}}$**, as the share-wise multiplication increases the degree of the polynomial from d to $2d$. For further information, we refer to Lemma 16 in the supplementary material. $\square$

We can prove Algorithm 5 to be SNI and fault-invariant with respect to $\mathcal{F}^+$ as follows.

Lemma 7. *The linear transformation gadget, Algorithm 5, is t-SNI.*

Proof (sketch). Considering the linear transformation gadget with respect to two inputs, the SNI property can be shown similarly to [ABEO24, Lem. 8], since we only needed to adapted the public parameters, which does not change the proof. However, as our linear transformation gadget considers more inputs than the (ψ_0, ψ_1)-gadget presented by Arnold et al., we have included a detailed proof of the SNI property in the supplementary material (cf. Supp. B). $\square$

Lemma 8. *All gadgets introduced in Table 4 are fault invariant with respect to* $\mathcal{F}^+$.

Proof. Since the fault set $\mathcal{F}^+$ consists of additive faults, all linear and affine operations are fault invariant. The only non-affine operation in Algorithm 5 can appear in $\ell_{i,j}^{\varphi_0,\ldots,\varphi_d}\left(F_i^{(0)}, \ldots, F_i^{(m-1)}\right)$. However, in all instances presented in Table 4, this is the only non-linear operation. Thus, all faults before the operation of $\ell_{i,j}^{\varphi_0,\ldots,\varphi_d}$ can be moved to the gadget's input, and all faults after the operation of $\ell_{i,j}^{\varphi_0,\ldots,\varphi_d}$ can be moved to the gadget's output as in [BEF+23, ABEO24]. $\square$

[9] We used a simplified version of the degree reduction from [FY92] that requires *degree-d* (instead of *degree-$2d$*) polynomials to generate zero encodings for secret re-sharing.

Moreover, we can show that our linear transformation gadget (cf. Alg. 5) instantiated with respect to Table 4 satisfies fault-robustness.

Lemma 9. *The linear transformation gadgets presented in Table 4 are e- fault robust with respect to $\mathcal{F}^+$.*

Proof. The first two gadgets of Table 4, **Refresh**$_{\mathrm{LT}}$ and **Permutation** consider one input sharing each. In comparison to [ABEO24, Table 5], we needed to change the public parameters, as well as the coefficients $\ell_{i,j}^{\varphi_0,\varphi_1}$ to maintain correctness. However, this does not impact the proof and we can proceed similarly to [ABEO24, Lem. 19]. Furthermore, the gadget **Mult**$_{\mathbf{BGW}}$ takes two input sharings into account. But similar to the previous case, we changed the public parameters in comparison to Arnold et al. and the coefficients $\ell_{i,j}^{\varphi_0,\varphi_1}$, which does not affect the proof. Thus, we can proceed similarly to [ABEO24, Lem. 18]. It is left to show that **Recombination** fulfills fault-robustness as well. Recall that every change to an output of ZEnc_n^d introduced by s internal faults can be achieved by s direct faults on the output. Moreover, each fault injected in the **Recombination** gadget can be transformed into an output fault since the gadget only uses share-wise addition and share-wise multiplication with public constants. Thus, e-fault robustness follows from the security of the underlying encoding. □

3.5 Refresh Gadget

Refresh$_{\mathbf{LT}}$ Gadget. As shown in Table 4, the **Gadget**$_{\mathbf{LT}}$ (cf. Alg. 5) can also be used to refresh or re-randomize shares. In more detail, we can input sharings $(F_i)_{i\in[n]}$ and consider the identity matrix as the transformation matrix. This procedure involves adding n zero encodings to $(F_i)_{i\in[n]}$, which does not change its underlying secret $\boldsymbol{v}$, but only re-randomizes the sharing. Note that correctness holds true since we add each element F_i only once to the corresponding H_i, and otherwise, we solely add the corresponding i-th share of the n zero encodings. The **Refresh**$_{\mathbf{LT}}$ is SNI, fault invariant and σ-fault robust, details are presented in Lemma 7, Lemma 8 and Lemma 9, respectively.

Optimized Refresh Gadget. We can perform a refresh of our shares using the linear transformation gadget **Gadget**$_{\mathbf{LT}}$ presented previously. However, this process requires n zero encodings ZEnc_n^d (cf. line $2-3$ in Alg. 5). Berndt et al. [BEF+23] introduced a stronger **Refresh** gadget that is not only SNI, but also region probing secure, requiring only d zero encodings. The region probing model introduced in [ISW03], offers stronger security than the notions of NI or SNI. Specifically, the region probing model extends the threshold probing model by applying the threshold of probes, t, to each gadget (or region) in the circuit, meaning that the total number of allowed probes increases with the number of gadgets. In this section, we adapt the **Refresh** gadget from [BEF+23], which was originally designed for Shamir's secret sharing [Sha79], to packed secret sharing and prove that the adapted gadget maintains the desired security properties using only $d+1-k$ zero encodings.

We construct the SNI refreshing gadget **Refresh** (cf. Alg. 6) using $\mathsf{sZEnc}_{\boldsymbol{n}}^{\boldsymbol{d}}$ (cf. Alg. 4). First, the gadget generates a zero encoding ($\mathsf{sZEnc}_{\boldsymbol{n}}^{\boldsymbol{d}}$), then it share-wise adds the zero encoding with the input encoding.

Lemma 10 ([BEF+23, adapted Thm. 17])**.** *The* **Refresh** *gadget is* $(d+1-k)$-frSNI*, w.r.t* $\mathcal{F}^+$.

Proof. We show that **Refresh** (cf. Alg. 6) is SNI and fault-invariant (cf. Def. 7). Let J be the set of output probes in $(Q_i)_{i\in[n]}$ and P be the internal and output probes in $(y_i)_{i\in[n]} \leftarrow \mathsf{sZEnc}_{\boldsymbol{n}}^{\boldsymbol{d}}$. By Lemma 5, we know that for any set of probes P, there is a subset $A \subset [n]$ with $|A| = n - |P|$ such that the values $(y_i)_{i\in[n]}$ are $(d+1-k) - |P|$-wise independent, independent from P and $(y_i)_{i\in[n]}$ and P can be perfectly simulated by $(y_i)_{i\in[n]}$. Thus, we can choose $(y_i)_{i\in[n]\setminus A}$ according to the set P. By Lemma 5, we can simulate P and $(y_i)_{i\in[n]\setminus A}$, which does not change the distribution of $(y_i)_{i\in A}$. The simulator obtains all the input shares of $(F_i)_{i\in[n]\setminus A}$, i.e. $S = \bigcup_{i\in[n]\setminus A}(F_i)$. We can simulate the output probes as follows:

- For any probe in $(Q_i)_{i\in A}$: choose Q_i uniformly at random. Note that the values $(y_i)_{i\in A}$ are $(d+1-k) - |P|$ wise independent by Lemma 5. Since the probe threshold is $(d+1-k)$ we can consider these values as uniform random for the simulation. As $(y_i)_{i\in A}$ re-randomizes the input value F_i, the $Q_i = F_i + y_i$ are indistinguishable from uniformly random.
- For any probe in $(Q_i)_{i\in[n]\setminus A}$: As already mentioned, the simulator can simulate $(y_i)_{i\in[n]\setminus A}$. Moreover, $F_i \in S$ by construction. Thus, the simulator can compute $Q_i = F_i + y_i$.

Since $S = \bigcup_{i\in[n]\setminus A}(F_i)$, we know that $|S| = |[n] \setminus A| = n - (n - |P|) = |P|$. Thus, the simulator requires $|P|$ input values, which is exactly the number of inner probes.

It is left to show that Algorithm 6 is fault-invariant. This follows directly by the fact that **Refresh** consists only of $\mathsf{sZEnc}_{\boldsymbol{n}}^{\boldsymbol{d}}$ together with a share-wise addition. For $\mathsf{sZEnc}_{\boldsymbol{n}}^{\boldsymbol{d}}$ we show in the supplementary material (Lem. 22) that every s internal faults can be pushed to s direct faults on the output. More details can be found in the Supplementary Material C.

□

Considering the work of Berndt et al., we even obtain region probing security [BEF+23, Thm. 15]. Consequently, this security property immediately implies security against more advanced attacks such as horizontal attacks.

Algorithm 6 Refresh for $n = d + e + 1$ and $k \le d$

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

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

4 Generic Compiler

We have introduced our linear compiler, realizing masked operations such as affine operations, general linear transformations, and re-randomization. To build a generic compiler, it remains to construct a multiplication gadget together with the degree reduction. In this section, we focus on constructing combined resilient gadgets for the multiplication operation using packed secret sharing [FY92]. As discussed in Section 3.2, share-wise multiplication of two *degree-d* polynomial sharings results in doubling the degree of output shares to $2d$. To guarantee the correctness of the computation, we need $n > 2d$ shares and must reduce the degree of the output shares from $2d$ back to d before the next multiplication operation. A classical solution for degree reduction was proposed by **B**en-Or-**G**oldwasser-**W**igderson [BGW88], which reduces the degree of output shares from $2d$ to d after the share-wise multiplication to prevent the degree from growing exponentially with the number of multiplication gadgets. In this work, we call multiplication gadgets using this degree reduction technique *BGW-based multiplication*. Recently, Berndt et al. [BEF+23] proposed another type of degree reduction technique that reduces the degree of the polynomial sharing from d to $\frac{d}{2}$ before the share-wise multiplication. We call the multiplication using this type of degree reduction the *LaOla-based multiplication.*

We begin by discussing a special type of degree reduction for packed secret sharing proposed by Damgård et al. [DIK10] originally aimed to reduce the communication and computational complexity of multi-party computation [Yao86, GMW87]. Then, we discuss the challenges identified by Grosso et al. [GSF14] that deploy this degree reduction [DIK10] and explain why it is vulnerable to fault attacks. Next, we revise one variant of the *BGW-based Multiplication* from [FY92], which is designed for packed secret sharing, and demonstrate how to adapt it to be combined resilient. Finally, we re-examined the original *LaOla-based multiplication* [BEF$^+$23] designed for Shamir's secret sharing and demonstrated how to adapt it to be compatible with packed secret sharing.

Degree Reduction from [DIK10]. A widely used degree-reduction technique of packed secret sharing for achieving perfectly secure MPC protocols [GPS21, GPS22] was proposed by Damgård et al. [DIK10]. Their core idea can be described as follows. Suppose we have n parties $(P_i)_{i\in[n]}$ involved in the computation. Before each party P_i receiving the input shares F_i and G_i for share-wise multiplication, the parties generated a pair of correlated shares $(R_i)_{i\in[n]}$ and $(R'_i)_{i\in[n]}$, which are generated using *degree-d* and *degree-*$2d$ polynomials encoding the same set of publicly unknown randomness $r_0, \ldots, r_d$. After receiving the input shares, each party P_i first share-wise multiplies these two *degree-d* input shares and obtains a *degree-*$2d$ polynomial sharing $F_i \cdot G_i$ as a result. Then, the *degree-*$2d$ multiplication result is *blinded* by each party P_i locally by adding it with the *degree-*$2d$ shares R'_i. Next, each party P_i sends the *blinded* result $F_i \cdot G_i + R'_i$ to one designated party who reconstructs the *blinded* results in plaintext and sends (or broadcasts) it back to each party. Finally, each party removes the randomness $r_0, \ldots, r_d$ in the received plaintext result by subtracting it with the *degree-d* shares R_i. However, this degree reduction technique faces challenges when being adapted to environments susceptible to side-channel attacks, particularly to active attacks (cf. Sec. 2.1). The vulnerability arises during the reconstruction of blinded values, where an active adversary could inject faults before (or during) the reconstruction of the *blinded* result. Such an injected fault cannot be detected using Shamir's secret sharing [Sha79] (or packed secret sharing [FY92]) and could further propagate through the *re-sending* (or broadcasting) step, affecting more than one output share and breaking the fault-robustness requirement (cf. Def. 3).

4.1 BGW-Based Multiplication

A classical solution for the degree reduction was proposed by **B**en-Or-**G**oldwasser-**W**igderson [BGW88], which reduces the degree from $2d$ to d after the share-wise multiplication to prevent the degree from growing exponentially with the number of multiplications. More specifically, the BGW reduction technique first takes *degree-*$2d$ polynomial sharings as input and *randomizes* it with *degree-*$2d$ zero encodings, in which the *degree-*$2d$ input shares are re-shared among all the parties. Then, the randomized (or re-shared) *degree-*$2d$ input shares are multiplied by a pre-computed constant matrix, which is equivalent to a secret reconstruction and re-sharing process. Importantly, after this re-sharing step, the outputs are *degree-d* shares that embed the same secrets as the original *degree-*$2d$ input shares.

Note that the original BGW degree reduction [BGW88] is designed for Shamir's secret sharing and requires the generation of *degree-*$2d$ zero encodings. Later, Gennaro et al. [GRR98] simplified this degree reduction technique by asking each party to re-share their *degree-*$2d$ shares using a random *degree-d* polynomial (instead of using *degree-*$2d$ zero encodings). Based on the original BGW degree reduction [BGW88], Franklin and Yung [FY92] proposed a degree reduction technique for packed secret sharing that (still) requires the generation of *degree-*$2d$ zero encodings. However, we can also improve their constructions for packed secret sharing using the same ideas as Gennaro et al. [GRR98].

Degree Reduction from [FY92]. Franklin and Yung [FY92] proposed a degree reduction approach for the multiplication operation of packed secret sharing, which requires $n = 2d+1$ shares and can be used in multi-party computation [Yao86, GMW87]. In the following discussion, we use uppercase letters F_i and G_i to represent the secret shares and lowercase letters f and g to represent the corresponding polynomials. In the MPC setting, each party P_i, given the shares F_i and G_i, performs the share-wise multiplication $F_i \cdot G_i$. The result of the multiplication $F_i \cdot G_i$ corresponds to the *degree*-$2d$ polynomial $f(X) \cdot g(X)$. Next, each party P_i generates a random *degree-d* polynomial $R^{(i)}(X)$ that encodes $F_i \cdot G_i$ at support point u_i and values of zero at other support points $\{u_0, \ldots, u_{i-1}, u_{i+1}, \ldots\}$. Then, each party P_i generates shares $\left(R_j^{(i)}\right)_{j\in[n]}$ by evaluating polynomial $r^{(i)}(X)$ at support points $\{x_0, \ldots, x_{n-1}\}$ and re-shares their $F_i \cdot G_i$ by sending $R_j^{(i)}$ to other parties P_j for $j \in [n] \setminus i$. Now, each party P_i has received shares of all the share-wise multiplication results of $F_0 \cdot G_0, \ldots, F_{n-1} \cdot G_{n-1}$ from other parties. Finally, the parties reduce the *degree*-$2d$ share vector $\boldsymbol{F} * \boldsymbol{G} = (F_i \cdot G_i)_{i\in[n]}$ and get *degree-d* share vector $\tilde{\boldsymbol{F}} * \tilde{\boldsymbol{G}}$ by computing $\tilde{\boldsymbol{F}} * \tilde{\boldsymbol{G}} = (\boldsymbol{F} * \boldsymbol{G}) \cdot M_{\boldsymbol{F}*\boldsymbol{G}\to\tilde{\boldsymbol{F}}*\tilde{\boldsymbol{G}}}$ in a MPC manner, i.e., each party P_i uses their received shares $R_i^{(j)}$ encoding $F_j \cdot G_j$ for $j \in [n] \setminus i$ and the locally computed $R_i^{(i)}$ as the inputs for the MPC protocol to compute the multiplication with a specially constructed matrix $M_{\boldsymbol{F}*\boldsymbol{G}\to\tilde{\boldsymbol{F}}*\tilde{\boldsymbol{G}}}$. Essentially, the secrets embedded in $(F_i \cdot G_i)_{i\in[n]}$ are reconstructed and re-shared as *degree-d* shares through the multiplication operation with the matrix $M_{\boldsymbol{F}*\boldsymbol{G}\to\tilde{\boldsymbol{F}}*\tilde{\boldsymbol{G}}}$. The share-wise product $\tilde{\boldsymbol{F}} * \tilde{\boldsymbol{G}}$ corresponding to a *degree-d* polynomial $\tilde{f}(X) \cdot \tilde{g}(X)$ that embeds the same secrets as the *degree*-$2d$ polynomial $f(X) \cdot g(X)$ at the same support points. The details of the construction of the matrix $M_{\boldsymbol{F}*\boldsymbol{G}\to\tilde{\boldsymbol{F}}*\tilde{\boldsymbol{G}}}$ are presented in Suplementary Material A.3. Note that we can *directly* use this degree reduction technique to construct gadgets for multiplication operations, and the gadgets can be adapted (by adding the error propagation term as in Section 3.4) and proved to be combined resilient.

BGW-Based Compiler for PSS. We can use our linear transformation gadget **Gadget$_{\mathsf{LT}}$** (cf. Alg. 5), refresh gadget **Refresh** (cf. Alg. 6), share-wise addition **SWAdd** (cf. Alg. 1) and the share-wise multiplication gadget **SWMult** (cf. Alg. 2) to build a circuit compiler (we name it *BGW-based PSS compiler*) that is combined secure against $t \leq d + 1 - k$ probes, where d denotes the degree of our polynomial and k the number of embedded secrets, and e additive faults $\mathcal{F}^{+}$ using $n = 2d + e + 1$ shares. It can be easily extended to resist wire-independent faults $\mathcal{F}^{\mathsf{ind}}$ launched by adaptive adversaries using Lemma 2.

4.2 LaOla-Based Multiplication

In this part, we first revisit the *degree reduction* techniques from the original LaOla circuit compiler [BEF+23] which is designed for Shamir's secret sharing [Sha79], then we introduce a gadget **SplitRed** (cf. Alg. 7) which is adapted from [BEF+23] to support packed secret sharing. Finally, we analyze the security properties of our modified gadget **SplitRed**.

SplitRed Gadget. Berndt et al. [BEF+23] proposed the *LaOla* circuit compiler which contains a novel multiplication gadget that avoids the doubling of the polynomial's degree *after* the share-wise multiplication by reducing the degree of two input polynomial sharings, $(F_i)_{i\in[n]}$ and $(G_i)_{i\in[n]}$, from d to $\frac{d}{2}$ *before* the share-wise multiplication. This degree reduction technique presents a technical challenge as it must preserve leakage- and fault resilience as discussed in Section 2.1. For simplicity, we use the polynomial representation of the shares to describe the multiplication. In a nutshell, **SplitRed** splits the *degree-d* polynomials intended for multiplication, $f(X)$ and $g(X)$, into four *degree-d* polynomials $f'(X)$, $f''(X)$, $g'(X)$ and $g''(X)$, such that $f(X) = f'(X) + f''(X)$ and

$\tilde{g}(X) = g'(X) + g''(X)$, where $\tilde{f}(X)$ and $\tilde{g}(X)$ are *degree-*$\frac{d}{2}$ polynomials that embed the same secrets as $f(X)$ and $g(X)$, i.e., $\tilde{f}(0) = f(0)$ and $\tilde{g}(0) = g(0)$. This degree reduction is executed using the gadget **SplitRed** proposed in [BEF+23], which effectively *splits* the polynomial and *reduces* the degree of polynomials $f(X)$ and $g(X)$. After performing the **SplitRed** on $f(X)$ and $g(X)$, we compute four times the share-wise multiplication to get

$$h_0(X) = (f'(X) \cdot g'(X)), \qquad h_1(X) = (f'(X) \cdot g''(X)),$$
$$h_2(X) = (f''(X) \cdot g'(X)), \qquad h_3(X) = (f''(X) \cdot g''(X)),$$

which leads to four polynomials that can be share-wise added $\sum_{i=0}^{3} h_i(X)$ such that it holds

$$\sum_{i=0}^{3} h_i(X) = (f'(X) + f''(X)) \cdot (g'(X) + g''(X)) = \tilde{f}(X) \cdot \tilde{g}(X),$$

where $\tilde{f}(X) \cdot \tilde{g}(X)$ hides the secret $f(0) \cdot g(0)$. Next, we describe how to modify the gadget **SplitRed** to be compatible with packed secret sharing [FY92]. In the following discussion, we assume polynomials $f(X)$ and $g(X)$ embed two vectors of secrets instead of two single secrets, respectively. For ease of description, we only consider a polynomial $f(X)$ that hides secret vector $\boldsymbol{v} = (v_0, \ldots, v_{k-1}, v_k, \ldots, v_d)$[10] at support points $(u_0, u_1, \ldots, u_d)$ such that $f(u_i) = v_i$ for all $i \in [d+1]$ and generates n packed secret shares $\boldsymbol{F} = (F_0, \ldots, F_{n-1})$ by evaluating at support points $x_0, \ldots, x_{n-1}$. Our goal is to generate n pair of *new* packed secret shares $\boldsymbol{F}' = \left(F'_0, \ldots, F'_{n-1}\right)$ and $\boldsymbol{F}'' = \left(F''_0, \ldots, F''_{n-1}\right)$ such that $\tilde{f}(X) = f'(X) + f''(X)$, where $\tilde{\boldsymbol{F}} = \left(\tilde{F}_0, \ldots, \tilde{F}_{n-1}\right)$ are shares generated with the *degree-*$\frac{d}{2}$ polynomial $\tilde{f}(X) = \tilde{f}_0 + \tilde{f}_1 X + \ldots + \tilde{f}_{\frac{d}{2}} X^{\frac{d}{2}}$ such that $\tilde{f}(x_i) = \tilde{F}_i$ for all $i \in [n]$ and $\tilde{f}(u_i) = v_i$ for all $i \in [k]$, and $\boldsymbol{F}'$ and $\boldsymbol{F}''$ are two vector of shares generated using $f'(X)$ and $f''(X)$. The key step is to *securely* calculate the coefficients of polynomial $\tilde{f}(X)$ using shares $F_0, \ldots, F_{n-1}$.

Our Modification. Recall the fundamental properties of packed secret sharing introduced in [FY92]. Let $\boldsymbol{v}$ denote the secrets $v_0, \ldots, v_d$, U^{-1} denote $\mathsf{Van}^{-1}_{(d+1)\times(d+1)}(u_0, \ldots u_d)$, and V denote $\mathsf{Van}^{-1}_{(d+1)\times n}(x_0, \ldots x_{n-1})$. Namely, we generate secret shares $\boldsymbol{F} = (F_0, \ldots, F_{n-1})$ through the following steps:

$$\boldsymbol{F} = \boldsymbol{v} \cdot U^{-1} \cdot V, \tag{7}$$

and the coefficients of the *degree-*$\frac{d}{2}$ polynomial $\tilde{f}(X)$ through,

$$\tilde{\boldsymbol{f}} = \left(\tilde{f}_0, \ldots, \tilde{f}_{\frac{d}{2}}\right) = \left(v_0, \ldots, v_{\frac{d}{2}}\right) \cdot \mathsf{Van}^{-1}_{\left(\frac{d}{2}+1\right)\times\left(\frac{d}{2}+1\right)} \left(u_0, \ldots, u_{\frac{d}{2}}\right). \tag{8}$$

We obtain the *degree-*$\frac{d}{2}$ shares $\tilde{\boldsymbol{F}}$ by first unmasking the packed secret shares $\boldsymbol{F}$ belonging to a *degree-*d polynomial $f(X)$ while simultaneously re-masking them with respect to a *degree-*$\frac{d}{2}$ polynomial $\tilde{f}(X)$. This is possible, as we can present masking and unmasking as linear transformations using vector-matrix multiplication. More specifically, we realize the degree reduction in two steps. In the first *unmasking* step, we want to find a *linear transformation* between $\boldsymbol{F}$ and $\tilde{\boldsymbol{f}}$ using a $n \times \left(\frac{d}{2}+1\right)$ matrix $M_{\boldsymbol{F}\to\tilde{\boldsymbol{f}}}$, i.e.,

$$\tilde{\boldsymbol{f}} = \boldsymbol{F} \cdot M_{\boldsymbol{F}\to\tilde{\boldsymbol{f}}}. \tag{9}$$

[10]Note that $v_0, \ldots, v_{k-1}$ represent the k ($< \frac{d}{2}$) secrets to be encoded and $v_k, \ldots, v_d$ represent the randomness (or redundancy) used for packed secret sharing.

In the second *re-masking* step, we use the coefficient vector $\tilde{\boldsymbol{f}}$ obtained in the first step to construct polynomials $f'(X)$ and $f''(X)$ using our modified **SplitRed** gadget (cf. Alg. 7). Finally, we can generate the desired n pair of packed secret shares $\boldsymbol{F}'$ and $\boldsymbol{F}''$. Since $\boldsymbol{F}$ is a vector of secret shares and **SplitRed** gadget only involves linear operation, the above two steps can be done as secret-sharing operations while satisfying the combined resilience requirement (cf. Sec. 2.1). Due to space limitation, the details of constructing the $n \times \left(\frac{d}{2}+1\right)$ matrix $M_{\boldsymbol{F}\to\tilde{\boldsymbol{f}}}$ are presented in Supplementary Material A.2. For ease of reference in the ensuing discussion, we simply refer to this matrix as M when discussing the linear transformation of gadget **SplitRed**. So far, we have found the matrix M for linear transformation between $\boldsymbol{F}$ and $\tilde{\boldsymbol{f}}$. Next we only need to modify the constant term $\hat{\lambda}^i_j = \lambda_{0,j} + \sum_{l=d+1}^{n-1} \lambda_{l,j} \cdot x_i^l$ in the original **SplitRed** [BEF⁺23] to

$$\hat{\lambda}^i_j = M_{j,0} \cdot x_i^0 + \ldots + M_{j,\frac{d}{2}} \cdot x_i^{\frac{d}{2}} + E_{j,i} = \sum_{l=0}^{\frac{d}{2}} M_{j,l} \cdot x_i^l + E_{j,i}, \tag{10}$$

where the $\lambda_{i,j}$ are the elements of the matrix $\mathsf{Van}_{(d+1)\times n}(x_0,\ldots,x_{n-1})$. Furthermore, we can verify that $\sum_{l=0}^{\frac{d}{2}} M_{j,l} \cdot x_i^l$ guarantees the condition that $\tilde{f}(X) = f'(X) + f''(X)$ and $E_{j,i}$ is the error propagation (or preserving) term [SFES18] that would preserve the error coefficients of f (i.e., the coefficients of the *degree* $> d$ terms in $f(X)$) such that the faulted inputs $(F_i + \Delta_i)_{i\in[n]}$ can be detected. Specifically, $E_{j,i}$ is defined as follows:

$$E_{j,i} = \begin{cases} \lambda_{j,n-i-1} & \text{if } 0 \le i < e \\ 0 & \text{if } e \le i \le n-1 \end{cases}. \tag{11}$$

Further, thanks to the efficiencies of packed secret sharing, our modified gadget **SplitRed** (cf. Alg. 7) offers improved amortized cost relative to the original **SplitRed** in [BEF⁺23] (cf. Tab. 3).

In the following part, we prove the correctness and security of our modified **SplitRed** gadget (cf. Alg. 7).

Lemma 11. *The modified gadget* **SplitRed** *is* correct*, i.e.,* $f'(u_m) + f''(u_m) = f(u_m)$ *for* $m \in [d/2]$*, if there are no faults in the inputs or gadgets.*

Proof. Within this proof, we define the following notations for the matrix that is used to mask a secret $M = U^{-1}_{(d+1)\times(d+1)}(u_0,\ldots,u_d) \cdot V_{(d+1)\times n}(x_0,\ldots,x_{n-1})$. Similar to Section 2.2, we denote by M^{-1} the matrix, used for unmasking (cf. Supp. A.1). Moreover, we denote $M_{\boldsymbol{F}\to\tilde{\boldsymbol{f}}}$ by $M^{\boldsymbol{F}\to\tilde{\boldsymbol{f}}}$ for simplicity. We begin by inserting the values defined in

SplitRed:

$$f'(u_m) + f''(u_m) = \sum_{i=0}^{n} \left(F'_i \cdot M^{-1}_{i,m}\right) + \sum_{i=0}^{n} \left(F''_i \cdot M^{-1}_{i,m}\right) \tag{12}$$

$$= \sum_{i=0}^{n} \left(\left(\sum_{j\in[\frac{n}{2}]} \mathcal{F}'^{j}_{i} \right) M^{-1}_{i,m} \right) + \left(\sum_{i=0}^{n} \left(\sum_{j\in[\frac{n}{2}]} \mathcal{F}'^{j+\frac{n}{2}}_{i} \right) M^{-1}_{i,m} \right) \tag{13}$$

$$= \sum_{i=0}^{n} \left(\left(\sum_{j\in[\frac{n}{2}]} \hat{\lambda}^{i}_{j} F_j + g^{j}_{i} \right) M^{-1}_{i,m} \right) \tag{14}$$

$$+ \left(\sum_{i=0}^{n} \left(\sum_{j\in[\frac{n}{2}]} \hat{\lambda}^{i}_{j+\frac{n}{2}} F_{j+\frac{n}{2}} - g^{j+\frac{n}{2}}_{i} \right) M^{-1}_{i,m} \right) \tag{15}$$

$$= \sum_{i=0}^{n} \left(\left(\sum_{j\in[n]} \hat{\lambda}^{i}_{j} F_j \right) M^{-1}_{i,m} \right) + \left(\sum_{i=0}^{n} \left(\sum_{j\in[n]} \tilde{g}^{j}_{i} \right) M^{-1}_{i,m} \right), \tag{16}$$

where the second term in the last equation equals zero because $\left(\tilde{g}^{j}_{i}\right)_{j\in[n]}$ are zero encodings. This second term is a linear combination with $M^{-1}_{i,m}$, which corresponds to the unmasking step presented in Section 2.2. By definition of $\hat{\lambda}^{i}_{j}$ it follows:

$$= \sum_{i=0}^{n} \left(\sum_{j\in[n]} \hat{\lambda}^{i}_{j} F_j \right) M^{-1}_{i,m} \tag{17}$$

$$= \sum_{i=0}^{n} \left(\sum_{j\in[n]} \left(\sum_{l=0}^{\frac{d}{2}} M^{\boldsymbol{F}\to\tilde{\boldsymbol{f}}}_{j,l} \cdot x^{l}_{i} + E_{j,i} \right) \cdot F_j \right) M^{-1}_{i,m} \tag{18}$$

$$= \sum_{i=0}^{n} \left(\sum_{j\in[n]} \left(\sum_{l=0}^{\frac{d}{2}} M^{\boldsymbol{F}\to\tilde{\boldsymbol{f}}}_{j,l} \cdot x^{l}_{i} \right) \cdot F_j \right) M^{-1}_{i,m} + \sum_{i=0}^{n} \left(\sum_{j\in[n]} E_{j,i} \cdot F_j \right) M^{-1}_{i,m}, \tag{19}$$

where the last equation is equal to zero in case no fault is injected. Moreover,

$$= \sum_{i=0}^{n} \left(\sum_{j\in[n]} \sum_{l=0}^{\frac{d}{2}} M^{\boldsymbol{F}\to\tilde{\boldsymbol{f}}}_{j,l} \cdot F_j \cdot x^{l}_{i} \right) M^{-1}_{i,m}$$

$$= \sum_{i=0}^{n} \left(\sum_{j\in[n]} \sum_{l=0}^{\frac{d}{2}} \left(v_j \cdot U^{-1}_{j,l} \right) \cdot x^{l}_{i} \right) M^{-1}_{i,m}$$

holds true, due to the definition of $M^{\boldsymbol{F}\to\tilde{\boldsymbol{f}}}$ (cf. Supp. A.2). Note that $x^{l}_{i} = V_{l,i}$, where V

denotes the Vandermonde matrix with respect to $(x_0, \ldots, x_{n-1})$. By definition,

$$= \sum_{i=0}^{n} \left(\sum_{j \in [n]} \sum_{l=0}^{\frac{d}{2}} \left(v_j \cdot U_{j,l}^{-1} \right) \cdot x_i^l \right) M_{i,m}^{-1} \tag{20}$$

$$= \sum_{j=0}^{\frac{d}{2}} v_j \cdot \left(\sum_{i=0}^{n} M_{j,i} \cdot M_{i,m}^{-1} \right) \tag{21}$$

$$= \sum_{j=0}^{\frac{d}{2}} v_j \cdot \begin{cases} 1 \text{ if } j = m \\ 0 \text{ else} \end{cases} \tag{22}$$

$$= v_m = f(u_m). \tag{23}$$

We remark that $f'(X) + f''(X)$ is a *degree-*$\frac{d}{2}$ polynomial by construction. Note that $f'(X) + f''(X)$ is a re-shared version of $f(X)$ with respect to $M_{\boldsymbol{F} \to \tilde{\boldsymbol{f}}}$. The re-sharing is achieved by share-wise additions to zero encodings of *degree-*$\frac{d}{2}$ as well as *degree-d* zero encodings $\left(\hat{g}_i^j\right)_{i \in [n]}$ that cancel out, as they are added to $f'(X)$ and subtracted from $f''(X)$. Furthermore, the matrix $M_{\boldsymbol{F} \to \tilde{\boldsymbol{f}}}$ for re-sharing f is a $n \times \left(\frac{d}{2} + 1\right)$ matrix, setting the highest $\frac{d}{2}$ coefficients of $f(X)$ to zero, as shown in Supplementary Material A.2. □

Next, we demonstrate that the modified gadget **SplitRed** is $(d - k + 1)$-frNI by proving it is NI and fault-invariant.

Lemma 12 ([BEF+23, Lem. 5])**.** *The modified gadget* **SplitRed** *is* $(d + 1 - k) -$ NI*.*

In comparison to [BEF+23, Lem. 5], we only change the constants to multiply the shares with. This modification does not affect the proof. The complete proof is provided in the supplementary material in Lemma 18.

Lemma 13 ([BEF+23, Thm. 17])**.** *The modified gadget* **SplitRed** *is fault-invariant with respect to* $\mathcal{F}^+$*.*

Proof. In **SplitRed**, all the operations are share-wise addition or multiplication with constant values. Because additive faults $\xi \in \mathcal{F}^+$ can be described with a fault value $a \in \mathbb{F}$ such that the faulted value is $\xi(x) = x + a$, and operation $+$ is commutative, we can shift the fault a from the input to the output and vice versa. Furthermore, the multiplication of a constant s can be seen as s-wise addition. Hence, we can push the faults injected into **SplitRed** to inputs or outputs. □

Compression Gadget. So far, we constructed the **SplitRed** gadget, which splits a *degree-d* polynomial $f(X)$ into two *degree-d* polynomials $f'(X)$ and $f''(X)$, such that their addition results in a *degree-*$\frac{d}{2}$ polynomial corresponding to the same secret values as $f(X)$. Doing this twice, we obtain polynomials $f'(X)$ and $f''(X)$, as well as $g'(X)$ and $g''(X)$, encoding the same secrets as $f(X)$ and $g(X)$, respectively. To obtain our desired multiplication gadget, it is left to share-wise multiply and add the results together, since we can rewrite

$$f(X) \cdot g(X) = f'(X) \cdot g'(X) + f'(X) \cdot g''(X) + f''(X) \cdot g'(X) + f''(X) \cdot g''(X)$$

Hence, to obtain our desired multiplication gadget, we integrate the gadget **SplitRed** with share-wise multiplication gadget **Mult** (cf. Alg. 2). The next crucial step involves securely summing the multiplication results of these splitted polynomials, i.e., to perform the additions in $f'(X) \cdot g'(X) + f'(X) \cdot g''(X) + f''(X) \cdot g'(X) + f''(X) \cdot g''(X)$. This can be achieved with the help of the **SWComp**, presented in Algorithm 8.

Algorithm 7 **SplitRed** for $n = d + e + 1$ and $k < \frac{d}{2}$

Input : *Degree-d* shares of $\boldsymbol{v} = (v_0, \ldots, v_d)$ 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 $f_j = f'_j + f''_j$ for $j \in \left[\frac{d}{2}+1\right]$

```
1  initialize $(F'_i)_{i\in[n]}$, $(F''_i)_{i\in[n]}$
2  for $j \in \left[\frac{n}{2}\right]$ do
3      $\left(\hat{g}_i^j\right)_{i\in[n]} \leftarrow \mathsf{ZEnc}_n^d$
4  for $j \in \left[\frac{n}{2}\right]$ do
5      $\left(\tilde{g}_i^j\right)_{i\in[n]} \leftarrow \mathsf{ZEnc}_n^{\frac{d}{2}}$
6      $\left(g_i^j\right)_{i\in[n]} \leftarrow \left(\tilde{g}_i^j\right)_{i\in[n]} + \left(\hat{g}_i^j\right)_{i\in[n]}$
7  for $j \in \left[\frac{n}{2}\right]$ do
8      for $i \in [n]$ do
9          $\mathcal{F}_i'^j \leftarrow \hat{\lambda}_j^i \cdot F_j$
10 for $j \in \left[\frac{n}{2}\right]$ do
11     $\left(\mathcal{F}_i^j\right)_{i\in[n]} \leftarrow \left(\mathcal{F}_i'^j\right)_{i\in[n]} + \left(g_i^j\right)_{i\in[n]}$
12 for $j \in \left[\frac{n}{2}\right]$ do
13     $(F'_i)_{i\in[n]} \leftarrow (F'_i)_{i\in[n]} + \left(\mathcal{F}_i^j\right)_{i\in[n]}$
14 for $j \in \left[\frac{n}{2}\right]$ do
15     $\left(\tilde{g}_i^{j+\frac{n}{2}}\right)_{i\in[n]} \leftarrow \mathsf{ZEnc}_n^{\frac{d}{2}}$
16     $\left(g_i^{j+\frac{n}{2}}\right)_{i\in[n]} \leftarrow \left(\tilde{g}_i^{j+\frac{n}{2}}\right)_{i\in[n]} - \left(\hat{g}_i^j\right)_{i\in[n]}$
17 for $j \in \left[\frac{n}{2}\right]$ do
18     for $i \in [n]$ do
19         $\mathcal{F}_i'^{j+\frac{n}{2}} \leftarrow \hat{\lambda}_{j+\frac{n}{2}}^i \cdot F_{j+\frac{n}{2}}$
20 for $j \in \left[\frac{n}{2}\right]$ do
21     $\left(\mathcal{F}_i^{j+\frac{n}{2}}\right)_{i\in[n]} \leftarrow \left(\mathcal{F}_i'^{j+\frac{n}{2}}\right)_{i\in[n]} + \left(g_i^{j+\frac{n}{2}}\right)_{i\in[n]}$
22 for $j \in \left[\frac{n}{2}\right]$ do
23     $(F''_i)_{i\in[n]} \leftarrow (F''_i)_{i\in[n]} + \left(\mathcal{F}_i^{j+\frac{n}{2}}\right)_{i\in[n]}$
24 return $(F'_i)_{i\in[n]}$, $(F''_i)_{i\in[n]}$
```

Algorithm 8 **SWComp** for $n = d + e + 1$ and $k \leq d$

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  $(Q_i)_{i\in[n]} \leftarrow \mathsf{sZEnc}_{\boldsymbol{n}}^{\boldsymbol{d}}$(cf. Alg. 3)
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 **SWComp** (cf. Alg. 8) follows directly by the fact that $\mathbf{sZEnc}_n^d$ (cf. Alg. 4) outputs a zero encoding and the correctness of share-wise addition. In a nutshell, security is ensured because we take share-wise multiplications as inputs and perform a secure re-randomization (or re-sharing) while adding the inputs share-wise. We present a detailed security analysis of **SWComp** in Supplementary Material D.

Multiplication Gadget for k Inputs. Now we have everything together for a detailed description of adapting the multiplication gadget [BEF+23, Alg. 6] to packed secret sharing [FY92]. We use **SplitRed** (cf. Alg. 7 in Sec. 4.2) to construct the multiplication gadget $\mathbf{Mult_{k-Input}}$ (cf. Alg. 9). Recall that we utilize the degree reduction technique presented in [BEF+23]. Namely, the multiplication algorithm obtains two *degree-d* shares $(F_i)_{i\in[n]}$ and $(G_i)_{i\in[n]}$ as inputs. Then, **SplitRed** is performed on both inputs, which results in new *degree-d* sharings $(F'_i)_{i\in[n]}$, $(F''_i)_{i\in[n]}$, $(G'_i)_{i\in[n]}$, $(G''_i)_{i\in[n]}$, such that $(F'_i)_{i\in[n]} + (F''_i)_{i\in[n]}$ (resp. $(G'_i)_{i\in[n]} + (G''_i)_{i\in[n]}$) is a *degree-$\frac{d}{2}$* sharing of the same k secrets of $(F_i)_{i\in[n]}$ (resp. $(G_i)_{i\in[n]}$). Afterward, a share-wise multiplication is performed on each combination. In the end, we can add the multiplication results together. The correctness directly follows from the correctness proof of **SplitRed** in Section 4.2, and the security analysis is given in Lemma 14 and Lemma 15.

Algorithm 9 $\mathbf{Mult_{k-Input}}$ for $n = d + e + 1$ and $k < \frac{d}{2}$

```
Input : Degree-d shares of $\boldsymbol{v} = (v_0, \dots, v_d)$ as $(F_i)_{i\in[n]}$ and degree-d shares of $\boldsymbol{v}' = (v'_0, \dots, v'_d)$ as $(G_i)_{i\in[n]}$
Output: Degree-d shares of $(v_0 \cdot v'_0, \dots, v_{k-1} \cdot v'_{k-1})$ as $(Q_i)_{i\in[n]}$
1 $\left((F'_i)_{i\in[n]}, (F''_i)_{i\in[n]}\right) \leftarrow$ SplitRed $\left((F_i)_{i\in[n]}\right)$
2 $\left((G'_i)_{i\in[n]}, (G''_i)_{i\in[n]}\right) \leftarrow$ SplitRed $\left((G_i)_{i\in[n]}\right)$
3 $(H^0_i)_{i\in[n]} \leftarrow$ SWMult $\left((F'_i)_{i\in[n]}, (G'_i)_{i\in[n]}\right)$
4 $(H^1_i)_{i\in[n]} \leftarrow$ SWMult $\left((F'_i)_{i\in[n]}, (G''_i)_{i\in[n]}\right)$
5 $(H^2_i)_{i\in[n]} \leftarrow$ SWMult $\left((F''_i)_{i\in[n]}, (G'_i)_{i\in[n]}\right)$
6 $(H^3_i)_{i\in[n]} \leftarrow$ SWMult $\left((F''_i)_{i\in[n]}, (G''_i)_{i\in[n]}\right)$
7 $(Q_i)_{i\in[n]} \leftarrow$ SWComp $\left((H^0_i)_{i\in[n]}, (H^1_i)_{i\in[n]}, (H^2_i)_{i\in[n]}, (H^3_i)_{i\in[n]}\right)$
8 return $(Q_i)_{i\in[n]}$
```

Lemma 14 ([BEF+23, Thm. 16])**.** *The gadget* $\mathbf{Mult_{k-Input}}$ *is* $(d+1-k)$-SNI.

Proof (sketch). The only difference between our gadget $\mathbf{Mult_{k-Input}}$ and the original multiplication gadget **Mult** in [BEF+23] is that we modified the public term $\hat{\lambda}^i_j$ (line 9 and 19) in **SplitRed** (cf. Alg. 7). Therefore, we only need to prove that our modified **SplitRed** (cf. Alg. 7) has the same and necessary security property, which is $(d+1-k)$-NI, as the original one in [BEF+23]. We prove that our modified **SplitRed** (cf. Alg. 7) satisfies $(d+1-k)$-SNI in Lemma 12. □

Lemma 15 ([BEF+23, Thm. 24])**.** *The gadget* $\mathbf{Mult_{k-Input}}$ *is e-f-robust.*

Proof (sketch). Since we only modified the **SplitRed** (cf. Alg. 7) gadget inside $\mathbf{Mult_{k-Input}}$ compared to the original **Mult** in [BEF+23]. We only need to show that our modified **SplitRed** (cf. Alg. 7) has the necessary security, which is given in Lemma 24, as the original **SplitRed** in [BEF+23]. In addition, we give another (slightly simpler) way to prove the e-f-robustness of $\mathbf{Mult_{k-Input}}$ in Lemma 25. □

LaOla-Based Compiler for PSS. We can use our multiplication gadget $\textsf{Mult}_{\textsf{k-Input}}$ (cf. Alg. 9), refresh gadget **Refresh** (cf. Alg. 6), linear transformation gadget $\textsf{Gadget}_{\textsf{LT}}$ (cf. Alg. 5), the share-wise addition **SWAdd** (cf. Alg. 1), and share-wise multiplication gadget **SWMult** (cf. Alg. 2) to build a circuit compiler (we name it *LaOla-based PSS compiler*) secure against $t < d + 1 + k$ probes, where d denotes the degree of the polynomial and k the number of embedded secrets. Furthermore, our compiler is simultaneously secure against e additive fault $\mathcal{F}^+$ using $n = d + e + 1$ shares. Moreover, our BGW-based PSS compiler and LaOla-based PSS compiler that are secure against the additive fault $\mathcal{F}^+$ can be extended to resist wire-independent faults $\mathcal{F}^{\textsf{ind}}$ launched by adaptive adversaries using Lemma 2. Details are provided in Supplementary Material E.

5 Exponentiation Gadget

The Frobenius optimization is a way to reduce the computational cost of certain exponentiations and was first presented by Prouff and Roche [PR11]. In the following, we first recap their method which is based on Shamir's secret sharing [Sha79]. Then, we demonstrate how it can be applied to packed secret sharing [FY92] for efficient gadgets. Finally, we analyze the leakage- and fault-resilience of our gadget **Exp** (cf. Alg. 10).

Frobenius Optimization. Let $p := |\mathbb{F}|$ denote the field characteristic of the underlying arithmetic circuit. Using the method of Prouff and Roche, we can compute the operation $y \mapsto y^p$ very efficiently. These operations are particularly useful when considering, for example, the AES SubBytes transformation. This technique requires the following condition: the support points α_i of our polynomial fulfill the *stability over Frobenius automorphism* property, which says that for every support point α_i, there exists a point $\alpha_{j(i)}$ such that $\alpha_{j(i)} = \alpha_i^p$.

In the case of Shamir's secret sharing, we can derive the following. Let $p \in \mathbb{F}[X]$ be a polynomial of degree d embedding one secret s_0, i.e.

$$p(x) = s_0 + \sum_{l=1}^{d} a_l x^l.$$

Now, consider a polynomial embedding the p-th power of secret s_0, i.e.

$$p'(x) = s_0^p + \sum_{l=1}^{d} a_l^p x^l.$$

Given that the coefficients $a_1, \ldots, a_d$ of p are distributed uniformly at random, the coefficients $a_1^p, \ldots, a_d^p$ are also distributed uniformly at random. Hence, we can obtain a sharing $(F_0', \ldots, F_{n-1}')$ of s_0^p from the sharing $(F_0, \ldots, F_{n-1})$ of s_0 through share-wise exponentiation of p, as follows:

$$F_i^p = (s_0 + \sum_{l=1}^{d} a_l \alpha_i^l)^p = s_0^p + \sum_{l=1}^{d} a_l^p \alpha_i^{lp} = s_0^p + \sum_{l=1}^{d} a_l^p \alpha_{j(i)}^l = F_i',$$

where $(a+b)^p = a^p + b^p$ for $a, b \in \mathbb{F}$, since we are in a field of characteristic p.

With these preliminaries, we can adapt the Frobenius optimization to packed secret sharing. The corresponding gadget is presented in Algorithm 10.

Algorithm 10 Exp using Frobenius optimization for $n = d + e + 1$ and $k \leq d$

Input : *Degree-d* shares of $\boldsymbol{v} = (v_0, \dots, v_d)$ as $(F_i)_{i \in [n]}$
Output : *Degree-d* shares of $(v_0^p, \dots, v_d^p)$ as $(F'_i)_{i \in [n]}$

```
1 for $i \in [n]$ do
2     $F'_i \leftarrow F_i^p$
3 return $(F'_i)_{i \in [n]}$
```

Frobenius Optimization in Packed Secret Sharing. In this section, we prove the correctness of Algorithm 10. We consider the secret vector $\boldsymbol{v} = (v_0, \dots, v_d)$ embedded in the shares $(F_i)_{i \in [n]}$. Recall that by definition of our packed secret sharing,

$$(F_0, \dots F_{n-1}) = \boldsymbol{v} \cdot U^{-1} \cdot V,$$

where U^{-1} denotes the inverse of a Vandermonde matrix with respect to the support points $(u_0, \dots, u_d)$, and V is the Vandermonde matrix considering the support points $(x_0, \dots x_{n-1})$. Assuming that $(u_0, \dots, u_d)$ and $(x_0, \dots x_{n-1})$ fulfill the stability over Frobenius automorphism property with respect to disjoint sets, the above exponentiation gadget (cf. Alg. 10) is correct. In more detail, assume that for every u_i there exists a point $u_{j(i)}$ such that $u_{j(i)} = u_i^p$, and for every x_i there exists a point $x_{j(i)}$ such that $x_{j(i)} = x_i^p$. Additionally, we suppose that $\{u_i, u_{j(i)} | i \in [n]\} \cup \{x_i, x_{j(i)} | i \in [n]\} = \emptyset$. Costes and Stam [CS21] showed that such disjoint sets do exist and can be used e.g., for AES. Note that in the following description, we assume that vector $\boldsymbol{v}$ and matrices U^{-1}, V are already necessarily padded with zeros to ensure the feasibility of matrix multiplication without affecting the operation correctness. Then, by setting $M = U^{-1} \cdot V$, we obtain

$$F'_i = F_i^p = \left(\sum_{l=0}^{n-1} v_l \cdot M_{l,i} \right)^p = \sum_{l=0}^{n-1} v_l^p \cdot M_{l,i}^p,$$

where the last equation holds, because we are in a field of characteristic p, and thus $(a+b)^p = a^p + b^p$ for $a, b \in \mathbb{F}$. Let us examine the matrix M in more detail. By definition, $M_{l,i} = \sum_{m=0}^{n-1} U_{l,m}^{-1} \cdot V_{m,i}$. Since we are operating in a field of characteristic p, we have $M_{l,i}^p = \sum_{m=0}^{n-1} \left(U^{-1}\right)_{l,m}^p \cdot V_{m,i}^p$. Given that V is a Vandermonde matrix, raising each entry of V to the power of p results in a Vandermonde matrix with respect to the support points $(x_0^p, \dots, x_{n-1}^p)$, which exist due to stability over the Frobenius automorphism. We still need to show that the same property holds for U^{-1}. By definition, $U \cdot U^{-1} = \mathbf{I}$, where $\mathbf{I}$ denotes the identity matrix. Let $\lambda_{j,k}$ denote the element in the j-th row and k-th column of U^{-1}. Then,

$$\sum_{j=0}^{d} u_{i,j} \cdot \lambda_{j,k} = \begin{cases} 1 \text{ if } i = k \\ 0 \text{ otherwise.} \end{cases}$$

It follows directly, that the matrix $\left(\lambda_{i,j}^p\right)_{i \in [d+1], j \in [d+1]}$ is the inverse of the Vandermonde matrix $U^p := \mathsf{Van}(u_0^p, \dots, u_{n-1}^p)$, since

$$\sum_{j=1}^{d} u_{i,j}^p \cdot \lambda_{j,k}^p = \left(\sum_{j=1}^{d} u_{i,j} \cdot \lambda_{j,k} \right)^p = \begin{cases} 1^p \text{ if } i = k \\ 0^p \text{ otherwise.} \end{cases}$$

Due to the stability over the Frobenius automorphism of u_i and x_i with respect to disjoint sets, we conclude that $(F'_0, \dots, F'_{n-1})$ is a valid encoding of $(v_0^p, \dots, v_k^p)$.

Note that in both Shamir's secret sharing and packed secret sharing, we obtain valid shares of the secret(s), but in polynomials with respect to different support points, namely a_i^p instead of a_i. Fortunately, we can use the linear transformation gadget **Gadget$_{\mathsf{LT}}$** (cf. Alg. 5) to *shift* the support points from a_i^p back to a_i. Otherwise, we can continue calculations with respect to the new support points a_i^p. Moreover, when aiming to multiply the shares F_i' with shares with respect to F_i' or F_i, we need to perform a refresh operation in order to satisfy the SNI property and maintain composability. In this case, the linear transformation gadget **Gadget$_{\mathsf{LT}}$** (cf. Alg. 5) can be used to both *shift* and *refresh* simultaneously without increasing the overall complexity.

Security Analysis. In a field of characteristic 2, we can use the share-wise multiplication gadget Algorithm 2 to compute the squaring without assuming the existence of a squaring gate. However, in the more general case of a finite field with characteristic p, we need to assume the existence of a gate evaluating $x \mapsto x^p$. With this assumption, the NI property follows directly, as we only perform share-wise operations. Consequently, each of the $t_{\text{int}} \leq t$ internal probes can be perfectly simulated with at most t_{int} shares of the input sharing. Furthermore, fault-invariance (cf. Def. 7) with respect to additive faults $\xi \in \mathcal{F}^+$ holds true, as $(F_i + \xi)^p = F_i^p + \xi^p$ in a field of characteristic p. Moreover, recall the concept of fault-robustness (cf. Def. 3): if we have s_{input} faults injected into the input and s_{int} faults injected during the computation of the gadget, we want to guarantee that either only s_{input} outputs are affected or the outputs contain high-order coefficients that can be detected. Due to the error-correcting property of packed secret sharing and the fact that we perform only share-wise operations, we can ensure with a high probability that any inserted error results in a higher-degree polynomial, which can be detected.

6 Fault Detection

In contrast to duplicated Boolean masking, which requires fault detection or correction within the gadgets to guarantee security (cf. [DN20, FRSG22, FGM+23]), our approach does not face this limitation. Due to error propagation [SFES18], we can first calculate the whole circuit and perform only one error detection right before the final decoding.

Our fault detection algorithm (cf. Alg. 13) can be summarized as follows. We hide the secrets embedded in our input polynomial $f(X) = \sum_i f_i X^i$, represented by the shares $(F_i)_{i\in[n]}$, by adding a completely random polynomial of degree d. Specifically, we perform a special type of the zero-encoding operation, **RandomEnc$_n^d$** (cf. Alg. 11), which encodes k random values, $r_0, \ldots, r_{k-1} \leftarrow_\$ \mathbb{F}^k$ (instead of encoding k zeros as in Section 3.5) into the random polynomial. Similar to a **Refresh**, we repeat this randomization $d+1-k$ times. Namely, we add $d+1-k$ shares $\left(R_i^{(j)}\right)_{i\in[n]}, j \in [d+1-k]$ of independent and uniformly random polynomials $R^{(j)}(X)$ to our input shares $(F_i)_{i\in[n]}$ in a share-wise manner. This procedure is called **RandomRefresh$_n^d$** (cf. Alg. 12). After this **RandomRefresh$_n^d$**, we recalculate the highest coefficients $g_i, i \in \{d+1, \ldots, n-1\}$ of the resulting randomized polynomial $g(X) = f(X) + \sum_{j=0}^{d+1-k} R^{(j)}(X)$, using our extended matrix V_n (cf. Supp. A.1). Finally, we check if any of these coefficients are zeros. In case there exists a non-zero coefficient, we abort[11]. We repeat the above steps e times to guarantee fault robustness (cf. Def. 3) and call the resulting algorithm **FaultDetect** (cf. Alg. 13). For further details on the algorithms, we refer to Supplementary Material F.

Intuitively, this approach does not reveal anything about the secrets underlying the polynomial $f(X)$, as we add $d+1-k$ polynomials with independent and uniformly random

[11]If there does not exist an implementation of the abort state, we assume the existence of a public vector $\perp = (\perp_0, \ldots, \perp_e)$, acting as abort flags, similar to [DN20]. This vector is initially set to zeros, and all values $\perp_0, \ldots, \perp_e$ are set to 1 if there is an abort.

coefficients in $\mathsf{RandomRefresh}_n^d$. Hence, these secrets are completely randomized and due to the probing threshold $t < d+1-k$, an adversary is not able to obtain any information about the underlying secrets during the computation of $(G_i)_{i\in[n]} := \left(F_i + \sum_{j\in[d+1-k]} R_i^{(j)}\right)_{i\in[n]}$. Due to our setting for combined security, the adversary can insert e faults. However, if a fault $\xi \in \mathcal{F}^+$, such that $\xi(x) = x + a$, is inserted in $\mathsf{RandomRefresh}_n^d$, one of the following two cases occurs:

- Either we can see the fault as if one of the secrets r_i embedded in the shares $\left(R_i^{(j)}\right)_{i\in[n]}$ for $j \in [d+1-k]$ is shifted by fault ξ. However, r_i was chosen uniformly at random. Hence, the result $r_i + a$ is distributed uniformly random as well.
- Or the degree of the underlying polynomial of the resulting shares $\left(R_i^{(j)}\right)_{i\in[n]}$, $j \in [d+1-k]$, increases.

In both cases, the underlying polynomial of shares $(G_i)_{i\in[n]}$ only contains the randomized secrets, which are independent of the secrets embedded in $(F_i)_{i\in[n]}$. Moreover, the error detection **FaultDetect** remains correct, as it is independent of the underlying secrets as well, because of fault robustness (cf. Def. 3).

Due to this independence, it is sufficient to consider fault resilience for the remaining steps, which is given, as we repeat the above randomization and coefficient reconstruction steps for e times in **FaultDetect**. Since all involved polynomials are of degree d in case no fault was inserted, an adversary must insert at least one fault to increase the degree of the underlying polynomial. Given the faulting threshold of e, repeating the procedure e times is sufficient. Detailed proofs for all mentioned properties can be found in Supplementary Material F. Concrete numbers for the Boolean circuit complexity of one error detection/ correction gadget for our LaOla-based PSS approach, our BGW-based PSS approach, as well as the duplicated Boolean masking approaches of [DN20] and [FRSG22, FGM+23] are provided in Table 5. We consider 16 inputs of elements in $\mathbb{F}_{2^8}$ for polynomial masking, and 16 bit secrets for the duplicated Boolean masking approaches. The numbers for our LaOla-based PSS approach are higher than those for the BGW-based PSS approach due to the choice of n. This is necessary to satisfy the condition $k < \frac{d}{2}$, which is required for correctness of the **SplitRed** gadget. Since [DN20] and [FRSG22, FGM+23] did not specify the majority function used for error detection/ correction, we use insertion sort, which is known to be efficient for a small number of faults e (cf. [CLRS22, YYG11]). For further details, we refer to Supplementary Materials G.2.

Table 5: Boolean circuit complexity for error-detection/correction of our *BGW-based PSS* and *LaOla-based PSS* compiler, and the duplicated Boolean masking [DN20, FRSG22, FGM+23] with respect to elements in $\mathbb{F}_{2^8}$. The security parameters are t probes and e faults with $e = t$ and $e, t \in [1, 9]$. The best complexity for achieving combined security is marked in bold. Please note that [FRSG22] shows that *NINA** is not combined secure.

$t = e$	Error Detection/Correction			
	LaOla-PSS (this work)	BGW-PSS (this work)	CINI-MINIs [FGM+23]	NINA* [DN20]
1	650896	59024	**16896**	224
2	1338388	137304	**117120**	816
3	2063106	**285910**	419328	1984
4	2825068	**526706**	1094400	3920
5	3624292	**884004**	2365440	6816
6	4460796	**1384564**	4507776	10864
7	5334598	**2057594**	7848960	16256
8	6245716	**2934750**	12768768	23184

7 Application

7.1 Secure AES Operations

In this section, we consider the block cipher AES-128 with 16 elements of the Galois field $\mathsf{GF}(2^8)$. The encryption process involves ten rounds of the following sequence of operations: First, the operation AddRoundKey is used to add a round key of length 128 to the internal state. The round keys involved here are derived from the secret key through key expansion. Next, we apply SubBytes, which applies the function $y \mapsto y^{254}$ to each byte of the internal state, followed by the affine transformation

$$\begin{aligned}\tau(y) =& 0\text{x}63 + (0\text{x}05 \cdot y) + (0\text{x}09 \cdot y^2) + (0\text{xf}9 \cdot y^4) + (0\text{x}25 \cdot y^8) \\ &+ (0\text{xf}4 \cdot y^{16}) + (0\text{x}01 \cdot y^{32}) + (0\text{xb}5 \cdot y^{64}) + (0\text{x}8\text{f} \cdot y^{128}).\end{aligned}$$

For a detailed description, please refer to [PR11]. Then the linear transformation ShiftRows is performed. When considering our 16 elements as 4×4-matrix, ShiftRows consists of a permutation of (some of) the rows, namely

$$A = \begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{pmatrix} \mapsto A' = \begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_5 & a_6 & a_7 & a_4 \\ a_{10} & a_{11} & a_8 & a_9 \\ a_{15} & a_{12} & a_{13} & a_{14} \end{pmatrix}$$

The last transformation used in a round (except from the last round) is called MixColumns. This transformation is applied to each column of the matrix A' at a time. Each column is multiplied by a fixed Maximum Distance Separable (MDS) matrix Z to produce the transformed state A''.

AES in Packed Secret Sharing. When applying our packed secret-sharing approach (cf. Sec. 2.2), we can perform the above transformations as follows. We choose the number of secrets $k = 16$ and encode these 16 secrets into one group of packed secret sharings with n shares as internal state shares. Note that we assume that the round keys are also encoded. The operation AddRoundKey can be achieved by a share-wise addition of the internal state shares and the round key shares.

SubBytes considers multiplication of the internal states with themselves, as well as a polynomial evaluation, $\tau(y)$, which can be realized by gadget $\textbf{Mult}_{\textbf{k}-\textbf{Input}}$ (cf. Alg. 9) together with matrix multiplication and share-wise addition. The linear transformation ShiftRows can be obtained using our gadget **Permutation** (cf. Alg. 5) with the corresponding function $\ell_{i,j}^{\varphi_0,\ldots,\varphi_d}$ (cf. Tab. 4).

Finally, for the MixColumns operation, we first split our internal state into four columns. Specifically, we consider four resharings of A', each containing one column of sensitive values $(v_0, \ldots, v_3), (v_4, \ldots, v_7), (v_8, \ldots, v_{11})$ and $(v_{12}, \ldots, v_{15})$, respectively. This can be performed by gadget **Permutation** (cf. Alg. 5) taking A' and a zero encoding as inputs. Subsequently, we perform the matrix multiplication with the MDS matrix and share-wise add all four resulting secret shares together to obtain A''. It remains to show how to efficiently compute the squaring required in SubBytes. We realize the optimization by adapting the Frobenius optimization (cf. Sec. 5) to packed secret sharing.

7.2 Complexity Analysis

We calculate the number of gadgets needed in the operations of AES similarly to [PR11, ABEO24], and summarize the results in Table 6. We denote the number of polynomials needed to evaluate one round of AES-128 by B. For example, using Shamir's secret sharing [Sha79], we need 16 secrets in $\mathsf{GF}(2^8)$, where each secret is encoded into one polynomial, resulting in $B = 16$ polynomials. Using the double-sharing approach presented by Arnold et al. [ABEO24], which encodes two secrets into one polynomial, we obtain $B = 8$ polynomials. Our method sets $k = 16$, allowing us to encode 16 secrets into one polynomial, resulting in $B = 1$.

When considering security against t-probing attacks, Shamir's secret sharing requires a *degree-t* polynomial, the double-sharing approach of Arnold et al. uses a *degree-*$(t+1)$ polynomial, and our approach uses a *degree-*$(t+k-1)$ polynomial.

Table 6: The number of gadget (providing probing security t) calls required to mask the operations of AES when operating on $B \in \{\{16, 8\} | \{1\}\}$ polynomials.

	AddRoundKey	$y \mapsto y^{254}$	τ	MixColumns	ShiftRows
Add(t)	B		$7B$	$\frac{15}{4}B \mid 15$	
Affine(t)			$8B$	$\frac{3}{4}B \mid 3$	
Square(t)		$7B$	$7B$		
Mult(t)		$4B$			

In Figure 1 and Table 6, we depict the complexity for one round of AES-128 with respect to various combined resilient compilers, the original *LaOla* compiler [BEF+23], the *double-sharing* compiler [ABEO24], our *BGW-based PSS* compiler (cf. Sec. 4.1), and our *LaOla-based PSS* compiler (cf. Sec. 4.2).

We set the number of fault injections e and the probing security t equal each other, i.e., $e = t$. Our *BGW-based* compiler outperforms the original *LaOla* compiler in one round of AES for $t = 5$ and the double-sharing approach of Arnold et al. for $t = 7$ in terms of the number of field operations. Regarding randomness complexity, our *BGW-based* compiler outperforms the BGW approach with respect to Shamir's secret sharing, the original *LaOLa* compiler, as well as the double-sharing approach for $t = 2$. Our *LaOla-based* compiler outperforms our *BGW-based* compiler with respect to both the number of operations and randomness complexity for $t = 13$.

7.3 Comparison to Duplicated Boolean Masking

In this section, we compare our packed secret sharing based masking approach (cf. Sec. 4) to the duplicated Boolean masking [DN20, FRSG22, FGM+23].

For comparison, we provide the concrete numbers of the Boolean gates required to perform a single multiplication and error detection/correction, followed by a brief discussion.

As discussed in [DN20, FRSG22, FGM+23], Boolean masking requires fault detection or correction mechanisms to be integrated during the computation. In contrast, our PSS-based approach only needs to detect faults right before the decoding phase due to error propagation [SFES18, BEF+23]. However, our fault detection algorithm, **FaultDetect** (cf. Alg. 13), can also be used to detect faults earlier within the circuit.

In the following analysis, we use the error detection approach from [DN20], as [FRSG22, FGM+23] does not provide a specific error detection or correction mechanism. However, Boolean masking with error detection is vulnerable to Statistical Ineffective Fault Attacks (SIFA) (cf. [FRSG22]). In order to circumvent this issue, more costly error correction mechanisms are used by [FRSG22, FGM+23]. This is the reason why we consider concrete numbers for complexity for error correction as well.

In Boolean masking, $n = (e+1) \cdot (t+1)$ shares are required for error detection, and $n = (2e+1) \cdot (t+1)$ shares are needed for error correction. As suggested by [FRSG22, FGM+23], errors can be corrected by duplicating each sharing of size $t+1$ for $2e+1$ times, then sorting the (faulted) duplicated shares and selecting the median. Since [FRSG22, FGM+23] does not specify a concrete sorting algorithm, we use the insertion sort algorithm for this purpose, given that it performs efficiently with small values of e (cf. [CLRS22, YYG11]).

Finally, although we only utilize error detection in our work, it is important to note that our PSS-based masking approach is inherently resistant to SIFA attacks. This is due to the fault robustness property (cf. Def. 3), which is specifically designed to counteract ineffective faults by ensuring that any faults lead to corrupted outputs [BEF+23].

For security parameters t and e, we consider elements in $\mathbb{F}_{2^8}$ that can be probed or faulted in our PSS-based approaches, where we set the parallelization factor $k = 16$ to be consistent with the AES-128 application in Section 7, where k is the number of secrets embedded in one polynomial using packed secret sharing (cf. Sec. 2.2). Since our operations are performed over the field $\mathbb{F}_{2^8}$, we transform these operations into Boolean operations and calculate the necessary number of Boolean gates. We use the Boolean circuits presented in [BDST20] for this transformation. As our approach does not require any error detection during or after the multiplication gadget (cf. Sec. 4), we calculate the numbers of Boolean gates without the detection gadget **FaultDetect** (and compute the numbers of the multiplication together with a detection gadget in brackets).

In contrast, we consider elements in $\mathbb{F}_2$ for the duplicated Boolean masking [DN20, FRSG22, FGM+23], where we run the multiplication gadget for 16 times to compute on 16 secrets as well. Note that here, a secret refers to a bit, not an element in $\mathbb{F}_{2^8}$. As the operations are also binary operations, we derive the gate counts by considering all operations within the multiplication gadget and the detection/correction gadgets.

For $t = e = 1$, we need to perform 1408 operations for one multiplication in [DN20] (with error detection), and 34560 in [FRSG22, FGM+23] (with error correction), which outperforms both, our *LaOla-based PSS* compiler, which requires 5997328 (6648224) binary operations, and our *BGW-based PSS* compiler which needs 436968 (495992) binary operations[12]. However, our *BGW-based PSS* approach already outperforms [FRSG22, FGM+23] for $t = e = 3$. Namely, our *BGW-based PSS* compiler needs 1147840 (1433750) binary gates, whereas [FRSG22, FGM+23] requires 1684480 binary gates for the multiplication with error correction gadgets in place. A table (cf. Tab. 9) summarizing all numbers is provided in Supplementary Material G.2.

Note that for $t = e = \{1, 2\}$, duplicated Boolean masking [FRSG22, FGM+23] achieves the best Boolean circuit complexity for the multiplication (or AND) gadget (cf. Tab. 9). However, the Boolean circuit for one round of AES-128 requires 2660 XOR gates. Recall that duplicated Boolean masking [FRSG22, FGM+23] duplicates each share $e + 1$ times, and the Boolean circuit complexity for one XOR gadget is $(2e + 1) \cdot (t + 1)$. This is the reason why duplicated Boolean masking [FRSG22, FGM+23] does not provide the best Boolean circuit complexity for AES-128 when $t = e = \{1, 2\}$ (cf. Tab. 1).

We emphasize that our compilers are (t, e)-combined resilient, which allows t probes and e faults on field elements. In contrast, the duplicated Boolean masking from [FRSG22, FGM+23] allows glitch-extended probes and binary faults on bits. Additionally, the duplicated Boolean masking schemes from [DN20, FRSG22, FGM+23] do require error detection or correction mechanisms to be integrated during multiplication, whereas our approach only requires fault detection once during the final decoding phase.

8 Conclusion

This paper introduces an efficient approach for combined leakage and fault resilient computation on packed secret sharing. We develop new gadgets and extend state-of-the-art gadgets to apply them to packed secret sharing. All constructions are not only purely

[12]Note that the differences in our approaches arise from the different choices we make for n.

leakage and fault resilient, but combined resilient allowing for simultaneous attacks where faults can compromise the effectiveness of the leakage countermeasures.

To this end, our work addresses the challenges identified by Grosso et al. [GSF14] and Arnold et al. [ABEO24], leading to the development of highly parallelized compilers. We evaluate our improvements with an efficient AES scheme, achieving security against combined attacks and improving the computational complexity by a factor of up to 4.5. These results are a first step towards more efficiently protected implementations against combined attacks. We leave the practical evaluation (both in terms of attack resistance and efficiency) as important future work. This includes not only standalone side-channel and fault attacks but also more sophisticated combined attacks where injected faults aim to undermine the effectiveness of leakage countermeasures.

Acknowledgement

This work was partly supported by the German Research Foundation (DFG) via the 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 via the Projects LEAK and CoVeriNoisy, and by the European Research Council (ERC) under the European Union's Horizon 2020 and Horizon Europe research and innovation programs (grant CRYPTOLAYER-101044770).

References

[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.

[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, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, 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, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 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 Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 23–39. Springer, 2016.

[BDST20] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. MOTION - A framework for mixed-protocol multi-party computation. *IACR Cryptol. ePrint Arch.*, page 1137, 2020.

[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 Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part III*, volume 14083 of *Lecture Notes in Computer Science*, pages 377–409. Springer, 2023.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10. ACM, 1988.

[BP12] Joan Boyar and René Peralta. A small depth-16 circuit for the AES s-box. In Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou, editors, *Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, Heraklion, Crete, Greece, June 4-6, 2012. Proceedings*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 287–298. Springer, 2012.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO*

'97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997.

[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.

[CLRS22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 2022.

[CS21] Nicolas Costes and Martijn Stam. Redundant code-based masking revisited. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):426–450, 2021.

[DDRT12] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic transient faults injection on a hardware and a software implementations of AES. In Guido Bertoni and Benedikt Gierlichs, editors, *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012*, pages 7–15. IEEE Computer Society, 2012.

[DEG$^+$18] Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Stefan Mangard, Florian Mendel, and Robert Primas. Statistical ineffective fault attacks on masked AES with fault countermeasures. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*, volume 11273 of *Lecture Notes in Computer Science*, pages 315–342. Springer, 2018.

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 445–465. Springer, 2010.

[DLM19] Mathieu Dumont, Mathieu Lisart, and Philippe Maurine. Electromagnetic fault injection : How faults occur. In *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2019, Atlanta, GA, USA, August 24, 2019*, pages 9–16. IEEE, 2019.

[DN20] Siemen Dhooghe and Svetla Nikova. My gadget just cares for me - how NINA can prove security against combined attacks. In Stanislaw Jarecki, editor, *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings*, 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, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 990–1004. ACM, 2023.

[FRSG22] Jakob Feldtkeller, Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. CINI MINIS: domain isolation for fault and combined security. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1023–1036. ACM, 2022.

[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, *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 699–710. ACM, 1992.

[GIP$^+$15] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. *IACR Cryptol. ePrint Arch.*, page 154, 2015.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2001.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987.

[GPS21] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Unconditional communication-efficient MPC via hall's marriage theorem. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part II*, volume 12826 of *Lecture Notes in Computer Science*, pages 275–304. Springer, 2021.

[GPS22] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Sharing transformation and dishonest majority MPC with packed secret sharing. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part IV*, volume 13510 of *Lecture Notes in Computer Science*, pages 3–32. Springer, 2022.

[GRR98] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 101–111. ACM, 1998.

[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.

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 444–461. Springer, 2014.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[LMW$^+$24] Jack Lloyd, René Meusel, Simon Warta, Daniel Neus, David CARLIER, Philippe Lieser, René Fischer, Fabian Albert, Daniel Seither, Matthias Gierlings, Nuno Goncalves, Tim Oesterreich, Tobias | Never, Alexander Bluhm, Amos Treiber, Jean-Paul Calderone, Juraj Somorovsky, Kris Kwiatkowski, Kai Michaelis, Mouse, Hannes Rantzsch, oviano, Patrick Schmidt, Pavol Zacik, Lauri Nurmi, Patrick Wildt, phwork, Fabrice Fontaine, fxdupont, and Marcus Brinkmann. *randombit/botan.* 10 2024.

[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, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 63–78. Springer, 2011.

[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.

[SA02] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 2002.

[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.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167. IEEE Computer Society, 1986.

[YYG11] You Yang, Ping Yu, and Yan Gan. Experimental study on the five sort algorithms. In *2011 Second International Conference on Mechanic Automation and Control Engineering*, pages 1314–1317. IEEE, 2011.

[ZDCT13] Loïc Zussa, Jean-Max Dutertre, Jessy Clédière, and Assia Tria. Power supply glitch induced faults on FPGA: an in-depth analysis of the injection mechanism. In *2013 IEEE 19th International On-Line Testing Symposium (IOLTS), Chania, Crete, Greece, July 8-10, 2013*, pages 110–115. IEEE, 2013.

A Matrix Transformation

A.1 Matrix Transformation For Error-Detection

In order to guarantee the error-detection property of packed secret sharing, we need to consider all n shares during reconstruction. Recall the description of packed secret sharing in Section 2.2. We use the same notation as presented there, namely, let $\boldsymbol{v} = (v_0, \dots, v_d)$ denote a vector consisting of k secrets and $d-k+1$ distinct random numbers. We denote the inverse of the Vandermonde matrix of our support points $u_0, \dots, u_d$ by $U^{-1} = \mathsf{Van}^{-1}_{(d+1)\times(d+1)}(u_0, \dots, u_d)$. As previously, the Vandermonde matrix of support points $x_0, \dots, x_{n-1}$ is $V = \mathsf{Van}_{(d+1)\times n}(x_0, \dots, x_{n-1})$, where u_i with $i \in [d+1]$ and x_i with $i \in [n]$ are pairwise distinct support points. The masking operation is defined as follows:

$$\mathsf{mask}(\boldsymbol{v}) = \boldsymbol{F} = \boldsymbol{v} \cdot M,$$

where $M := U^{-1} \cdot V$.

Note that M is not a square matrix. Thus, M cannot be directly inverted. Instead, we consider the $(d+1) \times n$-matrix V. We *extend* matrix V up and obtain a square matrix $V_n = \mathsf{Van}_{n\times n}(x_0, \dots, x_{n-1})$. Note that V_n is a square Vandermonde matrix and is thus invertible. Next, we define $V_n^{-1} = \mathsf{Van}^{-1}_{n\times n}(x_0, \dots, x_{n-1})$, and define V^{-1} to be a $n \times (d+1)$ sub-matrix of V_n^{-1} without the last $(n-d-1)$ rows. We can denote the unmasking procedure by

$$\mathsf{unmask}(\boldsymbol{F}) = \boldsymbol{F} \cdot M^{-1},$$

where $M^{-1} := V^{-1} \cdot U$. Correctness holds since

$$\mathsf{unmask}(\mathsf{mask}(\boldsymbol{v})) = \mathsf{unmask}(\boldsymbol{v} \cdot M) = \boldsymbol{v} \cdot M \cdot M^{-1} = \boldsymbol{v} \cdot U^{-1} \cdot V \cdot V^{-1} \cdot U$$

By definition, U is the inverse of U^{-1}. Now, we need to consider V. Recall that $\boldsymbol{f} = \boldsymbol{v} \cdot U^{-1}$ consists of $d+1$ elements $(f_0, \dots, f_d)$. Hence,

$$F_l = \sum_{i=0}^{d} f_i \cdot V_{l,i} = \sum_{i=0}^{d} \left(\sum_{j=0}^{d} F_j \cdot V_{j,i}^{-1} \right) \cdot V_{i,l} = \sum_{j=0}^{d} F_j \sum_{i=0}^{d} \cdot V_{j,i}^{-1} \cdot V_{i,l} = F_l,$$

Since $\sum_{i=0}^{d} \cdot V_{j,i}^{-1} \cdot V_{i,l} = 0$ if $l \neq j$ and 1 otherwise, we obtain the desired property for all $l \in [n]$.

A.2 SplitRed Matrix Transformation

Next, we describe how to get the $n \times \left(\frac{d}{2}+1\right)$ matrix $M_{\boldsymbol{F}\to\tilde{\boldsymbol{f}}}$. We start with Equation 7,

$$\boldsymbol{F} = \boldsymbol{v} \cdot U^{-1} \cdot \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ x_0 & x_1 & x_2 & \dots & x_{n-1} \\ x_0^2 & x_1^2 & x_2^2 & \dots & x_{n-1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_0^d & x_1^d & x_2^d & \dots & x_{n-1}^d \end{pmatrix}_{(d+1)\times n} \tag{24}$$

$$= \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_d \end{pmatrix}^T \cdot \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ x_0 & x_1 & x_2 & \dots & x_{n-1} \\ x_0^2 & x_1^2 & x_2^2 & \dots & x_{n-1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_0^d & x_1^d & x_2^d & \dots & x_{n-1}^d \end{pmatrix}_{(d+1)\times n} \tag{25}$$

$$= \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_d \\ \hline 0_{d+1} \\ \vdots \\ 0_{n-1} \end{pmatrix}^T \cdot \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ x_0 & x_1 & x_2 & \dots & x_{n-1} \\ x_0^2 & x_1^2 & x_2^2 & \dots & x_{n-1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_0^d & x_1^d & x_2^d & \dots & x_{n-1}^d \\ \hline x_0^d & x_1^d & x_2^d & \dots & x_{n-1}^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_0^{n-1} & x_1^{n-1} & x_2^{n-1} & \dots & x_{n-1}^{n-1} \end{pmatrix}_{n\times n}, \tag{26}$$

where $\{0_{d+1}, \dots, 0_{n-1}\}$ represent zero values. Then we can get

$$\boldsymbol{F} \cdot \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ x_0 & x_1 & x_2 & \dots & x_{n-1} \\ x_0^2 & x_1^2 & x_2^2 & \dots & x_{n-1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_0^d & x_1^d & x_2^d & \dots & x_{n-1}^d \\ \hline x_0^d & x_1^d & x_2^d & \dots & x_{n-1}^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_0^{n-1} & x_1^{n-1} & x_2^{n-1} & \dots & x_{n-1}^{n-1} \end{pmatrix}_{n\times n}^{-1} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_d \\ \hline 0_{d+1} \\ \vdots \\ 0_{n-1} \end{pmatrix}^T, \tag{27}$$

as the matrix in Equation 26 is a square and invertible Vandermonde matrix.

Let's denote the inverse of this Vandermonde matrix by

$$V^{-1} = \mathsf{Van}_{n\times n}^{-1}(x_0, \dots, x_{n-1}) = \begin{pmatrix} \lambda_{0,0} & \dots & \lambda_{0,d} & \dots & \lambda_{0,n-1} \\ \lambda_{1,0} & \dots & \lambda_{1,d} & \dots & \lambda_{1,n-1} \\ \lambda_{2,0} & \dots & \lambda_{2,d} & \dots & \lambda_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \lambda_{n-1,0} & \dots & \lambda_{n-1,d} & \dots & \lambda_{n-1,n-1} \end{pmatrix}_{n\times n}.$$

Further, we have

$$\boldsymbol{F} \cdot \begin{pmatrix} \lambda_{0,0} & \dots & \lambda_{0,d} & \dots & \lambda_{0,n-1} \\ \lambda_{1,0} & \dots & \lambda_{1,d} & \dots & \lambda_{1,n-1} \\ \lambda_{2,0} & \dots & \lambda_{2,d} & \dots & \lambda_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \lambda_{n-1,0} & \dots & \lambda_{n-1,d} & \dots & \lambda_{n-1,n-1} \end{pmatrix}_{n\times n} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_d \\ \hline 0_{d+1} \\ \vdots \\ 0_n \end{pmatrix}^T \tag{28}$$

$$\boldsymbol{F} \cdot \left(\begin{array}{ccc|ccc} \lambda_{0,0} & \dots & \lambda_{0,d} & 0 & \dots & 0 \\ \lambda_{1,0} & \dots & \lambda_{1,d} & 0 & \dots & 0 \\ \lambda_{2,0} & \dots & \lambda_{2,d} & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \lambda_{n-1,0} & \dots & \lambda_{n-1,d} & 0 & \dots & 0 \end{array}\right)_{n\times n} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_d \\ \hline 0_{d+1} \\ \vdots \\ 0_n \end{pmatrix}^T \tag{29}$$

$$\boldsymbol{F} \cdot \begin{pmatrix} \lambda_{0,0} & \dots & \lambda_{0,d} \\ \lambda_{1,0} & \dots & \lambda_{1,d} \\ \lambda_{2,0} & \dots & \lambda_{2,d} \\ \vdots & \ddots & \vdots \\ \lambda_{n-1,0} & \dots & \lambda_{n-1,d} \end{pmatrix}_{n\times (d+1)} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_d \end{pmatrix}^T \tag{30}$$

$$\tag{31}$$

where

$$M_1 = \begin{pmatrix} \lambda_{0,0} & \dots & \lambda_{0,d} \\ \lambda_{1,0} & \dots & \lambda_{1,d} \\ \lambda_{2,0} & \dots & \lambda_{2,d} \\ \vdots & \ddots & \vdots \\ \lambda_{n-1,0} & \dots & \lambda_{n-1,d} \end{pmatrix}_{n\times (d+1)} .$$

Since

$$\boldsymbol{v} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_d \end{pmatrix}^T \cdot U,$$

we have

$$\boldsymbol{F} \cdot M_1 = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_d \end{pmatrix}^T \tag{32}$$

$$\boldsymbol{F} \cdot M_1 \cdot U = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_d \end{pmatrix}^T \cdot U = \boldsymbol{v}. \tag{33}$$

Next, we transform Equation 8 as follows,

$$\begin{pmatrix} \tilde{f}_0 \\ \tilde{f}_1 \\ \tilde{f}_2 \\ \dots \\ \tilde{f}_{\frac{d}{2}} \end{pmatrix}^T = \begin{pmatrix} v_0 \\ \dots \\ v_{t-1} \\ \vdots \\ v_{\frac{d}{2}} \end{pmatrix}^T \cdot \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ u_0 & u_1 & u_2 & \dots & u_{\frac{d}{2}} \\ u_0^2 & u_1^2 & u_2^2 & \dots & u_{\frac{d}{2}}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ u_0^{\frac{d}{2}} & u_1^{\frac{d}{2}} & u_2^{\frac{d}{2}} & \dots & u_{\frac{d}{2}}^{\frac{d}{2}} \end{pmatrix}^{-1}_{\left(\frac{d}{2}+1\right)\times\left(\frac{d}{2}+1\right)} \tag{34}$$

$$\left(\begin{array}{c} \tilde{f}_0 \\ \tilde{f}_1 \\ \tilde{f}_2 \\ \dots \\ \tilde{f}_{\frac{d}{2}} \\ \hline 0_{\frac{d}{2}+1} \\ \vdots \\ 0_d \end{array}\right)^T = \left(\begin{array}{c} v_0 \\ \dots \\ v_{t-1} \\ \vdots \\ v_{\frac{d}{2}} \\ \hline v_{\frac{d}{2}+1} \\ \vdots \\ v_d \end{array}\right)^T \cdot \begin{pmatrix} \begin{bmatrix} U^{-1}_{\left(\frac{d}{2}+1\right)\times\left(\frac{d}{2}+1\right)} \end{bmatrix} & \begin{matrix} 0 & \dots & 0 \\ \vdots & \dots & \vdots \\ 0 & \dots & 0 \end{matrix} \\ \begin{matrix} 0 & \dots & 0 \\ \vdots & \dots & \vdots \\ 0 & \dots & 0 \end{matrix} & \begin{matrix} 0 & \dots & 0 \\ \vdots & \dots & \vdots \\ 0 & \dots & 0 \end{matrix} \end{pmatrix}_{(d+1)\times(d+1)} . \tag{35}$$

Finally, we combine Equation 33 and Equation 35 and get

$$\left(\begin{array}{c} \tilde{f}_0 \\ \tilde{f}_1 \\ \tilde{f}_2 \\ \dots \\ \tilde{f}_{\frac{d}{2}} \\ \hline 0_{\frac{d}{2}+1} \\ \vdots \\ 0_d \end{array}\right)^T = \boldsymbol{F} \cdot M_1 \cdot U \cdot M_2, \tag{36}$$

where

$$M_2 = \begin{pmatrix} \begin{bmatrix} U^{-1}_{\left(\frac{d}{2}+1\right)\times\left(\frac{d}{2}+1\right)} \end{bmatrix} & \begin{matrix} 0 & \dots & 0 \\ \vdots & \dots & \vdots \\ 0 & \dots & 0 \end{matrix} \\ \begin{matrix} 0 & \dots & 0 \\ \vdots & \dots & \vdots \\ 0 & \dots & 0 \end{matrix} & \begin{matrix} 0 & \dots & 0 \\ \vdots & \dots & \vdots \\ 0 & \dots & 0 \end{matrix} \end{pmatrix}_{(d+1)\times(d+1)} .$$

Note that we can change the matrix multiplication order of the right-hand side of Equation 36 to $\boldsymbol{F} \cdot (M_1 \cdot (U \cdot M_2))$ because of the associativity of matrix multiplication. Let's denote the multiplication result of $(M_1 \cdot (U \cdot M_2))$ by M_3. It is easy to verify that M_3 has the following form,

$$M_3 = \left(\begin{array}{ccc|ccc} M_{0,0} & \dots & M_{0,\frac{d}{2}} & 0 & \dots & 0 \\ M_{1,0} & \dots & M_{1,\frac{d}{2}} & 0 & \dots & 0 \\ M_{2,0} & \dots & M_{2,\frac{d}{2}} & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ M_{n-1,0} & \dots & M_{n-1,\frac{d}{2}} & 0 & \dots & 0 \end{array}\right)_{n\times(d+1)} . \tag{37}$$

Then we have

$$\left(\begin{array}{c} \tilde{f}_0 \\ \tilde{f}_1 \\ \tilde{f}_2 \\ \dots \\ \tilde{f}_{\frac{d}{2}} \\ \hline 0_{\frac{d}{2}+1} \\ \vdots \\ 0_d \end{array}\right)^T = \begin{pmatrix} F_0 \\ F_1 \\ F_2 \\ \dots \\ F_{n-1} \end{pmatrix}^T \cdot \left(\begin{array}{ccc|ccc} M_{0,0} & \dots & M_{0,\frac{d}{2}} & 0 & \dots & 0 \\ M_{1,0} & \dots & M_{1,\frac{d}{2}} & 0 & \dots & 0 \\ M_{2,0} & \dots & M_{2,\frac{d}{2}} & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ M_{n-1,0} & \dots & M_{n-1,\frac{d}{2}} & 0 & \dots & 0 \end{array}\right)_{n\times(d+1)} \tag{38}$$

$$\begin{pmatrix} \tilde{f}_0 \\ \tilde{f}_1 \\ \tilde{f}_2 \\ \dots \\ \tilde{f}_{\frac{d}{2}} \end{pmatrix}^T = \begin{pmatrix} F_0 \\ F_1 \\ F_2 \\ \dots \\ F_{n-1} \end{pmatrix}^T \cdot \begin{pmatrix} M_{0,0} & \dots & M_{0,\frac{d}{2}} \\ M_{1,0} & \dots & M_{1,\frac{d}{2}} \\ M_{2,0} & \dots & M_{2,\frac{d}{2}} \\ \vdots & \ddots & \vdots \\ M_{n-1,0} & \dots & M_{n-1,\frac{d}{2}} \end{pmatrix}_{n\times\left(\frac{d}{2}+1\right)} . \tag{39}$$

We denote matrix

$$\begin{pmatrix} M_{0,0} & \dots & M_{0,\frac{d}{2}} \\ M_{1,0} & \dots & M_{1,\frac{d}{2}} \\ M_{2,0} & \dots & M_{2,\frac{d}{2}} \\ \vdots & \ddots & \vdots \\ M_{n-1,0} & \dots & M_{n-1,\frac{d}{2}} \end{pmatrix}_{n\times\left(\frac{d}{2}+1\right)}$$

by $M_{\boldsymbol{F}\to\tilde{\boldsymbol{f}}}$ which is the matrix for the linear transformation in Equation 9. In the following text, we omit the subscript of $M_{\boldsymbol{F}\to\tilde{\boldsymbol{f}}}$ and use M instead when it is clear from the context.

A.3 [FY92] Matrix Transformation

To obtain the matrix corresponding to the $\textsf{Mult}_{\textsf{BGW}}$ Gadget, we can proceed similarly to the degree reduction matrix for **SplitRed** presented in Supplementary Material A.2, using $2d$ and d instead of d and $d/2$.

A.4 Permutation Matrix Transformation

We describe how to get the $n \times (d+1)$ matrix $M_{\boldsymbol{F}\to\tilde{\boldsymbol{f}}}$. Following the same steps as in Supplementary Material A.2 we get a $n \times (d+1)$ matrix M_1 such that

$$\boldsymbol{F} \cdot M_1 = \boldsymbol{v}. \tag{40}$$

Next, we transform Equation 8 as follows,

$$\begin{pmatrix} \tilde{f}_0 \\ \vdots \\ \tilde{f}_d \end{pmatrix}^T = \boldsymbol{v} \cdot U'^{-1}, \tag{41}$$

where $U'^{-1} = \mathsf{Van}^{-1}_{(d+1)\times(d+1)}(u'_0, \dots, u'_d)$.

Finally, we combine Equation 40 and Equation 41 and get

$$\begin{pmatrix} \tilde{f}_0 \\ \vdots \\ \tilde{f}_d \end{pmatrix}^T = \boldsymbol{F} \cdot M_1 \cdot U'^{-1}, \tag{42}$$

where $\tilde{\boldsymbol{f}} = (\tilde{f}_0 \dots \tilde{f}_d)$ are the coefficients of polynomial $\tilde{f}(X)$, that are used to generate another n packed secret shares $\boldsymbol{F}' = (F'_0, \dots, F'_{n-1})$ at support points $\boldsymbol{u}' = (u'_0, u'_1, \dots, u'_d)$.

A.5 Recombination Matrix Transformation

We begin to explain the construction of the Recombination Matrix with the help of a simple example. Assume we consider $m = 2$ input sharings $\boldsymbol{F}^{(0)} = \left(F_i^{(0)}\right)_{i\in[n]}$ that embeds the secret $\boldsymbol{v}^{(0)}$ and $\boldsymbol{F}^{(1)} = \left(F_i^{(1)}\right)_{i\in[n]}$ embedding $\boldsymbol{v}^{(1)}$. We know that

$$\boldsymbol{F}^{(0)} = \boldsymbol{v}^{(0)} \cdot U^{-1} \cdot V,$$

where U^{-1} denotes the inverse of the Vandermonde matix corresponding to $(u_0, \dots, u_d)$ and V denotes the Vandermonde matrix with respect to $(x_0, \dots, x_{n-1})$. Similarly,

$$\boldsymbol{F}^{(1)} = \boldsymbol{v}^{(1)} \cdot U^{-1} \cdot V.$$

For simplicity, we set $M = U^{-1} \cdot V$. If we now want to obtain shares $\boldsymbol{F}'$ embedding the secret $\boldsymbol{v}' = \left(v_0^{(0)}, v_1^{(1)}, v_2^{(0)}, \dots, v_{n-1}^{(0)}\right)$, we can consider

$$\begin{aligned}
\boldsymbol{v}' &= \left(v_0^{(0)}, v_1^{(1)}, v_2^{(0)}, \dots, v_{n-1}^{(0)}\right) \\
&= \left(\sum_{i=0}^{n-1} F_i^{(0)} \cdot M_{0,i}^{-1}, \sum_{i=0}^{n-1} F_i^{(1)} \cdot M_{1,i}^{-1}, \sum_{i=0}^{n-1} F_i^{(0)} \cdot M_{2,i}^{-1}, \dots, \sum_{i=0}^{n-1} F_i^{(0)} \cdot M_{n-1,i}^{-1}\right) \\
&= \left(\boldsymbol{F}^{(0)}, \boldsymbol{F}^{(1)}\right) \cdot \begin{pmatrix} M_{0,0}^{-1} & 0 & M_{0,2}^{-1} & \dots & M_{0,d}^{-1} \\ M_{1,0}^{-1} & 0 & M_{1,2}^{-1} & \dots & M_{1,d}^{-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M_{n-1,0}^{-1} & 0 & M_{n-1,2}^{-1} & \dots & M_{n-1,d}^{-1} \\ 0 & M_{0,1}^{-1} & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & M_{n-1,1}^{-1} & 0 & \dots & 0 \end{pmatrix}_{(n\cdot m)\times(d+1)} \\
&= \left(\boldsymbol{F}^{(0)}, \boldsymbol{F}^{(1)}\right) \cdot M'
\end{aligned}$$

Please recall that we argued that the inverse of M does exist in Supplementary Material A. We achieved a reconstruction of the secret $\boldsymbol{v}'$. Now, we need to reshare our new secret.

We can do so by calculating:

$$\begin{aligned}\boldsymbol{F}' &= \boldsymbol{v}' \cdot U^{-1} \cdot V \\ &= \left(\boldsymbol{F}^{(0)}, \boldsymbol{F}^{(1)}\right) \cdot M' \cdot U^{-1} \cdot V \\ &= \left(\boldsymbol{F}^{(0)}, \boldsymbol{F}^{(1)}\right) \cdot M_{\left(\boldsymbol{F}^{(0)},\dots,\boldsymbol{F}^{(1)}\right)\to\boldsymbol{F}'}\end{aligned}$$

This idea can be generalized by considering m input sharings $\boldsymbol{F}^{(0)}, \dots \boldsymbol{F}^{(m-1)}$ and an $(n \cdot m) \times (d+1)$ matrix M' that consists of columns of M and 0 entries. Thus, we only need to carefully choose where to put the columns of M and columns containing 0 to reconstruct the desired secret.

Furthermore, we can also multiply the resulting matrix $M_{\left(\boldsymbol{F}^{(0)},\dots,\boldsymbol{F}^{(1)}\right)\to\boldsymbol{F}'}$ with a permutation matrix (cf. Supp. A.4), to obtain a resharing together with a permutation.

A.6 Correctness of Table 4

In this section, we give a more detailed description of our instantiations of Algorithm 5, while keeping the matrices derived in the previous sections in mind.

Lemma 16. *Algorithm 5 is correct for all instantiations of* $\ell_{i,j}^{\varphi_0,\dots,\varphi_d}(\cdot,\dots,\cdot)$ *shown in Table 4 when using* $n > 2d$ *shares and polynomials of degree* d.

Proof. We aim to show that the functionality as denoted in Table 4 is correct when inserting $\ell_{i,j}^{\varphi_0,\dots,\varphi_d}$ as specified in Table 4 into Algorithm 5. For one specific j, over the iteration of i, we obtain

$$H_j = \sum_{i=0}^{n-1} \ell_{i,j}^{\varphi_0,\dots,\varphi_d}\left(F_i^{(0)},\dots,F_i^{(m-1)}\right) + \left(0,\dots,0,\sum_{i=0}^{n-1} r_{i,k},\dots,\sum_{i=0}^{n-1} r_{i,(n-1)}\right)\cdot M_j,$$

where M_j denotes the j-th column of M.
For correctness, we need to consider the first term of this equation, since the latter one is the j-th share of the sum of n zero encodings.

- **Refresh**$_{\text{LT}}$. When considering one sharing, $(F_i)_{i\in[n]}$, as input and setting $\ell_{i,j}^{\varphi_0,\dots,\varphi_d} = \mathbf{I}_{i,j}$ as denoted in the table, we obtain

$$H_j = F_j + \left(0,\dots,0,\sum_{i=0}^{n-1} r_{i,k},\dots,\sum_{i=0}^{n-1} r_{i,(n-1)}\right)\cdot M_j.$$

  Thus, if $(F_i)_{i\in[n]}$ is a masking of $(v_0,\dots,v_{k-1},r_k,\dots,r_{n-1})$, Algorithm 5 outputs a masking of $(v_0,\dots,v_{k-1},r_k^{prime},\dots,r_{n-1}^{prime})$, as desired.
- **Permutation**. Similar to **Refresh**$_{\text{LT}}$, we only consider one input sharing and obtain

$$H_j = \lambda_{j,i}^{\textsf{(P)}} + \left(0,\dots,0,\sum_{i=0}^{n-1} r_{i,k},\dots,\sum_{i=0}^{n-1} r_{i,(n-1)}\right)\cdot M_j.$$

  where $\lambda_{j,i}^{\textsf{(P)}}$ is calculated as described in the previous paragraph. We can easily verify that $\lambda_{j,i}^{\textsf{(P)}}$ permutes the underlying secrets $\boldsymbol{v}$ of the sharing $(F_i)_{i\in[n]}$ to $\boldsymbol{v}'$. Since we only add zero encodings, $(H_i)_{i\in[n]}$ is an encoding of $\boldsymbol{v}'$ as well.
- **Recombination**. When choosing the values $\lambda_{j+ln,i}^{\textsf{(R)}}$ according to the description in Supp. A.5, we obtain desired gadged, using the same reasoning as for the **Permutation** gadget.

- $\textsf{Mult}_{\textsf{BGW}}$. Within this gadget, we consider two input encodings, namely $\left(F_i^{(0)}\right)_{i\in[n]}$ and $\left(F_i^{(1)}\right)_{i\in[n]}$. When constructing $\lambda_{j,i}^{(\textsf{DegRed})}$ as described in the previous section, we obtain the desired multiplication and degree reduction.

□

B Probe-Resilience

For most of the proofs in this section, we follow the proof ideas in [BEF⁺23] and [ABEO24] with minor modifications.

The Linear Transformation Gadget is SNI. In this section, we show that gadget $\textsf{Gadget}_{\textsf{LT}}$ (cf. Alg. 5) for general linear transformations already satisfies the SNI property.

Lemma 17. $\textsf{Gadget}_{\textsf{LT}}$ *is t-SNI when using n shares and polynomials of degree $d > t-1+k$. When considering* $\textsf{Mult}_{\textsf{BGW}}$*, we need to consider $n > 2d$ shares and polynomials of degree d.*

Proof. We proceed similarly to the proof presented in [ABEO24, Lem. 8].
We consider t_1 internal probes and t_2 output probes, such that $t_1 + t_2 < t$. First, we define the input sets $I_0, \dots, I_{m-1}$ of size at most t_1. Afterwards, we construct a simulator that can perfectly simulate all t probes, while only using elements of $I_0, \dots, I_{m-1}$.
The sets $I_0, \dots, I_{m-1}$ are dependent on the intermediate variables that are probed and constructed as follows:

- If $F_i^{(0)}, \dots, F_i^{(m-2)}$ or $F_i^{(m-1)}$ are probed, add i to $I_0, \dots, I_{m-2}$ or I_{m-1}, respectively.
- If any value is probed during the computation of $\ell_{i,j}^{\varphi_0,\dots,\varphi_d}\left(F_i^{(0)}, \dots, F_i^{(m-1)}\right)$, add i to $I_0, \dots, I_{m-2}$ and I_{m-1}.
- If an element r_j in loop i (an element of $\textsf{ZEnc}_n^d$), for simplicity denoted as $r_{i,j}$, some intermediate value of $\textsf{ZEnc}_n^d$ (denoted as $\tilde{H}_{i,j}^k$), $k \in [d]$, or the sum with the share-wise transformation (denoted as $\tilde{H}_{i,j}^d$) is probed, add i to $I_0, \dots, I_{m-2}$ and I_{m-1}.

Since we add at most one index to $I_0, \dots, I_{m-2}$ and I_{m-1} for each internal probe, we obtain $|I_0| \le t_1, \dots, |I_{m-2}| \le t_1$ and $|I_{m-1}| \le t_1$.

Now, we construct the simulator. In case a value $r_{i,j}$ is needed for simulation, it will be sampled uniformly at random by the simulator and thus be fixed.

1. In case $F_i^{(0)}, \dots, F_i^{(m-2)}$ or $F_i^{(m-1)}$ are probed, we can perfectly simulate them, since for any $i \in I_l$, $l \in [m]$, the share F_i^l is a known value. Moreover, if an internal value of $\ell_{i,j}^{\varphi_0,\dots,\varphi_d}\left(F_i^{(0)}, \dots, F_i^{(m-1)}\right)$ is probed, we can perfectly simulate this value, because $i \in I_0, \cup \dots \cup I_{m-1}$ and $\ell_{i,j}^{\varphi_0,\dots,\varphi_d}\left(F_i^{(0)}, \dots, F_i^{(m-1)}\right)$ is only dependent on the public values $F_i^{(0)}, \dots, F_i^{(m-1)}$.
2. If $\tilde{H}_{i,j}^k$, $k \in [d]$ is probed, the simulator fixes the necessary $r_{i,j}$ and calculates $\tilde{H}_{i,j}^k = \sum_{l=0}^k r_{i,l} \cdot M_{j,l}$. Unknown $r_{i,l}$ are sampled uniformly at random, resulting in a perfect simulation.
3. The simulation of $\tilde{H}_{i,j}^d$ depends on $\tilde{H}_{i,j}^{d-1}$ and $\ell_{i,j}^{\varphi_0,\dots,\varphi_d}\left(F_i^{(0)}, \dots, F_i^{(m-1)}\right)$. In case $\tilde{H}_{i,j}^{d-1}$ has not been calculated before, we simulate it as described in 2.. Moreover, since $i \in I_0, \cup \dots \cup I_{m-1}$, the values $F_i^{(0)}, \dots, F_i^{(m-1)}$ are known and we can simulate $\ell_{i,j}^{\varphi_0,\dots,\varphi_d}\left(F_i^{(0)}, \dots, F_i^{(m-1)}\right)$. This results in a perfect simulation of $\tilde{H}_{i,j}^d$.

4. If $H_{i,j}$ is probed, we consider $H_{i,j} = \sum_{k=0}^{i} \tilde{H}^d_{k,j} = H_{j,l} + \sum_{k=l+1}^{i} \tilde{H}^d_{k,j}$. If $l < i$ has been probed before, we simulate only $\tilde{H}^d_{k,j}$, $l < k \leq i$, otherwise we consider all elements $\tilde{H}^d_{k,j}$.
   If $\ell^{\varphi_0,\ldots,\varphi_d}_{k,j}\left(F^{(0)}_k, \ldots, F^{(m-1)}_k\right)$ are known for all necessary k, $\tilde{H}^d_{k,j}$ can be perfectly simulated similar to case 3.
   Otherwise, there is at least one unknown $\ell^{\varphi_0,\ldots,\varphi_d}_{k,j}\left(F^{(0)}_k, \ldots, F^{(m-1)}_k\right)$. If this is the case, $r_{k,l}$ has not been probed for $l \in [d+1-k]$. Recall that the $r_{k,l}$ are uniformly random, thus $\left(\tilde{H}^d_{k,j}\right)_{j\in[n]}$ is $d+1-k$-wise independent. Due to our threshold t, $H_{j,i}$ can be simulated by an element which is chosen uniformly at random.

This finishes the proof since all elements can be perfectly simulated. □

Lemma 18. *The modified* **SplitRed** *is* $(d+1-k)-\mathsf{NI}$.

Proof. Let us consider three **groups of probes**

- Group K: All probes of the set $K = (\bigcup_{l\in[n]} K_l) \cup (\bigcup_{l\in[n/2]} K'_l)$ with
  - K_l contains the internal values in $\left(\tilde{g}^l_i\right)_{i\in[n]} \leftarrow \mathsf{ZEnc}^d_n$ and $\left(\tilde{g}^l_i\right)_{i\in[n]}$
  - K'_l contains the internal values of $\left(\hat{g}^l_i\right)_{i\in[n/2]} \leftarrow \mathsf{ZEnc}^{n/2}_n$ and $\left(\hat{g}^l_i\right)_{i\in[n/2]}$
- Group I: All probes of the set $I = \bigcup_{i\in[n]} I_l$, where $I_l = \{F_l, \mathcal{F}^l_0, \ldots, \mathcal{F}^l_{n-1}, \mathcal{F}'^l_0, \ldots, \mathcal{F}'^l_{n-1}\}$
- Group J: All probes of the set $J = \bigcup_{i\in[n]}(J'_i \cup J''_i)$, where $J'_i = \{F'^{(0)}_i, \ldots, F'^{(n/2-1)}_i\}$ and $J''_i = \{F''^{(0)}_i, \ldots, F''^{(n/2-1)}_i\}$

We begin by considering the cases where a random value g^l_i can not be generated uniformly at random due to probes in K.

- D.Case 1: If we have a probe in K_l and K'_l with $l < n/2$, then $\tilde{g}^l_i$ and $\hat{g}^l_i$ are dependent of these probes. Moreover, $g^l_i = \tilde{g}^l_i + \hat{g}^l_i$ is dependent on these probes as well.
- D.Case 2: If we have a probe in K_l and K'_l, $l > n/2$, then $\hat{g}^{l-n/2}_i$ and $\tilde{g}^l_i$ are not independent of the probes. Hence $g^l_i = \tilde{g}^l_i - \hat{g}^{l-n/2}_i$ is dependent on these probes as well.
- D.Case 3: If we have a probe in K'_l, but not in K_l or $K_{l+n/2}$ (otherwise, we are in Case 1 or Case 2), then $\hat{g}^l_i$ is dependent on this probe. Hence, g^l_i and $g^{l+n/2}_i$ are d' wise independent, where $n/2+1-k \leq d' < d+1-k$, since g^l_i is only $n/2+1-k$-wise randomized by $\left(\tilde{g}^l_i\right)_{i\in[n]}$, and $g^{n/2+l}_i$ by $\left(\tilde{g}^{n/2+l}_i\right)_{i\in[n]}$, respectively. Following the argument of [BEF+23], we only need to consider one of the two encodings and can assume the other one consists of uniform random values.

Next, we need to **construct the input set** S for the Simulator. Again, we discuss the cases one by one.

- Group K: we only need to consider the dependency cases, as otherwise, $\left(g^l_i\right)_{i\in[n]}$ can still be generated uniformly at random. Hence, we look at D.Case 1-3, where $\left(g^l_i\right)_{i\in[n]}$ is not $d-k+1$-wise independent. If there are at least two probes in $K_l, K_{l+n/2}, K'_l$, where $l < n/2$, we add F_l to S
$$S \leftarrow S \cup \{F_l\}$$
  Note that this covers D.Case 1 and D.Case 2. The third case is considered later together with group J. Moreover, observe that for each probe, the simulator gets at most one input.
- Group I: If there is a probe in I_l, we add F_l to S
$$S \leftarrow S \cup \{F_l\}$$
  Again, we add at most one element per probe in I.

- Group J: We need to consider the cases, where $\left(g_i^l\right)_{i\in[n]}$ and $\left(g_i^{l+n/2}\right)_{i\in[n]}$ are not $d+1-k$ wise independent. (Otherwise, the elements are a sum of a constant multiplication with an input and a sum to a random value unknown to the adversary.)
  - If there is a probe in K_l' but not in K_l or $K_{l+n/2}$ (D.Case 3) and there are more than $d/2+1-k$ probes in $\bigcup_{i=0}^{n/2-k}(I_i \cup J_i')$. Then add F_l to S.
  $$S \leftarrow S \cup \{F_l\}$$
  - Similarly for $I_{i+n/2} \cup J_i''$, i.e. If there is a probe in K_l' but not in K_l or $K_{l+n/2}$ (D.Case 3) and there are more than $d/2+1-k$ probes in $\bigcup_{i=0}^{n/2-k}\left(I_{i+n/2} \cup J_i''\right)$. Then add F_l to S.
  $$S \leftarrow S \cup \{F_l\}$$

Note that we add at most one element per probe in J_i.

The simulator obtains the input set S. Now, we explain group-wise how the simulator can simulate all necessary values.

Simulator with input set S

1. Group K: If $\left(g_i^l\right)_{i\in[n]}$ is not $d-k+1$-wise independent (D.Case 1-3), the simulator simulates the g_i^l as follows.
   - D.Case 1: If there are two probes in $K_l, K_{l+n/2}$ and K_l', together with a probe in I_l, where $l < n/2$, then the simulator simulates $g_i^l = \tilde{g}_i^l + \hat{g}_i^l$ by computing $\tilde{g}_i^j$ and $\hat{g}_i^j$ as mentioned in the algorithm.
   - D.Case 2: If there are two probes in $K_l, K_{l+n/2}$ and K_l', together with a probe in $I_{l+n/2}$, where $l < n/2$, the simulator simulates $g_i^{l+n/2} = \tilde{g}_i^{l+n/2} - \hat{g}_i^l$ by computing the distribution of $\tilde{g}_i^{j+n/2}$ and $\hat{g}_i^j$.
   - D.Case 3: If there is a probe in K_l' and not in K_l not $K_{l+n/2}$. If there are more than $d/2-k+1$ probes in the set $\bigcup_{i=0}^{n/2-1}((I_i \cup J_i')$ the simulator simulates $g_i^l = \tilde{g}_i^l + \hat{g}_i^l$ again the distribution of $\tilde{g}_i^j$ and $\hat{g}_i^j$. Similarly, if there are more than $d/2-k+1$ probes in the set the set $\bigcup_{i=0}^{n/2-1}\left((I_{i+n/2} \cup J_i''\right)$ the simulator calculates $g_i^{l+n/2} = \tilde{g}_i^{l+n/2} + \hat{g}_i^l$.

   Note that all required $\tilde{g}_i^j$ and $\hat{g}_i^j$ and all remaining probes in K can be simulated according to the algorithm.
2. Group I: For all probes in I_l, we have $F_l \in S$ by construction. Hence, all elements, $F_l, \mathcal{F}_l^i$, or $\mathcal{F}_l'^i$, for all $i \in [n]$ can be calculated according to the algorithm. In more detail, for any $i \in [n], l \in [n]$ if there was a probe in K_l, the simulator already simulated g_i^l and hence can calculate $\mathcal{F}_i'^l = \hat{\lambda}_l^i \cdot F_l$ and $\mathcal{F}_i^l = \mathcal{F}_i'^l + g_i^l$ for $l < n/2$, as well as $\mathcal{F}_i'^l = \hat{\lambda}_l^i \cdot F_l$ and $\mathcal{F}_i^l = \mathcal{F}_i'^l - g_i^l$ for $l \geq n/2$. Thus, all together, the simulator can simulate all elements in I.
3. Group J:
   - All elements in J' are sums of the form $\lambda_l^i \cdot F_l + g_i^l$, $l < n/2$.
     - In case g_i^l is not uniformly random due to probes in K or L, the simulator already calculated the corresponding values $g_i^l$and holds F_l. Hence, the simulator can perfectly simulate $\lambda_l^i \cdot F_l + g_i^l$.
     - Otherwise, if g_i^l is uniformly random, the element $\lambda_l^i \cdot F_l + g_i^l$ is a shift of a random number and thus is distributed uniformly at random as well. Since g_i^l is unknown to the adversary in this case, the simulator can generate the desired value uniformly at random.
   - All elements in J'' are sums of the form $\lambda_l^i \cdot F_l - g_i^l$, $l \geq n/2$. And we can proceed completely analogous to the previous step.

Note that the simulator can simulate any $d-k+1$ probes P with at most $d-1+k$ input values. Any probe in SplitRed is contained in Group K, I, or J. By definition of S, we

thus get $|S| \leq P \leq d - t + 1$.
It is left to show that S can perfectly simulate all values in P only using S.

- Group K: All probes in K are independent of the input and can be simulated perfectly.
- Group I: $F_l \in S$ for any probe in I_l. Any element in I_l is an intermediate value that can be simulated by the value F_l together with the random values $\left(g_i^l\right)_{i\in[n]}$. The values $\left(g_i^l\right)_{i\in[n]}$ have been simulated as described in Step 1 or can be chosen uniformly at random.
- Group J:
  - Simulating all elements in J' is equivalent to simulating all values $(\hat{\lambda}_l^i F_l) + g_i^l$, with $l \in [n/2]$. Note that by definition of S, if there is a probe in J_l', $F_l \in S$. As denoted in Step 3, either g_i^l has already been calculated in Step 1 or Step 2. Since F_l is known as well, $(\hat{\lambda}_l^i F_l) + g_i^l$ can be simulated perfectly. Otherwise, we can consider g_i^l as uniform random (the explanation follows).
  - Simulating all elements in J'' is completely analogous to the simulation of all elements in J'.

Note that if g_i^l was not generated in Step 1, we can consider it as uniformly random. The reason is as follows. Recall our threshold of $d - k + 1$ probes. The Simulator never uses more than $d + 1 - k$ random values of each encoding, and $d + 1 - k$ values in $\left(g_i^l\right)_{i\in[n]}$ ($\left(g_i^{l+n/2}\right)_{i\in[n]}$) are still independent and uniform random, if there is no probe in K_l' and K_l (or K_l' and $K_{l+n/2}$, respectively). In D.Case 3, the simulator uses at most $d/2$ values of at least one of the encodings. Thus at most one of the encodings in only one of the encodings $\left(g_i^l\right)_{i\in[n]}$, $\left(g_i^{l+n/2}\right)_{i\in[n]}$ leads to dependent random values. This case is covered in Step 1 in the simulator, and the simulator has all dependent (and required) g_i^l.

□

Theorem 2 ([BEF$^+$23, Theorem 12])**.** *A $(d + 1 - k)$ probing secure composition with $(d + 1 - k)$-NI and $(d + 1 - k)$-SNI gadgets is $\frac{d+1-k}{2}$ region probing secure if each gadget is refreshed using* **Refresh** *(cf. Alg. 6), and output refreshed sharings.*

Proof. The only difference to [BEF$^+$23, Theorem 12] is that we modified gadget **Refresh** (cf. Alg. 6). Hence it remains to prove that our modified **Refresh** satisfies $(d + 1 - k)$-SNI, which is already proved in Lemma 10.

□

C Fault-Invariance

Theorem 3 (Similar to Theorem 17 [BEF$^+$23])**.** *All gadgets of our compiler are fault-invariant with respect to $\mathcal{F}^+$.*

Proof. In a nutshell, all affine transformation of secret shares (e.g., share-wise addition, multiplication with constant values) are fault-invariant with respect to $\mathcal{F}^+$ as the corresponding operations are associative. Hence such faults are independent from the inputs and can be pushed to inputs or outputs. □

D Fault-Resilience

Fault-Resilience of ZEnc (cf. Alg. 3). In order to guarantee fault-resilience, we show that every change to the output of **ZEnc** (cf. Alg. 3) introduced by s internal faults can also be achieved by s direct faults on the output. For $j = k, \ldots, d$, we denote the value $\sum_{j=k}^{l} r_j \cdot Z_{j,i}$

by $e_i[l]$. Now, an adversary can use the faults to fault a set $J \subseteq \{k, \ldots, d+1-k\}$ of random values, i.e., r_j is faulted to $r_j + \Delta_j$ for $j \in J$. Furthermore, they can fault a set $I \subseteq \{0, \ldots, n-1\} \times \{k, \ldots, d+1-k\}$ of values $e_i[l]$, i.e., these values are now $e_i[l] + \Delta_{i,l}$. For two sets J and I, let $D_{J,I}$ be the output distribution of **ZEnc** (cf. Alg. 3) if the values are faulted according to these fault sets.

Lemma 19 ([BEF+23, Lem. 6]). *For all $J \subseteq \{k, \ldots, d+1-k\}$ and all $I \subseteq \{0, \ldots, n-1\} \times \{k, \ldots, d+1-k\}$, we have $D_{J,I} \equiv D_{\emptyset,I}$*

Proof. For $(e_i)_{i\in[n]} \leftarrow_\$ D_{J,I}$, it is easy to see that

$$e_i = \sum_{j \notin J} r_j \cdot Z_{j,i} + \sum_{j \in J} ((r_j + \Delta_j) \cdot Z_{j,i}) + \sum_{(i,l)\in I} \Delta_{i,l}.$$

Let r'_j with $r'_j = r_j$ for $j \notin J$ and $r'_j = r_j + \Delta_j$ for $j \in J$, then we get

$$e_i = \sum_{j\in[n]} r'_j \cdot Z_{j,i} + \sum_{(i,l)\in I} \Delta_{i,l}.$$

Since r_i is i.i.d., it holds $r'_j \equiv r_j$, and hence

$$e_i \equiv \sum_{j\in[n]} r_j \cdot Z_{j,i} + \sum_{(i,l)\in I} \Delta_{i,l},$$

which implies the statement. □

Lemma 20 ([BEF+23, Lem. 7]). *For all $I \subseteq \{0, \ldots, n-1\} \times \{k, \ldots, d+1-k\}$, there exist a sharing $(f_i)_{i\in[n]}$ with $|\mathsf{supp}\,(f_i)_{i\in[n]}| \leq |I|$ such that $D_{\emptyset,I} = D_{\emptyset,\emptyset} + (f_i)_{i\in[n]}$.*

Proof. For $(e_i)_{i\in[n]} \leftarrow_\$ D_{\emptyset,I}$, we have

$$e_i = \sum_{j\in[n]} r_j \cdot Z_{j,i} + \sum_{(i,l)\in I} \Delta_{i,l}.$$

Let's define $f_i = \sum_{(i,l)\in I} \Delta_{i,l}$, which means that $(e_i)_{i\in[n]} - (f_i)_{i\in[n]}$ is distributed according to $D_{\emptyset,\emptyset}$. Furthermore, we have $|\mathsf{supp}\,(f_i)_{i\in[n]}| \leq |I|$. □

Using the above lemmas, we can derive the subsequent lemma on the fault-resilience of ZEnc_n^d (cf. Alg. 3).

Lemma 21 ([BEF+23, Thm. 18]). *Every change to the output of ZEnc_n^d (cf. Alg. 3) introduced by s internal faults can be achieved by s direct faults on the output.*

Fault-Resilience of sZEnc_n^d (cf. Alg. 4). Because sZEnc_n^d (cf. Alg. 4) is only a share-wise addition of ZEnc_n^d (cf. Alg. 3), we have the following lemma for sZEnc_n^d (cf. Alg. 4).

Lemma 22 ([BEF+23, Thm. 19]). *Every change to the output of sZEnc_n^d (cf. Alg. 4) introduced by s internal faults can be achieved by s direct faults on the output.*

Proof. This Lemma follows directly from the fact that sZEnc_n^d consists of share-wise addition of zero encryption ZEnc_n^d. Note that for share-wise addition internal faults can be pushed to the output by associativity of the operation. □

Fault-Resilience of Refresh (cf. Alg. 6)

Lemma 23 ([BEF+23, Thm. 20]). *The gadget* **Refresh** *(cf. Alg. 6) is e-fault-robust (cf. Def. 3) w.r.t. $\mathcal{F}^+$.*

Proof. Because **Refresh** (cf. Alg. 6) satisfies the fault-invariance (cf. Def. 7), each fault can be pushed to an output which satisfies e-fault-robustness (cf. Def. 3) w.r.t. $\mathcal{F}^+$. □

Fault-Resilience of SplitRed (cf. Alg. 7). Before proving the fault-robustness of **SplitRed** (cf. Alg. 7), we first consider two scenarios in which faults can happen in **SplitRed** (cf. Alg. 7).

1. s_{input} faulted inputs without faults in **SplitRed** (cf. Alg. 7). From Lemma 11, we have shown that the faulted output $\tilde{F}'_i$, $\tilde{F}''_i$ of **SplitRed** (cf. Alg. 7) satisfy the following relation:
$$\tilde{F}'_i + \tilde{F}''_i = \begin{cases} F'_i + F''_i + f_{n-i-1} & \text{if } 0 \le i < e \\ F'_i + F''_i & \text{if } e \le i \le n-1 \end{cases},$$
where f_l denotes the l-th coefficient of polynomial $f(X)$, and $f(X)$ is used to generate the inputs $(F_i)_{i\in[n]}$ for **SplitRed** (cf. Alg. 7). More specifically, the faulted result caused by s_{input} (that is smaller than e) input faults can be seen as the coefficients of $x^{d+1}, x^{d+2}, \ldots, x^{n-s_{\text{input}}}$ that is generated by the random experiment defined in the Def. 3.
2. Non-faulted inputs with s_{int} faults in **SplitRed** (cf. Alg. 7). Because **SplitRed** (cf. Alg. 7) satisfies the fault-invariance (cf. Def. 7), all faults introduced during the computation by the adversary can be pushed either into the inputs or the outputs of the gadget.

Based on the above discussion, we obtain the following lemma.

Lemma 24 ([BEF+23, Thm. 21])**.** *If the (faulted) input to* **SplitRed** *(cf. Alg. 7) has degree at least $n - s_{\text{input}}$, the (faulted) outputs have degree at least $n - s_{\text{int}} - s_{\text{input}}$, if s_{int} faults are performed during the computation.*

Proof. Based on the fact that each fault (except the input faults) can only affect one single output share, s_{int} faults will cause the degree of output shares to be reduced at most s_{int}. □

Fault-Resilience of SWComp (cf. Alg. 8) and $\textsf{Mult}_{\textsf{k-Input}}$ (cf. Alg. 9). Since both gadgets satisfy the fault-invariance (cf. Def. 7), Lemma 24 also holds for them.

Lemma 25 ([ABEO24, Section 5.4])**.** $\textsf{Mult}_{\textsf{k-Input}}$ *(cf. Alg. 9) is e-f-robustness.*

Proof. The only difference to the proof in [ABEO24, Section 5.4] is that we adapted the gadgets **SplitRed** and **Refresh** (used in **SWComp**). Hence, it remains to prove that **SplitRed** and **Refresh** satisfy the necessary security requirements, which are already proven in Lemma 24 and Lemma 23. □

E LaOla-Based PSS Compiler

In this section, we show that our *LaOla-based PSS Compiler* using packed secret sharing [FY92] introduces only *mild* changes to the security properties compared to the original LaOla compiler in [BEF+23], which only supports Shamir's secret sharing [Sha79].

As shown in Lemma 26, we can use our multiplication gadget $\textsf{Mult}_{\textsf{k-Input}}$ (cf. Alg. 9), refresh gadget **Refresh** (cf. Alg. 6), and the share-wise addition **SWAdd** (cf. Alg. 1) and multiplication gadget **SWMult** (cf. Alg. 2) to build a circuit compiler secure against additive fault $\mathcal{F}^+$ using $n = d + e + 1$ shares.

Lemma 26 (Additive LaOla-Pack-Complier [BEF+23, Thm. 8])**.** *For any $d, e \in \mathbb{N}$ there is a circuit compiler that is given an arithmetic circuit C over $\mathbb{F}$ and outputs an arithmetic circuit C' over $\mathbb{F}^n$ where $n = d + e + 1$ with*

1. *C' has the same functionality as C,*
2. *$T[\mathsf{C}']$ is probing secure for any $T \in \mathsf{A}(\mathcal{F}^+)$ and*

   (a) up to $(d+1-k)$ probes in $T[\mathcal{C}']$ (threshold probing security), or
   (b) up to $(d+1-k)/2$ probes in every gadget of $T[\mathsf{C}']$ (region probing security).
3. *C' is e-fault-robust (cf. Def. 3) with respect to $\mathcal{F}^+$.*

Furthermore, using [BEF+23, Cor. 2], the above LaOla-Pack-Complier that is only secure against the additive fault $\mathcal{F}^+$ can also be extended to resist the wire-independent fault $\mathcal{F}^{\mathsf{ind}}$. This is shown in Lemma 27.

Lemma 27 (General LaOla-Pack-Complier [BEF+23, Thm. 8])**.** *For any $d, e \in \mathbb{N}$ there is a circuit compiler that is given an arithmetic circuit C over $\mathbb{F}$ and outputs an arithmetic circuit C' over $\mathbb{F}^n$ where $n = d+e+1$ with*
1. *C' has the same functionality as C,*
2. *$T[\mathsf{C}']$ is probing secure for any $T \in \mathsf{A}(\mathcal{F}^+)$ with $|T| < e$ and*
   (a) up to $(d+1-k-e)$ probes in $T[\mathcal{C}']$ (threshold probing security), or
   (b) up to $(d+1-k)/2$ probes in every gadget of $T[\mathsf{C}']$ when the faults are counted as probes (region probing security).
3. *$T[\mathsf{C}']$ is $(d+1-k-e)$ probing secure for any $T \in \mathsf{A}\left(\mathcal{F}^{\mathsf{ind}}\right)$ with $|T| < e$*
4. *C' is e-fault-robust (cf. Def. 3) with respect to $\mathcal{F}^{\mathsf{ind}}$.*

The only difference between our *packed* LaOla compiler and the original compiler in [BEF+23] is that we have modified the multiplication gadget $\mathbf{Mult_{k-Input}}$ (cf Alg. 9) and refresh gadget **Refresh** (cf. Alg. 6). More importantly, our modified gadgets satisfy all the necessary security requirements (cf. Sec. 3 and cf. Sec. 4) as the original gadgets [BEF+23]. Hence, the correctness and security of Lemma 26 and Lemma 27 can be proven as in [BEF+23].

F Building Blocks for Fault Detection

Recall that we would usually detect faults right before decoding. However, the following fault detection **FaultDetect** (cf. Alg. 13) algorithm can also be used in case one aims to detect errors during circuit operation. The idea of **FaultDetect** can be summarized as follows. We hide the secrets embedded in our input polynomial $f(X)$ by adding a completely random polynomial of degree d. (Namely, we compute a refresh, but instead of using a zero-encoding, we use an encryption of k random values, $r_0, \dots, r_{k-1}$.) Then, we recalculate the highest coefficients $f_i, i \in \{d+1, \dots, n-1\}$ of $f(X)$, using our extended matrix V, as presented in Section A.1. Now, we check if any of these coefficients are zeros. We abort if a coefficient is non-zero, otherwise we continue. This procedure is repeated e times.

We begin by providing an algorithm $\mathbf{RandomEnc}_n^d$ (cf. Alg. 11) to create *degree-d* random polynomials. In order to guarantee SNI, we use the algorithm **Refresh** (cf. Alg. 6) as a baseline and only change the underlying **ZEnc** (cf. Alg. 3) into $\mathbf{RandomEnc}_n^d$ to create random polynomials. Please note that it is sufficient that **FaultDetect** (cf. Alg. 13) is

Algorithm 11 $\mathbf{RandomEnc}_n^d$ for $n = d+e+1$ and $k \le d$

Output: Randomized sharing $(g_i)_{i\in[n]}$ of random vector $\boldsymbol{v}_{\mathsf{random}} = (r_0, \dots, r_{k-1}, r_k, \dots, r_d) \leftarrow_{\$} \mathbb{F}^{d+1}$.

```
initialize $(g_i)_{i\in[n]}$
$(r_0, \dots, r_d) \leftarrow_{\$} \mathbb{F}^{d+1}$
for $i \in [n]$ do
    for $j \in [d+1]$ do
        $g_i \leftarrow g_i \oplus r_j M_{j,i}$
```

secure against an adversary that inserts at most $e-1$ faults, since the fault injection threshold is e and an adversary must have faulted at least one element previously to obtain a polynomial of degree $> d$.

Since $\textsf{RandomEnc}_n^d$ is very similar to **sZEnc** (except for not choosing zeros as secrets, but random elements), Lemma 22 holds here as well.

Algorithm 12 $\textsf{RandomRefresh}_n^d$ for $n = d+e+1$ and $k \le d$

```
Input  : Degree-$d$ shares of $\boldsymbol{v} = (v_0, \dots, v_d)$ as $(F_i)_{i\in[n]}$
Output : A degree-$d$ shares of a randomized $\boldsymbol{v}$ as $(F_i')_{i\in[n]}$
1 initialize $(y_i)_{i\in[n]}$
2 for $j \in [d+1-k]$ do
3     $(g_i)_{i\in[n]} \leftarrow \textsf{RandomEnc}_n^d$(cf. Alg. 11)
4     $(y_i)_{i\in[n]} \leftarrow (y_i)_{i\in[n]} \oplus (g_i)_{i\in[n]}$
5 $(Q_i)_{i\in[n]} \leftarrow (F_i)_{i\in[n]} + (y_i)_{i\in[n]}$
6 return $(Q_i)_{i\in[n]}$
```

Moreover, $\textsf{RandomRefresh}_n^d$ uses $\textsf{RandomEnc}_n^d$ instead of **ZEnc**, but other than that, it is exactly the same algorithm as **Refresh** (cf. Alg. 6). Lemma 22 holds for both, **ZEnc** and $\textsf{RandomEnc}_n^d$, as the proof relies on the fact, that the shares $(R_i)_{i\in[n]}$ of **ZEnc** (resp. $\textsf{RandomEnc}_n^d$) are $d+1-k$-wise independent, which is clearly the case. With the help of Lemma 22, we can show that $\textsf{RandomRefresh}_n^d$ is frSNI.

Lemma 28 ([BEF+23, adapted from Theorem 17])**.** *The* $\textsf{RandomRefresh}_n^d$ *gadget is* $(d+1-k)$-frSNI*, w.r.t* $\mathcal{F}^+$*.*

Proof. The proof is identical to the one of Lemma 10, as the only change to **Refresh** (cf. Alg. 6) is the use of $\textsf{RandomEnc}_n^d$ instead of **ZEnc**. However, all properties needed for **ZEnc** within the proof of Lemma 10 are fulfilled by $\textsf{RandomEnc}_n^d$ as well, since we rely on the fact, that the shares $(R_i)_{i\in[n]}$ are $d+1-k$-wise independent, which is clearly the case, because $\textsf{RandomEnc}_n^d$ constructs a polynomial with $d+1$ uniformly random coefficients. Note that an adversary is able to insert a fault to the coefficients. However, this only results in a shift of a uniformly random value, which remains uniformly distributed. ☐

Note that similar to [DN20], we assume the existence of an abort state $\perp$ in **FaultDetect** (cf. Alg. 13). In case such a state is not already implemented, we can create an abort flag by using a global variable $\perp_f \in \{0,1\}^e$. This variable corresponds to a vector of size e and is initially set to the all-zero vector. In case an error is detected (line 6), all entries of $\perp_f$ are set to 1. As there are e entries, the adversary can not fault all of them, in case he did insert a previous fault. We begin by showing correctness under the existence of at most d probes and $e-1$ faults.

Lemma 29. *Algorithm 13 outputs $\perp$ if the degree of the polynomial $P(X)$ underlying the input shares $(F_i)_{i\in[n]}$ is greater than $d+1-k$, and an adversary can insert at most $e-1$ faults.*

Proof. We do the following e times. The algorithm begins by calculating shares of a random polynomial of degree d, which are share-wise added to the input shares. If $P(X)$ had degree $\geq d$, share-wise addition with a degree d polynomial does not change the degree of the underlying polynomial. Further, due to robustness, the error detection is independent of the underlying secret, meaning that $\textsf{RandomRefresh}_n^d$ does not have an impact on the higher degree coefficients.

Algorithm 13 FaultDetect

```
Input : Faulty shares $(F_i)_{i\in[n]}$, where $f_d, \dots, f_{n-1}$ are the coefficients of the underlying
        polynomial
1 for $k \in [e]$ do
2     $(G_i)_{i\in[n]} \leftarrow \textsf{RandomRefresh}_n^d(F_0, \dots, F_{n-1})$
3     for $j \in [d+1, n]$ do
4         for $i \in [n]$ do
5             $f_j^{(k)} \leftarrow f_j^{(k)} + G_i \cdot V_{i,j}^{-1}$
6         if $f_j^{(k)} \neq 0$ then
7             return $\perp$
```

Then, we recalculate the highest coefficients $f_j, j \in [d+1, n]$ as described in the decryption, i.e. $f_j = \sum_{i=0}^{n-1} G_i \cdot V_{i,j}^{-1}$, where $(G_i)_{i\in[n]}$ denotes our randomized shares.

All higher degree coefficients $f_j, j \in [d+1, n]$ are calculated e times. Thus, if any coefficient $f_j \neq 0, j \in [d+1, n]$, the adversary can only fault $e-1$ of the checks, but not e. This means that the algorithm returns $\perp$, if any of the coefficients was non-zero. In case there exists no abort $\perp$, we use a vector of size e to denote the abort. Namely, the vector is initialized to the all zero vector. In case we would return $\perp$, we instead set all elements of the vector to 1. Hence, with a faulting threshold of $e-1$ (as we assume the adversary faulted already at least once to increase the degree of the polynomial), the adversary cannot change all elements of the vector. If any element of the vector is non-zero, we set the abort-flag. □

Lemma 30. *For a probing threshold of $d+1-k$, Algorithm 13 does not reveal any information about the underlying secrets.*

Proof. Due to the frSNI property of $\textsf{RandomRefresh}_n^d$, the adversary does not learn anything about the underlying secret within this part of the algorithm. Now, after computing $\textsf{RandomRefresh}_n^d$, the shares $(G_i)_{i\in[n]}$ cannot learn any information about the secret anymore (with a probing threshold of $d+1-k$), since it embeds $v_i + \sum_{j\in[d+1-k]} r_i^{(j)}$, where v_i denotes the secret embedded in $(F_i)_{i\in[n]}$ and $r_i^{(j)} \in \mathbb{F}$ are uniformly chosen values. Further, due to robustness, the error detection is independent of the underlying secret, meaning that $\textsf{RandomRefresh}_n^d$ does not have an impact on the higher degree coefficients and the adversary cannot learn any information about the secrets from the higher degree coefficients. □

G Complexity Comparisons

G.1 Complexity Comparison for AES ($e = t \in \{1, \dots, 30\}$)

In Table 7, we provide the randomness and computational complexity for $t = e \in \{1, \dots, 30\}$. We calculate the computational complexity of one round of AES-128 (that takes 16 inputs, and each input is in the field $\mathbb{F}_{2^8}$) as follows: For duplicated Boolean masking [FRSG22, FGM+23] that requires $(t+1) \cdot (2e+1)$ shares, we use the small depth-16 circuit for AES S-Box from [BP12], as described by [FRSG22, FGM+23]. This AES S-Box Boolean circuit has 34 AND gates, 90 XOR gates, and 4 XNOR gates. For other AES operations (cf. [LMW+24]), e.g., AddRoundKey, ShiftRows and MixColumns, they require 32 AND gates and 1220 XOR gates. In total, one round of AES-128 for 16 inputs needs 576 AND gates and 2660 XOR gates and 64 NOT gates. Then, based on complexity provided by [FRSG22, Tab. 2], we can compute the Boolean circuit complexity for all the Boolean operations in one AES-128 round.

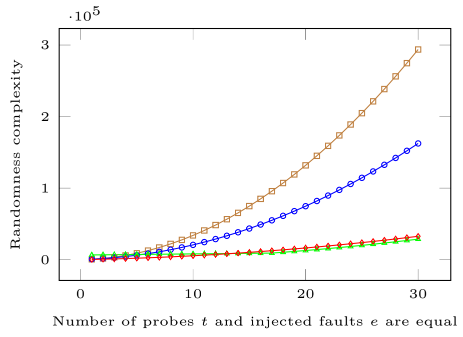
Please note that a crucial error correction component for achieving combined security in the duplicated Boolean masking [FRSG22, FGM+23] is a sorting-based majority function. Since no concrete sorting algorithm was chosen by [FRSG22, FGM+23], we choose insertion sort (with asymptotic complexity $\mathcal{O}(n^2)$ for n inputs), which is known to be efficient for a small number of faults e (cf. [CLRS22, YYG11]).

In Table 7, we calculate the random bit complexity and Boolean circuit complexity for different masking schemes. For the duplicated Boolean masking [FRSG22, FGM+23], we consider t glitch-extended bit probes and e bit faults. For the original *LaOla* [BEF+23] and the *double-sharing* compiler [ABEO24], and our *LaOla-based PSS* and *BGW-based PSS* compiler, we calculate the total number of arithmetic (or field) operations (i.e., addition and multiplication in $\mathbb{F}_{2^8}$) for one AES-128 round (cf. Sec. 7). Then, we transform these arithmetic operations into Boolean operations and calculate the necessary number of Boolean gates (using the Boolean circuits from [BDST20]) as the computational complexity. Here, we consider t probe and e fault on the field elements from $\mathbb{F}_{2^8}$. We calculate the random bit complexity in a similar manner.

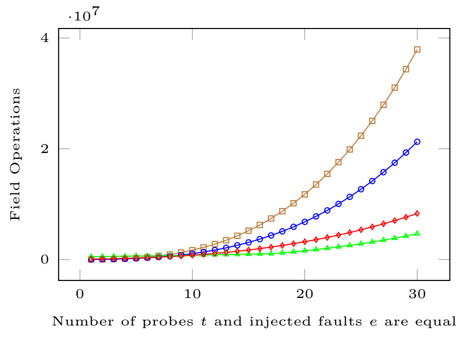
Table 7: Random bit complexity and Boolean circuit complexity for one AES round of our *BGW-based PSS* and *LaOla-based PSS* compiler, the original *LaOla* [BEF+23] and the *double-sharing* compiler [ABEO24], and the *Duplicated Boolean masking* [FRSG22, FGM+23] as in Figure 1. The security parameters are t probes and e faults with $e = t$ and $e, t \in \{1, \dots, 30\}$. The best complexity is marked in bold.

$t = e$	LaOla [BEF+23]		Double-Sharing [ABEO24]		LaOla-based PSS (this work)		BGW-based PSS (this work)		Dup. Boolean Mask [FRSG22, FGM+23]	
	Rand	Comp	Rand	Comp	Rand	Comp	Rand	Comp	Rand	Comp
1	4096	**803760**	5888	940576	53040	41981007	2448	4017967	**1152**	1260312
2	13312	2634320	12864	**2499408**	54328	44276032	5328	6840689	**3456**	12792860
3	27648	6262256	22528	**5240896**	55616	46631646	8640	10430860	**6912**	60716208
4	47104	12340368	34880	**9507760**	56904	49047847	12384	14871098	**11520**	197630676
5	71680	21521456	49920	**15642720**	58192	51524637	**16560**	20244025	17280	512023688
6	101376	34458320	67648	**23988496**	59480	54062014	**21168**	26632259	24192	1137670092
7	136192	51803760	88064	34887808	60768	56659980	**26208**	**34118422**	32256	2263032480
8	176128	74210576	111168	48683376	62056	59318533	**31680**	**42785132**	41472	4140661508
9	221184	102331568	136960	65717920	63344	62037675	**37584**	**52715011**	51840	7096596216
10	271360	136819536	165440	86334160	64632	64817404	**43920**	**63990677**	63360	11539764348
11	326656	178327280	196608	110874816	65920	**67657722**	**50688**	76694752	76032	17971382672
12	387072	227507600	230464	139682608	67208	**70558627**	**57888**	90909854	89856	26994357300
13	452608	285013296	267008	173100256	68496	**73520121**	**65520**	106718605	104832	39322684008
14	523264	351497168	306240	211470480	**69784**	**76542202**	73584	124203623	120960	55790848556
15	599040	427612016	348160	255136000	**71072**	**79624872**	82080	143447530	138240	77363227008
16	679936	514010640	392768	304439536	**72360**	**82768129**	91008	164532944	156672	105143486052
17	765952	611345840	440064	359723808	**73648**	**85971975**	100368	187542487	176256	140383983320
18	857088	720270416	490048	421331536	**82752**	**99781942**	110160	212558777	196992	184495167708
19	953344	841437168	542720	489605440	**92336**	**114940485**	120384	239664436	218880	239054979696
20	1054720	975498896	598080	564888240	**102400**	**131508804**	131040	268942082	241920	305818251668
21	1161216	1123108400	656128	647522656	**112944**	**149548099**	142128	300474337	266112	386726108232
22	1272832	1284918480	716864	737851408	**123968**	**169119570**	153648	334343819	291456	483915366540
23	1389568	1461581936	780288	836217216	**135472**	**190284417**	165600	370633150	317952	599727936608
24	1511424	1653751568	846400	942962800	**147456**	**213103840**	177984	409424948	345600	736720221636
25	1638400	1862080176	915200	1058430880	**159920**	**237639039**	190800	450801835	374400	897672518328
26	1770496	2087220560	986688	1182964176	**172864**	**263951214**	204048	494846429	404352	1085598417212
27	1907712	2329825520	1060864	1316905408	**186288**	**292101565**	217728	541641352	435456	1303754202960
28	2050048	2590547856	1137728	1460597296	**200192**	**322151292**	231840	591269222	467712	1555648254708
29	2197504	2870040368	1217280	1614382560	**214576**	**354161595**	246384	643812661	501120	1845050446376
30	2350080	3168955856	1299520	1778603920	**229440**	**388193674**	261360	699354287	535680	2176001546988

In addition to the random bit and Boolean circuit complexity in the binary field, we also compute the randomness and computational complexity in the field $\mathbb{F}_{2^8}$. Specifically, in Figure 2 and Table 8, we compare the randomness complexity and computational complexity for one AES round of our *BGW-based PSS* and *LaOla-based PSS* compiler, and the original *LaOla* [BEF+23] and the *double-sharing* compiler [ABEO24].



(a) Randomness Complexity



(b) Field Operations

Figure 2: Randomness complexity (a) and computational complexity (b) for one AES round of our *BGW-based PSS compiler* (◇) and our *LaOla-based PSS compiler* (△), the original *LaOla* [BEF+23] (□) and the *double-sharing* compiler [ABEO24] (○). The security parameters are t probes and e faults with $e = t$ and $e, t \in \{1, \ldots, 30\}$.

Table 8: Randomness complexity and computational complexity for one AES round of our *BGW-based PSS* compiler and our *LaOla-based PSS* compiler, and the original *LaOla* [BEF+23] and the *double-sharing* compiler [ABEO24]. The security parameters are t probes and e faults for values in field $\mathbb{F}_{2^8}$ with $e = t$ and $e, t \in \{1, \ldots, 30\}$. The best complexity is marked in bold.

$t = e$	LaOla [BEF+23]		Double-Sharing [ABEO24]		LaOla-based PSS (this work)		BGW-based PSS (this work)	
	Rand	Comp	Rand	Comp	Rand	Comp	Rand	Comp
1	512	**11256**	736	12688	6630	510331	**306**	53915
2	1664	35720	1608	**32904**	6791	538256	**666**	88309
3	3456	82712	2816	**67744**	6952	566919	**1080**	131830
4	5888	159912	4360	**121240**	7113	596319	**1548**	185448
5	8960	275000	6240	**197424**	7274	626458	**2070**	250136
6	12672	435656	8456	**300328**	7435	657335	**2646**	326866
7	17024	649560	11008	433984	7596	688950	**3276**	**416611**
8	22016	924392	13896	602424	7757	721302	**3960**	**520341**
9	27648	1267832	17120	809680	7918	754393	**4698**	**639029**
10	33920	1687560	20680	1059784	8079	788222	**5490**	**773647**
11	40832	2191256	24576	1356768	8240	**822789**	**6336**	925168
12	48384	2786600	28808	1704664	8401	**858093**	**7236**	1094562
13	56576	3481272	33376	2107504	8562	**894136**	**8190**	1282802
14	65408	4282952	38280	2569320	**8723**	**930917**	9198	1490860
15	74880	5199320	43520	3094144	**8884**	**968436**	10260	1719709
16	84992	6238056	49096	3686008	**9045**	**1006692**	11376	1970319
17	95744	7406840	55008	4348944	**9206**	**1045687**	12546	2243663
18	107136	8713352	61256	5086984	**10344**	**1210963**	13770	2540713
19	119168	10165272	67840	5904160	**11542**	**1392214**	15048	2862442
20	131840	11770280	74760	6804504	**12800**	**1590162**	16380	3209820
21	145152	13536056	82016	7792048	**14118**	**1805525**	17766	3583820
22	159104	15470280	89608	8870824	**15496**	**2039025**	19206	3985414
23	173696	17580632	97536	10044864	**16934**	**2291380**	20700	4415575
24	188928	19874792	105800	11318200	**18432**	**2563312**	22248	4875273
25	204800	22360440	114400	12694864	**19990**	**2855539**	23850	5365481
26	221312	25045256	123336	14178888	**21608**	**3168783**	25506	5887171
27	238464	27936920	132608	15774304	**23286**	**3503762**	27216	6441316
28	256256	31043112	142216	17485144	**25024**	**3861198**	28980	7028886
29	274688	34371512	152160	19315440	**26822**	**4241809**	30798	7650854
30	293760	37929800	162440	21269224	**28680**	**4646317**	32670	8308192

G.2 Complexity Comparison for Multiplication and Error Detection

We compare both the multiplication gadget and error-detection (or correction) gadget with the duplicated Boolean masking scheme [DN20, FRSG22, FGM+23].

For the original *LaOla* compiler [BEF+23], *double-sharing* compiler [ABEO24], and our *LaOla-based PSS* compiler and *BGW-based PSS* compiler, we first calculate the total number of arithmetic operations (i.e., addition and multiplication operations in the field $\mathbb{F}^{2^8}$) for single run of the multiplication gadget (cf. Alg. 9) embedding 16 secret values from field $\mathbb{F}_{2^8}$ into one polynomial, which is the same as the case in the AES-128 evaluation on field $\mathbb{F}_{2^8}$ (cf. Sec. 7). Then, we transform these arithmetic operations into Boolean operations and calculate the necessary number of Boolean gates (using the Boolean circuits from [BDST20]) as the computational complexity. Similarly, we compute the computational complexity for our error-detection gadget (cf. Alg. F).

For the duplicated Boolean masking [DN20, FRSG22, FGM+23], we calculate the total number of Boolean gates (e.g., AND and XOR gates) for 16$\times$ (corresponding to 16 secret bits) runs of the multiplication gadget ([DN20, Algo. 2], [FRSG22, FGM+23, Algo. 2]) the computational complexity. In addition, we compute the complexity of error detection ([DN20, Algo. 3, 4] and error correction ([FRSG22, FGM+23]). The error correction in [FRSG22, FGM+23] relies on the majority function, which can be realized using sorting-based algorithms. Specifically, [FRSG22, FGM+23] achieves secret sharing correction by re-setting each of the duplicated shares to the output of a majority function, which takes all the duplicated shares as input. However, [FRSG22, FGM+23] didn't provide the concrete algorithm or Boolean circuits for such a majority function. Hence, we choose the insertion sort [CLRS22] (with asymptotic complexity $\mathcal{O}(n^2)$ for n inputs) and calculate the corresponding complexity. More specifically, we compute the concrete number Boolean gates for realizing the insertion sort algorithm by transforming all the arithmetic operations (e.g., comparison, index subtraction) inside the insertion sort algorithm using the Boolean circuit from [BDST20]. In Figure 9, we compare the computational complexity for different compilers for t probes and e faults.

Table 9: Boolean circuit complexity for the multiplication operation and error-detection/correction in one AES-128 round of our *BGW-based PSS* and *LaOla-based PSS* compiler, and the duplicated Boolean masking [DN20, FRSG22, FGM+23]. The security parameters are t probes and e faults with $e = t$ and $e, t \in [1, 30]$. The best complexity for achieving combined security is marked in bold. Please note that [FRSG22] shows that *NINA** is not combined secure.

$t = e$	Multiplication				Error Detection/Correction			
	LaOla-PSS (this work)	BGW-PSS (this work)	CINI-MINIs [FGM+23]	NINA* [DN20]	LaOla-PSS (this work)	BGW-PSS (this work)	CINI-MINIs [FGM+23]	NINA* [DN20]
1	5997328	436968	**34560**	1408	650896	59024	**16896**	224
2	6303430	749768	**354240**	5520	1338388	137304	**117120**	816
3	6616944	**1147840**	1684480	15104	2063106	**285910**	419328	1984
4	6937870	**1640364**	5486400	33520	2825068	**526706**	1094400	3920
5	7266208	**2236520**	14217984	64896	3624292	**884004**	2365440	6816
6	7601958	**2945488**	31595200	114128	4460796	**1384564**	4507776	10864
7	7945120	**3776448**	62853120	186880	5334598	**2057594**	7848960	16256
8	8295694	**4738580**	115007040	289584	6245716	**2934750**	12768768	23184
9	8653680	**5841064**	197113600	429440	7194168	**4050136**	19699200	31840
10	9019078	**7093080**	320531904	614416	8179972	**5440304**	29124480	42416
11	9391888	**8503808**	499184640	853248	9203146	**7144254**	41581056	55104
12	**9772110**	10082428	749819200	1155440	10263708	**9203434**	57657600	70096
13	**10159744**	11838120	1092268800	1531264	**11361676**	11661740	77995008	87584
14	**10554790**	13780064	1549713600	1991760	**12497068**	14565516	103286400	107760
15	**10957248**	15917440	2148941824	2548736	**13669902**	17963554	134277120	130816
16	**11367118**	18259428	2920610880	3214768	**14880196**	21907094	171764736	156944
17	**11784400**	20815208	3899508480	4003200	**16127968**	26449824	216599040	186336
18	**13871728**	23593960	5124813760	4928144	**19369834**	31647880	269682048	219184
19	**16175160**	26604864	6640358400	6004480	**23066870**	37559846	331968000	255680
20	**18704896**	29857100	8494887744	7247856	**27258874**	44246754	404463360	296016
21	**21471136**	33359848	10742321920	8674688	**31987276**	51772084	488226816	340384
22	**24484080**	37122288	13442016960	10302160	**37295138**	60201764	584369280	388976
23	**27753928**	41153600	16659025920	12148224	**43227154**	69604170	694053888	441984
24	**31290880**	45462964	20464360000	14231600	**49829650**	80050126	818496000	499600
25	**35105136**	50059560	24935249664	16571776	**57150584**	91612904	958963200	562016
26	**39206896**	54952568	30155405760	19189008	**65239546**	104368224	1116775296	629424
27	**43606360**	60151168	36215280640	22104320	**74147758**	118394254	1293304320	702016
28	**48313728**	65664540	43212329280	25339504	**83928074**	133771610	1489974528	779984
29	**53339200**	71501864	51251270400	28917120	**94634980**	150583356	1708262400	863520
30	**58692976**	77672320	60444347584	32860496	**106324594**	168915004	1949696640	952816