

Masked CROSS: Masking the CROSS Digital Signature Scheme at Arbitrary Order

Khan Keren Mengwi¹, Puja Mondal², Achille
Ecladore Tchahou Tchendjeu³, Emmanuel Fouotsa¹, and Suparna Kundu⁴

¹ Centre for Cybersecurity and Mathematical Cryptology, University of Bamenda, Cameroon
khankeren@uniba.cm, emmanuel Fouotsa@yahoo.fr

² Department of Computer Science and Engineering, IIT Kanpur, India
pujamondal@cse.iitk.ac.in

³ NAHPI, University of Bamenda, Cameroon
tchahoutchendjeu@gmail.com

⁴ COSIC, KU Leuven, Kasteelpark Arenberg 10, Bus 2452, B-3001 Leuven-Heverlee, Belgium
suparna.kundu@esat.kuleuven.be

Abstract. CROSS is a code-based signature scheme built on the Restricted Syndrome Decoding Problem and is a second-round candidate in NIST’s additional digital signature standardization process. Recent work shows that its reference implementation is vulnerable to side-channel analysis, allowing an attacker to recover the long-term secret key from a single power trace. No masking countermeasure has so far been designed for CROSS’s restricted-syndrome framework, leaving its practical side-channel resistance unresolved. In this work, we present a full sensitivity analysis and a gadget-based masking schedule (G_0 – G_8) for the CROSS signing algorithm of the R-SDP(G) variant. One gadget in this schedule, the restriction-lift exponentiation, has no counterpart in prior masked multivariate- or lattice-based signatures: it masks $g^{\vec{v}[i]}$ via table lookup, share-serial conversion, and ISW-multiplication sub-gadgets rather than via a fixed linear or bilinear map. We rigorously prove t_{mask} -probing security for each gadget individually and for their full composition, using the NI, SNI, and NIO frameworks. We implement the resulting masked signer for an arbitrary number of shares d and evaluate it on an x86-64 platform, measuring cycle overheads of $21.73\times$ at first order and $78.05\times$ at second order relative to the unmasked baseline. At first order, our masked CROSS is cheaper than a comparably masked Dilithium-2 ($38\times$), but Dilithium-2 overtakes it at second order ($70\times$). This crossover cost is driven by the $O(d^3)$ chained-ISW cost of G_3/G_7 compared to Dilithium’s cheaper $O(d^2)$ scaling.

Keywords: Post-Quantum Cryptography · Digital Signature · CROSS · Side-Channel Attack · Secure implementations · Masking

1 Introduction

CROSS [3] is a post-quantum digital signature scheme based on the Restricted Syndrome Decoding Problem R-SDP or Restricted Syndrome Decoding Problem

with Subgroup (R-SDP(G)). This code-based scheme is constructed from a zero-knowledge identification protocol (CROSS-ID) to a non-interactive signature via the Fiat-Shamir transform. This scheme was among the second-round candidates in the ongoing National Institute of Standards and Technology (NIST) standardization procedure [1,2]. Although this scheme has not been advanced to the third round, it is a post-quantum code-based candidate scheme constructed from a zero-knowledge identification protocol and uses a restricted-syndrome-decoding structure.

Recent works have demonstrated that unprotected implementations of CROSS are vulnerable to powerful side-channel attacks, including single-trace horizontal and correlation power-analysis attacks. Schupp and Sigl [18] presented the first horizontal side-channel attack on CROSS, recovering the full secret key from a single power trace for nearly every published parameter set. Czuprynyko et al. [12] independently confirmed single-trace key recovery via correlation power analysis and reduced the number of rounds an attacker needs to observe by an order of magnitude. Both attacks exploit the same structural weakness in CROSS to recover the long-term secret key. However, a systematic side-channel security analysis of CROSS and corresponding implementation countermeasures remains lacking.

Contribution. In this work, we present a comprehensive sensitivity analysis of the CROSS signing algorithm. We propose a masked implementation to protect against side-channel attacks, specifically differential power attacks. Concretely, our main contributions are as follows:

- **Systematic Sensitivity Analysis (Section 3):** We perform a systematic sensitivity analysis of CROSS signing, adapting the methodology introduced for masking the Unbalanced Oil and Vinegar scheme [16] to CROSS’s restricted-syndrome-decoding structure. We classify every intermediate variable produced during signing, including the long-term restricted error vector, the per-round transitive map $\bar{\mathbf{v}}[i]$, and the ephemeral syndrome, according to whether its Hamming weight, if observed on a power or EM traces, can expose the long-term secret directly or transitively.
- **Complete Masking Schedule (Section 4):** We design a full gadget-based masking schedule, G0–G8, covering key expansion, per-round ephemeral sampling, the transitive-map subtraction, a masked discrete-exponentiation gadget for the restriction lift $g^{\bar{\mathbf{v}}[i]}$, the secret–secret multiplication that produces $\mathbf{u}[i]$, the syndrome computation, commitment formation, and the first-response step. The restriction-lift gadget has no counterpart in prior masked multivariate or lattice-based signature schemes. It masks bit-serial modular exponentiation via a chain of arithmetic-to-Boolean and Boolean-to-arithmetic conversions rather than a fixed linear or bilinear map.
- **Formal Probing Security Proofs (Sections 4 & 5):** We formally prove the t_{mask} -probing security of every individual gadget. In Section 5, we compose these into a unified probing-security proof for the complete masked signing algorithm, following the NI/SNI/NIo composability framework of Barthe et al. [4,6] based on the ISW masking model [14].
- **Implementation and Evaluation (Section 6):** We implement the resulting masked signer and evaluate its performance on an x86-64 platform, benchmarking

the overhead against the unmasked CROSS baseline and masked implementations of Dilithium-2 [15].

2 Preliminaries

2.1 Notation

We follow the notation of the CROSS Round 2 specification [3]. Let $\mathbb{F}_p$ be a finite field with p elements and let $\mathbb{F}_z$ be a finite field with z elements, $z < p$. Let $\mathbb{E} \subset \mathbb{F}_p^*$ be the cyclic subgroup of order z generated by g , and write $g^x \in \mathbb{E}$ for the exponent-to-group lift of $x \in \mathbb{F}_z$. The restriction set $\mathbb{E}^n$ is the object space for R-SDP; for R-SDP(G) we additionally work with a subgroup $\mathcal{G} \subset \mathbb{E}^n$ of size z^m , for some $0 < m < n$. We write n and $n-k$ for the code length and syndrome length, t for the number of parallel rounds of CROSS-ID, w for the weight of the second challenge, λ for the security parameter, and $c=2t-1$ for a fixed constant used throughout key and seed derivation.

Vectors and matrices are written in bold (e.g. $\mathbf{v}$, $\mathbf{H}$). A central notational convention, which we adopt directly from the specification, is the distinction between *exponent-domain* and *lifted* representations of a restricted object: an overlined symbol such as $\bar{e}$ or $\bar{\mathbf{v}}[i]$ denotes a quantity living in $\mathbb{F}_z$, while the corresponding non-overlined symbol, e.g. $\mathbf{v}[i] = g^{\bar{\mathbf{v}}[i]}$, denotes the same quantity after being lifted into $\mathbb{F}_p$ through g . This distinction is not cosmetic: every leakage-bearing operation in CROSS signing either acts on an exponent-domain quantity directly (the transitive-map subtraction) or acts on a lifted quantity (the syndrome computation and the first response). Our masking gadgets in Section 4 are organised around exactly this boundary. The public parity-check matrix is $\mathbf{H} \in \mathbb{F}_p^{(n-k) \times n}$ and for R-SDP(G), the public subgroup matrix is $\mathbf{M} \in \mathbb{F}_p^{m \times n}$. We write $\text{Seed}_{sk} \in \{0,1\}^{2\lambda}$ for the long-term secret-key seed, $\text{Salt} \in \{0,1\}^{2\lambda}$ for the salt per sign, $\text{Seed} \in \{0,1\}^\lambda$ for the root of the seed-tree per sign, and $\text{Seed}[i]$ for the round leaf i . For each (sub-)gadget X introduced later, $\mathbf{T}_X(\cdot)$ denotes its elementary operation count and $\mathbf{R}_X(\cdot)$ its fresh-randomness cost.

2.2 Masking and Security Notions

Ishai et al. [14] introduced the t -probing model, in which an adversary adaptively chooses t_{mask} intermediate wire values to observe in a circuit; the circuit is t_{mask} -probing secure if the joint distribution of any t_{mask} probed wires is independent of the secret input. Security against such an adversary is achieved by decomposing every sensitive variable x into a d -share decomposition $[[x]]$, for $d = t_{\text{mask}} + 1$ shares at a target masking order t_{mask} ; we use t_{mask} throughout this report to avoid clashing with the specification's own t , the number of CROSS-ID rounds. Because CROSS mixes Boolean and arithmetic computation, and because its arithmetic computation itself spans two algebraic domains, we use three distinct share types, distinguished by notation:

- **Boolean shares**, $[[x]]_B = (x_0, \dots, x_{d-1})$, satisfying $x = x_0 \oplus x_1 \oplus \dots \oplus x_{d-1}$, arising from CSPRNG/SHAKE output and from any computation performed directly on bitstrings.

- **Exponent-domain arithmetic shares**, $[[\bar{x}]]$, satisfying $\bar{x} = \sum_{k=0}^{d-1} \bar{x}_k \pmod{z}$, carrying quantities that live in $\mathbb{F}_z$ before the exponent-to-group lift, such as $\bar{e}$, $\bar{e}_G$, and $\bar{v}[i]$.
- **Field-domain arithmetic shares**, $[[x]]$, satisfying $x = \sum_{k=0}^{d-1} x_k \pmod{p}$, carrying quantities that live in $\mathbb{F}_p$ after the lift, such as $u[i]$, $v[i]$, $s'[i]$, and $y[i]$.

An adversary who observes at most t_{mask} shares of any single variable, under any one of these three sharing types, learns nothing about it.

The paper [4] introduced composable security notions that allow individually secure masked gadgets to be combined while preserving overall probing security, three of which are central to the masked CROSS construction developed in this report.

t_{mask} -Non-Interference (NI) A gadget is t_{mask} -Non-Interference if the joint distribution of any set of at most t_{mask} intermediate probes, including output wires, can be simulated from at most t_{mask} shares of each input; this ensures that a gadget does not amplify the adversary’s information across the share boundary.

t_{mask} -Strong Non-Interference (SNI) A gadget is t_{mask} -Strong Non-Interference if any set of t_i internal probes and t_o output probes, with $t_i + t_o \leq t_{\text{mask}}$, can be simulated from at most t_i shares of each input; SNI is strictly stronger than NI and permits direct composition of gadgets without an additional refresh between them, since the output of a SNI gadget propagates no more information than t_i input shares.

t_{mask} -Non-Interference with Public Outputs (Nlo) A gadget is t_{mask} -Non-Interference with Public Outputs if, for any set of at most t_{mask} probes, there exists a subset of input shares of size at most t_{mask} from which the probes can be simulated together with the gadget’s public output; this is the appropriate notion for operations such as rejection sampling, whose accept/reject decision is leaked by design.

2.3 The CROSS Signature Scheme

CROSS (Codes and Restricted Objects Signature Scheme) [3] is a Fiat–Shamir signature obtained from a five-pass zero-knowledge identification protocol, CROSS-ID, which proves knowledge of a solution to a restricted syndrome decoding problem. The secret key is a single seed Seed_{sk} , from which the long-term secret is expanded via ExpandSK (Algorithm 18 in Appendix B, line 1). The scheme admits two variants. In the R-SDP variant, ExpandSK generates the exponent-domain secret $\bar{e} \in \mathbb{F}_z^n$ directly, whose lift $\mathbf{e} = g^{\bar{e}} \in \mathbb{E}^n$ is the long-term restricted error vector satisfying the public syndrome relation $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$. In the R-SDP(G) variant, the secret is instead generated in a compact form $\bar{e}_G \in \mathbb{F}_z^m$, with the full exponent-domain vector recovered as $\bar{e} = \bar{e}_G \bar{\mathbf{M}}$ for the public subgroup matrix $\bar{\mathbf{M}}$, before being lifted the same way; this reduces the size of the secret key at the cost of an extra derivation step wherever $\bar{e}$ is needed. Both problems are believed hard against Information-Set-Decoding-style solvers, with the R-SDP(G) variant additionally relying on the subgroup structure of $\mathcal{G}$ not yielding a meaningful decoding shortcut.

CROSS-ID runs t independent parallel rounds to amplify soundness, and the Fiat–Shamir transform replaces the verifier’s two interactive challenges, $\text{chall}_1 \in \mathbb{F}_p^*$ and $\text{chall}_2 \in \{0,1\}^t$, with values derived non-interactively by hashing the prover’s commitments and the message. Algorithm 18 in Appendix B restates the signing procedure, retaining the specification’s own variable names and phase structure. Following the convention of the CROSS v2.0 specification, steps that are common to both variants are written once, spanning the full width of the algorithm. Steps that differ between the two variants are split into two columns separated by a dashed vertical rule: R-SDP(G)-specific content appears on the left, and R-SDP-specific content appears on the right with a dashed horizontal rule bounding the top and bottom of each such block.

Two structural points are worth making explicit before turning to sensitivity analysis. First, For R-SDP(G), the secret key and the per-round ephemeral material each exist in two related forms: a compact exponent-domain form $(\bar{e}_G, \bar{e}'_G[i])$ and a full exponent-domain form $(\bar{e}, \bar{e}'[i])$ derived from it via $\bar{\mathbf{M}}$, so the transitive map is likewise computed twice, once compact $(\bar{\mathbf{v}}_G[i])$ and once full $(\bar{\mathbf{v}}[i])$. The exponent-to-group lift is invoked twice per round on independent inputs, in Phases C and E of Algorithm 18 (Appendix B): on $\bar{\mathbf{v}}[i]$ in Phase C, to form the masked syndrome input, and on $\bar{e}'[i]$ in Phase E, to form the first response.

3 Sensitivity Analysis

A variable X is *sensitive* if knowledge of X or its Hamming weight, i.e., side-channel leakage (observable from a power or electromagnetic trace), enables an adversary to recover the long-term secret keys. For the CROSS signature, a variable X is *sensitive* if its side-channel leakage exposes CROSS long-term secrets Seed_{sk} , $\bar{e}$, $\bar{e}_G$ directly or algebraically. Across the signing data flows (Figures 1 and 2), intermediate variables are categorized as high (long-term secrets), medium (ephemeral secrets), or public, and analyzed step-by-step below.

Key Expansion In the beginning of the signing process, $\text{ExpandSK}(\text{Seed}_{sk})$ algorithm generates public $\mathbf{H}$ and secret $\bar{e}$ (for R-SDP(G) it uses two intermediate computed private value $\bar{e}_G$ and public value $\bar{\mathbf{M}}$). Since leakage during this expansion directly exposes the master key, Seed_{sk} , $\bar{e}$, and $\bar{e}_G$ are rated *high sensitivity*, whereas $\mathbf{H}$ and $\bar{\mathbf{M}}$ are *public*.

Per-Round Ephemeral Sampling and the Transitive Map Figure 1 traces the per-round computation for the R-SDP variant, and Figure 2 the corresponding computation for R-SDP(G).

In the ephemeral sampling process, first a master seed (Seed) and salt (salt) are sampled uniformly and used to generate t seeds $\text{Seed}[1], \dots, \text{Seed}[t]$ by invoking the SeedTreeLeaves algorithm, shown as the $\text{Seed}[i]$ node in Figures 1 and 2. As all of these t seeds are ephemeral, we classify these components as Medium sensitivity. In round i , evaluating a CSPRNG on $\text{Seed}[i]$ yields:

- $\bar{e}'[i]$ and $\mathbf{u}'[i]$ directly for R-SDP (Figure 1);

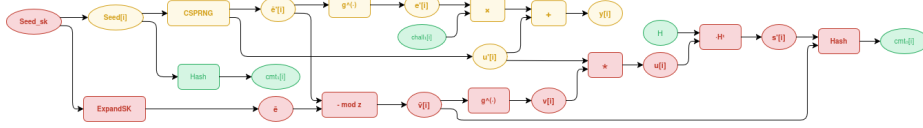


Fig. 1: Sensitivity of the per-round signing computation for the R-SDP variant. **Red** = HIGH (long-term-secret derived), **amber** = MEDIUM (ephemeral-secret), **green** = public.

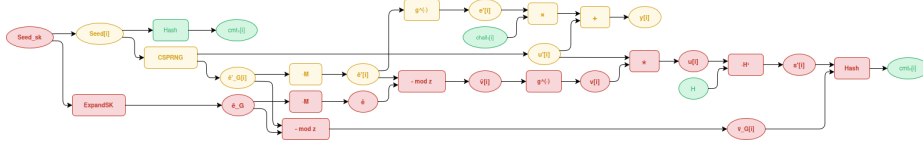


Fig. 2: Sensitivity of the per-round signing computation for the R-SDP(G) variant. The compact secret $\bar{e}_G$ and compact transitive map $\bar{\mathbf{v}}_G[i]$ are equally sensitive to their full counterparts $\bar{e}$ and $\bar{\mathbf{v}}[i]$, since either reveals the long-term secret through the same algebraic relation. The commitment $\text{cmt}_0[i]$ hashes the compact map $\bar{\mathbf{v}}_G[i]$ rather than $\bar{\mathbf{v}}[i]$.

- $\mathbf{u}'[i]$ and a compact $\bar{e}'_G[i]$ for R-SDP(G), where $\bar{e}'[i] = \bar{e}'_G[i] \bar{\mathbf{M}}$ via public matrix multiplication (Figure 2).

We classify $\bar{e}'[i]$ and $\mathbf{u}'[i]$ as Medium sensitivity: although refreshed per round, they transitively interact with the long-term secret $\bar{e}$ via the modular difference $\bar{\mathbf{v}}[i] = \bar{e} - \bar{e}'[i] \bmod z$ (and analogously $\bar{\mathbf{v}}_G[i] = \bar{e}_G - \bar{e}'_G[i] \bmod z$ for R-SDP(G)). This subtraction stage is High sensitivity because physical leakage (e.g., Hamming weight) immediately compromises $\bar{e}$ entrywise. The subsequent componentwise lift $\mathbf{v}[i]_j = g^{\bar{\mathbf{v}}[i]_j} \in \mathbb{F}_p$ (the $g^{(\cdot)}$ node) remains HIGH sensitivity, as it is an invertible function of the secret-dependent state $\bar{\mathbf{v}}[i]$.

Secret-Times-Secret Multiplication and the Ephemeral Syndrome In the next step, the vector $\mathbf{v}[i]$ is then combined with the ephemeral masking vector $\mathbf{u}'[i]$ to form $\mathbf{u}[i] = \mathbf{v}[i] \star \mathbf{u}'[i]$, shown as the $\star$ node in Figures 1 and 2. According to previous discussion, the long-term secret dependent vector $\mathbf{v}[i]$ is high sensitivity. Additionally, this is the only secret-times-secret multiplication in CROSS signing.

The ephemeral syndrome $\mathbf{s}'[i] = \mathbf{u}[i] \mathbf{H}^\top$ is derived from the high-sensitivity intermediate $\mathbf{u}[i]$ and the public matrix $\mathbf{H}$, therefore we classified it as HIGH sensitivity. Notably, $\mathbf{s}'[i]$ is the primary leakage target exploited in prior side-channel analyses of CROSS [18,12].

Commitment Computation The CROSS signature scheme generates two commitments: cmt_0 and cmt_1 . In round i , $\text{cmt}_0[i]$ hashes the syndrome $\mathbf{s}'[i]$, Salt, and the secret-dependent vector $\bar{\mathbf{v}}[i]$ (Figure 1); for R-SDP(G), it evaluates $\text{cmt}_0[i] = \text{Hash}(\mathbf{s}'[i] | \bar{\mathbf{v}}_G[i] | \text{Salt} | i+c)$ using $\bar{\mathbf{v}}_G[i]$ instead (Figure 2). This hash evaluation is classified as HIGH sensitivity because its pre-image combines two HIGH sensitivity inputs ($\mathbf{s}'[i]$ and $\bar{\mathbf{v}}[i]$ or $\bar{\mathbf{v}}_G[i]$) prior to compression.

The commitment $\text{cmt}_1[i] = \text{Hash}(\text{Seed}[i] \parallel \text{Salt}[i+c])$ is generated directly by $\text{Seed}[i]$ (see Figures 1 and 2). As it is computed directly from the ephemeral round seed, this step has MEDIUM sensitivity. Once the commitments are computed, the digests $\text{cmt}_0[i]$ and $\text{cmt}_1[i]$ are safe to publish due to hash one-wayness. Therefore, side-channel leakage is a risk only during pre-image assembly, not from the output digests.

The challenges $(\text{chall}_1, \text{chall}_2)$, commitment digests $(\text{digest}_{\text{cmt}_0}, \dots, \text{digest}_{\text{chall}_2})$, and Proof elements are strictly public hash/commitment outputs, while Path is assigned Medium sensitivity due to its dependence on the master Seed. For the i -th round with $\text{chall}_2[i] = 0$, the opened responses $(\mathbf{y}[i], \bar{\mathbf{v}}[i])$ (or $(\mathbf{y}[i], \bar{\mathbf{v}}_G[i])$ in $\text{R-SDP}(\mathbf{G})$) are mathematically zero-knowledge and safely published without exposing $\bar{e}$. Although $\bar{\mathbf{v}}[i]$ remains vulnerable to physical side-channel leakage during intermediate hardware execution (countermeasures are detailed in Sections 4.3 and 4.4). This pathway depends on the ephemeral exponent $\bar{e}'[i]$, which generates the Medium-sensitivity vector $\mathbf{e}'[i]$ through $\mathbf{e}'[i]_j = g^{\bar{e}'[i]_j}$. During the evaluation of $\mathbf{y}[i] = \text{chall}_1[i] \cdot \mathbf{e}'[i] + \mathbf{u}'[i]$, side-channel leakage on the product $\text{chall}_1[i] \cdot \mathbf{e}'[i]$ directly exposes $\bar{e}'[i]$, affecting the security of $\bar{e}$ through the linear relation $\bar{\mathbf{v}}[i] = \bar{e} - \bar{e}'[i] \bmod z$.

4 Masking Gadgets for CROSS Signing

In this section, we present a full masking schedule for CROSS signing (Algorithm 18). For each gadget, we present its t_{mask} -secure algorithm. We define security notions for their constituent sub-gadgets, and establish overall security via compositional arguments. Table 1 summarizes all gadgets and their security targets.

Table 1: Overview of the gadgets used in Masked CROSS signing algorithm

Step	Gadget	Notion	Theorem
G0	Masked key expansion	t_{mask} -probing	Theorem 6
G1	Masked per-round ephemeral sampling	t_{mask} -probing	Theorem 11
G2	Masked transitive-map subtraction	SNI	Theorem 12
G3	Masked restriction lift / lookup	SNI	Theorem 14
G4	Masked secret–secret multiplication	SNI	Theorem 15)
G5	Masked syndrome computation	SNI	Theorem 16
G6	Commitment computation	Nlo	Theorem 18
G7	Masked first response	SNI	Theorem 19
G8	Final output handling	ZK-safe	Theorem 20
MaskedCROSS_Sign (composition of G0 through G8)			t_{mask} -probing

Methodology. We establish probing security for each gadget by matching its sub-gadgets to proven t_{mask} -(S)NI/Nlo constructions (affine maps, one-time pad splits, ISW multiplications, table recomputations) and applying standard composition rules (affine maps are SNI, SNI after NI/Nlo restores probing security, and serial SNI composition preserves SNI), as in Theorems 6 and 11.

Algorithm 1 $\text{SampleSharedSeed}(\llbracket \text{Seed}_{sk} \rrbracket)$ — G0.1 ($t_{\text{mask}}\text{-SNI}$)

Require: Boolean shares $(\text{ms}_0, \dots, \text{ms}_{d-1}) \in (\{0,1\}^{2\lambda})^d$ such that $\text{ms}_0 \oplus \dots \oplus \text{ms}_{d-1} = \text{Seed}_{sk}$ for a uniformly random Seed_{sk} and share count $d = t_{\text{mask}} + 1$.
Ensure: Boolean shares $(s_0, \dots, s_{d-1}) \in (\{0,1\}^{2\lambda})^d$ such that $s_0 \oplus \dots \oplus s_{d-1} = \text{Seed}_{sk}$.
1: **for** $k=0, \dots, d-2$ **do**
2: $s_k \xleftarrow{\$} \{0,1\}^{2\lambda}$
3: $s_k \leftarrow s_k \oplus \text{ms}_k$ ▷ each share independently uniform; no plaintext seed constructed
4: $s_{d-1} \leftarrow s_{d-1} \oplus \text{ms}_k$
5: **return** $(s_0, \dots, s_{d-1})$

This shortcut is sound because the underlying sub-gadgets already have simulator-based proofs. While linear maps, OTP splits, and ISW products match trivially, G3’s per-share table lift ($w \leftarrow g^{\bar{v}_{j,k}} \bmod p$) pushes uniform shares through a nonlinear map. Uniformity is preserved only because g generates the order- z subgroup E , making the lift a bijection $\mathbb{Z}/z\mathbb{Z} \rightarrow E$. Thus, Theorem 14 uses a dedicated simulator-based proof via a bijection lemma before applying composition rules to the subsequent ISW accumulation.

4.1 G0: Masked Key Expansion

G0 is the first gadget to handle the long-term secret Seed_{sk} , which is vulnerable to side-channel leakage during initial key expansion (Section 3). To protect the master key and derived secrets ($\bar{e}$ and $\bar{e}_G$), G0 immediately turns shares of Seed_{sk} into arithmetic shares $\llbracket \bar{e} \rrbracket$ (and $\llbracket \bar{e}_G \rrbracket$) over $\mathbb{F}_z$ together with public matrices $\mathbf{H}$ (and $\bar{\mathbf{M}}$). Throughout this process, G0 follows one essential rule: no circuit wire ever carries Seed_{sk} , $\bar{e}$, or $\bar{e}_G$ as a whole, unmasked plaintext.

Sub-Gadgets of G0

G0.1 — SampleSharedSeed. To ensure Seed_{sk} never appears as a plaintext wire value, we generate its d Boolean shares $(s_0, \dots, s_{d-1})$ independently and uniformly at random such that it holds $\text{Seed}_{sk} = \bigoplus_{i=0}^{d-1} s_i$. This allows us to create a valid Boolean sharing without ever computing or holding Seed_{sk} in plaintext.

Theorem 1 (G0.1 $t_{\text{mask}}\text{-SNI}$). *SampleSharedSeed is $t_{\text{mask}}\text{-SNI}$.*

Proof. There are no internal wires beyond the d output wires themselves, so the only probes available to the adversary are output probes on the d shares. Since every share s_k is independently and uniformly sampled and added with a share of Seed_{sk} , any subset of at most $t_{\text{mask}} \leq d-1$ output shares is jointly uniformly distributed and statistically independent of $\text{Seed}_{sk} = s_0 \oplus \dots \oplus s_{d-1}$. $\square$

The cost of SampleSharedSeed follows directly from Algorithm 1:

$$T_{\text{G0.1}}(d) = 3d = \mathcal{O}(d), \quad R_{\text{G0.1}}(d) = d \cdot 2\lambda = \mathcal{O}(d).$$

G0.2: MaskedSHAKE. The Boolean shares $(s_0, \dots, s_{d-1})$ are absorbed into a masked Keccak- $f[1600]$ permutation. We use this gadget directly from [4], treated as $t_{\text{mask}}\text{-NI}$ following Bos et al. [7], and closed by the $t_{\text{mask}}\text{-SNI}$ refresh gadget G0.5 (Theorem 22).

Theorem 2 (G0.2 $t_{\text{mask}}\text{-NI}$). *MaskedSHAKE is $t_{\text{mask}}\text{-NI}$ $[4, 7]$.*

The complexity of MaskedSHAKE is the following.

$$T_{G0.2}(d) = \Theta(d^2) = \mathcal{O}(d^2), \quad R_{G0.2}(d) = \Theta(d^2) = \mathcal{O}(d^2).$$

G0.3: SecSampleFz. G0.2 gives us a Boolean-shared byte stream, and we must turn this into n (or m for $R\text{-SDP}(G)$) arithmetic shares in $\mathbb{F}z$. We do this with rejection sampling, taking one candidate byte b at a time and checking if $b < z$. This check must stay on Boolean shares the whole time and must never bring the shares of b together on one wire. If it did, one probe would reveal the full value of b and break probing security. To check $b < z$ safely we build a masked comparator from the **SecAdd** and **SecAnd** gadgets of [5]. We compute $b - z$ on Boolean shares using **SecAdd**, with the bits of the public value z used as plain constants. Every AND gate inside **SecAdd** is an instance of **SecAnd**, their masked AND gadget. We then read the sign bit of the result, which tells us if $b < z$, without ever putting the plaintext value of b on a wire. This gadget is $t_{\text{mask}}\text{-NI}$. The sign bit becomes the public flag $flag \in \{0, 1\}$. Since z is public, revealing $flag$ does not leak more than the protocol already exposes. Once the $flag$ is revealed, the full gadget is $t_{\text{mask}}\text{-NIO}$ in the sense of [5]: the public output $flag$ is given to the simulator for free, and any $t_1 \leq t_{\text{mask}}$ internal probes can be simulated from at most t_1 input shares plus $flag$.

Algorithm 2 $\text{SecSampleFz}([stream]_B, n)$ — G0.3 ($t_{\text{mask}}\text{-NIO}$)

Require: Boolean-shared byte stream $[stream]_B$; target count n ; public modulus z .

Ensure: Arithmetic shares $[[\bar{x}_1]], \dots, [[\bar{x}_n]]$ with $\bar{x}_j \in \mathbb{F}z$ for each j .

```

1: collected  $\leftarrow 0$ 
2: for  $\ell = 0, 1, \dots$  until collected  $= n$  do
3:    $(b_0[\ell], \dots, b_{d-1}[\ell]) \leftarrow [stream[\ell]]_B$   $\triangleright$  take one Boolean-shared candidate byte
4:    $(\delta_0, \dots, \delta_{d-1}) \leftarrow \text{SecAdd}((b_0[\ell], \dots, b_{d-1}[\ell]), -z)$   $\triangleright$  masked subtract,  $z$  public; SecAdd [5]
5:    $flag \leftarrow (\delta_0[\text{msb}] \oplus \dots \oplus \delta_{d-1}[\text{msb}])$   $\triangleright$  reveal sign bit only; public NIO output
6:   if  $flag = 1$  then
7:      $[[\bar{x}_{\text{collected}}]] \leftarrow \text{B2A}((b_0[\ell], \dots, b_{d-1}[\ell]), z)$   $\triangleright$  calls G0.4; no recombination
8:     collected  $\leftarrow$  collected  $+ 1$ 
9: return  $[[\bar{x}_1]], \dots, [[\bar{x}_n]]$ 
```

Theorem 3 (G0.3 $t_{\text{mask}}\text{-NIO}$). *SecSampleFz is $t_{\text{mask}}\text{-NIO}$.*

Proof. The only step touching the Boolean shares of b before the reveal is the call to **SecAdd**. Every linear step inside **SecAdd** is share-wise for free. Every AND gate inside **SecAdd** is an instance of **SecAnd**, which is $t_{\text{mask}}\text{-SNI}$ [5]. Composing these gives **SecAdd** itself $t_{\text{mask}}\text{-NI}$ on its own, so any $t_1 \leq t_{\text{mask}}$ probes on its internal wires can

be simulated from t_1 shares of b . Once the $flag$ is revealed, the composite gadget meets the t_{mask} -NIO definition of the paper [5]: the same t_1 probes can be simulated from t_1 shares of b plus the public value $flag$. The conditional call to B2A (G0.4) only runs when $flag=1$, which is a public event, so given $flag$ the B2A step is t_{mask} -SNI by [10] and adds no new information about b beyond what $flag$ already tells us. $\square$

The cost of SecSampleFz follows directly from Algorithm 2:

$$\begin{aligned} T_{G0.3}(d, n) &= \Theta(n) (d + T_{\text{SecAdd}}(d) + (d-1) + T_{\text{B2A}}(d) + 1) = \mathcal{O}(n\mu_z d^2). \\ R_{G0.3}(d, n) &= \Theta(n) (R_{\text{SecAdd}}(d) + R_{\text{B2A}}(d)) = \mathcal{O}(n\mu_z d^2). \end{aligned}$$

G0.4: B2A (Boolean-to-arithmetic conversion). After the masked comparator accepts a candidate byte, its Boolean shares must be converted to arithmetic shares over $\mathbb{F}_q$, for $q \in \{z, p\}$, before the value can take part in the rest of CROSS signing. We convert one bit at a time, following Schneider, Paglialonga, Oder, and Güneysu [17]. Each Boolean-shared bit is first turned into arithmetic shares of that same bit; since a bit only has two possible values, this step needs no table and no cross-share multiplication beyond a small, fixed pattern. The $\mu = \lceil \log_2 q \rceil$ per-bit results are then combined with Horner's method, most significant bit first: double the running total and add in the next bit's arithmetic shares, modulo q . For CROSS, μ is 3, 7, or 9, depending on the parameter set. The cost of the conversion depends only on μ , the bit length of q , rather than on q itself.

Algorithm 3 B2A($(e_0, \dots, e_{d-1}), q, \mu$) — G0.4 (t_{mask} -SNI)

Require: Boolean shares of a μ -bit value $x = \bigoplus_k e_k$ with $x < q$. Public prime $q \in \{z, p\}$.
Bit width $\mu = \lceil \log_2 q \rceil$.

Ensure: Arithmetic shares $a_0, \dots, a_{d-1} \in \mathbb{F}_q$ with $\sum_k a_k \equiv x \pmod{q}$.

- 1: $(a_0, \dots, a_{d-1}) \leftarrow (0, \dots, 0)$
 - 2: **for** $j = \mu - 1, \dots, 0$ **do** $\triangleright$ most significant bit first
 - 3: get the Boolean bit-shares $b_k \leftarrow \text{bit}_j(e_k)$ for all k
 - 4: $(\bar{b}_0, \dots, \bar{b}_{d-1}) \leftarrow \text{SecB2Aq-Bit}((b_0, \dots, b_{d-1}), q)$ $\triangleright$ [17], Alg. 5
 - 5: **for** $k = 0, \dots, d-1$ **do**
 - 6: $a_k \leftarrow (2a_k + \bar{b}_k) \bmod q$
 - 7: **return** $(a_0, \dots, a_{d-1})$
-

Theorem 4 (G0.4 t_{mask} -SNI). *B2A is t_{mask} -SNI at $d = t_{\text{mask}} + 1$ shares.*

Proof. Each call to SecB2Aq-Bit is t_{mask} -SNI on its own [17]. Each Horner step, $a \mapsto 2a + \bar{b}$, is a linear operation over $\mathbb{F}_q$, so it is t_{mask} -SNI for free [4], matching how [17] treats these same steps in proving their full multi-bit conversion SNI. Chaining SNI gadgets one after another keeps the result SNI [4]. $\square$

The cost of B2A follows directly from Algorithm 3:

$$\begin{aligned} T_{G0.4}(d, \mu) &= \mu \cdot T_{\text{1bitB2A}}(d) + \mathcal{O}(\mu d) = \mathcal{O}(\mu d^2) \text{ ([17])}. \\ R_{G0.4}(d, \mu) &= \mu \cdot R_{\text{1bitB2A}}(d) = \mathcal{O}(\mu d^2). \end{aligned}$$

G0.5: Refresh. At the end of G0's key expansion, and at several points in G1, we need to re-randomize a set of arithmetic shares without changing the value they sum to. We draw one fresh random value for every pair of shares $i < j$, add it to share i , and subtract it from share j . For this, we follow the Refresh algorithm proposed in [4] of Appendix A. This gadget is t_{mask} -SNI and we use this gadget wherever the rest of the paper needs a standalone t_{mask} -SNI refresh.

The cost of Refresh follows directly from Algorithm 17:

$$T_{G0.5}(d) = \frac{d(d-1)}{2} \cdot \mathcal{O}(1) = \mathcal{O}(d^2), \quad R_{G0.5}(d) = \frac{d(d-1)}{2} = \mathcal{O}(d^2).$$

G0.6: SecPubMatVecFz (R-SDP(G) only). In the R-SDP(G) variant, G0.3 and G0.4 produce shares of the compact secret $[[\bar{e}_G]] \in \mathbb{F}z^m$. The full restricted error vector is then derived as $[[\bar{e}]] = [[\bar{e}_G]]\bar{\mathbf{M}}$, where $\bar{\mathbf{M}} \in \mathbb{F}z^{m \times n}$ is the public subgroup matrix. Since $\bar{\mathbf{M}}$ is public, this multiplication decomposes share-wise: $\bar{e}_k = \bar{e}_{G,k}\bar{\mathbf{M}}$ for each share index k . No cross-share interaction occurs, and no fresh randomness is needed.

Algorithm 4 SecPubMatVecFz($(\bar{e}_{G,0}, \dots, \bar{e}_{G,d-1}), \bar{\mathbf{M}}$) — G0.6 (t_{mask} -SNI [4])

Require: Arithmetic shares $\bar{e}_{G,0}, \dots, \bar{e}_{G,d-1} \in \mathbb{F}z^m$ with $\sum_k \bar{e}_{G,k} \equiv \bar{e}_G \pmod{z}$; public matrix $\bar{\mathbf{M}} \in \mathbb{F}z^{m \times n}$.

Ensure: Arithmetic shares $\bar{e}_0, \dots, \bar{e}_{d-1} \in \mathbb{F}z^n$ with $\sum_k \bar{e}_k \equiv \bar{e}_G \bar{\mathbf{M}} \pmod{z}$.

1: **for** $k=0, \dots, d-1$ **do**

2: $\bar{e}_k \leftarrow \bar{e}_{G,k}\bar{\mathbf{M}} \pmod{z}$ $\triangleright$ public matrix; touches only share k

3: **return** $(\bar{e}_0, \dots, \bar{e}_{d-1})$

Theorem 5 (G0.6 t_{mask} -SNI). *SecPubMatVecFz is t_{mask} -SNI.*

Proof. Each output share $\bar{e}_k = \bar{e}_{G,k}\bar{\mathbf{M}}$ is a linear function of one input share $\bar{e}_{G,k}$ and the public matrix $\bar{\mathbf{M}}$. There are no secret-dependent branches, no memory accesses dependent on the secret indices, and no cross-share multiplications. Any t_i internal probes and t_o output probes with $t_i + t_o \leq t_{\text{mask}}$ can therefore be simulated from t_i shares of $[[\bar{e}_G]]$, satisfying the SNI definition of the paper [4]. $\square$

The cost of SecPubMatVecFz follows directly from Algorithm 4:

$$T_{G0.6}(d, m, n) = d \cdot \Theta(mn) = \mathcal{O}(dmn), \quad R_{G0.6}(d, m, n) = 0.$$

Security Analysis of G0

Theorem 6 (G0 t_{mask} -Probing Security). *Let $d = t_{\text{mask}} + 1$. The gadget MaskedExpandPrivateSeed is t_{mask} -probing secure. The joint distribution of any set of at most t_{mask} probes inside G0 is statistically independent of Seed_{sk} , $\bar{e}$, and $\bar{e}_G$. No intermediate wire inside G0 carries any of these values in plaintext form.*

Algorithm 5 MaskedExpandPrivateSeed($[[\text{Seed}_{sk}]]$) — G0 master flow

Require: $[[\text{Seed}_{sk}]]$; CROSS parameters n, m (R-SDP(G) only), z, p, λ, t .

Ensure: $([[\bar{e}]], [[\bar{e}_G]])$ (R-SDP(G)), $\mathbf{H}, \bar{\mathbf{M}}$ (R-SDP(G)).

Sharing the master seed

1: $(s_0, \dots, s_{d-1}) \leftarrow \text{SampleSharedSeed}([[\text{Seed}_{sk}]])$ ▷ G0.1

Expanding Seed_{sk} into $\text{Seed}_{\bar{e}}$ and Seed_{pk}

2: $\text{state}_1 \leftarrow \text{MaskedSHAKE.init}()$

3: $\text{MaskedSHAKE.absorb}(\text{state}_1, (s_0, \dots, s_{d-1}) \parallel 3t+1)$ ▷ G0.2

4: $[[\text{stream}_1]]_B \leftarrow \text{MaskedSHAKE.squeeze}(\text{state}_1, 4\lambda \text{ bits})$ ▷ G0.2

5: $[[\text{Seed}_{\bar{e}}]]_B, \text{Seed}_{pk} \leftarrow \text{split } [[\text{stream}_1]]_B \text{ at } 2\lambda \text{ bits}$

Public matrices from Seed_{pk} (unmasked)

6: $\mathbf{H}, \bar{\mathbf{M}} \leftarrow \text{ExpandPublic}(\text{Seed}_{pk} \parallel 3t+2)$

Expanding $\text{Seed}_{\bar{e}}$ into a fresh Boolean-shared stream

7: $\text{state}_2 \leftarrow \text{MaskedSHAKE.init}()$

8: $\text{MaskedSHAKE.absorb}(\text{state}_2, [[\text{Seed}_{\bar{e}}]]_B \parallel 3t+3)$ ▷ G0.2

9: $[[\text{stream}_2]]_B \leftarrow \text{MaskedSHAKE.squeeze}(\text{state}_2, \cdot)$ ▷ G0.2

Rejection sampling and Boolean-to-arithmetic conversion

10: **if** R-SDP **then**

11: $[[\bar{e}]] \leftarrow \text{SecSampleFz}([[\text{stream}_2]]_B, n)$ ▷ G0.3 (calls G0.4 internally)

12: **else** ▷ R-SDP(G)

13: $[[\bar{e}_G]] \leftarrow \text{SecSampleFz}([[\text{stream}_2]]_B, m)$ ▷ G0.3 (calls G0.4 internally)

14: $[[\bar{e}]] \leftarrow \text{SecPubMatVecFz}([[\bar{e}_G]], \bar{\mathbf{M}})$ ▷ G0.6

Closing the composition

15: $[[\bar{e}]] \leftarrow \text{Refresh}([[\bar{e}]], z)$ ▷ G0.5

16: **return** $([[\bar{e}]], [[\bar{e}_G]], \mathbf{H}, \bar{\mathbf{M}})$

Proof. We apply the composition theorem of [4] to the gadget chain.

$$\underbrace{\text{G0.1}}_{\text{SNI}} \rightarrow \underbrace{\text{G0.2}}_{\text{NI}} \rightarrow \underbrace{\text{G0.3}}_{\text{NIO}} \rightarrow \underbrace{\text{G0.4}}_{\text{SNI}} \xrightarrow{\text{R-SDP(G)}} \underbrace{\text{G0.6}}_{\text{SNI}} \rightarrow \underbrace{\text{G0.5}}_{\text{SNI}}.$$

Step 1 (G0.1 to G0.2). G0.1 is t_{mask} -SNI. Its output shares are independently uniform. Zero input shares are enough to simulate any set of at most t_{mask} probes. G0.2 is t_{mask} -NI. We adopt this from Bos et al. [7], who treat the same masked Keccak- $f[1600]$ construction as NI in their own composition. Feeding an SNI output into an NI gadget yields an NI composite [4].

Step 2 (G0.2 to G0.3). G0.3 is t_{mask} -NIO. Its only public output is the flag *flag*. This flag is a fixed function of the public threshold z and the shared candidate byte. By the NIO composition rule of [4], a public output like this does not lower the probing security of the surrounding composition.

Step 3 (G0.3 to G0.4). G0.4 is t_{mask} -SNI. It takes the conditional output of G0.3 as input. An NIO gadget behaves like an NI gadget on its shared output. Its public flag costs nothing extra to simulate. Feeding this NIO output into an SNI gadget still yields an NI composite.

Step 4 (G0.4 to G0.6, R-SDP(G) only). G0.6 is t_{mask} -SNI for free. It is an affine gadget, so [4] applies directly. Feeding the NI output of G0.4 through this SNI linear map still yields an NI composite.

Step 5 (G0.4 or G0.6 to G0.5). G0.5 is t_{mask} -SNI. Placing an SNI sink at the end of an otherwise NI chain re-establishes full t_{mask} -probing security for the whole composition [4]. Take any t_i internal probes and t_o output probes on the full G0 circuit, with $t_i + t_o \leq t_{\text{mask}}$. These can be simulated from t_i shares of G0's implicit input. That input is empty: $(\text{Seed}_{sk}, \bar{e}, \bar{e}_G) = \emptyset$. G0 has no circuit inputs beyond the fresh randomness sampled inside G0.1. So simulation from the empty set is enough. This is trivially possible.

The invariant that no wire inside G0 ever carries a plaintext sensitive value follows from each sub-gadget in turn. G0.1 never constructs Seed_{sk} . G0.2 processes only Boolean shares. G0.3 checks $b < z$ using **SecAdd** and **SecAnd**, and never recombines the candidate byte b . G0.4 converts one bit at a time and never reconstructs the plaintext byte. G0.5 and G0.6 are both purely share-wise. $\square$

The cost of **MaskedExpandPrivateSeed** follows directly from Algorithm 5, combining the costs of its constituent sub-gadgets:

$$\begin{aligned} T_{G0} &= T_{G0.1} + \mathcal{O}(1) \cdot T_{G0.2} + T_{G0.3}(d, m) + T_{G0.6} + T_{G0.5} = \mathcal{O}(m\mu_z d^2 + dmn + d\lambda). \\ R_{G0} &= R_{G0.1} + \mathcal{O}(1) \cdot R_{G0.2} + R_{G0.3}(d, m) + R_{G0.5} = \mathcal{O}(m\mu_z d^2 + d\lambda). \end{aligned}$$

4.2 G1: Masked Per-Round Ephemeral Sampling

G1 produces the per-round ephemeral material $[[\bar{e}'[i]]] \in \mathbb{F}z^n$, $[[\mathbf{u}'[i]]] \in \mathbb{F}p^n$, and, for R-SDP(G), $[[\bar{e}'_G[i]]] \in \mathbb{F}z^m$. A subtlety distinguishes G1 from G0. The seed-tree leaf $\text{Seed}[i]$ used by G1 is only *conditionally* public. It will be revealed for rounds where $\text{chall}_2[i] = 1$. But chall_2 is determined by the Fiat-Shamir hash only *after* all t rounds of G1 through G6 have already executed. When G1 executes, no round has been designated open or closed. Every leaf must therefore be treated as sensitive. This is an instance of the general vulnerability identified by Seker et al. [19]. Their attack targets the unopened party in the Picnic signature scheme and allows full key recovery from side-channel probes. Their countermeasure introduces a new $(N-1)$ -private MPCitH paradigm, named SNI-in-the-head. Instead of revealing $N-1$ MPC shares as in the unprotected case, the signing algorithm reveals only $N-d-1$ shares. This yields a d -probing secure signature scheme by construction, through reduced disclosure rather than through masking every gadget. We adopt a related discipline here, but through direct masking rather than reduced disclosure. We generate each $\text{Seed}[i]$ directly in d -shared Boolean form (G1.0). No leaf value ever appears as a whole plaintext on any wire, regardless of whether the round is eventually opened.

Sub-Gadgets of G1

G1.0: MaskedSeedLeaves. The seed-tree evaluation $\text{SeedTree}(\text{Seed}|\text{Salt})$ produces t leaf values $\text{Seed}[1], \dots, \text{Seed}[t]$. No leaf and no internal node of the tree may ever exist as

Algorithm 6 MaskedSeedLeaves() — G1.0 ($t_{\text{mask}}\text{-SNI}$)

Require: Public salt $\text{Salt} \in \{0,1\}^{2\lambda}$; tree depth $\log_2 t$; share count $d = t_{\text{mask}} + 1$.

Ensure: Boolean-shared leaves $([[\text{Seed}[1]]]_B, \dots, [[\text{Seed}[t]]]_B)$, each a valid d -sharing of the corresponding leaf of the tree rooted at a uniformly random, never-instantiated Seed.

```

1:  $(\text{root}_0, \dots, \text{root}_{d-1}) \xleftarrow{\$} (\{0,1\}^{2\lambda})^d$ 
2:  $\text{level}[0] \leftarrow \{(\text{root}_0, \dots, \text{root}_{d-1})\}$ 
3: for  $\ell = 0, \dots, \log_2 t - 1$  do
4:   for each node  $[[\text{node}]]$  at position  $\text{idx}$  in  $\text{level}[\ell]$  do
5:      $([[\text{left}]], [[\text{right}]]) \leftarrow \text{MaskedTreePRG}([[\text{node}]], \text{Salt}, \ell, \text{idx})$   $\triangleright t_{\text{mask}}\text{-NI}$  [4]
6:      $[[\text{left}]] \leftarrow \text{Refresh}([[\text{left}]]); [[\text{right}]] \leftarrow \text{Refresh}([[\text{right}]])$   $\triangleright \text{G0.5}$ 
7:     add  $[[\text{left}]], [[\text{right}]]$  to  $\text{level}[\ell+1]$ 
8:  $([[\text{Seed}[1]]]_B, \dots, [[\text{Seed}[t]]]_B) \leftarrow \text{level}[\log_2 t]$ 
9: return  $([[\text{Seed}[1]]]_B, \dots, [[\text{Seed}[t]]]_B)$ 

```

a whole plaintext value on any wire. Note that a leaf is not a fresh value the signer is free to define implicitly as the XOR of independently-sampled shares. It is a *deterministic* output of the tree, and that determinism is load-bearing. The verifier reconstructs disclosed leaves for $\text{chall}_2[z] = 1$ rounds from a small number of revealed internal nodes, using ordinary, unmasked evaluations of the same public tree PRG. Feneuil, Rivain, and Warmé-Janville [13] analyze exactly this leakage surface for GGM-tree-based commitments. Recovering any parent seed, including the root lets an attacker recompute every child seed beneath it, and hence the full secret sharings derived from that subtree. Masking such a tree means masking the PRG evaluation itself at every node, as we do here. This is their standard, format-preserving construction, at $O(d^2)$ cost per node under standard masking gadgets, since the non-linear step of a masked Keccak permutation requires an $O(d^2)$ multiplication gadget. Their paper’s own contribution is a set of tweaks to bring this down. For the GGM variant specifically, their tweak replaces the single d -masked tree with $d+1$ separate, unmasked seed trees, one per masking share of the root seed, at the cost of including $d+1$ sibling paths in the signature instead of one. This brings the GGM-variant commitment cost to $O(d)$, at a linear cost in signature size. This tweak changes the signature format. We adopt the standard, format-preserving construction here and leave the cheaper, format-changing tweak as a design option outside the scope of a masking-gadget-level rewrite. Concretely, we reuse the same composition strategy as G0.2: the permutation at each tree node is the higher-order masked Keccak construction of [4], treated as $t_{\text{mask}}\text{-NI}$ following Bos et al. [7]. We call this per-node application **MaskedTreePRG**. That permutation is only $t_{\text{mask}}\text{-NI}$, hence we insert the pairwise-random **Refresh** gadget (G0.5, Theorem 22) after every level before the next level’s masked-PRG call consumes the shares.

Theorem 7 (G1.0 $t_{\text{mask}}\text{-SNI}$, conditional). *As before, the masked Keccak- $f[1600]$ permutation used by Bos et al. [7] is $t_{\text{mask}}\text{-NI}$. Hence the gadget **MaskedSeedLeaves** is $t_{\text{mask}}\text{-SNI}$.*

Proof. The root shares are produced randomly. At each level, **MaskedTreePRG** is $t_{\text{mask}}\text{-NI}$ by the identical argument as Theorem 2. The **Refresh** call immediately

following is t_{mask} -SNI (Theorem 22), placed as a sink after each level's NI gadget [4]. This re-establishes SNI for the composite of that level. By induction on tree depth, every level's output and hence every leaf is delivered as a valid t_{mask} -SNI sharing. $\square$

The cost of **MaskedSeedLeaves** follows directly from Algorithm 6:

$$\mathsf{T}_{\mathsf{G1.0}}(d, T) = \Theta(T)(\mathsf{T}_{\mathsf{G0.2}} + \mathsf{T}_{\mathsf{G0.5}}) = \mathcal{O}(Td^2), \quad \mathsf{R}_{\mathsf{G1.0}}(d, T) = \Theta(T)(\mathsf{R}_{\mathsf{G0.2}} + \mathsf{R}_{\mathsf{G0.5}}) = \mathcal{O}(Td^2).$$

G1.1 — MaskedSHAKE (round-seed expansion). Each Boolean-shared leaf $[[\text{Seed}[i]]]_B$ from G1.0 is absorbed share-wise into a masked Keccak- $f[1600]$ permutation [7] to produce a Boolean-shared byte stream. The linear absorb step XORs each leaf share $\text{Seed}[i]_k$ into the corresponding Keccak state share: $\text{state}_k \leftarrow \text{state}_k \oplus \text{Seed}[i]_k \oplus \text{Salt} \oplus \langle i+c \rangle$, where Salt and $i+c$ are public and absorbed without masking. This absorb is t_{mask} -SNI for free [4]. The subsequent masked permutation call is t_{mask} -NI by the same assumption used in G0.2 and G1.0 [7]; composing the SNI absorb with the NI permutation yields an NI composite [4].

Theorem 8 (G1.1 t_{mask} -NI, conditional). *Under the assumption that the masked Keccak- $f[1600]$ implementation is t_{mask} -NI [7], and given that $[[\text{Seed}[i]]]_B$ is a valid Boolean d -sharing from G1.0 (Theorem 7), MaskedSHAKE is t_{mask} -NI.*

Proof. Identical in structure to the proof of Theorem 2, with $[[\text{Seed}[i]]]_B$ replacing $[[\text{Seed}_{sk}]]_B$. $\square$

The cost of MaskedSHAKE (round-seed expansion) follows directly from the same accounting as G0.2:

$$\mathsf{T}_{\mathsf{G1.1}}(d) = \mathsf{T}_{\mathsf{G0.2}}(d) = \mathcal{O}(d^2), \quad \mathsf{R}_{\mathsf{G1.1}}(d) = \mathsf{R}_{\mathsf{G0.2}}(d) = \mathcal{O}(d^2).$$

G1.2 — SecSampleFz (masked rejection sampling over $\mathbb{F}z$). The Boolean-shared byte stream from G1.1 is converted into n (or m for R-SDP(G)) arithmetic shares of elements of $\mathbb{F}z$ by masked rejection sampling. The threshold check $b < z$ is evaluated on Boolean shares using a comparator built from the **SecAdd** and **SecAnd** gadgets of [5]: we compute $b - z$ with **SecAdd**, using the bits of the public z as plain constants, and every AND gate inside **SecAdd** is a **SecAnd** call; the sign bit of the result tells us if $b < z$. The only public output of the comparator is the accept/reject flag $\text{flag} \in \{0, 1\}$; since z is a public parameter, leaking flag reveals nothing about b beyond what the protocol already exposes. This gadget is shared verbatim with G0.3; Algorithm 2 and Theorem 3 cover both instantiations, with $[[e'[i]]]$ or $[[e'_G[i]]]$ in place of $[[\bar{x}]]$. The t_{mask} -NIo claim is per [5].

$$\mathsf{T}_{\mathsf{G1.2}}(d, L) = \mathsf{T}_{\mathsf{G0.3}}(d, L) = \mathcal{O}(L\mu_z d^2) \quad (\text{Algorithm 2, } L \in \{n, m\}).$$

$$\mathsf{R}_{\mathsf{G1.2}}(d, L) = \mathsf{R}_{\mathsf{G0.3}}(d, L) = \mathcal{O}(L\mu_z d^2).$$

G1.3 — SecSampleFq (sampling over $\mathbb{F}p$). G1.3 produces $[[\mathbf{u}'[i]]] \in \mathbb{F}p^n$ from the same masked SHAKE instance already running in G1.1. This mirrors G1.2 exactly, with p in place of z : the candidate stays Boolean-shared throughout, and only a single accept/reject bit is revealed. We build the comparator from the same SecAdd and SecAnd gadgets of [5], computing $rnd - p$ on Boolean shares and reading the sign bit as $flag$. Once accepted, the Boolean-shared candidate is converted to arithmetic shares in $\mathbb{F}p$ using the same B2A gadget as G0.4, targeting $\mathbb{F}p$ instead of $\mathbb{F}z$. We then apply the pairwise-random Refresh gadget (G0.5, Theorem 22) to each accepted output share, closing the chain with a genuine SNI sink.

Algorithm 7 SecSampleFq($[[stream]]_B, n$) — G1.3 ($t_{\text{mask-SNI}}$)

Require: Boolean-shared SHAKE stream $[[stream]]_B$ from G1.1; output length n ; public prime p .

Ensure: Arithmetic shares $[[\mathbf{u}'[i]]]$ with each $u'[i]_j$ uniform in $\mathbb{F}p$.

```

1:  $j \leftarrow 0$ 
2: while  $j < n$  do
3:    $(r_0, \dots, r_{d-1}) \leftarrow [[stream]]_B$ , next  $\lceil \log_2 p \rceil$  bits ▷ Boolean-shared candidate
4:    $(\delta_0, \dots, \delta_{d-1}) \leftarrow \text{SecAdd}((r_0, \dots, r_{d-1}), -p)$  ▷ SecAdd [5];  $p$  public
5:    $flag \leftarrow \delta_0[\text{msb}] \oplus \dots \oplus \delta_{d-1}[\text{msb}]$  ▷ public NIO output; 1 iff  $rnd < p$ 
6:   if  $flag = 1$  then
7:      $[[\mathbf{u}'[i]_j]] \leftarrow \text{B2A}((r_0, \dots, r_{d-1}), p)$  ▷ G0.4, targeting  $\mathbb{F}p$ 
8:      $[[\mathbf{u}'[i]_j]] \leftarrow \text{Refresh}([[ \mathbf{u}'[i]_j ]])$  ▷ G0.5; closes with a genuine SNI sink
9:      $j \leftarrow j + 1$ 
10: return  $[[\mathbf{u}'[i]]]$ 

```

Theorem 9 (G1.3 $t_{\text{mask-SNI}}$). *SecSampleFq is $t_{\text{mask-SNI}}$.*

Proof. The candidate-generation and acceptance steps, up to and including B2A, form a $t_{\text{mask-NIO}}$ core: SecAdd is $t_{\text{mask-NI}}$, since every AND gate inside it is a $t_{\text{mask-SNI}}$ SecAnd call [5], and B2A is $t_{\text{mask-SNI}}$ (Theorem 4), conditioned on the public $flag$. Refresh sits as a sink immediately after this NIO core, on every accepted output share re-establishing $t_{\text{mask-SNI}}$ for the sub-gadget. $\square$

The cost of SecSampleFq follows directly from Algorithm 7:

$$\begin{aligned}
T_{G1.3}(d, n) &= \Theta(n) (T_{\text{SecAdd}}(d) + T_{G0.4}(d, \mu_p) + T_{G0.5}(d)) = \mathcal{O}(n\mu_p d^2) \text{ (SecAdd [5])}. \\
R_{G1.3}(d, n) &= \Theta(n) (R_{\text{SecAdd}}(d) + R_{G0.4}(d, \mu_p) + R_{G0.5}(d)) = \mathcal{O}(n\mu_p d^2).
\end{aligned}$$

G1.4 — SecPubMatVecFz (R-SDP(G) only). For R-SDP(G), the full ephemeral $[[\bar{e}'[i]]]$ is derived from $[[e'_G[i]]]$ by the share-wise linear map $\bar{e}'[i]_k = e'_G[i]_k \mathbf{M}$ for each share index k . Since $\mathbf{M}$ is public, no cross-share interaction occurs and no fresh randomness is needed. The argument and theorem are identical to G0.6 (Theorem 5).

Theorem 10 (G1.4 $t_{\text{mask-SNI}}$). *SecPubMatVecFz($[[\bar{e}'_G[i]]], \bar{\mathbf{M}}$) is $t_{\text{mask-SNI}}$.*

Proof. Identical to the proof of Theorem 5. $\square$

The cost of `SecPubMatVecFz` mirrors `G0.6` directly (Theorem 5, Algorithm 4):

$$T_{G1.4}(d, m, n) = T_{G0.6}(d, m, n) = \mathcal{O}(dmn), \quad R_{G1.4}(d, m, n) = 0.$$

Algorithm 8 `MaskedRoundSample`($i, \llbracket \text{Seed}[i] \rrbracket_B, \text{Salt}, c, \bar{\mathbf{M}}$) — G1

Require: Round index i ; that round's leaf $\llbracket \text{Seed}[i] \rrbracket_B$, from Algorithm 6; public Salt, constant c ; public matrix $\bar{\mathbf{M}}$ (R-SDP(G) only).

Ensure: ($\llbracket \bar{e}'[i] \rrbracket, \llbracket \bar{e}'_G[i] \rrbracket$) (R-SDP(G)), ($\llbracket \mathbf{u}'[i] \rrbracket$).

```

1:  $state \leftarrow \text{MaskedSHAKE.init}()$ 
2:  $\text{MaskedSHAKE.absorb}(state, \llbracket \text{Seed}[i] \rrbracket_B \mid \text{Salt} \mid i + c)$  ▷ G1.1
3:  $\llbracket stream \rrbracket_B \leftarrow \text{MaskedSHAKE.squeeze}(state)$ 
4: if R-SDP(G) then
5:    $\llbracket \bar{e}'_G[i] \rrbracket \leftarrow \text{SecSampleFz}(\llbracket stream \rrbracket_B, m)$  ▷ G1.2
6:    $\llbracket \bar{e}'[i] \rrbracket \leftarrow \text{SecPubMatVecFz}(\llbracket \bar{e}'_G[i] \rrbracket, \bar{\mathbf{M}})$  ▷ G1.4
7: else
8:    $\llbracket \bar{e}'[i] \rrbracket \leftarrow \text{SecSampleFz}(\llbracket stream \rrbracket_B, n)$  ▷ G1.2
9:    $\llbracket \mathbf{u}'[i] \rrbracket \leftarrow \text{SecSampleFq}(\llbracket stream \rrbracket_B, n)$  ▷ G1.3
10: return ( $\llbracket \bar{e}'[i] \rrbracket, \llbracket \bar{e}'_G[i] \rrbracket, \llbracket \mathbf{u}'[i] \rrbracket$ )
```

G1 Master Algorithm

Security Analysis of G1

Theorem 11 (G1 t_{mask} -Probing Security). *Let $d = t_{\text{mask}} + 1$. Under the assumption that the masked Keccak- $f[1600]$ permutation is $t_{\text{mask}}\text{-NI}$ [7], `MaskedRoundSample` is t_{mask} -probing secure with respect to the ephemeral outputs $\bar{e}'[i]$, $\bar{e}'_G[i]$, and $\mathbf{u}'[i]$. No intermediate wire inside G1 carries any of these values in plaintext form.*

Proof. We apply the composition theorem of [4]:

$$\underbrace{G1.0}_{\text{SNI}} \rightarrow \underbrace{G1.1}_{\text{NI}} \rightarrow \underbrace{G1.2}_{\text{NIo}} \rightarrow \underbrace{G1.3}_{\text{SNI}} \xrightarrow{\text{R-SDP(G)}} \underbrace{G1.4}_{\text{SNI}}.$$

G1.0 is $t_{\text{mask}}\text{-SNI}$ (Theorem 7); feeding its output into the $t_{\text{mask}}\text{-NI}$ gadget G1.1 yields an NI composite. G1.2 is $t_{\text{mask}}\text{-NIo}$ (Theorem 3), and by Rule 3 of [4] its public accept/reject output does not degrade the composition. G1.3 is $t_{\text{mask}}\text{-SNI}$ (Theorem 9); as an SNI sink after the NI/NIo chain, it re-establishes t_{mask} -probing security. In R-SDP(G), G1.4 is $t_{\text{mask}}\text{-SNI}$ for free (Theorem 10), and the composition remains t_{mask} -probing secure. The plaintext invariant follows from: G1.0 never recombines any tree node (Theorem 7); G1.1 processes only Boolean shares; G1.2 uses the masked comparator built from `SecAdd` and `SecAnd` of [5]; G1.3 uses the same masked comparator and converts accepted candidates directly to arithmetic shares via `B2A`, without ever recombining a candidate into plaintext; G1.4 is purely share-wise. $\square$

The cost of `MaskedRoundSample` follows directly from Algorithm 8, combining the per-round costs of its sub-gadgets:

$$\begin{aligned} T_{G1}^{\text{round}} &= T_{G1.1} + T_{G1.2}(L) + T_{G1.3}(n) + T_{G1.4} = \mathcal{O}((n+m)\mu d^2 + dmn). \\ R_{G1}^{\text{round}} &= R_{G1.1} + R_{G1.2}(L) + R_{G1.3}(n) = \mathcal{O}((n+m)\mu d^2). \end{aligned}$$

4.3 G2: Masked Transitive-Map Subtraction

Protecting the subtraction $\bar{v}[i] = \bar{e} - \bar{e}'[i] \bmod z$ is the role of G2. Both published power-analysis attacks on CROSS target exactly this step [18,12], since it is the first point where the long-term secret and the round's ephemeral material meet in direct arithmetic. The masked version of this subtraction works share by share. Each output share $c_{j,k}$ only depends on the matching input shares $a_{j,k}$ and $b_{j,k}$, and on the public modulus z . No share ever touches a different share index, and no fresh randomness is needed to compute it.

The `SecRestrSub` Gadget

Theorem 12 (`SecRestrSub` t_{mask} -SNI). *`SecRestrSub` is t_{mask} -SNI, as long as $[[a]]$ and $[[b]]$ are valid full-threshold d -sharings.*

Proof. Each output share is an affine function of the matching input shares: no two shares are multiplied together, and no share depends on another share's index. This makes the gadget t_{mask} -SNI directly [4]. Any t_i internal probes and t_o output probes with $t_i + t_o \leq t_{\text{mask}}$ touch at most $d-1$ distinct share indices, which the simulator reproduces from t_i shares of each input alone. $\square$

Algorithm 9 `SecRestrSub`($[[a]], [[b]]$) — G2 (t_{mask} -SNI)

Require: Arithmetic shares $[[a]] = (a_{j,k})_{j,k}$ and $[[b]] = (b_{j,k})_{j,k}$ of vectors over $\mathbb{F}z$, length n ; both are valid full-threshold d -sharings delivered by G0 and G1 respectively.

Ensure: $[[c]]$ with $\sum_k c_{j,k} \equiv a_j - b_j \pmod{z}$ componentwise.

- 1: **for** $j = 1, \dots, n$ **do**
 - 2: **for** $k = 0, \dots, d-1$ **do**
 - 3: $c_{j,k} \leftarrow (a_{j,k} + (z - b_{j,k})) \bmod z$ $\triangleright$ share-wise; no cross-share interaction
 - 4: **return** $[[c]]$
-

The cost of `SecRestrSub` follows directly from Algorithm 9:

$$T_{G2}(d, n) = nd = \mathcal{O}(nd), \quad R_{G2}(d, n) = 0.$$

Security Analysis of G2

Theorem 13 (G2 t_{mask} -Probing Security). *Let $d = t_{\text{mask}} + 1$. As long as G0 and G1 deliver valid full-threshold d -sharings, G2 is t_{mask} -probing secure across all t rounds. Any set of at most t_{mask} probes on the wires of every G2 call, together, tells the attacker nothing about $\bar{e}$, $\bar{e}_G$, $\bar{e}'[i]$, or $\bar{e}'_G[i]$.*

Proof. G0 hands $\llbracket \bar{e} \rrbracket$ to t separate calls of SecRestrSub, one per round, and the attacker's budget of t_{mask} probes is shared across all of them, not reset per call. Since each call is t_{mask} -SNI (Theorem 12), the composition rules of [4] guarantee no split of the probe budget across the t calls ever recovers t_{mask} or more shares of $\llbracket \bar{e} \rrbracket$.

The inputs required by Theorem 12 are valid d -sharings by construction: $\llbracket \bar{e} \rrbracket$ and $\llbracket \bar{e}_G \rrbracket$ come from G0 (Theorem 6), and $\llbracket \bar{e}'[i] \rrbracket$, $\llbracket \bar{e}'_G[i] \rrbracket$ come from G1 (Theorem 11).

For R-SDP(G), SecRestrSub is called twice per round: once on the compact values and once on their full-size images. G0.6 and G1.4 both apply $\bar{\mathbf{M}}$ share by share, so

$$\bar{\mathbf{v}}[i]_k = \bar{e}_k - \bar{e}'[i]_k = (\bar{e}_{G,k} - \bar{e}'_{G,k}[i])\bar{\mathbf{M}} = \bar{\mathbf{v}}_G[i]_k\bar{\mathbf{M}}.$$

holds exactly, share by share. Probing share k of one call therefore gives nothing beyond probing share k of the other, since the two are tied by a public linear operation, so both calls together are covered by the same SNI argument as Theorem 12 at the same share count. A non-linear or index-mixing link between G0.6 and G1.4 would break this shortcut, requiring a refresh between the two calls, or the probe-isolating framework of Cassiers and Standaert [9], instead. $\square$

4.4 G3: Masked Restriction Lift

G3 evaluates the group-element map $g^{(\cdot)} : \mathbb{F}z \rightarrow \mathbb{E} \subset \mathbb{F}p^*$ on each component of the shared transitive-map vector $\llbracket \bar{\mathbf{v}}[i] \rrbracket \in \mathbb{F}z^n$, producing the shared lifted vector $\llbracket \bar{\mathbf{v}}[i] \rrbracket \in \mathbb{F}p^n$. $\bar{\mathbf{v}}[i]_j$ must never appear as a plaintext index. g generates the order- z subgroup $\mathbb{E}$, so the exponentiation telescopes over an additive sharing of the exponent: $g^{\bar{v}_j} = \prod_{k=0}^{d-1} g^{\bar{v}_{j,k}} \bmod p$. This lets each share be lifted independently through a small public table ($z \in \{7, 127\}$ for R-SDP and R-SDP(G)) and then combined via the masked multiplication gadget of G4.

Lemma 1 (The lift is a bijection). *The map $\phi : \mathbb{Z}/z\mathbb{Z} \rightarrow \mathbb{E}$, $\phi(x) = g^x \bmod p$, is a bijection.*

Proof. g generates $\mathbb{E}$, and $|\mathbb{E}| = z$. So $g^0, \dots, g^{z-1}$ are the z distinct elements of $\mathbb{E}$, one each. That makes ϕ injective and surjective. A map between two finite sets of the same size that is injective is a bijection. $\square$

Corollary 1 (Uniformity is preserved). *If X is uniform over $\mathbb{Z}/z\mathbb{Z}$, then $\phi(X)$ is uniform over $\mathbb{E}$.*

Proof. For any $y \in \mathbb{E}$, exactly one x satisfies $\phi(x) = y$. So $\Pr[\phi(X) = y] = \Pr[X = x] = 1/z$, the same for every y . $\square$

G3.1 — TableLift. The public lookup table $T[y] = g^y \bmod p$ is precomputed once for $y = 0, \dots, z-1$. For share $\bar{v}_{j,k}$, **TableLift** looks up $w \leftarrow T[\bar{v}_{j,k}]$, then immediately splits w into a fresh d -sharing, the same construction as **SecSampleFq** in G1.3. No wire besides the transient w ever holds an unmasked function of $\bar{v}_{j,k}$, and w is erased right after splitting.

$$T_{G3.1}(d) = \mathcal{O}(d), \quad R_{G3.1}(d) = d-1 = \mathcal{O}(d).$$

G3.2 — SecMult (accumulate). The d per-share liftings $[[g^{\bar{v}_{j,0}}]], \dots, [[g^{\bar{v}_{j,d-1}}]]$ are combined into one additively-shared product, using $d-1$ sequential calls to the ISW gadget **SecMult** of G4 (Theorem 15): initialize the accumulator to the first lifting, then fold in each later lifting via $[[acc_j]] \leftarrow [[acc_j]] \star [[g^{\bar{v}_{j,k}}]]$. This is the only secret $\times$ secret multiplication per share in the schedule, and its output is $[[\mathbf{v}[i]_j]]$ directly.

$$T_{G3.2}(d) = (d-1) \cdot T_{G4}(d) = \mathcal{O}(d^3), \quad R_{G3.2}(d) = (d-1) \cdot R_{G4}(d) = \mathcal{O}(d^3).$$

Algorithm 10 **SecExpLift**($[[\bar{\mathbf{v}}[i]]]$) — G3 (t_{mask} -SNI)

Require: Arithmetic shares $[[\bar{\mathbf{v}}[i]]]$ over $\mathbb{F}z^n$, valid full-threshold d -sharings from G2; public generator g of the order- z subgroup $\mathbb{E} \subset \mathbb{F}p^*$.

Ensure: $[[\mathbf{v}[i]]]$ with $\mathbf{v}[i]_j = g^{\bar{\mathbf{v}}[i]_j}$ in shared form over $\mathbb{F}p^n$.

```

1:  $T[y] \leftarrow g^y \bmod p$  for all  $y \in \{0, \dots, z-1\}$   $\triangleright$  public table, computed once & reused across  $\forall j, i$ 
2: for  $j = 1, \dots, n$  do
3:   for  $k = 0, \dots, d-1$  do
4:      $w \leftarrow T[\bar{\mathbf{v}}[i]_{j,k}]$   $\triangleright$  G3.1: lookup on one share, safe to reveal
5:     for  $\ell = 0, \dots, d-2$  do
6:        $\ell_k[\ell] \xleftarrow{\$} \mathbb{F}p$ ;  $w \leftarrow (w - \ell_k[\ell]) \bmod p$ 
7:        $\ell_k[d-1] \leftarrow w$   $\triangleright$  fresh  $d$ -sharing  $[[g^{\bar{\mathbf{v}}[i]_{j,k}}]] \leftarrow \ell_k$ 
8:       Zero( $w$ )
9:      $[[acc_j]] \leftarrow \ell_0$ 
10:    for  $k = 1, \dots, d-1$  do
11:       $[[acc_j]] \leftarrow \text{SecMult}([acc_j], \ell_k)$   $\triangleright$  G3.2: one genuine ISW call (G4) per share
12:     $[[\mathbf{v}[i]_j]] \leftarrow [[acc_j]]$ 
13: return  $[[\mathbf{v}[i]]]$ 

```

Security Analysis of G3

Theorem 14 (G3 t_{mask} -SNI). *SecExpLift is t_{mask} -SNI, provided $[[\bar{\mathbf{v}}[i]]]$ is a valid full-threshold d -sharing from G2. No intermediate wire carries $\bar{\mathbf{v}}[i]_j$ in plaintext form.*

Proof. Fix a component j and a share k . Write $\bar{v} = \bar{\mathbf{v}}[i]_j$ and $\bar{v}_k$ for its k -th share.

Consider the wire $w = T[\bar{v}_k]$, the first wire of that G3.1 call. Build a simulator Sim_k for it. If a probe touches w , Sim_k asks for the single share $\bar{v}_k$ and computes

Algorithm 11 $\text{SecMult}(\llbracket v \rrbracket, \llbracket w \rrbracket) \leftarrow \text{G4 } (t_{\text{mask}}\text{-SNI}, [14, 4])$

Require: Additive shares $\llbracket v \rrbracket = (v_0, \dots, v_{d-1})$ and $\llbracket w \rrbracket = (w_0, \dots, w_{d-1})$ in $\mathbb{F}p$; both valid full-threshold d -sharings.

Ensure: $\llbracket u \rrbracket$ with $\sum_k u_k \equiv (\sum_k v_k)(\sum_k w_k) \pmod{p}$.

```

1: for  $k=0, \dots, d-1$  do
2:    $u_k \leftarrow v_k \cdot w_k \pmod{p}$  ▷ diagonal terms; one per share
3: for  $k=0, \dots, d-2$  do
4:   for  $\ell=k+1, \dots, d-1$  do
5:      $r_{k,\ell} \xleftarrow{\$} \mathbb{F}p$ 
6:      $u_k \leftarrow (u_k + r_{k,\ell}) \pmod{p}$ 
7:      $u_\ell \leftarrow (u_\ell + v_k w_\ell + v_\ell w_k - r_{k,\ell}) \pmod{p}$ 
8: return  $\llbracket u \rrbracket$ 

```

w the same way the real circuit does. This uses one input share. If no probe touches w , Sim_k draws a value $\hat{w}$ uniformly from $\mathbb{E}$ on its own. This uses no input share. By Corollary 1, $\hat{w}$ has the same distribution as the real w , since $\bar{v}_k$ is uniform by the full-threshold property of G2's output. Either way, one input share is enough.

The remaining wires of that call, $\ell_k[0], \dots, \ell_k[d-1]$, form a one-time-pad split of w . This is the same structure already used in Theorem 1. Any probe among them is simulated with fresh uniform draws, using no further input shares.

So each G3.1 call needs at most one share of $\llbracket \bar{v} \rrbracket$ to simulate any number of its own probes. Across the d calls, a budget of $t_i \leq t_{\text{mask}}$ probes touches at most t_i of the wires w , so it costs at most t_i shares in total.

Each G3.2 call is t_{mask} -SNI by Theorem 15. Chaining $d-1$ SNI calls, and chaining them after the G3.1 stage above, stays SNI by the same composition rule used in Theorem 6. SecMult is a sink gadget, so the last call in the chain re-establishes SNI for the whole composite, and $\llbracket \mathbf{v}[i]_j \rrbracket$ can be taken directly as $\llbracket \text{acc}_j \rrbracket$. So the full per-component chain is t_{mask} -SNI.

The plaintext invariant holds throughout. w never persists past its own split and erasure. G3.2 uses fresh ISW randomness per call (Theorem 15). No wire ever computes $\bar{v} = \sum_k \bar{v}_k$ itself. $\square$

The cost of SecExpLift follows directly from Algorithm 10:

$$\text{T}_{\text{G3}}(d, n) = n(\mathcal{O}(d^2) + (d-1) \cdot \text{T}_{\text{G4}}(d)) = \mathcal{O}(nd^3).$$

$$\text{R}_{\text{G3}}(d, n) = n(\mathcal{O}(d^2) + (d-1) \cdot \text{R}_{\text{G4}}(d)) = \mathcal{O}(nd^3).$$

4.5 G4: Masked Secret–Secret Multiplication

G4 computes the component-wise product $\mathbf{u}[i] = \mathbf{v}[i] \star \mathbf{u}'[i] \in \mathbb{F}p^n$, using the full ISW multiplication gadget. G3.2 reuses this same gadget. All other multiplications in the pipeline involve a public operand and uses cheaper share-wise scaling instead.

Theorem 15 (G4 t_{mask} -SNI at linear share count). *Let $d = t_{\text{mask}} + 1$. The multiplication structure of Ishai, Sahai, and Wagner [14], instantiated with d shares, is t_{mask} -SNI. This follows the tight-share-count SNI analysis of [4].*

Proof. Take any t_i internal probes and t_o output probes with $t_i + t_o \leq t_{\text{mask}}$. Each probed cross-term wire, either $r_{k,\ell}$ or $v_k w_\ell + v_\ell w_k - r_{k,\ell}$, involves at most two share indices k, ℓ . Each fresh random $r_{k,\ell}$ is used in exactly one cross-term, masking it independently of every other cross-term. The simulator of [4] builds a perfect joint distribution of all probed wires from at most t_i shares of each input, $[[v]]$ and $[[w]]$, regardless of t_o . This is the defining property of SNI, unlike the weaker NI notion, where simulation cost would grow with $t_i + t_o$. It is also what lets the share count drop from ISW’s original $2t_{\text{mask}} + 1$ to the linear $t_{\text{mask}} + 1$ used throughout this paper. $\square$

The cost of **SecMult** follows directly from Algorithm 11:

$$T_{G4}(d) = d + \frac{d(d-1)}{2} \cdot \mathcal{O}(1) = \mathcal{O}(d^2) \text{ (ISW [14])}, \quad R_{G4}(d) = \frac{d(d-1)}{2} = \mathcal{O}(d^2).$$

4.6 G5: Masked Syndrome Computation

G5 computes $[[s'[i]]] = [[u[i]]] \mathbf{H}^\top \in \mathbb{F}p^{n-k}$, where $\mathbf{H} \in \mathbb{F}p^{(n-k) \times n}$ is the public parity-check matrix. This operation is targeted by both published power-analysis attacks on CROSS [18,12]. Because $\mathbf{H}$ is public, the matrix-vector product decomposes share-wise: $s'[i]_k = u[i]_k \mathbf{H}^\top$ for each share index k . No cross-share multiplication occurs, and no fresh randomness is needed. We process one share index end-to-end before moving to the next. This temporal share separation does not affect the formal proof, but it reduces cross-share leakage in the physical time domain [16,5].

Algorithm 12 SecPubMatVec($[[u[i]]], \mathbf{H}$) — G5 (t_{mask} -SNI)

Require: Arithmetic shares $[[u[i]]]$ in $\mathbb{F}p^n$; public parity-check matrix $\mathbf{H} \in \mathbb{F}p^{(n-k) \times n}$ in systematic form $[I_{n-k} \mid P]$.

Ensure: $[[s'[i]]]$ with $\sum_k s'[i]_k \equiv u[i] \mathbf{H}^\top \pmod{p}$.

```

1: for  $k=0, \dots, d-1$  do                                 $\triangleright$  temporal share separation: one share index at a time
2:   for  $j=0, \dots, n-k-1$  do                                 $\triangleright$  systematic part:  $\mathbf{H}_{j,j}=1$ 
3:      $s'[i]_{k,j} \leftarrow u[i]_{k,j}$ 
4:   for  $\ell=n-k, \dots, n-1$  do                                 $\triangleright$  non-systematic accumulation
5:     for  $j=0, \dots, n-k-1$  do
6:        $s'[i]_{k,j} \leftarrow (s'[i]_{k,j} + u[i]_{k,\ell} \cdot \mathbf{H}[j][\ell - (n-k)]) \pmod{p}$ 
7: return  $[[s'[i]]]$ 

```

Theorem 16 (G5 t_{mask} -SNI). *SecPubMatVec is t_{mask} -SNI.*

Proof. Every output share $s'[i]_{k,j}$ is a $\mathbb{F}p$ -linear combination of shares $u[i]_{k,\ell}$ at the same share index k , and public matrix coefficients $\mathbf{H}[j][\ell - (n-k)]$. No two different share indices of $[[u[i]]]$ appear in the same expression, and no secret-dependent branch or memory access occurs. By the (AFFINE) typing rule of [4], this makes the gadget t_{mask} -SNI directly.

For completeness: take any t_i internal probes and t_o output probes with $t_i + t_o \leq t_{\text{mask}}$. These probed wires touch at most $t_i + t_o \leq d-1$ distinct share indices of $[[u[i]]]$.

Algorithm 13 SecA2B($[[x]], q$) — G6.0 (t_{mask} -SNI, PINI core + t_{mask} -SNI refresh [4])

Require: Arithmetic shares $[[x]] = (x_0, \dots, x_{d-1})$ of $x \in \mathbb{F}q$; bit length $k = \lceil \log_2 q \rceil$.

Ensure: Boolean shares $(x'_0, \dots, x'_{d-1}) \in (\{0, 1\}^k)^d$ with $x'_0 \oplus \dots \oplus x'_{d-1} = x$.

- 1: reduce each $x_k \leftarrow x_k \bmod q$ share-wise
 - 2: $(x'_0, \dots, x'_{d-1}) \leftarrow \text{SecA2BModp}((x_0, \dots, x_{d-1}), q)$ $\triangleright$ [8]; t_{mask} -PINI
 - 3: $(x'_0, \dots, x'_{d-1}) \leftarrow \text{Refresh}((x'_0, \dots, x'_{d-1}))$ $\triangleright$ G0.5, over $(\{0, 1\}^k, \oplus)$
 - 4: **return** $(x'_0, \dots, x'_{d-1})$
-

Since $[[\mathbf{u}[i]]]$ is a valid full-threshold d -sharing from G4 (Theorem 15), any subset of at most $d-1$ same-index shares is jointly uniform and independent of $\mathbf{u}[i]$. The simulator reproduces the joint distribution of all probed wires from t_i shares of $[[\mathbf{u}[i]]]$ and the public matrix $\mathbf{H}$, satisfying the SNI definition of [4]. $\square$

The cost of SecPubMatVec follows directly from Algorithm 12:

$$T_{G5}(d, n, k) = d \cdot \Theta((n-k)n) = \mathcal{O}(d(n-k)n), \quad R_{G5}(d, n, k) = 0.$$

4.7 G6: Masked Commitment Computation

G6 forms the per-round commitment values $\text{cmt}_0[i]$ and $\text{cmt}_1[i]$. It represents the boundary between masked internal computation and the public signature transcript. $\text{cmt}_0[i]$ is a protocol-public hash digest, published in the first challenge phase, but its pre-image assembles the two HIGH-sensitivity inputs before compression. The hash primitive inside G6 is the same masked Keccak- $f[1600]$ construction used in G0.2 and G1.1 [7]. It operates on a Boolean-masked state and absorbs its inputs share-wise, without ever recombining them. $[[s'[i]]]$ from G5 and $[[\bar{\mathbf{v}}[i]]]$ (or $[[\bar{\mathbf{v}}_G[i]]]$) from G2 are arithmetic shares, over $\mathbb{F}p$ and $\mathbb{F}z$ respectively, so each component must first be converted to a Boolean sharing before it can be absorbed. $\text{cmt}_1[i]$ needs no such conversion: its sole sensitive input, $\text{Seed}[i]$, is already Boolean-shared from G1.0.

Sub-Gadgets of G6

G6.0 SecA2B. Each component of $[[s'[i]]]$ and $[[\bar{\mathbf{v}}[i]]]$ (or $[[\bar{\mathbf{v}}_G[i]]]$) is converted to a Boolean sharing using SecA2BModp, the bitsliced arithmetic-to-Boolean conversion of Bronchain and Cassiers [8], applied over $\mathbb{F}q$ for $q \in \{p, z\}$. This construction is proven t_{mask} -PINI by Bronchain and Cassiers, a composable notion introduced by Cassiers and Standaert [9] alongside SNI. We close each call with one Refresh (G0.5), applied over $(\{0, 1\}^{\lceil \log_2 q \rceil}, \oplus)$ the same way it is already applied to Boolean shares in G1.0's seed tree. This closing refresh is what lets SecA2B slot into the same t_{mask} -SNI composition chain as every other gadget in this schedule.

Lemma 2 (PINI implies t_{mask} -NI). *If a gadget is t_{mask} -PINI [9], it is t_{mask} -NI.*

Proof. Take any $t' \leq t_{\text{mask}}$ probes, split into t_1 internal probes and t_2 output-share indices A , $t_1 + t_2 = t'$. By the PINI definition, there is a set B of at most t_1 indices such that all probed wires and the output shares indexed by A can be simulated from input shares indexed by $A \cup B$. Since $|A \cup B| \leq |A| + |B| \leq t_2 + t_1 = t' \leq t_{\text{mask}}$, this is exactly the t_{mask} -NI simulator requirement. $\square$

Theorem 17 (G6.0 t_{mask} -SNI). *SecA2B is t_{mask} -SNI at $d = t_{\text{mask}} + 1$ shares.*

Proof. SecA2BModp is t_{mask} -PINI [8], hence t_{mask} -NI by Lemma 2. Refresh is t_{mask} -SNI (Theorem 22), and the same argument holds over $(\{0,1\}^k, \oplus)$ as over $\mathbb{F}_z$. Placing an SNI sink at the end of an NI gadget re-establishes full t_{mask} -probing security for the composite [4], the same rule used for masked Keccak throughout this schedule (Theorem 6, Step 5; Theorem 7). This gives SecA2B the same t_{mask} -SNI interface used by every other gadget in this composition tree. $\square$

The cost of SecA2B follows directly from Algorithm 13:

$$\mathsf{T}_{\text{G6.0}}(d, \mu) = \mathcal{O}(\mu d^2) \text{ (SecA2BModp [8])}, \quad \mathsf{R}_{\text{G6.0}}(d, \mu) = \mathcal{O}(\mu d^2).$$

G6.1–G6.2 MaskedSHAKE squeeze and Reveal. MaskedSHAKE.squeeze returns d Boolean shares of the digest, one per share of the sponge state, since the permutation never recombines shares internally. The final commitment is obtained by calling Reveal on these d shares. Reveal is t_{mask} -NIO: it takes a valid d -sharing and outputs the plaintext value, safe precisely because that value is already protocol-public.

$$\begin{aligned} \mathsf{T}_{\text{G6.1}}(d) &= \mathsf{T}_{\text{G0.2}}(d) = \mathcal{O}(d^2), & \mathsf{R}_{\text{G6.1}}(d) &= \mathsf{R}_{\text{G0.2}}(d) = \mathcal{O}(d^2). \\ \mathsf{T}_{\text{G6.2}}(d) &= d = \mathcal{O}(d), & \mathsf{R}_{\text{G6.2}}(d) &= 0. \end{aligned}$$

Theorem 18 (G6 t_{mask} -NIO). *Under the assumption that the masked Keccak- $f[1600]$ permutation of Bos et al. is t_{mask} -NI [7], and given Theorem 17, the gadget SecCommit is t_{mask} -NIO with public outputs $\text{cmt}_0[i]$, $\text{cmt}_1[i]$.*

Proof. Each SecA2B call is t_{mask} -SNI by Theorem 17. Each absorb step XORs one input share into the corresponding Keccak state share, with no cross-share interaction, so the absorb is t_{mask} -SNI for free. Composing the SNI SecA2B output with the SNI absorb, and that with the t_{mask} -NI masked permutation [7], yields an NI composite [4], for both the $\text{cmt}_0[i]$ chain and the $\text{cmt}_1[i]$ chain, up to the point squeeze returns $[[h]]$ (or $[[h']]]$). Reveal is t_{mask} -NIO by definition, since the value it outputs is protocol-public, and sinking an NIO gadget after an NI composite preserves NIO for the whole chain. Both outputs are protocol-public, so they are given to the simulator for free. This is exactly t_{mask} -NIO. $\square$

The cost of SecCommit follows directly from Algorithm 14:

$$\begin{aligned} \mathsf{T}_{\text{G6}} &= (n-k) \cdot \mathsf{T}_{\text{G6.0}}(d, p) + L \cdot \mathsf{T}_{\text{G6.0}}(d, z) + \mathcal{O}(1) \cdot \mathsf{T}_{\text{G6.1}} + \mathsf{T}_{\text{G6.2}} = \mathcal{O}(n \mu d^2), \\ \mathsf{R}_{\text{G6}} &= (n-k) \cdot \mathsf{R}_{\text{G6.0}}(d, p) + L \cdot \mathsf{R}_{\text{G6.0}}(d, z) + \mathcal{O}(1) \cdot \mathsf{R}_{\text{G6.1}} = \mathcal{O}(n \mu d^2). \end{aligned}$$

Algorithm 14 SecCommit($[[s'[i]]], [[\bar{v}[i]]], [[Seed[i]]]_B, \text{Salt}, i, c$) — G6 ($t_{\text{mask-NIo}}$)

Require: Arithmetic shares $[[s'[i]]]$ in $\mathbb{F}p^{n-k}$ from G5; arithmetic shares $[[\bar{v}[i]]]$ in $\mathbb{F}z^n$ (or $[[\bar{v}_G[i]]]$ in $\mathbb{F}z^m$ for R-SDP(G)) from G2; Boolean shares $[[Seed[i]]]_B$ from G1.0; public $\text{Salt} \in \{0,1\}^{2\lambda}$, round index i , constant $c = 2t - 1$.

Ensure: Public commitments $\text{cmt}_0[i]$, $\text{cmt}_1[i]$.

```

1: for  $j = 1, \dots, n-k$  do
2:    $[[s'[i]_j]]_B \leftarrow \text{SecA2B}([s'[i]_j], p)$  ▷ G6.0: one call per component, over  $\mathbb{F}p$ 
3: if R-SDP then
4:   for  $j = 1, \dots, n$  do
5:      $[[\bar{v}[i]_j]]_B \leftarrow \text{SecA2B}([\bar{v}[i]_j], z)$ 
6: else
7:   for  $j = 1, \dots, m$  do
8:      $[[\bar{v}_G[i]_j]]_B \leftarrow \text{SecA2B}([\bar{v}_G[i]_j], z)$ 
9:  $\mathcal{H}_0 \leftarrow \text{MaskedSHAKE.init}()$ 
10:  $\text{MaskedSHAKE.absorb}(\mathcal{H}_0, [[s'[i]]]_B)$ 
11: if R-SDP then
12:    $\text{MaskedSHAKE.absorb}(\mathcal{H}_0, [[\bar{v}[i]]]_B)$ 
13: else
14:    $\text{MaskedSHAKE.absorb}(\mathcal{H}_0, [[\bar{v}_G[i]]]_B)$ 
15:  $\text{MaskedSHAKE.absorb}(\mathcal{H}_0, \text{Salt} \mid i+c)$  ▷ public inputs absorbed unmasked
16:  $[[h]] \leftarrow \text{MaskedSHAKE.squeeze}(\mathcal{H}_0)$  ▷ G6.1:  $d$  Boolean shares of the digest, still masked
17:  $\text{cmt}_0[i] \leftarrow \text{Reveal}([h])$  ▷ G6.2: the one declassification step
18:  $\text{Zero}([s'[i]]); \text{Zero}([s'[i]]_B)$ 
19: if R-SDP then
20:    $\text{Zero}([\bar{v}[i]]); \text{Zero}([\bar{v}[i]]_B)$ 
21: else
22:    $\text{Zero}([\bar{v}_G[i]]); \text{Zero}([\bar{v}_G[i]]_B)$ 
23:  $\mathcal{H}_1 \leftarrow \text{MaskedSHAKE.init}()$ 
24:  $\text{MaskedSHAKE.absorb}(\mathcal{H}_1, [[Seed[i]]]_B)$ 
25:  $\text{MaskedSHAKE.absorb}(\mathcal{H}_1, \text{Salt} \mid i+c)$ 
26:  $[[h']] \leftarrow \text{MaskedSHAKE.squeeze}(\mathcal{H}_1)$ 
27:  $\text{cmt}_1[i] \leftarrow \text{Reveal}([h'])$ 
28: return  $\text{cmt}_0[i]$ ,  $\text{cmt}_1[i]$ 

```

4.8 G7: Masked First-Response Computation

G7 computes $[[y[i]]] = [[u'[i]]] + \text{chall}_1[i] \cdot [[e'[i]]]$, where $[[e'[i]]] = [[g^{\bar{e}'[i]}]]$ is the shared group-element lift of the ephemeral exponent vector, and $\text{chall}_1[i] \in \mathbb{F}p^*$ is a public scalar derived by Fiat–Shamir. $y[i]$ is published as part of the signature. G7 covers the device-side computation leading to $[[y[i]]]$; the security implications of subsequently revealing $y[i]$ is addressed in G8.

Sub-Gadgets of G7

G7.1 — SecExpLift. The ephemeral exponent vector $[[e'[i]]] \in \mathbb{F}z^n$ from G1 is lifted to $[[e'[i]]] \in \mathbb{F}p^n$ using the same masked exponentiation gadget developed in G3 (Algo-

rithm 10), applied here to the structurally identical input $[[\bar{e}'[i]]]$. The two invocations use independent inputs and independent fresh randomness, so Theorem 14 applies without modification.

$$T_{G7.1}(d, n) = T_{G3}(d, n) = \mathcal{O}(nd^3), \quad R_{G7.1}(d, n) = R_{G3}(d, n) = \mathcal{O}(nd^3).$$

G7.2 — SecLinMap. The shared lifted vector $[[\mathbf{e}'[i]]]$ is multiplied share-wise by the public scalar $\text{chall}_1[i]$: $s_{j,k} \leftarrow \text{chall}_1[i] \cdot \mathbf{e}'[i]_{j,k} \bmod p$. Each output share depends on exactly one input share and one public constant.

$$T_{G7.2}(d, n) = nd = \mathcal{O}(nd), \quad R_{G7.2}(d, n) = 0.$$

G7.3 — SecAdd. The result is combined with $[[\mathbf{u}'[i]]]$ from G1 share-wise: $\mathbf{y}[i]_{j,k} \leftarrow (s_{j,k} + \mathbf{u}'[i]_{j,k}) \bmod p$.

$$T_{G7.3}(d, n) = nd = \mathcal{O}(nd), \quad R_{G7.3}(d, n) = 0.$$

Algorithm 15 $\text{SecFirstResp}([[\mathbf{u}'[i]]], \text{chall}_1[i], [[\bar{e}'[i]]]) \text{ — G7 } (t_{\text{mask-SNI}})$

Require: Arithmetic shares $[[\mathbf{u}'[i]]]$ in $\mathbb{F}p^n$ from G1; public scalar $\text{chall}_1[i] \in \mathbb{F}p^*$; arithmetic shares $[[\bar{e}'[i]]]$ in $\mathbb{F}z^n$ from G1.

Ensure: Arithmetic shares $[[\mathbf{y}[i]]]$ in $\mathbb{F}p^n$ with $\sum_k \mathbf{y}[i]_k \equiv \mathbf{u}'[i] + \text{chall}_1[i] \cdot \mathbf{e}'[i] \pmod{p}$.

- 1: $[[\mathbf{e}'[i]]] \leftarrow \text{SecExpLift}([[\bar{e}'[i]]])$ $\triangleright$ G7.1: reuses G3; $\bar{e}'[i]$ never recombined
 - 2: **for** $j = 1, \dots, n$ **do**
 - 3: **for** $k = 0, \dots, d-1$ **do**
 - 4: $s \leftarrow \text{chall}_1[i] \cdot \mathbf{e}'[i]_{j,k} \bmod p$ $\triangleright$ G7.2: transient register; one share $\times$ public scalar
 - 5: $\mathbf{y}[i]_{j,k} \leftarrow (\mathbf{u}'[i]_{j,k} + s) \bmod p$ $\triangleright$ G7.3: share-wise add; s consumed and not stored
 - 6: **return** $[[\mathbf{y}[i]]]$
-

Theorem 19 (G7 $t_{\text{mask-SNI}}$). *The gadget SecFirstResp is $t_{\text{mask-SNI}}$.*

Proof. We apply the composition rules of [4] to the sub-gadget chain

$$\underbrace{\text{G7.1}}_{\text{SNI}} \rightarrow \underbrace{\text{G7.2}}_{\text{SNI}} \rightarrow \underbrace{\text{G7.3}}_{\text{SNI}}.$$

G7.1 (SecExpLift) is $t_{\text{mask-SNI}}$ by Theorem 14, since G3 was proven SNI and the present invocation is an independent application of the same construction to $[[\bar{e}'[i]]]$. G7.2 is $t_{\text{mask-SNI}}$ for free [4]: each output share depends on one input share and the public scalar $\text{chall}_1[i]$. The sequential composition of two $t_{\text{mask-SNI}}$ gadgets is itself $t_{\text{mask-SNI}}$; thus G7.1 composed with G7.2 is $t_{\text{mask-SNI}}$. G7.3 is $t_{\text{mask-SNI}}$ for free by the same argument; composing the SNI output of G7.1+G7.2 through the SNI G7.3 yields an SNI composite. The full gadget G7 is therefore $t_{\text{mask-SNI}}$.

The plaintext invariant holds throughout: G7.1 never recombines $\bar{e}'[i]$ (Theorem 14); G7.2 and G7.3 operate share-wise on individual shares and public constants, introducing no cross-share interaction. $\square$

The cost of `SecFirstResp` follows directly from Algorithm 15:

$$T_{G7}(d, n) = T_{G3}(d, n) + \mathcal{O}(nd) = \mathcal{O}(nd^3), \quad R_{G7}(d, n) = R_{G3}(d, n) = \mathcal{O}(nd^3).$$

Theorem 20 (G8 Transcript Safety). *CROSS-ID is zero-knowledge [3], $\text{cmt}_0[i]$ is computationally binding (collision resistance of the underlying SHAKE-based hash), and $\text{cmt}_0[i]$ and $\text{cmt}_1[i]$ are fixed and published before chall_2 is determined. Given this, publishing $\mathbf{y}[i]$ right after $G7$, and then running `TranscriptPublish`, reveals no more information than the public signature transcript of the unmasked CROSS signing algorithm. In particular, the `Reveal` calls do not violate t_{mask} -probing security.*

$$T_{G8} = T \cdot \mathcal{O}(nd) = \mathcal{O}(Tnd), \quad R_{G8} = 0.$$

5 Masked CROSS Sign — Top-Level Algorithm

Algorithm 16 shows the full masked signing flow for CROSS. The pseudocode follows the gadget schedule above. All HIGH and MEDIUM sensitivity variables appear in shared form. `Reveal` appears only at points that are protocol-public.

5.1 Gadget Composition and Security Proof

This section ties the per-gadget theorems together into a single end-to-end security claim for `MaskedCROSS_Sign`, following the NI/SNI/NIo composition framework [4].

Composition Rules Used Three facts about NI, SNI, and NIo gadgets are used throughout the composition argument, all from [4]. Feeding the output of an SNI gadget into an NI gadget gives an NI composite. Chaining several SNI gadgets in sequence gives another SNI gadget. And placing an SNI gadget as the last step of an otherwise NI or NIo chain restores full t_{mask} -probing security for the whole chain, since a public NIo output never costs the simulator anything extra.

5.2 Main Theorem

Theorem 21 (Probing Security of MaskedCROSS_Sign). *Let $d = t_{\text{mask}} + 1$. Assume the masked Keccak-f[1600] permutation of Bos et al. is t_{mask} -NI in the probing model [7], and that the CROSS identification protocol CROSS-ID is zero-knowledge, so G8 publication is safe [3]. Given these two assumptions, together with the t_{mask} -SNI ISW multiplication gadget (Theorem 15) and the t_{mask} -SNI SecA2B gadget (Theorem 17), both already established above, the masked signing algorithm `MaskedCROSS_Sign` (Algorithm 16) is t_{mask} -probing secure: the joint distribution of any set of at most t_{mask} probes inside the signing computation is statistically independent of Seed_{sk} , $\bar{e}$, and $\bar{e}_G$.*

Algorithm 16 MaskedCROSS_Sign($[[\text{Seed}_{sk}]], \text{Msg}$)

Require: Shares of Seed_{sk} , $[[\text{Seed}_{sk}]]$ and message Msg ; masking order t_{mask} .

Ensure: Same public signature object as Algorithm 18.

```

1:  $[[\text{Seed}_{sk}]] \leftarrow \text{SampleSharedSeed}([[\text{Seed}_{sk}]])$  ▷ G0.1 (SNI)
2:  $([[\bar{e}]], [[\bar{e}_G]], \mathbf{H}, \bar{\mathbf{M}}) \leftarrow \text{MaskedExpandPrivateSeed}([[\text{Seed}_{sk}]])$  ▷ G0 ( $t_{\text{mask}}$ )-probing
3:  $\text{Salt} \xleftarrow{\$} \{0,1\}^{2\lambda}; \text{Seed} \xleftarrow{\$} \{0,1\}^\lambda$ 
4:  $([[\text{Seed}[1]]]_B, \dots, [[\text{Seed}[t]]]_B) \leftarrow \text{MaskedSeedLeaves}()$  ▷ G1.0 (SNI)
5: for  $i = 1, \dots, t$  do
6:    $([[\bar{e}'_G[i]]], [[\bar{e}'[i]]], [[\mathbf{u}'[i]]]) \leftarrow \text{MaskedRoundSample}(i, [[\text{Seed}[i]]]_B, \text{Salt}, c, \bar{\mathbf{M}})$  ▷
    $G1(t_{\text{mask}}\text{-probing})$ 
7:   if R-SDP( $G$ ) then
8:      $[[\bar{\mathbf{v}}_G[i]]] \leftarrow \text{SecRestrSub}([[\bar{e}_G]], [[\bar{e}'_G[i]]])$  ▷ G2 (SNI)
9:      $[[\bar{\mathbf{v}}[i]]] \leftarrow \text{SecRestrSub}([[\bar{e}]], [[\bar{e}'[i]]])$  ▷ G2 (SNI)
10:     $[[\mathbf{v}[i]]] \leftarrow \text{SecExpLift}([[\bar{\mathbf{v}}[i]]])$  ▷ compute  $g^{\bar{\mathbf{v}}[i]}$ , G3 (SNI)
11:     $[[\mathbf{u}[i]]] \leftarrow \text{SecMult}([[\mathbf{v}[i]]], [[\mathbf{u}'[i]]])$  ▷ ISW, G4(SNI)
12:     $[[\mathbf{s}'[i]]] \leftarrow \text{SecPubMatVec}([[\mathbf{u}[i]]], \mathbf{H})$  ▷ G5 (SNI)
13:    if R-SDP then
14:       $(\text{cmt}_0[i], \text{cmt}_1[i]) \leftarrow \text{SecCommit}([[\mathbf{s}'[i]]], [[\bar{\mathbf{v}}[i]]], [[\text{Seed}[i]]]_B, \text{Salt}, i, c)$ 
15:    else
16:       $(\text{cmt}_0[i], \text{cmt}_1[i]) \leftarrow \text{SecCommit}([[\mathbf{s}'[i]]], [[\bar{\mathbf{v}}_G[i]]], [[\text{Seed}[i]]]_B, \text{Salt}, i, c)$  ▷ G6 (Nlo)

17: Public challenge phase (no masking required)
18:  $\text{chall}_1 \leftarrow \text{CSPRNG}_{(\mathbb{F}_p^*)^t}(\text{Hash}(\text{Hash}(\text{Msg}) \mid \text{digest}_{\text{cmt}} \mid \text{Salt}))$ 
19: for  $i = 1, \dots, t$  do
20:   recompute  $([[\bar{e}'[i]]], [[\mathbf{u}'[i]]])$  via G1
21:    $[[\mathbf{y}[i]]] \leftarrow \text{SecFirstResp}([[\mathbf{u}'[i]]], \text{chall}_1[i], [[\bar{e}'[i]]])$  ▷ G7 (SNI)
22:  $\text{chall}_2 \leftarrow \text{FixedWeightChall}_w(\text{Hash}([[\mathbf{y}]] \mid \text{digest}_{\text{chall}_1}))$ 
23:  $(\text{Salt}, \text{digest}_{\text{cmt}}, \text{digest}_{\text{chall}_2}, \text{Path}, \text{Proof}, \text{resp}) \leftarrow$ 
    $\text{TranscriptPublish}(\text{chall}_2, \{[[\mathbf{y}[i]]], [[\bar{\mathbf{v}}[i]]], [[\bar{\mathbf{v}}_G[i]]], \text{Seed}\})$  ▷ G8
24: return public signature  $(\text{Salt}, \text{digest}_{\text{cmt}}, \text{digest}_{\text{chall}_2}, \text{Path}, \text{Proof}, \text{resp})$ 

```

Proof. By Theorems 6–20, each of G0–G8 satisfies the notion recorded in Table 1.

$$\underbrace{G0}_{t\text{-prob}} \rightarrow \underbrace{G1}_{t\text{-prob}} \rightarrow \underbrace{G2}_{\text{SNI}} \rightarrow \underbrace{G3}_{\text{SNI}} \rightarrow \underbrace{G4}_{\text{SNI}} \rightarrow \underbrace{G5}_{\text{SNI}} \rightarrow \underbrace{G6}_{\text{Nlo}} \rightarrow \underbrace{G7}_{\text{SNI}} \rightarrow \underbrace{G8}_{\text{ZK}}.$$

We verify composition using the rules of [4]. G0 and G1 are each t_{mask} -probing secure by their own composition arguments (Theorems 6, 11), so the shares they deliver satisfy every downstream precondition. G2–G5 are each t_{mask} -SNI, so the sub-chain is t_{mask} -SNI. G6 is t_{mask} -Nlo: its public outputs $\text{cmt}_0[i]$, $\text{cmt}_1[i]$ cost the simulator nothing extra, so the chain’s probing security is unaffected. G7, t_{mask} -SNI, sits as an SNI sink right after G6 and re-establishes full t_{mask} -probing security up through G7. G8 reveals only protocol-safe values (Theorem 20): its **Reveal** calls are justified by the zero-knowledge property of CROSS-ID [3] and the binding of the commitment hash, exposing no shares of $[[\bar{e}]]$, $[[\bar{e}_G]]$, or any ephemeral secret. Since Seed_{sk} , $\bar{e}$, and $\bar{e}_G$ appear only inside the $[[\cdot]]$ types produced by G0 and consumed by G2–G7, the

simulator definition of t_{mask} -probing security lets any t_{mask} probes be reproduced without knowledge of those values, completing the proof. $\square$

The cost of the masked signing algorithm follows from 16.

$$\begin{aligned} T_{\text{Sign}} &= T_{G0} + T_{G1.0} + T \cdot (T_{G1}^{\text{round}} + T_{G2} + T_{G3} + T_{G4}^{\text{vec}} T_{G5} + T_{G6} + T_{G7} + T_{G8}^{\text{round}}) = \mathcal{O}(T(n\mu d^2 + nd^3)), \\ R_{\text{Sign}} &= R_{G0} + R_{G1.0} + T \cdot (R_{G1}^{\text{round}} + R_{G3} + R_{G4}^{\text{vec}} + R_{G6} + R_{G7}) = \mathcal{O}(T(n\mu d^2 + nd^3)). \end{aligned}$$

6 Implementation Results

We implement MaskedCROSS_Sign (Algorithm 16) for the R-SDP(G) variant `cross-sha3-r-sdpg-1-fast`. Every gadget G0–G8 is implemented as a share-generic C routine parameterized by the number of shares d , which is specified as a compile-time constant. The same code base runs at any masking order $t_{\text{mask}} = d - 1$ without gadget-level changes.

We benchmark on an x86-64 laptop, compiled with `gcc -O3 -march=native -std=c11`. We time Sign two ways for each run, the CPU’s cycle counter (`rdtsc`) and wall-clock time (`CLOCK_MONOTONIC`). The unmasked reference Sign implementation is substantially more expensive, so we average its performance over 30 executions. The masked Sign is far more expensive to run, so we average it over 30 calls instead. In both cases we report the mean and standard deviation, computed with Welford’s running-average method, so the numbers stay numerically stable even for the larger, noisier masked runs.

Table 2: Sign cost, unmasked vs. masked, for `cross-sha3-r-sdpg-1-fast` ($n=55$, $k=36$, $p=509$, $z=127$, $T=147$ rounds, weight $w=76$), measured on an x86-64 host. ρ is masked mean divided by unmasked mean.

	Unmasked	$d=2$	$d=3$
Sign, kCycles (ρ)	3050.77 (1X)	66287.11 (21.73X)	238127.09 (78.05X)
Sign, wall-clock (ms)	1.087	23.607	84.804

We build and run three versions of Sign for this variant: the unmasked reference, and the masked version at $d=2$ and $d=3$ shares. $d=2$ is first-order masking, $t_{\text{mask}}=1$, safe against a single probe. $d=3$ is second-order masking, $t_{\text{mask}}=2$, safe against any two probes. For both values of d , KeyGen, Sign, and Verify executed correctly end-to-end. A masked signature still verifies against the unmasked verifier.

Public key size (54 bytes) and signature size (11980 bytes) do not change under masking. Masking only changes how Sign computes internally, not the format of what it outputs. The secret key does grow. It now stores d shares instead of a single plaintext seed, so it scales linearly with d , 32 bytes per share. Going from $d=2$ to $d=3$ multiplies Sign’s cost by about $3.6\times$ as shown in Table 2 (ρ column).

A detailed performance breakdown by phase is provided in Table 3. This data is derived from a separate, instrumented run that utilized per-phase timers, with

Table 3: Sign phase breakdown for `cross-sha3-r-sdpg-1-fast`. Values are mean kCycles per Sign. Top block: exclusive phases (sum to the Sign wall up to $\approx 1\%$ of unprofiled glue). Bottom block: nested B2A/SecA2B costs already included in G1/G6.

Phase / sub-gadget	$d=2$	$d=3$
G0 key expansion	200	412
G1.0 seed tree	557	1306
G1 ephemeral sample	33981	122108
G2 transitive subtract	143	192
G3 restriction lift	1109	3151
G4 ISW SecMult	574	1138
G5 syndrome	731	1068
G6 commit	28529	89614
G7 first response	1102	3285
Public FS / Merkle	350	376
G8 reveal	16	18
Sign wall (profiled)	67546	223100
<i>Nested conversions (inside G1 / G6; not additive)</i>		
B2A _z /B2A _p	9684/6575	22639/62065
SecA2B _z /SecA2B _p	9496/9256	35407/33663

results averaged over 10 calls. As a result, the totals are close but not identical to those presented in Table 2. Almost all of the Sign cost is split between two phases: ephemeral sampling (G1) and commitments (G6). Their combined share of the Sign wall increases with d : 93% at $d=2$ and 95% at $d=3$. Both phases are expensive mainly because of Boolean-to-arithmetic (B2A) and arithmetic-to-Boolean (SecA2B) conversion. B2A’s share of Sign grows from 24% ($d=2$) to 38% ($d=3$). SecA2B’s share grows from 28% ($d=2$) to 31% ($d=3$).

Why the cost jumps this way. Section 5.1 gives the overall cost of Sign as $O(T(n\mu d^2 + nd^3))$. Most gadgets, including B2A, SecA2B, and the masked Keccak calls inside G1 and G6, cost $O(d^2)$. G3 (and G7, which reuses G3) is the exception: it chains $d-1$ ISW multiplications together, one per share, and each ISW call already costs $O(d^2)$ on its own, so the chain costs $O(d^3)$ per component. As the chain length is $d-1$, the cubic term increases at a faster rate than the quadratic terms as d increases. This is why the cost of moving from $d=2$ to $d=3$ is higher than a uniform per-share increase would indicate, because one term in the total scales with d^3 rather than d^2 .

6.1 Comparison with Dilithium

We compare masked CROSS implementations against masked implementations of Dilithium2 [11], which is built and run on the same machine as masked CROSS.

On masking overhead itself, Dilithium-2 costs noticeably more than CROSS at $d=2$ ($38\times$ vs. $21.73\times$, about $1.75\times$ higher), but by $d=3$ CROSS’s overhead ratio has overtaken Dilithium-2’s ($78.05\times$ vs. $70\times$, about $1.12\times$ higher). Second, in terms

Table 4: Sign cost, unmasked vs. masked, CROSS R-SDP(G)-1-fast vs. Dilithium-2 with the same x86-64 host and same compiler flags. ρ denotes masking overhead.

Scheme	Unmasked (kCycles)	$d=2$		$d=3$	
		(kCycles, ρ)		(kCycles, ρ)	
CROSS fast	3050.77	66287.11	(21.73 $\times$)	238127.09	(78.05 $\times$)
Dilithium-2 [11]	1436	55029	(38 $\times$)	100794	(70 $\times$)

of raw cost, a masked Sign call with CROSS is more expensive than Dilithium-2 across both orders measured: about 1.20 $\times$ more at $d=2$, increasing to about 2.36 $\times$ more at $d=3$. CROSS’s masked Sign costs more cycles than Dilithium-2’s masked Sign at every masking order measured here.

7 Conclusion

We presented the first masking countermeasure for the CROSS signature scheme. We derived a gadget-based masking schedule (G0–G8) from a sensitivity analysis of CROSS signing and proved t_{mask} -probing security for each gadget and their full composition in the NI/SNI/NIo framework. One gadget in this schedule has no direct counterpart in prior masked multivariate- or lattice-based signatures: the restriction-lift exponentiation (G3), which masks $g^{\tilde{v}[i]}$ via table-lookup, share-serial conversion, and ISW-multiplication sub-gadgets rather than a fixed linear or bilinear map. The masked CROSS signing implementation incurs overheads of 21.73 $\times$ at first order and 78.05 $\times$ at second order. The overhead of the first-order masked implementation of CROSS is cheaper than that of masked Dilithium-2 [11] (38 $\times$) but is overtaken by it at second order (70 $\times$). This crossover is driven by the $O(d^3)$ chained-ISW cost of G3/G7 compared to Dilithium’s more efficient $O(d^2)$ scaling.

Acknowledgments

This work was partially funded by the Horizon 2020 ERC Advanced Grant (101020005 Belfort) and the Cybersecurity Research Program Flanders (CRPF), with reference number VOEWICS02.

References

1. Alagic, G., Bros, M., Ciadoux, P., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Liu, Y.K., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Silberg, H., Smith-Tone, D., Waller, N.: Status Report on the First Round of the Additional Digital Signature Schemes for the NIST Post-Quantum Cryptography Standardization Process. NIST Internal Report NIST IR 8528, National Institute of Standards and Technology (2024). <https://doi.org/10.6028/NIST.IR.8528>, <https://doi.org/10.6028/NIST.IR.8528>

2. Alagic, G., Bros, M., Ciadoux, P., Dang, Q., Dang, T.H., Kelsey, J., Lichtinger, J., Liu, Y.K., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Silberg, H., Smith-Tone, D., Waller, N.: Status Report on the Second Round of the Additional Digital Signature Schemes for the NIST Post-Quantum Cryptography Standardization Process. NIST Internal Report NIST IR 8610, National Institute of Standards and Technology (May 2026). <https://doi.org/10.6028/NIST.IR.8610>, <https://doi.org/10.6028/NIST.IR.8610>
3. Baldi, M., Barenghi, A., Battagliola, M., Bitzer, S., Gianvecchio, M., Karl, P., Manganiello, F., Pavoni, A., Pelosi, G., Pintore, F., Santini, P., Schupp, J., Signorini, E., Slaughter, F., Wachter-Zeh, A., Weger, V.: CROSS: Codes and restricted objects signature scheme — reference and optimized implementation, release 2.0. https://www.cross-crypto.com/CROSS_submission_package_v2.0.zip (2024), NIST PQC Additional Digital Signature Schemes, Round 2 submission package
4. Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P.A., Grégoire, B., Strub, P.Y., Zucchini, R.: Strong non-interference and type-directed higher-order masking. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 116–129. ACM Press (Oct 2016). <https://doi.org/10.1145/2976749.2978427>
5. Barthe, G., Belaïd, S., Espitau, T., Fouque, P.A., Grégoire, B., Rossi, M., Tibouchi, M.: Masking the GLP lattice-based signature scheme at any order. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part II. LNCS, vol. 10821, pp. 354–384. Springer, Cham (Apr / May 2018). https://doi.org/10.1007/978-3-319-78375-8_12
6. Barthe, G., Belaïd, S., Espitau, T., Fouque, P.A., Grégoire, B., Rossi, M., Tibouchi, M.: Masking the GLP lattice-based signature scheme at any order. *Journal of Cryptology* **37**(1), 5 (Jan 2024). <https://doi.org/10.1007/s00145-023-09485-z>
7. Bos, J.W., Gourjon, M., Renes, J., Schneider, T., van Vredendaal, C.: Masking Kyber: First- and higher-order implementations. *IACR TCHES* **2021**(4), 173–214 (2021). <https://doi.org/10.46586/tches.v2021.i4.173-214>, <https://tches.iacr.org/index.php/TCHES/article/view/9064>
8. Bronchain, O., Cassiers, G.: Bitslicing arithmetic/boolean masking conversions for fun and profit with application to lattice-based KEMs. *IACR TCHES* **2022**(4), 553–588 (2022). <https://doi.org/10.46586/tches.v2022.i4.553-588>
9. Cassiers, G., Standaert, F.X.: Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Transactions on Information Forensics and Security* **15**, 2542–2555 (2020). <https://doi.org/10.1109/TIFS.2020.2971153>
10. Coron, J.S.: High-order conversion from Boolean to arithmetic masking. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 93–114. Springer, Cham (Sep 2017). https://doi.org/10.1007/978-3-319-66787-4_5
11. Coron, J.S., Gérard, F., Lepoint, T., Trannoy, M., Zeitoun, R.: Improved high-order masked generation of masking vector and rejection sampling in Dilithium. *IACR TCHES* **2024**(4), 335–354 (2024). <https://doi.org/10.46586/tches.v2024.i4.335-354>
12. Czuprynski, M., Mukherjee, A., Roy, S.S.: Correlation power analysis of LESS and CROSS. In: Nitaj, A., Petkova-Nikova, S., Rijmen, V. (eds.) AFRICACRYPT 25. LNCS, vol. 15651, pp. 270–295. Springer, Cham (Jul 2025). https://doi.org/10.1007/978-3-031-97260-7_13
13. Feneuil, T., Rivain, M., Warmé-Janville, A.: Masking-friendly post-quantum signatures in the threshold-computation-in-the-head framework. *Cryptology ePrint Archive*, Report 2025/520 (2025), <https://eprint.iacr.org/2025/520>
14. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: Securing hardware against probing attacks. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 463–481. Springer, Berlin, Heidelberg (Aug 2003). https://doi.org/10.1007/978-3-540-45146-4_27

15. Kannwischer, M.J., Krausz, M., Petri, R., Yang, S.Y.: pqm4: Benchmarking NIST additional post-quantum signature schemes on microcontrollers. Cryptology ePrint Archive, Report 2024/112 (2024), <https://eprint.iacr.org/2024/112>
16. Kundu, S., Norga, Q., Ojha, U.K., Ganguly, A., Karmakar, A., Verbaauwhede, I.: mUOV: Masking the unbalanced oil and vinegar digital signature scheme at first- and higher-order. Cryptology ePrint Archive, Report 2024/1875 (2024), <https://eprint.iacr.org/2024/1875>
17. Schneider, T., Paglialonga, C., Oder, T., Güneysu, T.: Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In: Lin, D., Sako, K. (eds.) PKC 2019, Part II. LNCS, vol. 11443, pp. 534–564. Springer, Cham (Apr 2019). https://doi.org/10.1007/978-3-030-17259-6_18
18. Schupp, J., Sigl, G.: A horizontal attack on the codes and restricted objects signature scheme (CROSS). Cryptology ePrint Archive, Report 2025/116 (2025), <https://eprint.iacr.org/2025/116>
19. Seker, O., Berndt, S., Wilke, L., Eisenbarth, T.: SNI-in-the-head: Protecting MPC-in-the-head protocols against side-channel analysis. Cryptology ePrint Archive, Report 2020/544 (2020), <https://eprint.iacr.org/2020/544>

A The Refresh gadget

Algorithm 17 Refresh $((a_0, \dots, a_{d-1}), z)$ — G0.5 (t_{mask} -SNI, matching RefreshM of [4])

Require: Arithmetic shares $a_0, \dots, a_{d-1}$ with $\sum_k a_k \equiv x \pmod{z}$.

Ensure: Fresh arithmetic shares $c_0, \dots, c_{d-1}$ with $\sum_k c_k \equiv x \pmod{z}$.

```

1: for  $k=0, \dots, d-1$  do
2:    $c_k \leftarrow a_k$ 
3: for  $i=0, \dots, d-2$  do
4:   for  $j=i+1, \dots, d-1$  do
5:      $r \xleftarrow{\$} \mathbb{F}z$ 
6:      $c_i \leftarrow (c_i + r) \bmod z$ 
7:      $c_j \leftarrow (c_j - r) \bmod z$ 
8: return  $(c_0, \dots, c_{d-1})$ 

```

Theorem 22 (G0.5 t_{mask} -SNI). *Refresh is t_{mask} -SNI.*

Proof. We apply Proposition 1 of [4] to our group $\mathbb{F}z$ in place of their generic ring K . Take any set of probes with t_1 internal probes and t_o output probes, where $t_1 + t_o \leq t_{\text{mask}}$. Each internal probe touches at most one line of the update matrix. So we can find at most t_1 shares whose full line we must simulate exactly. For those t_1 shares, we simulate their line using the real input share and fresh random values, exactly like the real computation. Every other output share still has at least one un-fixed random value in its line. We simulate each of those outputs with one fresh random value. So all probes, internal and output, are simulated using only t_1 input shares. This meets the SNI definition.

The cost of Refresh follows directly from Algorithm 17:

$$T_{\text{G0.5}}(d) = \frac{d(d-1)}{2} \cdot \mathcal{O}(1) = \mathcal{O}(d^2), \quad R_{\text{G0.5}}(d) = \frac{d(d-1)}{2} = \mathcal{O}(d^2)$$

B The CROSS Signing Algorithm

Algorithm 18 $\text{Sign}(sk, Msg)$

Require: sk : secret key $\text{Seed}_{sk} \in \{0,1\}^{2\lambda}$; $Msg \in \{0,1\}^*$: message.

Ensure: Sgn : signature.

Data: λ : security parameter; $c=2t-1$; $g \in \mathbb{F}_p^*$: generator of $\mathbb{E}$; t : number of rounds;
 w : weight of the second challenge.

Phase A. Key expansion

1: $\bar{e}, \bar{e}_G, \bar{\mathbf{H}}, \bar{\mathbf{M}} \leftarrow \text{ExpandSK}(\text{Seed}_{sk})$ $\tau \bar{e}, \tau \bar{\mathbf{H}} \leftarrow \text{ExpandSK}(\text{Seed}_{sk})$

Phase B. Sample salt, seed tree (public/protocol-level)

2: $\text{Seed} \xleftarrow{\$} \{0,1\}^\lambda$; $\text{Salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$
3: $(\text{Seed}[1], \dots, \text{Seed}[t]) \leftarrow \text{SeedTreeLeaves}(\text{Seed}|\text{Salt})$

Phase C. Per-round commitments

4: **for** $i=1, \dots, t$ **do**
5: $\bar{e}'_G[i], \mathbf{u}'[i] \leftarrow \tau \bar{e}'[i], \mathbf{u}'[i]$
 $\text{CSPRNG}_{\mathbb{F}_z^m \times \mathbb{F}_p^n}(\text{Seed}[i]|\text{Salt}[i+c]) \parallel \text{CSPRNG}_{\mathbb{F}_z^n \times \mathbb{F}_p^n}(\text{Seed}[i]|\text{Salt}[i+c])$
 $\bar{\mathbf{v}}_G[i] \leftarrow \bar{e}_G - \bar{e}'_G[i]$
 $\bar{e}'[i] \leftarrow \bar{e}'_G[i] \bar{\mathbf{M}}$
 $\bar{\mathbf{v}}[i] \leftarrow \bar{e} - \bar{e}'[i]$
6: **for** $j=1, \dots, n$ **do**
7: $\mathbf{v}[i]_j \leftarrow g^{\bar{\mathbf{v}}[i]_j}$
8: $\mathbf{u}[i] \leftarrow \mathbf{v}[i] * \mathbf{u}'[i]$
9: $\mathbf{s}'[i] \leftarrow \mathbf{u}[i] \mathbf{H}^\top$
10: $\text{cmt}_0[i] \leftarrow \text{Hash}(\mathbf{s}'[i]|\bar{\mathbf{v}}_G[i]|\text{Salt}[i+c])$ $\text{cmt}_0[i] \leftarrow \text{Hash}(\mathbf{s}'[i]|\bar{\mathbf{v}}[i]|\text{Salt}[i+c])$
 $\text{cmt}_1[i] \leftarrow \text{Hash}(\text{Seed}[i]|\text{Salt}[i+c])$

Phase D. First challenge chall_1 (public)

11: $\text{digest}_{\text{cmt}_0} \leftarrow \text{TreeRoot}(\text{cmt}_0[1] \dots \text{cmt}_0[t])$
12: $\text{digest}_{\text{cmt}_1} \leftarrow \text{Hash}(\text{cmt}_1[1] \dots \text{cmt}_1[t])$
13: $\text{digest}_{\text{cmt}} \leftarrow \text{Hash}(\text{digest}_{\text{cmt}_0}|\text{digest}_{\text{cmt}_1})$
14: $\text{digest}_{Msg} \leftarrow \text{Hash}(Msg)$
15: $\text{digest}_{\text{chall}_1} \leftarrow \text{Hash}(\text{digest}_{Msg}|\text{digest}_{\text{cmt}}|\text{Salt})$
16: $\text{chall}_1 \leftarrow \text{CSPRNG}_{(\mathbb{F}_p^*)^t}(\text{digest}_{\text{chall}_1}|t+c)$

Phase E. First response

17: **for** $i=1, \dots, t$ **do**
18: **for** $j=1, \dots, n$ **do**
19: $\mathbf{e}'[i]_j \leftarrow g^{\bar{e}'[i]_j}$
20: $\mathbf{y}[i] \leftarrow \mathbf{u}'[i] + \text{chall}_1[i] \cdot \mathbf{e}'[i]$

Phase F. Second challenge chall_2 (public)

21: $\text{digest}_{\text{chall}_2} \leftarrow \text{Hash}(\mathbf{y}[1] \dots \mathbf{y}[t]|\text{digest}_{\text{chall}_1})$
22: $\text{chall}_2 \leftarrow \text{CSPRNG}_{\mathcal{B}_{(t,w)}}(\text{digest}_{\text{chall}_2}|t+c+1) \in \{0,1\}^t$

Phase G. Second response and signature assembly

23: $\text{Proof} \leftarrow \text{TreeProof}(\text{cmt}_0[1] \dots \text{cmt}_0[t]|\text{chall}_2)$
24: $\text{Path} \leftarrow \text{SeedPath}(\text{Seed}|\text{Salt}|\text{chall}_2)$
25: **for** $i=1, \dots, t$ **do**
26: **if** $\text{chall}_2[i]=0$ **then**
27: $\text{resp}[i]_0 \leftarrow (\mathbf{y}[i], \bar{\mathbf{v}}_G[i])$ $\text{resp}[i]_0 \leftarrow (\mathbf{y}[i], \bar{\mathbf{v}}[i])$
 $\text{resp}[i]_1 \leftarrow \text{cmt}_1[i]$
28: $\text{Sgn} \leftarrow (\text{Salt}, \text{digest}_{\text{cmt}}, \text{digest}_{\text{chall}_2}, \text{Path}, \text{Proof}, \text{resp})$
29: **return** Sgn
