# LogVOLE: Succinct and Efficient Chosen-Input VOLE for ZK and Beyond

Lucien K. L. Ng*
Georgia Institute of Technology

Peter Rindal†
Category Labs

Akash Shah‡
University of California, Los Angeles

**Abstract**

Random Vector Oblivious Linear Evaluation (VOLE) correlations can be generated efficiently for zero-knowledge proofs and secure computation. Applications that need correlations on the receiver's actual input typically derandomize random VOLE by sending a linear-size vector, a main communication bottleneck.

We study chosen-input VOLE (CI-VOLE), where the receiver privately chooses a large vector $\mathbf{x}$, the sender fixes $\Delta$, and the parties obtain shares of $\mathbf{x} \cdot \Delta$ without communicating a linear-size object. This work presents LogVOLE, a concretely efficient CI-VOLE protocol with polylogarithmic end-to-end communication under Ring-LWE. The construction uses a recursive shrink/expand design: it authenticates short digests of the chosen input and then uses a succinct telescope to expand those relations back to the full vector. The protocol has $O(\lambda \log^2(|\mathbf{x}| + m_{\mathsf{msg}}))$ one-time setup and query communication and $O(|\mathbf{x}|/n)$ ring operations. Here, $n$ is the underlying ring degree, $\lambda$ is the computational security parameter, and $m_{\mathsf{msg}}$ is the size of the shares.

LogVOLE also supports a public-key non-interactive mode: for a fixed $\Delta$, the sender publishes reusable parameters, and a receiver sends one compact $\mathbf{x}$-dependent message to obtain the matching VOLE shares. This gives a route to non-interactive VOLE-based ZK with polylogarithmic communication for arbitrary circuits. We also give a malicious-security extension in the random oracle model.

Using LogVOLE, we construct the first concretely efficient VOLE-based (designated-verifier) ZK protocol for arbitrary circuits with polylogarithmic communication. At 128-bit computational and 40-bit statistical security, our implementation achieves online throughput of 13.9 million $\mathbb{Z}_p$ correlations/s and 12.9 million ZK multiplication gates/s using 16 cores per party. With golden-seed optimization and reusable setup, our GPU implementation processes 532.3 million $\mathbb{Z}_p$ correlations/s and 451.7 million ZK multiplication gates/s online on an H200 GPU. Using a CPU with a single thread, our implementation proves $1024 \times 1024$ matrix multiplication in about 4 seconds with 226 kB of communication.

*kng68@gatech.edu
†peterrindal@gmail.com
‡akashshah08@g.ucla.edu

# Contents


**1 Introduction** **5**
- 1.1 Motivation and Overview . . . 5
- 1.2 Our Contributions . . . 6
- 1.3 Roadmap . . . 7

**2 Related Work** **7**

**3 Preliminaries** **11**
- 3.1 Notation and Assumptions . . . 11
- 3.2 Detailed Description of Security Model . . . 12
- 3.3 Ring-LWE . . . 12
- 3.4 Linear Laconic Encryption . . . 16
- 3.5 Linearly Homomorphic Encryption . . . 16
- 3.6 Detailed Definitions for LEnc and LHE . . . 17
- 3.7 Zero-Knowledge Arguments . . . 19
- 3.8 Non-Interactive Arguments of Knowledge . . . 20

**4 High-Level Technical Overview** **21**
- 4.1 Revisiting TinyLabels . . . 21
- 4.2 Overview of LogVOLE . . . 23
- 4.3 Overview of Realizing the Recursion . . . 26
- 4.4 Supporting Non-interactive CI-VOLE . . . 27
- 4.5 Ideal Functionality and Modeling . . . 28
- 4.6 Application: IT-MAC-based ZKP . . . 28
- 4.7 Malicious Security . . . 28

**5 The Shrink/Expand Core** **29**
- 5.1 Shrink/Expand Interface . . . 29
- 5.2 Revisiting Linear Laconic Encryption . . . 30
- 5.3 One Leaf Path of Expand . . . 32
- 5.4 Bounded Openings Enable Translation . . . 34
- 5.5 Composes the Pieces Together . . . 35
- 5.6 The Complete Shrink/Expand Interface . . . 35

**6 Lower-Level Realization** **36**
- 6.1 CRT-Decomposition for Type Mismatch . . . 36
- 6.2 Gadget Decomposition for Noise Handling . . . 38
- 6.3 Summary of Transition Between Levels . . . 39
- 6.4 Wiring the Randomized Root Digest . . . 41

**7 Optimizations for Practical Efficiency** **42**
- 7.1 One-Digit Gadget Truncation . . . 42
- 7.2 Golden-Seed Grinding . . . 44

**8 Generalized CI-VOLE Functionality** **46**

**9 The Construction** **47**
9.1 Core CI-VOLE over $\mathcal{R}_q$ . . . 48

**10 Malicious Security** **48**
10.1 Protocol-Level Changes . . . 48

**11 IT-MAC-based ZKP** **58**
11.1 Upgrading to Malicious Verifier Security . . . 59

**12 Performance and Evaluation** **59**
12.1 Cost Analysis . . . 59
12.2 Evaluation . . . 60

**13 Discussion** **63**

**References** **65**

**Appendices** **74**

**A Digest Properties** **74**
A.1 Roadmap . . . 74
A.2 Hiding of the randomized root digest . . . 74
A.3 Digest Openings and Norm Bounds . . . 75
A.4 Binding of bounded openings . . . 76
A.5 Smudging of large openings . . . 77

**B Formalization and Semi-honest Security** **78**
B.1 Correctness . . . 78
B.2 Semi-honest proof of CI-VOLE . . . 79
B.3 Proof of Semi-Honest Correctness . . . 80
B.4 Proof of Semi-Honest Security of the Core Protocol . . . 86

**C Correctness of Truncation Optimization** **94**

**D Security Analysis for Truncation Optimization** **100**
D.1 Near-collision binding . . . 100
D.2 Truncated LHE simulation . . . 105
D.3 Truncated LacE simulation . . . 106
D.4 Composed truncated protocol . . . 107

**E Proofs for Malicious Security** **109**
E.1 Roadmap and the malicious interface . . . 109
E.2 Boundary Transcripts . . . 109
E.3 Boundary objects for the simulators . . . 110
E.4 Knowledge Extraction of $x$ . . . 113
E.5 Malicious security against a corrupted receiver . . . 116
E.6 Malicious security against a corrupted sender . . . 123
E.7 Putting the Corruption Cases Together . . . 127

**F Opened ZK Protocol and Proof Sketch** **128**
F.1 Why the Opening Is Sound . . . . . 129
F.2 Proof Sketch . . . . . 129

**G Omitted Details about Abrar, Malavolta, and Roy 2025** **130**
G.1 Cost Analysis of Abrar, Malavolta, and Roy 2025 . . . . . 130
G.2 Detailed Comparison about Constructions . . . . . 131

# 1 Introduction

## 1.1 Motivation and Overview

Vector Oblivious Linear Evaluation (VOLE) correlations are a central backend in modern zero-knowledge proofs, secure computation, PSI, and authenticated computation. In a VOLE instance, the sender holds a global key $\Delta$, the receiver holds a vector $\mathbf{x}$, and the parties obtain additive shares of $\mathbf{x} \cdot \Delta$, where the receiver's share is also an authenticated label for $\mathbf{x}$. Random VOLE is now a major success story: PCG and silent-correlation techniques generate random VOLE, or closely related correlations, with sublinear communication and strong concrete efficiency [BCGI18, BCG$^+$19b, SGRR19, BCG$^+$19a, YWL$^+$20, CRR21, RRT23].

Chosen-input VOLE (CI-VOLE), the focus of this work, asks for a stronger functionality. The receiver chooses a private length-$w$ vector $\mathbf{x}$, the sender fixes a private key $\Delta$, and the parties obtain a length-$w$ sharing of $\mathbf{x} \cdot \Delta$. Thus, both the chosen input and the output correlation have size $O(w)$. The challenge is to achieve communication *sublinear* in $w$. Standard MPC techniques do not readily yield such a primitive through a generic transformation. OT- or Beaver-triple-based approaches normally process the receiver's chosen input entry by entry. A naive homomorphic-encryption approach can let the receiver compute encrypted functions of a short seed or digest, but it still leaves the problem of turning the encrypted result into sender and receiver VOLE shares without revealing $\mathbf{x}$ or incurring linear-size communication. These obstacles do not establish a lower bound, but they explain why the functionality is remarkable.

Prior CI-VOLE constructions show that sublinear, and even logarithmic, communication is possible [ARS24, AMR25]. That theoretical possibility is already striking; the remaining question is whether it can yield a practical primitive with small end-to-end communication, SIMD-friendly computation, and an interface suited to real applications.

Concrete efficiency of CI-VOLE matters because it is a widely used backend for many crypto applications. For example, in VOLE-based ZK, random VOLE gives fast authenticated shares for a random vector, but the prover must still derandomize these correlations to match the actual execution trace by sending a linear-size correction vector. The same pattern appears in other applications where random correlations are easy to generate, but the parties must later correct them to match a privately chosen input. A succinct and concretely efficient CI-VOLE primitive removes this linear derandomization step.

Our closest concretely efficient precursor is TinyLabels [DLL25], which introduced a powerful shrink/expand technique for authenticated label (a.k.a. garbled label) delivery to reduce input-dependent communication in garbled-circuit protocols. At a high level, TinyLabels compresses a long input to a short digest, derives a short digest-dependent key, and uses structured ciphertexts to expand that key into selected authenticated labels. This practically efficient technique is the starting point of our construction. Nevertheless, TinyLabels does not by itself realize (staged) CI-VOLE with logarithmic end-to-end communication. Moreover, its target functionality delivers authenticated input labels with the inputs *revealed in the clear*.

LogVOLE realizes the full potential of this shrink/expand idea for CI-VOLE. Our first observation is that the input need not be revealed, eliminating both the input leakage and the linear-size query communication. Second, the (input, authenticated label) and (digest, digest-dependent key) pairs share the same structure. This self-similarity lets the parties reuse the same material recursively during the shrink/expand process, reducing end-to-end communication from linear to polylogarithmic in the input size. Together with our other optimizations, these observations yield a concretely efficient CI-VOLE protocol with polylogarithmic end-to-end communication under the Ring-LWE assumption.

**Parameter:** A finite field $\mathcal{M}$.
**State:** A global tuple $(\Delta, w)$, initially $\perp$.

$\mathcal{F}^{\mathsf{basic}}_{\mathsf{CI\text{-}VOLE}}.\mathsf{set\text{-}delta}(\mathbb{S} : \Delta \in \mathcal{M},\ w \in \mathbb{N})$:

- If the global tuple is unset:
  - Store $(\Delta, w)$ and send $w$ to $\mathbb{R}$.

$\mathcal{F}^{\mathsf{basic}}_{\mathsf{CI\text{-}VOLE}}.\mathsf{set\text{-}x}(\mathbb{R} : \mathbf{x} \in \mathcal{M}^w)$:

- If $(\Delta, w)$ is set:
  - Sample $\mathbf{k} \leftarrow \mathcal{M}^w$.
  - Compute $\mathbf{m} := \mathbf{k} + \mathbf{x} \cdot \Delta$.
  - Send $\mathbf{k}$ to $\mathbb{S}$ and $\mathbf{m}$ to $\mathbb{R}$.

Figure 1: Basic semi-honest CI-VOLE functionality.

LogVOLE also supports flexible share release orderings: a party may obtain its shares before interacting with the other party or before the input is chosen. In the *staged* mode, the sender can obtain its share before the receiver selects $\mathbf{x}$, which is the ordering needed by garbling-style applications. In the public-key non-interactive mode, the sender publishes a public key for a fixed $\Delta$; later, any receiver with a chosen $\mathbf{x}$ derives its VOLE share locally and sends the digest so that the sender can recover its corresponding share. Prior work also establishes the feasibility of such non-interactive interfaces, but LogVOLE combines this public-key mode with the concrete efficiency and polylogarithmic end-to-end cost needed for applications such as non-interactive VOLE-based ZK.

We demonstrate the strong concrete performance of our semi-honest interactive LogVOLE protocol. For ZK, we combine semi-honest CI-VOLE with the standard ZKP techniques of witness binding and IT-MAC checks. Our CPU implementation offers computational performance comparable to prior work on prover-efficient IT-MAC-based ZK, while reducing communication from linear to polylogarithmic in the verification circuit size. With the communication bottleneck removed and little interaction between parties, our protocols are also well suited to GPU acceleration. Our GPU implementations of CI-VOLE and ZK achieve an order of magnitude higher throughput than prior work. For garbled-input compression, we show concrete efficiency with substantial communication savings over prior work.

Finally, we propose extensions toward malicious security.

Figure 1 states the basic semi-honest CI-VOLE functionality. The generalized functionality with flexible release orderings appears in Section 8.

## 1.2 Our Contributions

- We construct LogVOLE, the first concretely efficient CI-VOLE protocol we know of with private chosen inputs, polylogarithmic end-to-end communication, and SIMD-friendly Ring-LWE computation. Our practical parameters support IT-MAC-based ZK with a designated verifier and garbling applications. More precisely, the parties can generate CI-VOLE shares for length-$w$ inputs and outputs over a message space of size $m$ with $O(\lambda \cdot (\log w + m)^2)$ communication for both one-time setup and each query.

- We propose two modes of CI-VOLE: staged and public-key non-interactive. In the staged mode, the sender obtains its shares before the receiver chooses $\mathbf{x}$. In the public-key non-interactive mode, the sender publishes a public key for its chosen global key $\Delta$. Any receiver can then

generate VOLE shares for its chosen input $\mathbf{x}$ locally, without interacting with the sender. The sender recovers its shares upon receiving the receiver's digest. This interface supports applications such as non-interactive VOLE-based ZK.

- Using LogVOLE, we obtain the first concretely efficient IT-MAC-based ZK protocol we know of with polylogarithmic communication for arbitrary circuits.
- We obtain garbled-input compression and authenticated input-label delivery with logarithmic end-to-end communication.
- We propose extensions toward malicious security for our CI-VOLE while preserving succinct communication. Our approaches include standard random linear checks, succinct non-interactive ZK proofs, and a knowledge-extraction argument.
- We implement LogVOLE and evaluate it in VOLE-based ZK and garbled-input compression, demonstrating that concrete communication savings can coexist with high computational throughput. Our implementation is available at `https://github.com/osu-crypto/libOTe`.

### 1.3 Roadmap

Figure 2 summarizes suggested reading paths. The high-level overview in Section 4 gives the shrink/expand idea; a short route continues directly to the performance results in Section 12. For the technical route, Section 5 explains the recursive mechanism, and Section 6 supplies the CRT decomposition, gadget decomposition, and root randomization used by the remaining paths. Section 3 provides the notation, security model, and encryption tools; Section 2 situates the construction among prior work.

The remaining paths cover core correctness and semi-honest security, practical optimizations, and the ZK application. The core security path first presents the generalized interface and full construction in Sections 8 and 9, then the digest properties in Section A, followed by the security statements and proofs in Section B. The malicious extension in Section 10 builds on this background; Section E gives its proofs. Section 7 presents lower digit truncation and golden-seed grinding. We separately prove the correctness and security of our protocol with lower digit truncation in Sections C and D, respectively. The security proof builds on the core argument in Section B.4. Section 11 presents the ZK application, and Section F gives its opened-setup proof sketch. The remaining appendices provide detailed comparisons and application results.

## 2 Related Work

**Random VOLE.** A long line of work [BCGI18, BCG$^+$19b, SGRR19, BCG$^+$19a, YWL$^+$20, BCG$^+$20b, CRR21, BCG$^+$22, BGIK22, BBMHS22, BCG$^+$23, BCCD23, RRT23, BBC$^+$24, BGH$^+$25, KPRR25, LXYY25, KPR$^+$25, GLH$^+$26, LLX$^+$26] develops sublinear-communication generation of random VOLE correlations and related silent correlations. [BCG$^+$19a] gives a concretely efficient two-round silent-correlation framework. PCFs support on-demand correlation generation [BCG$^+$20a, BCM$^+$25], while public-key constructions let parties derive correlations without further interaction after publishing public keys [OSY21, BCM$^+$24, BDSS25, CDH$^+$25]. These advances make random correlations efficient in many practical settings when the receiver can accept a random vector in advance. However, they do not directly address the chosen-input setting that we need: once the receiver later wants shares of a specific vector $\mathbf{x}$, prior VOLE-based applications still need a linear-size derandomization step to convert random VOLE into shares of

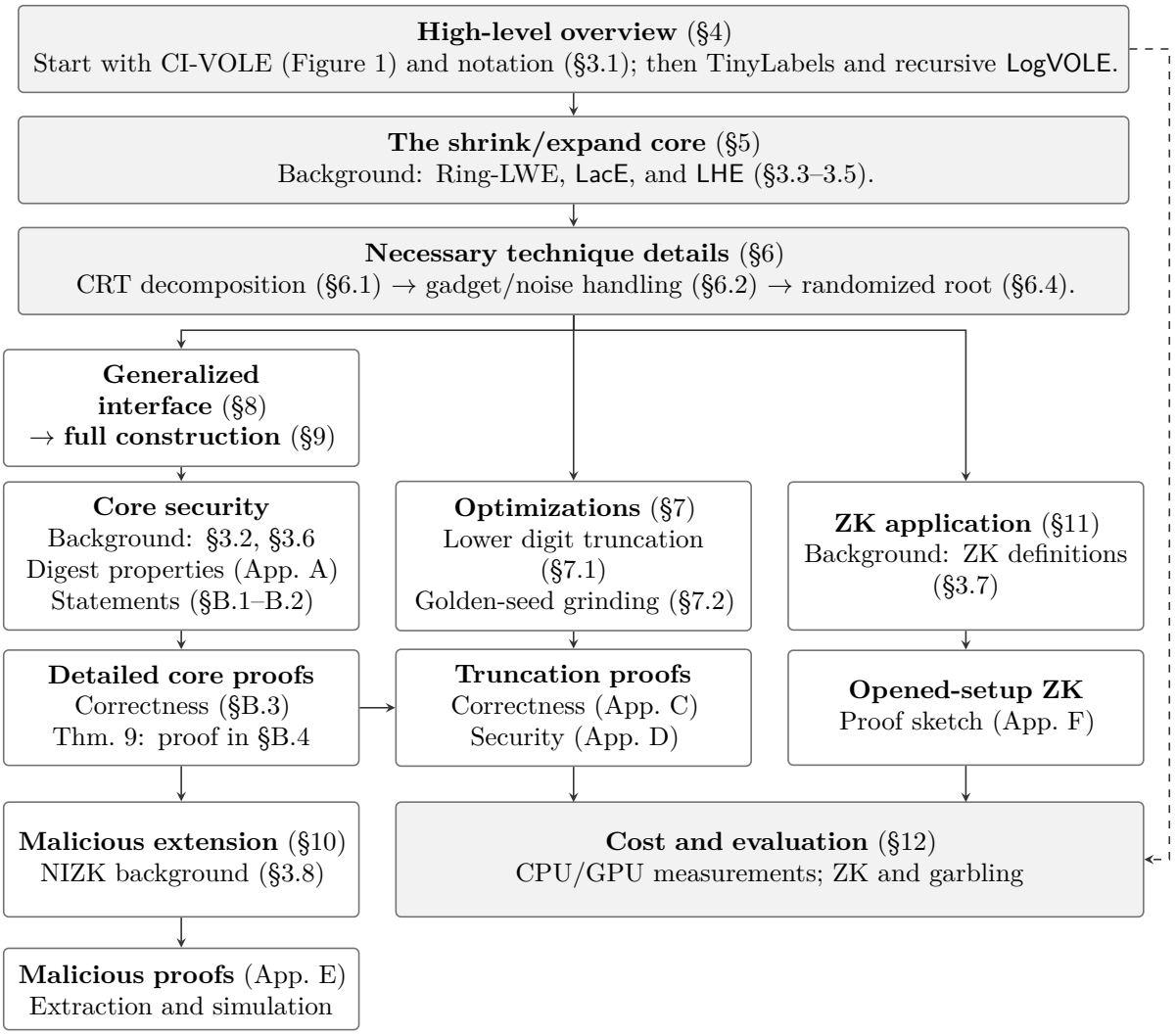


Figure 2: Reading roadmap. Solid arrows suggest a detailed reading order; background references identify prerequisites. The dashed arrow gives a short route from the overview to performance. We separately prove the correctness and security of our protocol with lower digit truncation in Appendices C and D, respectively.

Table 1: Comparison of VOLE-related protocols.

| Scheme | Functionality | Assumptions | Total Communication | Computation | Support Non-Inter. | Concrete Efficiency |
|---|---|---|---|---|---|---|
| [BCGI18, BCG+19b] | Random-VOLE | LPN / code-based | Random: $o(\|\mathbf{x}\|) + \mathsf{poly}(\lambda)$<br>Chosen $\mathbf{x}$: $\Theta(\|\mathbf{x}\|) + \mathsf{poly}(\lambda)$ | $O(\|\mathbf{x}\|)$ Efficient Comp. | × | ✓ |
| [ARS24] | CI-VOLE | DCR, QR, Power-DDH, LWE | $O_\lambda(\|\mathbf{x}\|^{2/3})$ | $O_\lambda(\|\mathbf{x}\|^{4/3})$ $\mathsf{Mult}_G/\mathsf{Mult}_{\mathbb{Z}}$ | ✓ | × |
| [ARS24] | CI-VOLE | Power-RLWE | $O_\lambda(\|\mathbf{x}\|^{1/2})$ | $O(\|\mathbf{x}\|^{3/2})$ $\mathsf{Mult}_{\mathcal{R}}$ | ✓ | × |
| [AMR25] | CI-VOLE | LWE | $\tilde{O}(\lambda^5 \log \|\mathbf{x}\|)$ | $\tilde{O}(\lambda^4 \|\mathbf{x}\|)$ $\mathsf{Mult}_{\mathbb{Z}}$ | ✓ | × |
| TinyLabels [DLL25] | Staged CI-VOLE | RLWE | $O_\lambda(\|\mathbf{x}\|)$ | $O(\|\mathbf{x}\| \log \|\mathbf{x}\|/n)$ $\mathsf{Mult}_{\mathcal{R}}$ | × | ✓ |
| **Ours** | Staged CI-VOLE | RLWE | $O(\lambda \log^2 \|\mathbf{x}\|)$ | $O(\|\mathbf{x}\|/n)$ $\mathsf{Mult}_{\mathcal{R}}$ | × | ✓ |
| **Ours** | CI-VOLE | RLWE | $O(\lambda^3)$ | $O(\|\mathbf{x}\|/n)$ $\mathsf{Mult}_{\mathcal{R}}$ | ✓ | ✓ |

• $\tilde{O}(\cdot)$ denotes $O(\cdot)$ with $\mathsf{poly}(\log \lambda)$ factors skipped. $O_\lambda(\cdot)$ denotes $O(\cdot)$ with $\mathsf{poly}(\lambda)$ factors skipped. $n$ is the degree of $\mathcal{R}$.
• For the random-VOLE row, the first line counts only random-correlation generation. The second line is the end-to-end communication after adding the standard derandomization step for a chosen vector $\mathbf{x}$.
• "Supporting Non-Inter." means the protocol can be used in a non-interactive way, *e.g.*, by generating a long random correlation and then derandomizing it for multiple $\mathbf{x}$'s.
• For our protocol, the staged and non-interactive variants are listed separately because staging and non-interactive use are incompatible.

$\mathbf{x} \cdot \Delta$. Our work is complementary to this line. Rather than further optimizing random correlations, we target the missing functionality and realize private chosen-input correlations directly, thereby removing the linear derandomization bottleneck.

**Chosen-Input (CI) VOLE.** Recent work studies CI-VOLE [ARS24, BCM+24, AMR25, DLL25]. [ARS24] showed the feasibility of chosen-input VOLE with sublinear communication, obtaining $O(|\mathbf{x}|^{2/3})$ communication under DDH/LWE-like assumptions and $O(|\mathbf{x}|^{1/2})$ under power-RLWE assumptions. This is conceptually important because it eliminates the derandomization step used in prior VOLE-based protocols, but the resulting constructions are computationally expensive; for example, the $O(|\mathbf{x}|^{1/2})$ regime incurs $O(|\mathbf{x}|^{3/2})$ computation. [AMR25] later obtained logarithmic communication for a more general outer-product primitive that can be specialized to CI-VOLE. However, once their parameters are instantiated for the large, noise-free labels needed in ZK and GC, the cost is about $O(\lambda^5 \log |\mathbf{x}|)$ communication and $O(\lambda^4 |\mathbf{x}|)$ arithmetic, so we view it mainly as a feasibility result. Appendix G gives the detailed parameter comparison.

**Staged CI-VOLE.** If one strengthens CI-VOLE so that the sender must learn its share before the receiver's input is fixed, then the closest prior line is not phrased as VOLE but as authenticated input-label delivery for garbled circuits [AIKW13, DLL25]. In these works, the garbler fixes the 0-labels (the labels representing wire value 0) and the global offset $\Delta$ during the one-time setup, and later the evaluator recovers labels of the form $K_{x_i}^{(i)} = K_0^{(i)} + x_i \cdot \Delta$. This is the closest prior analogue to the "commitment-style" aspect of our CI-VOLE functionality, namely that the sender-side masks are available before the actual input is chosen. TinyLabels [DLL25] is the main practical precursor: it gives a concretely efficient shrink-and-expand mechanism based on laconic encryption and linearly homomorphic encryption, but in a garbled-circuit setting where the input vector $\mathbf{x}$ is *revealed* to the sender, the one-time setup remains $O(|\mathbf{x}|\lambda + \mathsf{poly}(\lambda))$, and rate $1 + o(1)$ is reached only after amortizing this setup across many instances. Our protocol lifts this viewpoint to CI-VOLE: LogVOLE can hide the receiver's input in the staged functionality while reducing one-time setup and per-query communication costs to $O(\lambda \log^2 |\mathbf{x}|)$ bits. Table 1 summarizes the landscape across random VOLE, prior CI-VOLE, staged predecessors, and our construction.

**IT-MAC-based ZKP.** A sequence of works [DIO21a, YSWW21, DILO22, LHLA25a] shows that VOLE-generated information-theoretic MACs yield constant-round, fast, scalable designated-verifier zero-knowledge proofs with low prover overhead; QuickSilver and its successors [YSWW21, DILO22, LHLA25a] are the main examples. These systems still begin with random VOLE correlations and

Table 2: Comparison of garbled-input compression schemes.

| Scheme | Assumptions | One-Time Setup Communication | Per-Query Communication | Computation | Concrete Efficiency |
|---|---|---|---|---|---|
| Naive | – | – | $\lvert\mathbf{x}\rvert \cdot \lambda$ | $O(\lvert\mathbf{x}\rvert)$ Hashing | ✓ |
| [AIKW13] | DDH | $O_\lambda(\lvert\mathbf{x}\rvert)$ | $\lvert\mathbf{x}\rvert + \frac{\lvert\mathbf{x}\rvert}{\lambda}$ | $O(\lvert\mathbf{x}\rvert \cdot \mathsf{poly}(\lambda))$ $\mathsf{Mult}_G$ | × |
| [AIKW13] | (R)LWE | $O_\lambda(\lvert\mathbf{x}\rvert)$ | $\lvert\mathbf{x}\rvert + \frac{\lvert\mathbf{x}\rvert}{\lambda}$ | $O(\lvert\mathbf{x}\rvert \cdot \mathsf{poly}(\lambda))$ $\mathsf{Add}_q$ | × |
| [ABI+23] | Bilinear DDH | $O_\lambda(\lvert\mathbf{x}\rvert^2)$ | $\lvert\mathbf{x}\rvert + \frac{\lvert\mathbf{x}\rvert}{\lambda}$ | $O(\lvert\mathbf{x}\rvert\lambda)$ $\mathsf{Mult}_G$ + $O(\lvert\mathbf{x}\rvert)$ Bilinear ops | × |
| [ABI+23] | iO + SSBH | – | $\lvert\mathbf{x}\rvert + \mathsf{poly}(\log \lvert\mathbf{x}\rvert, \lambda)$ | $O(\lvert\mathbf{x}\rvert \cdot \mathsf{poly}(\lambda))$ iO-eval | × |
| [ABI+23] | RSA | – | $\lvert\mathbf{x}\rvert + \mathsf{poly}(\lambda)$ | $O(\lvert\mathbf{x}\rvert \log \lvert\mathbf{x}\rvert)$ $\mathsf{Mult}_{\mathsf{RSA}}$ | × |
| [AMR25] | LWE | $\tilde{O}(\lvert\mathbf{x}\rvert \cdot \lambda^5)$ | $\lvert\mathbf{x}\rvert + \tilde{O}(\lambda^2(\log \lvert\mathbf{x}\rvert + \lambda))$ | $\tilde{O}(\lvert\mathbf{x}\rvert \cdot \lambda^4)$ $\mathsf{Mult}_{\mathbb{Z}}$ | × |
| TinyLabels [DLL25] | RLWE | $O_\lambda(\lvert\mathbf{x}\rvert)$ | Theory: $\lvert\mathbf{x}\rvert + \mathsf{poly}(\lambda)$<br>Concrete: $6\lvert\mathbf{x}\rvert + \mathsf{poly}(\lambda)$ | $O(\lvert\mathbf{x}\rvert \log \lvert\mathbf{x}\rvert/n)$ $\mathsf{Mult}_{\mathcal{R}}$ | ✓ |
| **Ours** | RLWE | $O(\lambda \log^2 \lvert\mathbf{x}\rvert)$ | $O(\lambda \log^2 \lvert\mathbf{x}\rvert)$ | $O(\lvert\mathbf{x}\rvert/n)$ $\mathsf{Mult}_{\mathcal{R}}$ | ✓ |

must derandomize them to the actual witness by sending a correction vector, incurring communication linear in the trace size. AntMan [WYY+22] reduces this cost by combining VOLE with additively homomorphic encryption, achieving sublinear communication, but its communication is still $O(|\mathbf{x}|^{3/4})$, and it did not demonstrate concrete efficiency experimentally. Bui et al. [BCC+23] subsequently give a generic compiler from SIMD ZK to general circuits, reducing AntMan's communication to $O(|\mathbf{x}|^{1/2})$ and experimentally evaluating the resulting protocol. The implementation still incurs noticeable performance overhead from homomorphic encryption and polynomial multiplication. [BCM+24, LXY+25] observe that CI-VOLE can yield succinct zero-knowledge proofs. However, the CI-VOLE constructions they consider are not concretely efficient, making the resulting ZKP protocol impractical. They provide no experimental evaluation. In contrast, our work replaces the random-VOLE generation step itself with a concretely efficient CI-VOLE protocol, so the standard IT-MAC / QuickSilver-style checking layer can be retained while the end-to-end communication drops to $O(\lambda \log^2(|\mathbf{x}| + \sigma))$ for arbitrary circuits. Table 3 summarizes the comparison.

**Garbled Input Compression.** Garbled circuits (GC) are a fundamental building block for secure multiparty computation [Yao86, BHR12]. While substantial effort has gone into reducing the end-to-end communication of GC [KS08, ZRE15, RR21, HK20, HK21a, HK21b, HKP21, HHK+22, BGH+23, GHJY26, GLOS15, HKO22, HKN24, LLZ25, NK25, HKN+25, GTRS25, LLLH26, Hea24, CHHK25, MORS25], less work targets the online communication needed to deliver input labels. To reduce this online communication, garbled-input compression[1] was initiated by [AIKW13], which showed that the evaluator's input labels can be transferred with query communication $|\mathbf{x}| + \mathsf{poly}(\lambda)$ under assumptions such as RSA, LWE, or DDH, but only at the cost of $\Omega(|\mathbf{x}|\lambda)$ one-time setup communication. Follow-up works [GS18, GOS18] obtained comparable query guarantees under weaker assumptions such as Factoring or CDH, though via non-black-box techniques that are prohibitively expensive in practice. [ABI+23] later achieved overall rate $1 + o(1)$ using stronger tools such as RSA, bilinear DDH, or iO with SSBH, but these constructions remain far from concretely efficient. The closest prior work to ours is TinyLabels [DLL25], which is the first concretely efficient Ring-LWE-based scheme in this line. Yet it suffers from the drawbacks listed in the staged CI-VOLE section above: an expensive one-time setup and leakage of the input vector to the sender. Table 2 compares the communication and computation costs.

[1]These schemes do not necessarily imply staged CI-VOLE, because garbled inputs are not always correlated: for example, the input label pairs for an input bit can be pseudorandom and independent, rather than related by XOR with a global key $\Delta$.

# 3 Preliminaries

## 3.1 Notation and Assumptions

We use 1-based indexing for vectors and matrices.

- $\lambda$ denotes the computational security parameter (e.g. 128).
- $\sigma$ denotes the statistical security parameter (e.g. 40).
- $\mathbb{S}$ and $\mathbb{R}$ are the sender and receiver, respectively.
- $\mathbb{P}$ and $\mathbb{V}$ are the prover and verifier, respectively.
- Recursion-tree levels are numbered from the root to the leaves. The root level is 1, and the leaf or bottom level is $\ell$.
- The base level is the same as the leaf level: it acts on the original input blocks and uses the global key $\Delta$. Internal levels $1, \ldots, \ell - 1$ use the retained key $\mathsf{sk_r}$.
- The root wrapper is an extra layer above level 1, not another recursion-tree level. The base case of $\mathsf{OnlineRec}$ is the stopping case at level 1.
- $x \leftarrow X$ indicates $x$ is sampled according to $X$.
- $x := y$ is a deterministic assignment of value $y$ to $x$.
- $\mathcal{R}_q$ is the ring $\mathbb{Z}_q[X]/(X^n + 1)$.
- $q$ is the ciphertext modulus of the LWE/LEnc ciphertexts.
- $q_i$ is the $i$-th CRT modulus such that $q = \prod_{i \in [\rho]} q_i$, and each $q_i = 1 \bmod n$ is NTT-friendly to $\mathcal{R}_q$.
- $\rho \approx \log q / \log p$ is the number of CRT moduli constituting $q$.
- $g$ is the gadget base.
- $\tau = \log_g q$ is the number of gadget digits for base $g$.
- $n$ denotes the polynomial degree of the rings $\mathcal{R}$ and $\mathcal{R}_q$.
- $\mathbf{x}$ denotes a scalar vector, and $x[i]$ is the $i$-th element of $\mathbf{x}$.
- $\mathbf{x} \odot \mathbf{y} = (x[1] \cdot y[1], x[2] \cdot y[2], \ldots)$ is element-wise multiplication.
- $\mathbf{x} \cdot y = (x[1] \cdot y, x[2] \cdot y, \ldots)$ denotes the broadcast multiplication of a vector $\mathbf{x}$ by a scalar $y$.
- $\mathbf{x} \cdot \mathbf{y}^T$ and $\mathbf{x}^T \cdot \mathbf{y}$ are the outer product and inner product, resp.
- $\langle x \rangle$ is a subtractive secret sharing of $x$. $\mathbb{S}$ and $\mathbb{R}$ hold shares $\langle x \rangle^{\mathbb{S}}$ and $\langle x \rangle^{\mathbb{R}}$, respectively, such that $\langle x \rangle^{\mathbb{R}} - \langle x \rangle^{\mathbb{S}} = x$.
- $(\mathbb{S}{:}x, \mathbb{R}{:}y) \leftarrow f(\mathbb{S}{:}a, \mathbb{R}{:}b)$ denotes that $\mathbb{S}$ and $\mathbb{R}$ input $a$ and $b$ to the function $f$ and receive outputs $x$ and $y$, respectively. Similar notation is defined for other parties, *e.g.*, $\mathbb{P}$ and $\mathbb{V}$.
- $(a_i)_{i \in [n]}$ denotes the concatenation $a_1 || \ldots || a_n$.
- $(a_{ij})_{i \in [n], j \in [m]}$ denotes $(a_{1j})_{j \in [m]} || \ldots || (a_{nj})_{j \in [m]}$.
- $[n]$ denotes the array $(1, 2, \ldots, n)$.
- $\triangleright$ `COMMENT` denotes a comment in an algorithm.
- $\mathbf{g} = (1, g, \ldots, g^{\tau - 1}) \in \mathbb{Z}^{\tau}$ is the gadget vector.
- $\mathbf{g}^{-1}(x)$ is the gadget decomposition of $x$. It maps an element $x \in \mathcal{R}_q$ to a vector $\mathbf{x} \in \mathcal{R}_q^{\tau}$ such that $\|\mathbf{x}\|_{\infty} < g$ and $\mathbf{x} \cdot \mathbf{g} = x$.
- $\mathbf{g}_{\mathsf{hi}} = (g, g^2, \ldots, g^{\tau - 1}) \in \mathbb{Z}^{\tau - 1}$ is the truncated gadget vector, and $\mathbf{g}_{\mathsf{hi}}^{-1}(x)$ denotes the corresponding truncated digit vector (dropping the least-significant base-$g$ digit); we write $\mathsf{low}_g(x)$ for the omitted digit and $\mathsf{high}_g(x) := x - \mathsf{low}_g(x)$ for the remaining high-digit part.
- $\tilde{O}(\cdot)$ denotes $O(\cdot)$ with $\mathsf{poly}(\log \lambda)$ factors skipped.
- $O_{\lambda}(\cdot)$ denotes $O(\cdot)$ with $\mathsf{poly}(\lambda)$ factors skipped.

**Circuits.** In this work, we consider arithmetic circuits. A circuit $\mathcal{C}$ is represented as a list of 2-input, 1-output gates $g = (a, b, c, T)$, where $a$ and $b$ are the identifiers of the input wires, $c$ is the identifier of the output wire, and $T \in \{\text{Add}, \text{Mult}\}$ is the gate type.

### 3.2 Detailed Description of Security Model

A static **semi-honest** adversary $\mathcal{A}$ corrupts one of the two parties participating in the 2PC protocol at the outset and attempts to infer as much information as possible from the protocol execution while honestly following its specifications.

In the context of CI-VOLE (Figure 1), security is defined in the real/ideal simulation paradigm in the random oracle model. The protocol uses the random oracle in the usual way to derive pseudorandom ciphertext components and Fiat–Shamir challenges. Separately, Section A analyzes the concrete laconic digest itself and isolates the smudging, hiding, and binding properties needed in the security proofs.

In the real interaction, the parties execute the protocol $\Pi$ in the presence of $\mathcal{A}$ and the environment $\mathcal{Z}$. We write $\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}$ to denote the binary distribution ensemble describing $\mathcal{Z}$'s output in the real interaction. In the ideal execution, the parties send their inputs to an ideal functionality $\mathcal{F}$. We denote the adversary in the ideal world by $\mathsf{Sim}$. We write $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ to denote the binary distribution ensemble describing $\mathcal{Z}$'s output in the ideal interaction. A protocol $\Pi$ securely realizes a functionality $\mathcal{F}$ if for every adversary $\mathcal{A}$ in the real interaction, there exists an adversary $\mathcal{S}$ in the ideal interaction, such that no environment $\mathcal{Z}$ can tell apart the real and ideal interactions, except with negligible probability.

A static **malicious** adversary $\mathcal{A}$ corrupts one of the two parties participating in the 2PC protocol at the outset and may deviate arbitrarily from the protocol in an attempt to learn information or force outputs beyond what is allowed by the ideal functionality. One challenge in the malicious proof is that the simulator is not handed the corrupted party's effective input; it must recover that input from the adversarial transcript before invoking the ideal functionality.

We defer malicious security to the dedicated sections. Section 10 states the malicious protocol-level changes, and Section 3.8 records the NIZK interface used there. Sections E.1 to E.3 set up the proof roadmap and transcript notation. Sections E.4, E.4.1 and 10.1.6 give the receiver-extraction interfaces used by the malicious proof. Sections E.5 and E.6 prove the corrupted-receiver and corrupted-sender cases, respectively, and Section E.7 combines them into the full malicious CI-VOLE theorem of Theorem 14.

### 3.3 Ring-LWE

We borrow the following Ring-LWE notation and auxiliary definitions from [DKL+23a, DLL25]. Let $\mathcal{R} = \mathbb{Z}[X]/\Phi_m(X)$ denote the cyclotomic ring used throughout, where $\Phi_m(X)$ is the $m$-th cyclotomic polynomial and has power-of-2 degree $n = \phi(m)$. For modulus $q \in \mathbb{N}$, let $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$.

**Definition 1** ($\mathsf{LWE}_{\mathcal{R},m,q,\chi}$ Assumption)**.** *For a ring $\mathcal{R}$, number of samples $m$, modulus $q$, and error distribution $\chi$ over $\mathcal{R}_q$ (all parameterized by $\lambda$), the $\mathsf{LWE}_{\mathcal{R},m,q,\chi}$ assumption states that the following two distributions are computationally indistinguishable:*

$$\left\{ (\boldsymbol{a},\boldsymbol{y}) \middle| \begin{array}{c} \boldsymbol{a} \xleftarrow{\$} \mathcal{R}_q^m \\ s \xleftarrow{\$} \mathcal{R}_q \\ \boldsymbol{e} \xleftarrow{\$} \chi^m \\ \boldsymbol{y} = \boldsymbol{a}\cdot s + \boldsymbol{e} \pmod q \end{array} \right\}_\lambda \approx_c^{2^\lambda} \left\{ (\boldsymbol{a},\boldsymbol{y}) \middle| \begin{array}{c} \boldsymbol{a} \xleftarrow{\$} \mathcal{R}_q^m \\ \boldsymbol{y} \xleftarrow{\$} \mathcal{R}_q^m \end{array} \right\}_\lambda .$$

Here, $\chi$ is the discrete Gaussian distribution described below.

**Definition 2** (SIS Assumption)**.** *For modulus $q$, width $m$, and norm bound $\beta$ (all parameterized by $\lambda$), the problem $\mathsf{SIS}_{q,m,\beta}$ is: given a uniformly random matrix $A \xleftarrow{\$} \mathcal{R}_q^{1\times m}$, find a nonzero vector*

$z = (z_1, \ldots, z_m) \in \mathcal{R}_q^m$ *such that*

$$Az = 0 \in \mathcal{R}_q \qquad \textit{and} \qquad \|z\|_2 \leq \beta.$$

*We say that* $\mathsf{SIS}_{q,m,\beta}$ *is hard if no PPT adversary can solve this problem with non-negligible probability. Throughout the paper, when we invoke SIS, we mean this rank-*1 *module form over* $\mathcal{R}_q$*.*

**g-ary gadget vector.** To constrain noise growth, at times it will be useful to decompose an element $a \in \mathcal{R}_q$ (of arbitrary norm) into several ring elements $a_1, \ldots, a_m$ of norm $\|a_i\|_\infty < g$, where $m = \lceil \log_g q \rceil$. By $\boldsymbol{g}^{-1}$ we denote such a decomposition (written as a column vector), and its inverse is given by the vector

$$\boldsymbol{g}^T = (1, g, g^2, \ldots, g^{m-1}) \in \mathcal{R}_q^m.$$

We also use the following auxiliary ring notation. Writing

$$\Phi_m(X) = \prod_{i \in \mathbb{Z}_m^*} (X - \omega_m^i) \in \mathbb{Z}[X],$$

we let $\omega_m$ denote an $m$-th root of unity in $\mathbb{C}$.

The coefficient embedding $\Phi : \mathcal{R} \to \mathbb{R}^n$ maps $\Phi(\sum_{i=0}^{n-1} a_i X^i) = (a_0, \ldots, a_{n-1})^T$. The infinity norm of an element $a \in \mathcal{R}$ with respect to its coefficient embedding is defined as $\|a\|_\infty = \|\Phi(a)\|_\infty = \max_{i=0,\ldots,d-1} |a_i|$ for $a = \sum_{i=0}^{d-1} a_i X^i$. The ring expansion factor of $\mathcal{R}$ is given by $\gamma_{\mathcal{R}} := \max_{a,b \in \mathcal{R}\setminus\{0\}} \frac{\|a \cdot b\|_\infty}{\|a\|_\infty \cdot \|b\|_\infty}$.

**Lemma 1.** *If the cyclotomic ring* $\mathcal{R}$ *has degree* $n = 2^k$ *(i.e., the degree is a power of 2), then the expansion factor is bounded by* $\gamma_{\mathcal{R}} \leq n$*.*

We denote $\mathcal{R}/q\mathcal{R}$ by $\mathcal{R}_q$, where modulus $q \in \mathbb{N}$. For an element $a \in \mathcal{R}_q$, i.e., $a = \sum_{i=0}^{d-1} a_i X^i$, each coefficient $a_i \in \mathbb{Z}_q$ is identified with the corresponding representative element in $[-\frac{q}{2}, \frac{q}{2}) \subseteq \mathbb{Z}$. $\lfloor \frac{a}{\Delta} \rceil$ denotes the element $b \in \mathcal{R}_q$, where the $i^{\texttt{th}}$ coefficient of $b$ ($b_i$) is equal to $\frac{a_i}{\Delta}$, rounded to the nearest integer.

**Gaussian Distribution.** Like [DKL+23a, DLL25], we use discrete Gaussians for error distributions. Recall that the continuous Gaussian distribution over $\mathbb{R}$ with parameter $s$ is defined by the density function $\rho_s(x) = e^{-\pi x^2/s^2}$. The discrete Gaussian distribution over $\mathbb{Z}$ with parameter $s$ is defined by the density function $\mathcal{D}_{\mathbb{Z},s}(x) = \frac{\rho_s(x)}{\sum_{x' \in \mathbb{Z}} \rho_s(x')}$. To sample an element of the ring $\mathcal{R}$ using the discrete Gaussian distribution, we first sample a vector $\boldsymbol{a} = (a_1, \ldots, a_n)$ in $\mathbb{R}^n$ using $\mathcal{D}_{\mathbb{Z},s}^n$. We then map the result to an element in $\mathcal{R}$ using the coefficient embedding, i.e., compute $a = \sum_{i=0}^{n-1} a_i X^i$. Gaussians may also be generalized to a covariance matrix $\Sigma \in \mathbb{R}^{n \times n}$ which allows individual coordinates to be correlated. For a distribution $\chi$, $\max \chi$ denotes the maximum norm $\|a\|$ of any value $a \leftarrow \chi$ sampled from this distribution.

**Error-leakage Ring-LWE.** This variant of Ring-LWE was introduced in [DKL+23a]. In addition to the Ring-LWE sample $(\boldsymbol{a}, \boldsymbol{y})$, the adversary receives leakage $\boldsymbol{l} = \boldsymbol{Z} \cdot \boldsymbol{e} + \overline{\boldsymbol{e}}$, where $\boldsymbol{e}$ is the error in $\boldsymbol{y} = \boldsymbol{a} \cdot s + \boldsymbol{e}$, $\overline{\boldsymbol{e}}$ is fresh hiding noise, and the leakage matrix $\boldsymbol{Z}$ is chosen from an allowed family $\mathcal{L} \subset \mathcal{R}^{k \times m}$.

**Definition 3** ($\mathsf{elLWE}_{\mathcal{R},q,\chi,\overline{\chi},\mathcal{L}}$ Assumption)**.** *For ring* $\mathcal{R}$*, modulus* $q$*, error distributions* $\chi, \overline{\chi}$ *over* $\mathcal{R}$*, and an efficiently decidable set* $\mathcal{L} \subset \mathcal{R}^{k \times m}$ *(all parameterized by* $\lambda$*),* $\mathsf{elLWE}_{\mathcal{R},q,\chi,\overline{\chi},\mathcal{L}}$ *holds if for*

*any probabilistically polynomial-time (PPT) adversary $\mathcal{A}$ with running time bounded by $t(\lambda)$, the following holds:*

$$\left|Pr[\mathsf{elLWE}^{\mathcal{A},0}_{\mathcal{R},q,\chi,\overline{\chi},\mathcal{L}} = 1] - Pr[\mathsf{elLWE}^{\mathcal{A},1}_{\mathcal{R},q,\chi,\overline{\chi},\mathcal{L}} = 1]\right| \leq \frac{poly(\lambda)}{2^\lambda} \cdot t(\lambda),$$

*where the experiment $\mathsf{elLWE}^{\mathcal{A},b}_{\mathcal{R},q,\chi,\overline{\chi},\mathcal{L}}$ is defined as follows.*

1. *The challenger samples $\boldsymbol{a} \xleftarrow{\$} \mathcal{R}_q^m$ and sends $\boldsymbol{a}$ to $\mathcal{A}$.*
2. *$\mathcal{A}$ outputs a matrix $\boldsymbol{Z} \in \mathcal{L}$. The challenger samples $\boldsymbol{e} \xleftarrow{\$} \chi^m$ and $\overline{\boldsymbol{e}} \xleftarrow{\$} \overline{\chi}^k$, and computes the leakage $\boldsymbol{l} = \boldsymbol{Z} \cdot \boldsymbol{e} + \overline{\boldsymbol{e}}$.*
3. *If $b = 0$, the challenger samples $s \xleftarrow{\$} \mathcal{R}$ and computes $\boldsymbol{y} \leftarrow \boldsymbol{a} \cdot s + \boldsymbol{e} \pmod q$. If $b = 1$, the challenger samples $\boldsymbol{y} \xleftarrow{\$} \mathcal{R}_q^m$.*
4. *The challenger sends $(\boldsymbol{y}, \boldsymbol{l})$ to $\mathcal{A}$.*
5. *The output of the experiment is the bit $b'$ output by $\mathcal{A}$.*

**Adaptive error-leakage Ring-LWE** (a-elLWE)**.** In a-elLWE, the adversary may choose each leakage matrix after seeing the Ring-LWE sample, and may do so adaptively up to $T$ times.

**Definition 4** ($\mathsf{a\text{-}elLWE}_{\mathcal{R},q,\chi,\overline{\chi},T,\mathcal{L}}$ Assumption)**.**
*We say that the adaptive assumption $\mathsf{a\text{-}elLWE}_{\mathcal{R},q,\chi,\overline{\chi},T,\mathcal{L}}$ holds, for an efficiently decidable set $\mathcal{L} \subset \mathcal{R}^{k\times m}$, if for any PPT adversary $\mathcal{A}$ with running time bounded by $t(\lambda)$, we have*

$$\left|Pr[\mathsf{a\text{-}elLWE}^{\mathcal{A},0}_{\mathcal{R},q,\chi,\overline{\chi},T,\mathcal{L}} = 1] - Pr[\mathsf{a\text{-}elLWE}^{\mathcal{A},1}_{\mathcal{R},q,\chi,\overline{\chi},T,\mathcal{L}} = 1]\right| \leq \frac{poly(\lambda)}{2^\lambda} \cdot t(\lambda),$$

*where the experiment $\mathsf{a\text{-}elLWE}^{\mathcal{A},b}_{\mathcal{R},q,\chi,\overline{\chi},T,\mathcal{L}}$ is defined as follows.*

1. *The challenger samples $\boldsymbol{a} \xleftarrow{\$} \mathcal{R}_q^m$ and $\boldsymbol{e} \xleftarrow{\$} \chi^m$, and sends $\boldsymbol{a}$ to $\mathcal{A}$.*
2. *If $b = 0$, the challenger samples $s \xleftarrow{\$} \mathcal{R}$ and computes $\boldsymbol{y} \leftarrow \boldsymbol{a} \cdot s + \boldsymbol{e} \pmod q$. If $b = 1$, the challenger samples $\boldsymbol{y} \xleftarrow{\$} \mathcal{R}_q^m$.*
3. *The challenger sends $(\boldsymbol{a}, \boldsymbol{y})$ to $\mathcal{A}$.*
4. *For each $t \in [T]$:*
   - *$\mathcal{A}$ outputs a leakage matrix $\boldsymbol{Z}^{(t)} \in \mathcal{L}$.*
   - *The challenger samples $\overline{\boldsymbol{e}}^{(t)} \xleftarrow{\$} \overline{\chi}^k$ and sends the leakage $\boldsymbol{l}^{(t)} = \boldsymbol{Z}^{(t)} \cdot \boldsymbol{e} + \overline{\boldsymbol{e}}^{(t)}$ to $\mathcal{A}$.*
5. *The output of the experiment is the bit $b'$ output by $\mathcal{A}$.*

The adaptive version follows directly from standard Ring-LWE when the hiding noise statistically masks $\boldsymbol{Z} \cdot \boldsymbol{e}$ via noise flooding, at the cost of increasing the modulus bitlength by $\lambda$.

The work of [DLL25] uses truncated discrete Gaussians to ensure perfect correctness in its constructions. The truncated distribution $\overline{\mathcal{D}}_{\mathcal{R},s}$ is identical to $\mathcal{D}_{\mathcal{R},s}$ except that it rejects the sample if $\|a\|_\infty > \sqrt{\lambda} \cdot s$, and instead continues sampling until it finds an $a \in \mathcal{R}$ with $\|a\|_\infty \leq \sqrt{\lambda} \cdot s$. Observe that $\max \overline{\mathcal{D}}_{\mathcal{R},s} = \sqrt{\lambda} \cdot s$. As our construction builds on the construction of [DLL25], we also consider truncated discrete Gaussians. We recall the following lemma, which shows that discrete Gaussians and truncated discrete Gaussians are statistically indistinguishable.

**Lemma 2** (Truncated discrete Gaussians are statistically close)**.** *For a ring $\mathcal{R}$ with degree $n = poly(\lambda)$, and for any parameters, the following two distributions are $2^{-\lambda}$-statistically close: $\mathcal{D}_{\mathcal{R},s} \approx_s^{2^{-\lambda}} \overline{\mathcal{D}}_{\mathcal{R},s}$.*

Below, we recall the hardness result of elLWE under the standard assumption of Ring LWE [DKL+23a, DLL25].

**Definition 5** (Maximal Singular Value)**.** *For a ring $\mathcal{R}$ of degree $n$ and a vector $\boldsymbol{x} = (x_1, \ldots, x_\ell) \in \mathcal{R}^\ell$, $\Phi(\boldsymbol{x}) \in \mathbb{R}^{\ell n}$ denotes the vector $(\Phi(x_1), \ldots, \Phi(x_\ell))$, where $\Phi : \mathcal{R} \to \mathbb{R}^n$ is an embedding of $\mathcal{R}$ in $\mathbb{R}^n$.*

*For a matrix $\boldsymbol{Z} \in \mathcal{R}^{k \times m}$, the linear transformation $\Theta_{\boldsymbol{Z}}(\boldsymbol{x}) : \mathbb{R}^{mn} \to \mathbb{R}^{kn}$ (with corresponding matrix $\boldsymbol{A}_{\boldsymbol{Z}}$) is defined as:*

$$\Theta_{\boldsymbol{Z}}(\boldsymbol{x}) = \Phi(\boldsymbol{Z} \cdot \Phi^{-1}(\boldsymbol{x})) = \boldsymbol{A}_{\boldsymbol{Z}} \cdot \boldsymbol{x}.$$

*For a set $\mathcal{L} \subset \mathcal{R}^{k \times m}$, the maximal singular value $\sigma_{max}(\mathcal{L})$ is defined as:*

$$\sigma_{max}(\mathcal{L}) = \max_{\boldsymbol{Z} \in \mathcal{L}} \sigma_{max}(\boldsymbol{A}_{\boldsymbol{Z}}).$$

**Theorem 1.** *Let $\epsilon > 0$ be negligible. Let $\mathcal{R}$ be a suitable ring with an embedding as a lattice in $\mathbb{R}^n$, and let $\boldsymbol{\Sigma}_0$ be a covariance matrix with $\sqrt{\boldsymbol{\Sigma}_0} \geq \eta_\epsilon(\mathcal{R})$. Let $\mathcal{L} \subset \mathcal{R}^{k \times m}$ be an efficiently decidable set. Let $s, t \geq 2\sqrt{2}$ be such that*

$$t^2 \sigma_{min}(\boldsymbol{\Sigma}_0) \geq \frac{(s^2+1)(s^2+2)}{s^2} \sigma_{max}(\boldsymbol{\Sigma}_0) \cdot (\sigma_{max}(\mathcal{L}))^2.$$

*Now let $\chi = \mathcal{D}_{\mathcal{R}, t\sqrt{(s^2+1)\boldsymbol{\Sigma}_0}}$, $\overline{\chi} = \mathcal{D}_{\mathcal{R}, t\sqrt{(t^2+1)\boldsymbol{\Sigma}_0}}$, and $\chi^* = \mathcal{D}_{\mathcal{R}, s/2 \cdot \sqrt{\boldsymbol{\Sigma}_0}}$. Then, assuming that $\mathsf{LWE}_{\mathcal{R},m,q,\chi^*}$ is hard, $\mathsf{elLWE}_{\mathcal{R},q,\chi,\overline{\chi},\mathcal{L}}$ is also hard. More precisely, if there exists an adversary $\mathcal{A}$ with advantage $\delta$ against $\mathsf{elLWE}_{\mathcal{R},q,\chi,\overline{\chi},\mathcal{L}}$, then there exists an adversary $\mathcal{A}A'$ with roughly the same running time and advantage $\delta - \epsilon$ against $\mathsf{LWE}_{\mathcal{R},m,q,\chi^*}$.*

We consider the same set of leakage matrices as in [DLL25]. This set $\mathcal{L}_{k,m}(\beta) \subset \mathcal{R}^{k \times m}$ consists of leakage matrices whose ring elements all have infinity norm bounded by $\beta$ under the coefficient embedding. We recall the following lemma on the bound for the maximal singular values required in Theorem 1.

**Lemma 3.** *Let $\mathcal{R}$ be a cyclotomic ring of degree $n$ that is a power of 2, and let $\boldsymbol{Z} \in \mathcal{R}^{k \times m}$ be a matrix such that each coefficient of $\boldsymbol{Z}$ has $\|\cdot\|_\infty$-norm bounded by $\beta$ in the coefficient embedding. Then the maximal singular value of the matrix $\boldsymbol{A}_{\boldsymbol{Z}} \in \mathbb{R}^{kn \times mn}$ is bounded by $\sigma_{max}(\boldsymbol{A}_{\boldsymbol{Z}}) \leq \beta n \sqrt{km}$. This implies*

$$\sigma_{max}(\mathcal{L}_{k,m}(\beta)) \leq \beta n \sqrt{km}.$$

We also consider the sparse version of leakage matrices described below. Let $\mathcal{L}^{\times w}_{k,m}(\beta) \subset \mathcal{R}^{wk \times wm}$ be the set of matrices consisting of $w$ separate blocks, each with infinity norm bounded by $\beta$, of size $k \times m$ on the diagonal, i.e.,

$$\mathcal{L}^{\times w}_{k,m}(\beta) = \{diag(\boldsymbol{Z}_1, \ldots, \boldsymbol{Z}_w) | \boldsymbol{Z}_i \in \mathcal{L}_{k,m}(\beta)\}.$$

The following lemma from [DLL25] shows that the bound for the maximal singular values for the sparse version of leakage matrices is the same as the bound of Lemma 3.

**Lemma 4.** *Let $\mathcal{R}$ be a cyclotomic ring of degree $n$ that is a power of 2. Then the maximal singular value of any matrix in $\mathcal{L}^{\times w}_{k,m}(\beta)$ is bounded by*

$$\sigma_{max}(\mathcal{L}^{\times w}_{k,m}(\beta)) \leq \beta n \sqrt{km}.$$

### 3.4 Linear Laconic Encryption

We now record the full formal interface of Linear Laconic Encryption.

**Definition 6** (Linear Laconic Encryption)**.** *A* $\mathsf{LacE}$ *scheme with associated message space* $\mathcal{R}_q(\lambda)$ *and error bound* $B_{\mathsf{LacE}}(\lambda)$ *comprises the following efficient algorithms:*

- $\mathsf{pp} \xleftarrow{\$} \mathsf{LacE.Setup}(1^\lambda, w)$*: The algorithm takes* $\lambda$ *and input vector dimension* $w$ *as input, and outputs public parameters* $\mathsf{pp}$*. All the remaining algorithms of* $\mathsf{LacE}$ *implicitly take* $\mathsf{pp}$ *as input.*
- $(\boldsymbol{r}, \mathsf{ct}) \xleftarrow{\$} \mathsf{LacE.Enc}(\boldsymbol{s})$*: On input vector* $\boldsymbol{s} \in \mathcal{R}_q^w$*, the algorithm outputs randomness vector* $\boldsymbol{r} \in \mathcal{R}_q^w$ *and ciphertext* $\mathsf{ct}$*.*
- $d_{\boldsymbol{a}} \leftarrow \mathsf{LacE.Digest}(\boldsymbol{a})$*: This deterministic algorithm takes vector* $\boldsymbol{a} \in \mathcal{R}_q^w$ *as input, and outputs a digest* $d_{\boldsymbol{a}} \in \mathcal{R}_q$*.*
- $\mathsf{LacE.Eval}(\mathsf{ct}, \boldsymbol{a})$*: On input* $\mathsf{ct}$ *and vector* $\boldsymbol{a} \in \mathcal{R}_q^w$*, it outputs* $\boldsymbol{\delta} \in \mathcal{R}_q^w$*.*

*A scheme* $\mathsf{LacE}$ *is a Linear Laconic Encryption scheme if it satisfies the following properties:*

- **Correctness (Informal).** *Following the notation above,* $|\boldsymbol{\delta} - (\boldsymbol{r} \cdot d_{\boldsymbol{a}} - \boldsymbol{s} \odot \boldsymbol{a})| < B_{\mathsf{LacE}}$ *for an error bound* $B_{\mathsf{LacE}}$ *with probability 1. The full formal statement appears in Appendix 3.6.1.*
- **Security (Informal).** *For any sequence of query vectors* $\boldsymbol{a}^{(1)}, \ldots, \boldsymbol{a}^{(T)}$*, the joint distribution of the ciphertext and the corresponding noisy leakage terms can be simulated from the public parameters and the queries alone, up to the prescribed noise-hiding distribution. The full simulation-based definition appears in Appendix 3.6.2.*

#### 3.4.1 LacE from Ring-LWE

Lemma 5 summarizes the results of [DLL25] on Linear Laconic Encryption.

**Lemma 5** ($\mathsf{LacE}$ from Ring-$\mathsf{LWE}$ [DLL25, Lemma 5])**.** *Assume* $\mathsf{LWE}_{\mathcal{R},O(1),q,\mathcal{D}_{\mathcal{R},q^{0.1}}}$ *holds (for a cyclotomic ring* $\mathcal{R}$ *of degree* $n \geq \lambda$ *that is a power of* 2*, modulus* $q \geq n^{15}$*, and modulus-to-noise ratio* $q^{0.9}$*). Then, there exists an* $\mathsf{LacE}$ *scheme over message space* $\mathcal{R}_q$ *with error bound* $B_{\mathsf{LacE}} \leq \sqrt{q}$ *that is* $2^\lambda$*-simulation secure under* $\sqrt{q}$*-noise leakage with respect to a noise distribution* $\bar{\chi}_{\mathsf{LacE}}$ *whose values are bounded by max* $\bar{\chi}_{\mathsf{LacE}} = 1000\sqrt{q}$*.*

*For encrypting message vectors of length* $w$*, the sizes of the public parameters* $\mathsf{pp}$ *and ciphertext* $\mathsf{ct}$ *are* $O(1)$ *and* $O(w \log w)$ *ring elements, respectively. The runtimes (in terms of ring multiplications) of the* $\mathsf{Enc}$*,* $\mathsf{Digest}$*, and* $\mathsf{Eval}$ *algorithms are* $O(w \log w)$*,* $O(w)$*, and* $O(w \log w)$*, respectively.*

### 3.5 Linearly Homomorphic Encryption

We now record the full formal interface of Linearly Homomorphic Encryption. The reason the interface has two encryption algorithms is that the target decryption value is an *affine* expression

$$\boldsymbol{m}_1 \cdot y + \boldsymbol{m}_2.$$

Accordingly, $\mathsf{LHE.Enc}_1$ encrypts the coefficient vector $\boldsymbol{m}_1$ that will later be scaled by the public query $y$, while $\mathsf{LHE.Enc}_2$ encrypts the offset vector $\boldsymbol{m}_2$ that is carried through additively. The role of $\mathsf{LHE.KeyGen}$ is then to combine the two hidden states with the chosen $y$ so that decryption returns this affine combination and not the two terms separately.

**Definition 7** (Noisy Linearly Homomorphic Encryption over Rings ($\mathsf{LHE}$))**.** *An* $\mathsf{LHE}$ *scheme over* $\mathcal{R}_q(\lambda)$ *with error bound* $B_{\mathsf{LHE}}(\lambda)$ *comprises the following algorithms:*

- $\mathsf{pp} \xleftarrow{\$} \mathsf{LHE.Setup}(1^\lambda, 1^w)$*: The algorithm takes* $\lambda$ *and input vector dimension* $w$ *as input, and outputs public parameters* $\mathsf{pp}$*. All the remaining algorithms of* $\mathsf{LHE}$ *implicitly take* $\mathsf{pp}$ *as input.*

- $(\mathsf{ct}_1, \mathsf{st}_1) \xleftarrow{\$} \mathsf{LHE.Enc}_1(\boldsymbol{m}_1)$*: On input vector* $\boldsymbol{m}_1 \in \mathcal{R}_q^w$*, the algorithm outputs ciphertext* $\mathsf{ct}_1$ *and a state* $\mathsf{st}_1$ *for the term later multiplied by* $y$*.*
- $(\mathsf{ct}_2, \mathsf{st}_2) \xleftarrow{\$} \mathsf{LHE.Enc}_2(\boldsymbol{m}_2)$*: On input vector* $\boldsymbol{m}_2 \in \mathcal{R}_q^w$*, the algorithm outputs ciphertext* $\mathsf{ct}_2$ *and a state* $\mathsf{st}_2$ *for the additive offset term.*
- $\mathsf{sk}_y \leftarrow \mathsf{LHE.KeyGen}(\mathsf{st}_1, \mathsf{st}_2, y)$*: On input the two states* $st_1$ *and* $st_2$ *output by* $Enc_1$ *and* $Enc_2$*, respectively, and a ring element* $y \in \mathcal{R}_q$*, the algorithm outputs a decryption key* $sk_y$*.*
- $\boldsymbol{m}_{\mathsf{res}} \leftarrow \mathsf{LHE.Dec}(\mathsf{sk}_y, \mathsf{ct}_1, \mathsf{ct}_2, y)$*: The algorithm takes decryption key* $\mathsf{sk}_y$*, the two ciphertexts* $\mathsf{ct}_1, \mathsf{ct}_2$*, and ring element* $y$*. It outputs a decrypted message* $\boldsymbol{m}_{\mathsf{res}} \in \mathcal{R}_q^w$*.*

*A scheme* LHE *is a Linearly Homomorphic Encryption scheme if it satisfies the following properties:*

- **Correctness (Informal).** *Following the notation above,* $|\mathbf{m}_{\mathsf{res}} - (\boldsymbol{m}_1 \cdot y + \boldsymbol{m}_2)| < B_{\mathsf{LHE}}$ *for an error bound* $B_{\mathsf{LHE}}$ *with overwhelming probability. The full formal statement appears in Appendix 3.6.3.*
- **Security (Informal).** *The receiver's view consisting of ciphertexts and functional decryption keys can be simulated from the queried values* $\{y^{(t)}\}_{t\in[T]}$ *and the corresponding outputs* $\{\boldsymbol{m}_1 \cdot y^{(t)} + \boldsymbol{m}_2^{(t)}\}_{t\in[T]}$ *alone. The full simulation-based definition appears in Appendix 3.6.4.*

### 3.5.1 LHE from Ring-LWE

Lemma 6 summarizes the results of [DLL25] on linearly homomorphic encryption over rings.

**Lemma 6** (LHE from Ring-LWE[DLL25, Lemma 6])**.** *Assume* $\mathsf{LWE}_{\mathcal{R},w,q,\mathcal{D}_{\mathcal{R},q^{0.1}}}$ *holds (for a cyclotomic ring* $\mathcal{R}$ *of degree* $n \geq \lambda$ *that is a power of* 2*, modulus* $q \geq n^{15}$*, and modulus-to-noise ratio* $q^{0.9}$*). Then, there exists an* LHE *scheme over message space* $\mathcal{R}_q$ *with error bound* $B_{\mathsf{LHE}} \leq 10\sqrt{q}$ *that is* $\sqrt{q}$*-times* $2^\lambda$*-simulation secure.*

*For encrypting message vectors of length* $w$*, the sizes of the public parameters* pp*, ciphertexts* $\mathsf{ct}_1$ *and* $\mathsf{ct}_2$*, and secret key* sk *are* $O(w)$*,* $O(w)$*,* $O(w)$ *and* $O(1)$ *ring elements, respectively. The runtimes (in terms of ring multiplications) of the* $\mathsf{Enc}_1$*,* $\mathsf{Enc}_2$*,* KeyGen *and* Dec *algorithms are* $O(w)$*,* $O(w)$*,* $O(1)$ *and* $O(w)$*, respectively.*

## 3.6 Detailed Correctness and Security Definitions for LacE and LHE

The main text keeps only compact API summaries for these primitives. We record the precise correctness statements and simulation-based security definitions here.

### 3.6.1 LacE Correctness

A LacE scheme from Definition 6 is correct if for all $\lambda \in \mathbb{N}$, $w \leq 2^\lambda$, and vectors $\boldsymbol{s}, \boldsymbol{a} \in \mathcal{R}_q^w$ with $d_{\boldsymbol{a}} = \mathsf{LacE.Digest}(\boldsymbol{a})$:

$$\Pr\left[\begin{array}{r} |\boldsymbol{\delta} - (\boldsymbol{r} \cdot d_{\boldsymbol{a}} - \boldsymbol{s} \odot \boldsymbol{a})| \leq B_{\mathsf{LacE}}, \\ \mathsf{pp} \leftarrow \mathsf{LacE.Setup}(1^\lambda, w), \\ \boldsymbol{r}, \mathsf{ct} \leftarrow \mathsf{LacE.Enc}(\boldsymbol{s}), \\ \boldsymbol{\delta} \leftarrow \mathsf{LacE.Eval}(\mathsf{ct}, \boldsymbol{a}) \end{array}\right] = 1.$$

### 3.6.2 LacE Security

A LacE scheme from Definition 6 is *adaptively* $2^\lambda$-simulation secure under $T(\lambda)$-noise leakage with respect to the noise-hiding distribution $\overline{\chi}_{\mathsf{LacE}}$ if there exist efficient simulators $\mathsf{LacE.Sim}_1, \mathsf{LacE.Sim}_2$

such that, for every efficient adaptive adversary $\mathcal{A}$, the two experiments $\mathsf{Real}_{\mathsf{LacE}}$ and $\mathsf{Sim}_{\mathsf{LacE}}$ are $2^\lambda$-computationally indistinguishable.

$\mathsf{Real}_{\mathsf{LacE}}$:
1. $\mathcal{A}(\lambda)$ decides width $w$ and receives $\mathsf{pp} \leftarrow \mathsf{LacE.Setup}(1^\lambda, w)$.
2. $\mathcal{A}$ decides a message vector $\mathbf{s} \in \mathcal{R}_q^w$ and receives $\mathsf{ct}$, where $(\boldsymbol{r}, \mathsf{ct}) \leftarrow \mathsf{LacE.Enc}(\boldsymbol{s})$.
3. For $t = 1, \ldots, T$,
   - $\mathcal{A}$ outputs $\boldsymbol{a}^{(t)} \in \mathcal{R}_q^w$ and receives a leakage $\boldsymbol{e}^{(t)} + \overline{\boldsymbol{e}}^{(t)}$, where

$$\begin{aligned} d_{\boldsymbol{a}}^{(t)} &\leftarrow \mathsf{LacE.Digest}(\boldsymbol{a}^{(t)}), \\ \boldsymbol{\delta}^{(t)} &\leftarrow \mathsf{LacE.Eval}(\mathsf{ct}, \boldsymbol{a}^{(t)}), \\ \boldsymbol{e}^{(t)} &:= \boldsymbol{\delta}^{(t)} - (\boldsymbol{r} \cdot d_{\boldsymbol{a}}^{(t)} - \boldsymbol{s} \odot \boldsymbol{a}^{(t)}), \\ \overline{\boldsymbol{e}}^{(t)} &\xleftarrow{\$} \overline{\chi}_{\mathsf{LacE}}^w, \end{aligned}$$

$\mathsf{Sim}_{\mathsf{LacE}}$:
1. $\mathcal{A}(\lambda)$ decides width $w$ and receives $\mathsf{pp} \leftarrow \mathsf{LacE.Setup}(1^\lambda, w)$.
2. $\mathcal{A}$ decides a message vector $\mathbf{s} \in \mathcal{R}_q^w$ and receives $\mathsf{ct}$, where $(\mathsf{ct}, \mathsf{st}) \leftarrow \mathsf{LacE.Sim}_1(\mathsf{pp})$.
3. For $t = 1, \ldots, T$,
   - $\mathcal{A}$ outputs $\boldsymbol{a}^{(t)} \in \mathcal{R}_q^w$ and receives a leakage $\widetilde{\boldsymbol{e}}^{(t)} + \overline{\boldsymbol{e}}^{(t)}$, where

$$\widetilde{\boldsymbol{e}}^{(t)} := \mathsf{LacE.Sim}_2(\mathsf{st}, \boldsymbol{a}^{(t)}), \qquad \overline{\boldsymbol{e}}^{(t)} \xleftarrow{\$} \overline{\chi}_{\mathsf{LacE}}^w,$$

### 3.6.3 LHE Correctness

An LHE scheme from Definition 7 is correct if for all $\lambda, w \in \mathbb{N}$, messages $\boldsymbol{m}_1$, $\boldsymbol{m}_2 \in \mathcal{R}_q^w$, and $y \in \mathcal{R}_q$:

$$\Pr\left[\begin{array}{l|r} \|\boldsymbol{m}_{\mathsf{res}} - (\boldsymbol{m}_1 \cdot y + \boldsymbol{m}_2)\| \leq B_{\mathsf{LHE}} & \\ & \mathsf{pp} \leftarrow \mathsf{LHE.Setup}(1^\lambda, 1^w), \\ & (\mathsf{ct}_1, \mathsf{st}_1) \leftarrow \mathsf{LHE.Enc}_1(\boldsymbol{m}_1), \\ & (\mathsf{ct}_2, \mathsf{st}_2) \leftarrow \mathsf{LHE.Enc}_2(\boldsymbol{m}_2), \\ & \mathsf{sk}_y \leftarrow \mathsf{LHE.KeyGen}(\mathsf{st}_1, \mathsf{st}_2, y), \\ & \boldsymbol{m}_{\mathsf{res}} \leftarrow \mathsf{LHE.Dec}(\mathsf{sk}_y, \mathsf{ct}_1, \mathsf{ct}_2, y) \end{array}\right] = 1.$$

### 3.6.4 LHE Security

An LHE scheme from Definition 7 is *adaptively* $T(\lambda)$-time $2^\lambda$-simulation secure if there exist efficient simulators $\mathsf{LHE.Sim}_1$, $\mathsf{LHE.Sim}_2$ and $\mathsf{LHE.Sim}_3$ such that, for every efficient adaptive adversary $\mathcal{A}$, the two experiments $\mathsf{Real}_{\mathsf{LHE}}$ and $\mathsf{Sim}_{\mathsf{LHE}}$ are $2^\lambda$-computationally indistinguishable.

$\mathsf{Real}_{\mathsf{LHE}}$:
1. $\mathcal{A}(\lambda)$ decides width $w$ and receives $\mathsf{pp} \leftarrow \mathsf{LHE.Setup}(1^\lambda, 1^w)$.
2. $\mathcal{A}$ decides a message vector $\boldsymbol{m}_1 \in \mathcal{R}_q^w$ and receives $\mathsf{ct}_1$, where $(\mathsf{ct}_1, \mathsf{st}_1) \leftarrow \mathsf{LHE.Enc}_1(\boldsymbol{m}_1)$.
3. For $t = 1, \ldots, T$,
   - $\mathcal{A}$ outputs $\boldsymbol{m}_2^{(t)} \in \mathcal{R}_q^w$ and receives $\mathsf{ct}_2^{(t)}$, where

$$(\mathsf{ct}_2^{(t)}, \mathsf{st}_2^{(t)}) \leftarrow \mathsf{LHE.Enc}_2(\boldsymbol{m}_2^{(t)}),$$

- $\mathcal{A}$ outputs $y^{(t)} \in \mathcal{R}_q$ and receives $\mathsf{sk}_{y^{(t)}}^{(t)}$

$$\mathsf{sk}_{y^{(t)}}^{(t)} \leftarrow \mathsf{LHE.KeyGen}(\mathsf{st}_1, \mathsf{st}_2^{(t)}, y^{(t)}).$$

$\mathsf{Sim}_{\mathsf{LHE}}$:
1. $\mathcal{A}(\lambda)$ decides width $w$ and receives $\mathsf{pp} \leftarrow \mathsf{LHE.Setup}(1^\lambda, 1^w)$.
2. $\mathcal{A}$ decides a message vector $\boldsymbol{m}_1 \in \mathcal{R}_q^w$ and receives $\widetilde{\mathsf{ct}}_1$, where $(\widetilde{\mathsf{ct}}_1, \mathsf{st}_1) \leftarrow \mathsf{LHE.Sim}_1(\mathsf{pp})$.
3. For $t = 1, \ldots, T$,
   - $\mathcal{A}$ outputs $\boldsymbol{m}_2^{(t)} \in \mathcal{R}_q^w$ and receives $\widetilde{\mathsf{ct}}_2^{(t)}$, where

$$(\widetilde{\mathsf{ct}}_2^{(t)}, \mathsf{st}_2^{(t)}) := \mathsf{LHE.Sim}_2(\mathsf{st}_1).$$

   - $\mathcal{A}$ outputs $y^{(t)} \in \mathcal{R}_q$ and receives $\widetilde{\mathsf{sk}}_{y^{(t)}}^{(t)}$,

$$\boldsymbol{m}_{\mathsf{res}}^{(t)} := \boldsymbol{m}_1 \cdot y^{(t)} + \boldsymbol{m}_2^{(t)}, \qquad \widetilde{\mathsf{sk}}_{y^{(t)}}^{(t)} := \mathsf{LHE.Sim}_3(\mathsf{st}_2^{(t)}, \boldsymbol{m}_{\mathsf{res}}^{(t)}, y^{(t)}).$$

### 3.7 Zero-Knowledge Arguments

We recall the standard definition of zero-knowledge arguments from the literature [GMW87, Kil92, Gol01, IKOS07] below. An NP relation $R(x, w)$ is an efficiently decidable binary relation which is polynomially bounded, meaning that there exists a polynomial $p(\cdot)$ such that $R(x, w)$ is satisfied then $|w| \leq p(|x|)$. We view $R(x, \cdot)$ as an arithmetic circuit $C_x$ comprising addition and multiplication gates. For $x$ in the NP-language, $C_x$ on a valid witness outputs 0, i.e., $C_x(w) = 0$ indicates $R(x, w) = 1$.

A zero-knowledge argument system for an NP relation $R(x, w)$ is defined by two probabilistic polynomial-time (PPT) algorithms, a prover $\mathbb{P}$ and a verifier $\mathbb{V}$. Both $\mathbb{P}$ and $\mathbb{V}$ hold the common input statement $x$, while $\mathbb{P}$ additionally holds a witness $w$.

Both $\mathbb{P}$ and $\mathbb{V}$ interact using the next-message function defined by their respective protocols. The next-message function takes the inputs, the messages received so far from the other party, and the randomness, and outputs the next message to be sent to the other party.

**Definition 8** (Zero-Knowledge Arguments)**.**
*The protocol $(\mathbb{P}(x, w), \mathbb{V}(x))$ is a zero-knowledge argument for an NP relation $R$ if it satisfies the following:*

- ***Perfect Completeness.*** *If a statement is true, an honest prover $\mathbb{P}$ can always convince an honest verifier $\mathbb{V}$. For any pair $(x, w)$ such that $R(x, w) = 1$ (meaning $x \in L$), $\mathbb{V}$ accepts the proof.*

$$\Pr\left[(\mathbb{P}(x, w), \mathbb{V}(x)) = \mathit{accept}\right] = 1.$$

- ***Computational Soundness.*** *A malicious PPT prover $\mathbb{P}^*$ cannot convince an honest verifier $\mathbb{V}$ of a false statement except with negligible probability. Formally, for every PPT corrupt prover $\mathbb{P}^*$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for any $x \notin L$ (i.e. $\nexists w$ such that $R(x, w) = 1$),*

$$\Pr\left[(\mathbb{P}^*, \mathbb{V})(x) = \mathit{accept}\right] \leq \mathsf{negl}(\lambda).$$

- ***Zero-Knowledge.*** *A malicious verifier $\mathbb{V}^*$ learns nothing from the interaction other than the fact that the statement is true. Formally, for every PPT corrupt verifier $\mathbb{V}^*$, there exists a PPT simulator $\mathsf{Sim}_{\mathbb{V}^*}$ that takes as input a statement $x \in L$ and outputs a transcript that is computationally indistinguishable from the view of $\mathbb{V}^*$ in a real interaction between $\mathbb{P}$ and $\mathbb{V}$. That is, for all pairs $(x, w)$ such that $R(x, w) = 1$,*

$$\{view_{\mathbb{V}^*}(\mathbb{P}(x, w), \mathbb{V}^*(x))\} \approx_c \{\mathsf{Sim}_{\mathbb{V}^*}(x)\},$$

*where $view_{\mathbb{V}^*}$ denotes the verifier's complete view of the interaction (all received messages and its internal random coin tosses) and the symbol $\approx_c$ denotes computational indistinguishability between the two probability ensembles.*

**Remark.** A protocol $(\mathbb{P}(x, w), \mathbb{V}(x))$ is an *honest-verifier* zero-knowledge argument system for an NP relation $R$ if it satisfies perfect completeness and the computational soundness property as defined in Definition 8. However, zero-knowledge simulability is required only with respect to an honest verifier. Formally, there exists a PPT simulator $\mathsf{Sim}$ such that for all pairs $(x, w)$ such that $R(x, w) = 1$,

$$\{\text{view}_V(\mathbb{P}(x, w), \mathbb{V}(x))\} \approx_c \{\mathsf{Sim}(x)\}.$$

### 3.8 Non-Interactive Arguments of Knowledge

In the malicious protocol we use Fiat–Shamir non-interactive arguments of knowledge for bounded linear relations over $\mathcal{R}_q$, instantiated by the lattice proof systems of Bootle–Lyubashevsky–Seiler and Lyubashevsky–Nguyen–Plançon [BLS19, LNP22]. We use the following standard random-oracle interface.

Let $R(x, w)$ be an efficiently decidable relation. A Fiat–Shamir NIZK argument of knowledge for $R$ consists of PPT algorithms

$$\mathsf{ZK.Setup}, \qquad \mathsf{ZK.Prove}^H, \qquad \mathsf{ZK.Verify}^H,$$

together with a simulator $\mathsf{ZK.Sim}^H$ and an extractor $\mathsf{ZK.Ext}^H$. Setup outputs public proof parameters $\mathsf{pp_{zk}}$. The prover, given $(x, w)$ with $R(x, w) = 1$, outputs $\pi \leftarrow \mathsf{ZK.Prove}^H(\mathsf{pp_{zk}}, x, w)$. The verifier outputs a bit $\mathsf{ZK.Verify}^H(\mathsf{pp_{zk}}, x, \pi)$. We require the following properties, for polynomially many statements in one random-oracle execution.

- **Completeness.** For every $(x, w)$ with $R(x, w) = 1$, an honestly generated proof verifies except with negligible probability.
- **Zero knowledge / simulated proofs.** There is a PPT simulator $\mathsf{ZK.Sim}^H$, with the usual ability to program the Fiat–Shamir random-oracle challenge points, that on input a public statement $x$ outputs an accepting simulated proof. For true statements, simulated proofs are computationally indistinguishable from honest proofs to any malicious verifier. We will also invoke this simulator on public statements generated by other simulators; in that use, we only rely on acceptance and on the fact that the simulated proof is produced without a witness.
- **Knowledge soundness.** For every PPT prover $P^\star$, if $P^\star$ produces an accepting proof for $x$ with non-negligible probability, then the extractor $\mathsf{ZK.Ext}^H$, with rewindable access to $P^\star$ and the same public parameters, outputs $w$ such that $R(x, w) = 1$, except with negligible extraction error.

For the setup proofs in Section 10.1.3, $R$ is the bounded exact modular linear relation certified by the corresponding public matrix, target, and coefficient bounds.

# 4 High-Level Technical Overview

The goal of LogVOLE is to efficiently distribute CI-VOLE correlated randomness. Our starting point is TinyLabels [DLL25], a specialized protocol for GC input-label delivery with succinct communication. We provide an overview of TinyLabels in Section 4.1. Then, in Section 4.2, we use it as a building block to construct LogVOLE, our polylogarithmic-communication CI-VOLE scheme. This overview is mostly for our *staged* construction, except that in Section 4.4, we present the non-interactive variant. Finally, we describe its applications in detail in Section 4.6.

## 4.1 Revisiting TinyLabels

We use TinyLabels as the conceptual starting point for our construction. For our purposes, however, it is more convenient to present it in a simplified form.

**What we want the reader to keep in mind.** The mental model is the following. $\mathbb{S}$ holds a global secret $\Delta \in \mathcal{M}$ and wants $\mathbb{R}$ to obtain a vector

$$\mathbf{m} = \Delta \cdot \mathbf{x} + \mathbf{k} \in \mathcal{M}^w,$$

where $\mathbf{x} \in \mathcal{M}^w$ is chosen by $\mathbb{R}$ and $\mathbf{k} \in \mathcal{M}^w$ is a mask known to $\mathbb{S}$. We call such an affine equation a *relation*: $\mathbb{S}$ holds the mask $\mathbf{k}$, and $\mathbb{R}$ holds the value $\mathbf{m}$ satisfying the displayed equation. This is the CI-VOLE shape we ultimately care about. VOLE can also be thought of as a secret sharing of $\mathbf{x} \in \mathcal{M}^w$ times $\Delta \in \mathcal{M}$, where $\mathbf{m}, \mathbf{k} \in \mathcal{M}^w$ are the shares. We will sometimes denote this secret sharing of $\mathbf{x}$ times $\Delta$ as $\langle \mathbf{x} \cdot \Delta \rangle$, where $\mathbf{m}, \mathbf{k}$ are implicit. This is the basic CI-VOLE functionality captured in Figure 1.

TinyLabels does *not* quite solve this problem in our intended private-input setting. Instead, in the garbled-circuit setting that motivates TinyLabels, the evaluator's input $\mathbf{x}$ is already handled in a somewhat public way: the garbler effectively knows the masked input vector $\mathbf{x}$ and may send it in the clear. So the object being transferred is better viewed as a *semi-public linear correlation* than as a private VOLE correlation. For understanding our construction, that distinction is important: TinyLabels gives a semi-succinct way to deliver

$$\mathbf{m} = \Delta \cdot \mathbf{x} + \mathbf{k},$$

but in a setting where $\mathbf{x}$ is not hidden from $\mathbb{S}$.

**The laconic starting point.** The key technical ingredient behind TinyLabels is laconic encryption [DKL+23a]. At a moral level, it gives $\mathbb{S}$ a way to encrypt a vector of repeated secrets and later lets anyone locally combine that ciphertext with an input vector $\mathbf{x}$. The result looks like a masked linear relation:

$$\mathsf{LacE.Eval}(\mathsf{ct}_{\mathsf{lac}}, \mathbf{x}) \approx \mathbf{r} \cdot d_{\mathbf{x}} - \Delta \cdot \mathbf{x},$$

where

- $\mathsf{ct}_{\mathsf{lac}} = \mathsf{LacE.Enc}(\Delta; \mathbf{r})$ is a ciphertext prepared by $\mathbb{S}$,
- $\mathbf{r} \in \mathcal{M}^w$ is encryption randomness, and
- $d_{\mathbf{x}} = \mathsf{LacE.Digest}(\mathbf{x})$ is a *short* digest of $\mathbf{x}$.

So laconic encryption already has a clear VOLE-like flavor: it almost gives a sharing of $\Delta \cdot \mathbf{x}$, but with an annoying mask term $\mathbf{r} \cdot d_{\mathbf{x}}$ attached to it. This is the central point.

**The core obstacle.** That mask term depends on $d_{\mathbf{x}}$, hence on the eventual runtime input $\mathbf{x}$. Therefore, laconic encryption by itself does not yet give $\mathbb{S}$ a *fixed* mask $\mathbf{k}$ known in advance. But in

garbled circuits, $\mathbb{S}$ needs exactly such a fixed mask ahead of time, because those values are part of the garbling material. So the problem is:

> How can we replace the $\mathbf{x}$-dependent mask $\mathbf{r} \cdot d_{\mathbf{x}}$ by a mask $\mathbf{k}$ that is fixed before $\mathbf{x}$ is chosen?

**TinyLabels' answer: a short key for the correction term.** TinyLabels resolves this by adding a linearly homomorphic encryption layer. Here is the simplified TinyLabels flow in dependency order. For this overview, we keep the abstract spaces already used above:[2]

$$\Delta \in \mathcal{M}, \qquad \mathbf{x}, \mathbf{k} \in \mathcal{M}^w, \qquad \mathbf{r} \in \mathcal{C}^w, \qquad d_{\mathbf{x}}, \mathsf{sk}_{\mathsf{r}}, \mathsf{sk}_{\mathsf{k}}, \mathsf{sk}_{\mathbf{x}} \in \mathcal{C}.$$

As above, we intentionally suppress the encoding details that let these quantities interact across $\mathcal{M}$ and $\mathcal{C}$.

1. $\mathbb{S}$**'s setup.** $\mathbb{S}$ samples the global secret $\Delta \in \mathcal{M}$, the laconic randomness $\mathbf{r} \in \mathcal{C}^w$, a static mask vector $\mathbf{k} \in \mathcal{M}^w$, and two short LHE-side keys $\mathsf{sk}_{\mathsf{r}} \in \mathcal{C}$ and $\mathsf{sk}_{\mathsf{k}} \leftarrow \mathcal{C}$. It then prepares:

   - a laconic ciphertext
   $$\mathsf{ct}_{\mathsf{lac}} = \mathsf{LacE.Enc}((\Delta, \ldots, \Delta); \mathbf{r}), \tag{1}$$
   from $\Delta$ and $\mathbf{r}$, arranged so that later evaluation on $\mathbf{x} \in \mathcal{M}^w$ yields
   $$\mathsf{LacE.Eval}(\mathsf{ct}_{\mathsf{lac}}, \mathbf{x}) = \mathbf{r} \cdot d_{\mathbf{x}} - \Delta \cdot \mathbf{x},$$
   - $\mathsf{ct}_{\mathsf{r}} = \mathsf{LHE.Enc}(\mathsf{sk}_{\mathsf{r}}, \mathbf{r})$ encrypting a length-$w$ vector, and
   - $\mathsf{ct}_{\mathsf{k}} = \mathsf{LHE.Enc}(\mathsf{sk}_{\mathsf{k}}, \mathbf{k})$ encrypting a length-$w$ vector.

2. $\mathbb{R}$**'s input.** Later, $\mathbb{R}$ chooses the vector $\mathbf{x} \in \mathcal{M}^w$ and computes its public digest
   $$d_{\mathbf{x}} = \mathsf{LacE.Digest}(\mathbf{x}) \in \mathcal{C}.$$
   The size of $d_{\mathbf{x}}$ is a constant, independent of $\mathbf{x}$.

3. $\mathbb{S}$**'s online response.** Using this digest, $\mathbb{S}$ forms the short derived key
   $$\mathsf{sk}_{\mathbf{x}} = \mathsf{sk}_{\mathsf{r}} \cdot d_{\mathbf{x}} + \mathsf{sk}_{\mathsf{k}} \in \mathcal{C}.$$
   This is the only input-dependent secret sent online. Notice that $\mathsf{sk}_{\mathsf{k}}$ is uniformly random, serving as a random mask for the derived key $\mathsf{sk}_{\mathbf{x}}$.

4. $\mathbb{R}$**'s LHE decryption.** From the encryptions of $\mathbf{r}$ and $\mathbf{k}$, together with the derived key $\mathsf{sk}_{\mathbf{x}}$, $\mathbb{R}$ recovers
   $$\mathbf{r} \cdot d_{\mathbf{x}} + \mathbf{k} = \mathsf{LHE.Dec}(\mathsf{sk}_{\mathbf{x}}, \mathsf{ct}_{\mathsf{r}} \cdot d_{\mathbf{x}} + \mathsf{ct}_{\mathsf{k}}). \tag{2}$$

5. $\mathbb{R}$**'s laconic evaluation.** Independently, $\mathbb{R}$ evaluates the laconic ciphertext on $\mathbf{x}$ and gets
   $$\mathbf{r} \cdot d_{\mathbf{x}} - \Delta \cdot \mathbf{x} = \mathsf{LacE.Eval}(\mathsf{ct}_{\mathsf{lac}}, \mathbf{x}).$$

[2] In the full TinyLabels construction, the internal LacE output dimension need not literally equal the plaintext input length. In this overview, we identify these dimensions and write both as $w$ in order to keep the dependency flow readable.

6. **Final combination.** Subtracting the laconic output from the LHE output cancels the digest-dependent mask:

$$\underbrace{(\mathbf{r} \cdot d_{\mathbf{x}} + \mathbf{k})}_{\text{From } \mathsf{LHE}} - \underbrace{(\mathbf{r} \cdot d_{\mathbf{x}} - \Delta \cdot \mathbf{x})}_{\text{From } \mathsf{LacE}} = \underbrace{\Delta \cdot \mathbf{x} + \mathbf{k}}_{\mathsf{CI\text{-}VOLE} \text{ output}} . \tag{3}$$

Thus $\mathbb{R}$ obtains the desired authenticated labels, while $\mathbb{S}$ knows the matching mask $\mathbf{k}$.

**Why this is succinct.** The key asymmetry is between the large offline objects and the short online object. $\mathbb{S}$'s offline material—namely the laconic ciphertext $\mathsf{ct}_{\mathsf{lac}}$ and the LHE ciphertexts $\mathsf{ct}_{\mathsf{r}}, \mathsf{ct}_{\mathsf{k}}$—still has size linear in $w$, since it encrypts vectors of length $w$. What becomes succinct is the $\mathbf{x}$-*dependent* part. Once $\mathbf{x}$ is fixed and public, $\mathbb{S}$ does not send a length-$w$ correction vector. Instead, it sends only the short derived key

$$\mathsf{sk}_{\mathbf{x}} = \mathsf{sk}_{\mathsf{r}} \cdot d_{\mathbf{x}} + \mathsf{sk}_{\mathsf{k}},$$

which depends on $\mathbf{x}$ only through the short digest $d_{\mathbf{x}}$. Thus the online dependence on $\mathbf{x}$ has been compressed from a vector-sized object to a single digest-sized key.

**What matters for us.** For $\mathsf{LogVOLE}$, the important takeaway is not the exact garbled-circuit packaging, but the structural pattern underneath it. TinyLabels first *shrinks* $\mathbb{R}$'s long input vector $\mathbf{x}$ to a short digest $d_{\mathbf{x}}$, and then uses a short key derived from $d_{\mathbf{x}}$ to *expand* back to the desired linear correlation ($\Delta \cdot \mathbf{x} + \mathbf{k}$). This shrink/expand viewpoint is the real conceptual bridge to our construction. It isolates the part that depends on the full vector $\mathbf{x}$ into a short digest, and postpones the recovery of the full label vector to a later local computation.

**What TinyLabels does *not* give us.** From the CI-VOLE viewpoint, TinyLabels still falls short in two ways. First, it does not protect the privacy of $\mathbb{R}$'s input: in TinyLabels' target use case, $\mathbf{x}$ is effectively available to $\mathbb{S}$, so the resulting correlation is better viewed as semi-public than as private chosen-input VOLE. Second, although the online response is short, the offline setup remains linear in $|\mathbf{x}|$, because $\mathbb{S}$ must still prepare and transmit vector-sized ciphertexts such as $\mathsf{ct}_{\mathsf{lac}}$, $\mathsf{ct}_{\mathsf{r}}$, and $\mathsf{ct}_{\mathsf{k}}$. These are exactly the two limitations that our construction addresses next: we hide $\mathbf{x}$ from $\mathbb{S}$, and we compress the offline cost from linear to polylogarithmic.

## 4.2 Overview of LogVOLE

Our construction $\mathsf{LogVOLE}$ drastically reduces the (one-time) setup and online communication costs to $O(\log |\mathcal{R}_q|)$, where $\mathcal{R}_q$ is the ring used by the underlying RLWE ciphertexts. In this section, we explain the structural moves that make this possible. From now on, we work in the private-input setting: $\mathbb{R}$ knows $\mathbf{x}$ and must not reveal it to $\mathbb{S}$.

**Step 1: remove the public-x assumption.** The first obstacle is that TinyLabels, as stated, assumes the digest input is effectively public. In particular, if one writes

$$\mathsf{sk}_{\mathbf{x}} = \mathsf{sk}_{\mathsf{r}} \cdot d_{\mathbf{x}} + \mathsf{sk}_{\mathsf{k}} \qquad \text{where} \qquad d_{\mathbf{x}} = \mathsf{Digest}(\mathbf{x}),$$

then any party that computes $\mathsf{sk}_{\mathbf{x}}$ from $\mathsf{sk}_{\mathsf{r}}, \mathsf{sk}_{\mathsf{k}}$ and $d_{\mathbf{x}}$ must learn the digest input $\mathbf{x}$ itself, or at least some information about it. In particular, TinyLabels only used the digest in a functional way that authenticated $x$ but did not hide it. This is incompatible with CI-VOLE, where $\mathbb{R}$ must keep $\mathbf{x}$ hidden from $\mathbb{S}$.

Our approach leans on the fact that $d_{\mathbf{x}} = \mathsf{Digest}(x)$ is essentially the root of a Merkle tree. We prove that for this particular instantiation of $\mathsf{Digest}$, if one randomizes the input, then the digest is hiding. Concretely, $\mathbb{R}$ samples fresh randomness $\zeta$ and computes

$$d_{\mathbf{x}} := \mathsf{Digest}(\mathbf{x}; \zeta).$$

The protocol then proceeds exactly as before, except that all digest-dependent quantities are defined with respect to this randomized digest value, a commitment to $\mathbf{x}$.

The key point is that the randomness $\zeta$ raises the entropy of the digest input, and our later analysis shows that $\mathsf{Digest}$ is hiding on such inputs. Thus, $d_{\mathbf{x}} = \mathsf{Digest}(\mathbf{x}; \zeta)$ can be revealed and used in the TinyLabels derivation $\mathsf{sk}_{\mathbf{x}} = \mathsf{sk}_{\mathsf{r}} \cdot d_{\mathbf{x}} + \mathsf{sk}_{\mathsf{k}}$ without exposing $\mathbf{x}$ to $\mathbb{S}$. Section 6.4 gives the digest-randomization details.

With randomization, the digest serves not only as a compression mechanism but also as a privacy-preserving handle.

**Step 2: recall the TinyLabels ping-pong in the form we will reuse.** The main communication bottleneck inherited from TinyLabels is the linear-size ($|\mathbf{x}|$) offline setup: $\mathbb{S}$ must prepare vector-sized ciphertexts, in particular

$$\begin{aligned} \mathsf{ct}_{\mathsf{lac}} &= \mathsf{LacE.Enc}((\Delta, \ldots, \Delta); \mathbf{r}), \\ \mathsf{ct}_{\mathsf{r}} &= \mathsf{LHE.Enc}(\mathsf{sk}_{\mathsf{r}}, \mathbf{r}), \\ \mathsf{ct}_{\mathsf{k}} &= \mathsf{LHE.Enc}(\mathsf{sk}_{\mathsf{k}}, \mathbf{k}). \end{aligned}$$

Before we compress this setup, it is worth pausing and restating the TinyLabels flow in the form we will reuse repeatedly. The interaction has a very simple "ping-pong" structure:

1. $\mathbb{R}$ sends the short digest $d_{\mathbf{x}} = \mathsf{Digest}(\mathbf{x})$ of its chosen input $\mathbf{x}$.
2. $\mathbb{S}$ replies with the short key $\mathsf{sk}_{\mathbf{x}} = \mathsf{sk}_{\mathsf{r}} \cdot d_{\mathbf{x}} + \mathsf{sk}_{\mathsf{k}}$.
3. Using $\mathsf{sk}_{\mathbf{x}}$ and Equation (2), $\mathbb{R}$ turns this *digest-side relation* into the corresponding *output-side relation*
$$\mathbf{c}_{\mathbf{x}} = \mathbf{r} \cdot d_{\mathbf{x}} + \mathbf{k}.$$
It then uses the same cancellation as in Equation (3) to obtain
$$\mathbf{c}_{\mathbf{x}} - \mathsf{LacE.Eval}(\mathsf{ct}_{\mathsf{lac}}, \mathbf{x}) = \Delta \cdot \mathbf{x} + \mathbf{k}.$$

**Step 3: separate reusable setup from fresh correction.** Now we can make the next structural observation. Among the large offline objects above, the pair

$$(\mathsf{ct}_{\mathsf{lac}}, \mathsf{ct}_{\mathsf{r}})$$

depends only on the fixed secret $\Delta$ and the associated reusable randomness $\mathbf{r}$, as defined in Equation (1). This part can therefore be viewed as a *reusable setup*. By contrast, the correction side must be fresh for each invocation: $\mathbb{S}$ samples a fresh mask vector $\mathbf{k}$ and a fresh offset key $\mathsf{sk}_{\mathsf{k}}$, and forms the corresponding correction ciphertext $\mathsf{ct}_{\mathsf{k}}$. These masks enable TinyLabels to reuse $\mathbf{r}$ and $\mathsf{sk}_{\mathsf{r}}$ across queries. To avoid the linear-size communication due to $\mathsf{ct}_{\mathsf{k}}$, $\mathsf{TinyLabels}$ found a way to avoid sending it: In the random-oracle view, one can equivalently think of this fresh correction ciphertext as being sampled pseudorandomly, for example as

$$\mathsf{ct}_{\mathsf{k}} = H(\mathsf{seed}),$$

with $\mathbb{S}$ using $\mathsf{sk}_{\mathsf{k}}$ to recover the associated mask $\mathbf{k} = \mathsf{LHE.Dec}(\mathsf{sk}_{\mathsf{k}}, \mathsf{ct}_{\mathsf{k}})$. Section 7.2 gives the validity conditions for the random ciphertext.

So once $\Delta$ is fixed, the $\Delta$-dependent material can be cached and reused across many choices of $\mathbf{x}$. Only $d_{\mathbf{x}}$ and $\mathsf{ct}_{\mathsf{k}}$ change from instance to instance.

**Step 4: write the digest-side and output-side relations in parallel.** This reuse pattern becomes clearest when we look at many inputs at once. Suppose $\mathbb{R}$ wants CI-VOLE correlations for blocks $\mathbf{x}_1, \ldots, \mathbf{x}_\mu \in \mathcal{M}^w$. $\mathbb{R}$ sends the digest $d_{\mathbf{x}_i}$ for each $\mathbf{x}_i$ to $\mathbb{S}$. The corresponding *batched digest-side relation* for the expansion mechanism is

$$\begin{pmatrix} \mathsf{sk}_{\mathbf{x}_1} \\ \vdots \\ \mathsf{sk}_{\mathbf{x}_\mu} \end{pmatrix} = \mathsf{sk}_{\mathsf{r}} \cdot \begin{pmatrix} d_{\mathbf{x}_1} \\ \vdots \\ d_{\mathbf{x}_\mu} \end{pmatrix} + \begin{pmatrix} \mathsf{sk}_{k,1} \\ \vdots \\ \mathsf{sk}_{k,\mu} \end{pmatrix} \in \mathcal{C}^\mu. \tag{4}$$

From this digest-side relation, expansion produces the corresponding *batched output-side relation*

$$\begin{pmatrix} \mathbf{m}_{\mathbf{x}_1} \\ \vdots \\ \mathbf{m}_{\mathbf{x}_\mu} \end{pmatrix} = \Delta \cdot \begin{pmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_\mu \end{pmatrix} + \begin{pmatrix} \mathbf{k}_1 \\ \vdots \\ \mathbf{k}_\mu \end{pmatrix} \in \mathcal{M}^{w\mu}. \tag{5}$$

That is, for each block $\mathbf{x}_i$, $\mathbb{R}$ obtains a length-$w$ correlation

$$\mathbf{m}_{\mathbf{x}_i} = \Delta \cdot \mathbf{x}_i + \mathbf{k}_i \in \mathcal{M}^w.$$

These two displays should be read in that order: first the digest-side relation, then the output-side relation. Equation (4) is the batched *digest-side relation* for the TinyLabels-style expansion step: it is the short relation on the digests. Equation (5) is the batched *output-side relation*: it is the full relation on the original blocks. They have the same form. Only the ambient objects have changed: the global key $\Delta$ on the output side is replaced by the reusable key $\mathsf{sk}_{\mathsf{r}}$ on the digest side, and the long vectors $\mathbf{x}_i$ are replaced by their short digests $d_{\mathbf{x}_i}$.

Next, we show the power of the reuse idea alone to obtain $\sqrt{|\mathbf{x}|}$ communication. After that, we also use the self-similarity between Equations (4) and (5) to recurse, which drives the communication down to logarithmic.

**Warm-up: the balancing trick.** Once the reusable setup is separated from the fresh correction side, a simple amortization tradeoff becomes visible. The basic approach with $\mu = 1$ has linear offline communication, coming from the reusable setup, and constant online communication, coming from the short digest-dependent messages $d_{\mathbf{x}}, \mathsf{sk}_{\mathbf{x}}$. If one prepares a reusable setup for *blocks* of size $\mu$ and applies it across $|\mathbf{x}|/\mu$ blocks, then the total communication is

$$O_\lambda(\mu + |\mathbf{x}|/\mu).$$

Choosing $\mu = \Theta(\sqrt{|\mathbf{x}|})$ balances the two terms and gives total communication $O_\lambda(\sqrt{|\mathbf{x}|})$.

This warm-up is not yet the main result. Its role is to isolate the structural lesson: once the reusable setup is separated from the fresh correction side, one can trade setup size against the number of invocations.[3] The real gain comes next.

**Recursion.** The recursion in LogVOLE makes the setup cost logarithmic. The key observation is that the batched digest-side relation in (4) itself has the same CI-VOLE shape as the batched output-side relation in (5), but over a smaller domain.

It is helpful to isolate this as a shrink/expand picture. The expansion step starts from a digest-side relation, i.e., Equation (4), and returns an output-side relation on the underlying blocks, i.e.,

[3] In fact, TinyLabels already noted that this tradeoff yields a setup cost of $O_\lambda(\sqrt{|\mathbf{x}|})$. However, its authors did not exploit the self-similarity between the input and output layers to recurse, so they stopped at this warm-up.

Equation (5). This suggests recursion. We can generate the digest-side relation as the output-side relation of an even smaller instance. If we repeat this operation recursively, we obtain

$$\begin{array}{ccccccc} \mathbf{x} & \xrightarrow{\mathsf{Shrink}} & d_{\mathbf{x}} & \xrightarrow{\mathsf{Shrink}} & \cdots & \xrightarrow{\mathsf{Shrink}} & d_{\mathbf{x}}^{(\ell)} \\ \langle \mathbf{x} \cdot \Delta \rangle & \xleftarrow{\mathsf{Expand}} & \langle d_{\mathbf{x}} \cdot \mathsf{sk}_r \rangle & \xleftarrow{\mathsf{Expand}} & \cdots & \xleftarrow{\mathsf{Expand}} & \langle d_{\mathbf{x}}^{(\ell)} \cdot \mathsf{sk}_r^{(\ell)} \rangle \end{array}$$

The recursive idea is to avoid generating the digest-side relation in Equation (4) directly by a fresh linear-size setup. Instead, we produce it by running the same kind of mechanism again, but now on the digest vector

$$\mathbf{d}_{\mathbf{x}} = (d_{\mathbf{x}_1}, \ldots, d_{\mathbf{x}_\mu}).$$

This vector is the input to another, smaller shrink/expand instance.

Concretely, the vector

$$\mathbf{sk}_{\mathbf{x}} = (\mathsf{sk}_{\mathbf{x}_1}, \ldots, \mathsf{sk}_{\mathbf{x}_\mu})$$

already has exactly the algebraic form of a CI-VOLE output: it is a global secret times a message vector, plus a mask vector. The only difference is that the message vector is now the shorter digest vector rather than the original blocks. So the output of one level naturally serves as the input of the next.

This is the master recursion step. At the top level, we want authenticated outputs for the original blocks $\mathbf{x}_1, \ldots, \mathbf{x}_\mu$. To obtain them, it suffices to have authenticated inputs for the digests $d_{\mathbf{x}_1}, \ldots, d_{\mathbf{x}_\mu}$. But those authenticated digest inputs can themselves be produced by the same construction, after grouping the digests into smaller blocks. Repeating this process forms a tree of shrinking instances. Each level reduces the description length, while the reusable setup at that level is amortized across many children. After $O(\log |\mathbf{x}|)$ levels, the remaining instance has constant size, so the total setup cost becomes logarithmic.

This is why the earlier input/output parallelism matters. The reuse idea alone gives the $\sqrt{|\mathbf{x}|}$ warm-up. The additional observation that the input layer and output layer have the same algebraic form is what makes full recursion possible. Section 5 gives the detailed shrink/expand derivation.

**Further reusing the setup material.** Above, we described the recursion in a way that treats each level as a separate shrink/expand instance with a different key $\mathsf{sk_r}$. Thus, $\mathbb{S}$ should send $O(\log_2 |\mathbf{x}|)$ sets of $(\mathsf{ct_{lac}}, \mathsf{ct_r})$ for the $O(\log_2 |\mathbf{x}|)$ levels. However, observe that by reusing the same $\mathsf{sk_r}$ across all non-base levels, *i.e.*,

$$\mathsf{sk_r} = \mathsf{sk_r}^{(1)} = \cdots = \mathsf{sk_r}^{(\ell-1)},$$

$\mathbb{S}$ only needs to send one set of $(\mathsf{ct_{lac}}, \mathsf{ct_r})$ for the entire recursion, thus saving a factor of $O(\log_2 |\mathbf{x}|)$ in the setup cost.

Yet, this reuse does not entirely eliminate the logarithmic factor in communication. The remaining factor is due to our golden-seed optimization, which reduces the ring element size from $O(\lambda^3)$ to $O(\lambda \log^2 |\mathbf{x}|)$. While this optimization reintroduces dependence on $\log |\mathbf{x}|$ in communication, it actually improves performance since $\log |\mathbf{x}| \ll \lambda$. Section 7.2 gives the optimization details.

### 4.3 Overview of Realizing the Recursion

To aid understanding, the previous section deliberately abstracted away from the concrete RLWE instantiation and presented the recursion in a clean algebraic form. To instantiate this interface, each level must preserve the same input/output shape while handling three concrete issues: messages and ciphertexts live in different domains, ciphertext evaluations introduce noise, and the concrete evaluation must be optimized to make the scheme practical.

**Matching domains.** The first issue appears at the boundary between two recursive levels. Expansion at one level needs $\mathbb{R}$ to hold the short digest-side relation in the first form below, while ShrinkExpand produces the output-side relation in the second form:

$$\mathsf{sk}_{\mathbf{x}} = \mathsf{sk}_{\mathsf{r}} \cdot \mathbf{d}_{\mathbf{x}} + \mathbf{sk}_k \in \mathcal{C}^{w'}, \qquad \mathbf{m}_{\mathbf{x}} \ = \Delta \cdot \mathbf{x} + \mathbf{k} \in \mathcal{M}^{w}.$$

The recursive sketch reuses the first relation ($\mathsf{sk}_{\mathbf{x}}$) as a smaller instance of the second. This reuse is type-correct only if digests and keys can be represented as messages for the next shrink/expand call. In the concrete scheme, however, digests and LHE keys are in $\mathcal{C}$, while the messages are in $\mathcal{M}$. We handle this mismatch by using CRT decomposition to represent each $\mathcal{C}$ element as residues that fit the message domain, running shrink/expand on those residues, and recombining the results after expansion. Because CRT preserves addition and multiplication, the decomposed relations carry the same $\mathbf{d}_{\mathbf{x}} \cdot \mathsf{sk}_{\mathsf{r}}$ product as the original relation.

**Controlling noise.** Once the interface is type-correct, the next obstruction is ensuring the correctness of the ciphertext evaluations inside each recursive level. In our recursive computation, every level evaluates LacE and LHE ciphertexts on digests. A direct multiplication by a digest $d \in \mathcal{R}_q$ would use large coefficients and could push LHE noise past the decryption bound, damaging the messages inside. We therefore replace this multiplication by products against $\mathbf{g}^{-1}(d)$, whose digits are bounded by a small gadget base $g$, to avoid blowing up the noise. Also, after each level, the parties scale and round their decrypted shares, as in many LWE-based schemes, to produce noise-free shares of the needed keys, so evaluation noise is cleared and does not grow with recursion depth. Finally, we also keep the gadget decomposition compatible with the CRT decomposition used in shrink/expand across recursion levels.

**Optimizing the concrete cost.** Now, the remaining work is to reduce the dominant concrete costs without changing what each level exports. We propose two optimizations: First, one-digit gadget truncation reduces the overhead from gadget decomposition: for our implementation choice $\tau = 2$, it reduces one-time setup communication by about $4\times$ and query computation by about $2\times$. Second, *golden-seed grinding* uses rejection sampling to reduce the ring-element size from $O(\lambda^3)$ to $O(\lambda \log^2 |\mathbf{x}|)$. Sections 6 and 7 give the domain conversions, noise analysis, and optimization details.

### 4.4 Supporting Non-interactive CI-VOLE

Now we discuss how to modify the above interactive protocol to obtain a non-interactive public-key version. In this setting, $\mathbb{S}$ prepares the reusable setup material $(\mathsf{ct}_{\mathsf{r}}, \mathsf{ct}_{\mathsf{lac}})$ with its chosen global key $\Delta$. It then publishes this material as a public key while keeping the secret state $(\mathsf{sk}_{\mathsf{r}}, \Delta)$. Once $\mathbb{R}$ obtains the public key, it can generate the correlation $m_{\mathbf{x}}$ according to its chosen input $\mathbf{x}$. Then, $\mathbb{R}$ can send a single message to $\mathbb{S}$ to let $\mathbb{S}$ recover the matching share $\mathbf{k}$ such that $m_{\mathbf{x}} - \mathbf{k} = \Delta \cdot \mathbf{x}$. Thus, one sender public key for a fixed $\Delta$ can serve many receiver executions; the protocol domain-separates these executions by their identifiers.

Our observation enabling the non-interactive version is that the only online communication is the ping-pong protocol: $\mathbb{R}$ sends $d'_{\mathsf{rt}}$ to $\mathbb{S}$, and $\mathbb{S}$ replies with the key $\mathsf{sk}_{\mathsf{rt}}{}' = d'_{\mathsf{rt}} \cdot \mathsf{sk}_{\mathsf{r}} + \mathsf{sk}_k$. Notice that $\mathsf{sk}_k$ is sampled uniformly from $\mathcal{R}_q$ and thus masks $\mathsf{sk}_{\mathsf{rt}}{}'$, making it also uniformly random in $\mathcal{R}_q$.

Changing our perspective, $\mathbb{S}$ could instead sample $\mathsf{sk}_{\mathsf{rt}}{}'$ from $\mathcal{R}_q$ and then choose[4]

$$\mathsf{sk}'_k = \mathsf{sk}_{\mathsf{rt}} - \mathsf{sk}_{\mathsf{r}} \cdot d'_{\mathsf{rt}}$$

as the corresponding one-time key. This does not change the distribution of $(\mathsf{sk}'_{\mathsf{rt}}, \mathsf{sk}_k)$.

[4]For simplicity, we omit here the fact that $\mathsf{sk}_{\mathsf{rt}}$ should be $\mathsf{sk}_r^{\top} \cdot \mathbf{g}^{-1}(d'_{\mathsf{rt}}) + \mathsf{sk}_k$ to handle gadget decomposition. Section 6.1 gives the details.

Finally, we enable the non-interactive version by letting both parties compute

$$\mathsf{sk}'_{\mathsf{rt}} = H(d'_{\mathsf{rt}}).$$

Then, $\mathbb{R}$ can derive $\mathsf{sk}'_{\mathsf{rt}}$ without interacting with $\mathbb{S}$, and $\mathbb{S}$ can derive $\mathsf{sk}'_{\mathsf{rt}}$ after obtaining $d'_{\mathsf{rt}}$.

**Reducing Online Communication in Non-Staged Mode.** This key derivation trick can also reduce online communication in the interactive protocol when an application does not need the sender's mask before the receiver queries, *e.g.*, in interactive ZKP. The sender derives $\mathsf{sk}_k$ from $\mathsf{sk}_{\mathsf{rt}}{}'$ and the received digest, so it need not transmit $\mathsf{sk}_{\mathsf{rt}}{}'$, halving the online communication. (Yet, when running with the golden-seed optimization, the sender still needs to respond with a $\lambda$-bit string as the seed.)

### 4.5 Ideal Functionality and Modeling

Our basic functionality is the basic CI-VOLE interface in Figure 1. The sender fixes a global correlation value $\Delta \in \mathcal{M}$ and a width $w$. On a receiver input $\mathbf{x} \in \mathcal{M}^w$, the functionality samples a fresh sender mask $\mathbf{k} \leftarrow \mathcal{M}^w$, gives $\mathbf{k}$ to the sender, and gives the receiver

$$\mathbf{m} = \mathbf{k} + \mathbf{x} \cdot \Delta \in \mathcal{M}^w.$$

Since this overview focuses on the high-level ideas, this functionality idealizes only the correlation generation needed by applications such as the ZKP. The generalized functionality further idealizes the protocol's ability to generate correlations across multiple sessions, run non-interactively, and let the sender pre-compute its shares. Section 8 gives its formal description.

### 4.6 Application: IT-MAC-based ZKP

Prior IT-MAC-based ZKP protocols [WYKW21, BMRS21, DIO21a, YSWW21, DILO22, LHLA25a] first obtain random VOLE correlations under the verifier's hidden key $\Delta$. The prover $\mathbb{P}$ then sends a *linear-size* derandomization vector to move those labels to the actual extended witness, *i.e.*, the execution trace for the verification circuit. In our instantiation, $\mathbb{V}$ is CI-VOLE's sender, and $\mathbb{P}$ is the receiver. The point of LogVOLE is that the labels are generated directly on $\mathbb{P}$'s chosen witness, skipping the linear-size derandomization communication. The communication is thus reduced to $O(\lambda \log^2 |\mathbf{x}|)$ bits for the one-time setup and the exchange of the digest and decryption key during the online phase.

For a non-interactive ZKP, we use the public-key CI-VOLE variant in Section 4.4 to derive the decryption key locally and Fiat–Shamir to generate the proof challenges. With the maliciously secure reusable setup below, the prover sends the digest and proof in one message, giving a designated-verifier NIZK in the random oracle model.

Section 4.7 explains how to obtain a maliciously secure ZKP using maliciously secure CI-VOLE. Sections F and 11 present the concrete ZKP protocol and proof sketch.

### 4.7 Malicious Security

**Malicious Receiver.** We assume digest-knowledge extractability: given access to $\mathbb{R}$, an extractor recovers from its digest the input underlying its successful use of the shares. This assumption is analogous in spirit to extraction in the algebraic group model (AGM) [FKL18] and adds no protocol steps, computation, or communication. If readers prefer to avoid this less standard assumption, we can instead use random-linear correlation checks [KOS15, YWL+20, Roy22], which require one additional linear scan of each party's shares. In the ZKP application, the check can be combined

with proof verification: its challenge is derived via Fiat–Shamir, and the receiver's response, one field element per check, accompanies the prover's proof without an extra round.

**Malicious Sender.** There are two separate approaches. First, we can adapt the commitment step of the GC-to-ZK compiler [JKO13]. After $\mathbb{R}$ obtains its shares and prepares its application messages, *e.g.*, the proof in the ZKP application, it sends a commitment to these messages to $\mathbb{S}$. The sender then reveals the seed used to generate the $\mathsf{LogVOLE}$ setup material, allowing $\mathbb{R}$ to recompute and verify it before opening the commitment. This reveals $\Delta$, so the setup material can only be used once. The other approach is that $\mathbb{S}$ can use a lattice-friendly NIZK [BLS19, LNP22] to prove that the setup material is correctly generated. This keeps $\Delta$ hidden and allows the certified setup to be reused.

Sections E and 10 give the construction and security proofs.

# 5 The Shrink/Expand Core

This section prepares the reader for the low-level overview in Section 6 and Section 7. It opens the box of the core $\mathsf{Shrink}/\mathsf{Expand}$ mechanism, a critical component of our protocol, and focuses on its algebraic structure. We do not introduce any new ideas from our work here.

**Basic Terminology.** In general, a value $m = s \cdot x + k$ is an authenticated label of $x$, or equivalently an IT-MAC of $x$; we use the two terms interchangeably. Here, the sender owns the authentication key $s$ and the additive mask $k$, while the receiver owns the label $m$ and the authenticated value $x$.

## 5.1 Shrink/Expand Interface

**The Shrink/Expand Mechanism.** We start by stating the interface of our shrink/expand gadget. At the base level, $\mathbb{R}$ inputs the encoded $\mathbf{x} \in \mathcal{M}^w$ and $\mathbb{S}$ inputs $\Delta \in \mathcal{M}^w$ to the shrink gadget, which outputs to $\mathbb{R}$ a vector of digests $d_{\mathbf{x}} \in \mathcal{C}^{w'}$:

$$(\mathbb{R} : \mathbf{x} \in \mathcal{M}^w, \mathbb{S} : \mathbf{s} \in \mathcal{M}^w) \xrightarrow{\mathsf{Shrink}} (\mathbb{R} : d_{\mathbf{x}} \in \mathcal{C}^{w'})$$

Here, we have $w' < w$ to achieve shrinking. Similarly, $\mathbb{R}$ and $\mathbb{S}$ need to acquire the key shares $\mathsf{sk}_{\mathbf{x}} = \mathbf{d}_{\mathbf{x}} \cdot \mathsf{sk}_1 + \mathsf{sk}_k \in \mathcal{C}^{w'}$ and $\mathbf{k} \in \mathcal{C}^{w'}$, respectively, and use them as inputs to the expand gadget to obtain $\mathbf{m}_{\mathbf{x}} = \mathbf{x} \odot \mathbf{s} + \mathbf{k} \in \mathcal{M}^w$ and $\mathbf{k} \in \mathcal{M}^w$, *i.e.*,

$$(\mathbb{R} : \mathsf{sk}_{\mathbf{x}} \in \mathcal{C}^{w'}, \mathbb{S} : \mathsf{sk}_k \in \mathcal{C}^{w'}) \xrightarrow{\mathsf{Expand}} (\mathbb{R} : \mathbf{m}_{\mathbf{x}} \in \mathcal{M}^w, \mathbb{S} : \mathbf{k} \in \mathcal{M}^w)$$

Here is a slight detour about the functionality provided by the shrink/expand gadget. Previously, we only mentioned $\mathsf{LacE}$'s ability to compute broadcast multiplications. In fact, it also supports (more general) element-wise multiplications. The shrink/expand interface accepts the global key $\mathbf{s}$ as a vector of values, instead of a scalar value. (A scalar global key is handled by duplicating it across the vector).

**Dissecting the Shrink/Expand Interface.** We first focus on a simplified interface. (Later, Section 5.6 describes the true interface, which is presented in detail in Section 6.) $\mathsf{Shrink}$ compresses a block $\mathbf{x} \in \mathcal{R}_q^\mu$ of size $\mu$ to a digest

$$d_{\mathbf{x}} := \mathsf{Digest}(\mathbf{x}) \in \mathcal{R}_q.$$

It repeats this process for every block of the input vector, thus achieving shrinking.

Expand requires the parties to first generate an authenticated label of $d_{\mathbf{x}} \in \mathcal{R}_q$ via recursion. This label is the input to Expand. In the simplified interface, it has one multiplicative factor and one additive mask:

$$\widetilde{m} = \widetilde{s}\, d_{\mathbf{x}} + \widetilde{k} \in \mathcal{R}_q, \tag{6}$$

with $\widetilde{s}, \widetilde{k}, \widetilde{m} \in \mathcal{R}_q$.

Given this authenticated digest label, the block $\mathbf{x}$, and reusable LacE/LHE material, Expand produces the output-side IT-MAC form

$$\mathbf{m} = \mathbf{s} \odot \mathbf{x} + \mathbf{k} \in \mathcal{R}_q^{\mu}.$$

Here, we deliberately suppresses the low-level details, including the decomposed digest representation, coordinate ordering, CRT recombination, and scale-and-round details. The final subsection will state our complete interface and points to the lower-level realization that restores those details.

Once the materials $(\mathsf{ct}_{\mathsf{lac}}, \mathsf{ct}_{\mathsf{r}})$ are prepared, anyone with the digest preimage $\mathbf{x}$ can use them to translate the authenticated digest label into the self-similar form $\mathbf{m} = \mathbf{s} \odot \mathbf{x} + \mathbf{k}$ on $\mathbf{x}$, conceptually implementing

$$((\mathsf{ct}_{\mathsf{lac}}, \mathsf{ct}_{\mathsf{r}}), \mathbf{x}, \widetilde{m} = \widetilde{s}\, d_{\mathbf{x}} + \widetilde{k}) \mapsto (\mathbf{m} = \mathbf{s} \odot \mathbf{x} + \mathbf{k}).$$

The reusable materials $(\mathsf{ct}_{\mathsf{lac}}, \mathsf{ct}_{\mathsf{r}})$ are prepared independently of $\mathbf{x}$. Indeed, we crucially rely on the fact that the same materials can be used to translate many different $(d_{\mathbf{x}^{(1)}}, \mathbf{x}^{(1)}), \ldots, (d_{\mathbf{x}^{(\mu)}}, \mathbf{x}^{(\mu)})$ pairs. In that sense, these materials act as a vector of reusable garbled gates for this degree-1 label-translation map. These gates can be evaluated for *all* valid $(d_{\mathbf{x}}, \mathbf{x})$ pairs. In contrast, evaluation effectively outputs $\perp$ for any input that is not a valid digest-preimage pair. This remarkable construction originates in TinyLabels [DLL25]. We repeat it here because it is integral to our construction and to the various optimizations that interact with it. The rest of this section explains how the structure of the digest itself enables *bounded openings* that allow the translation to be performed only for valid digest-preimage pairs.

### 5.2 Revisiting Linear Laconic Encryption

First let us recall the full LacE construction that underlies TinyLabels. The shrink/expand core uses this telescoping structure, and Section 6.4 later modifies the top level to randomize the root digest. Here, we suppose the input length for the encrypted secrets is $\mu = 2^{\ell}$ for some $\ell$.

**Setup.** LacE setup samples the public node key $(\mathbf{b}_1, \mathbf{b}_2) \in \mathcal{R}_q^{\tau} \times \mathcal{R}_q^{\tau}$. We write the resulting global node key as

$$B := (B_1\ B_2) := (\mathbf{b}_1^{\top}\ \mathbf{b}_2^{\top}).$$

**Encryption.** To encrypt a vector $\mathbf{s} \in \mathcal{R}_q^{\mu}$, $\mathbb{S}$ views $\mathbf{s}$ as the leaf layer of a complete binary tree of depth $\ell$. It first samples intermediate masking vectors

$$\mathbf{r}^{(1)}, \mathbf{r}^{(2)}, \ldots, \mathbf{r}^{(\ell)} \in \mathcal{R}_q^{\mu}$$

and sets

$$\mathbf{r}^{(\ell+1)} := \mathbf{s}.$$

The top-level mask is denoted by

$$\mathbf{r} := \mathbf{r}^{(1)}.$$

These masks are the values that will telescope across the tree during evaluation. Then, for each level $i \in [\ell]$, $\mathbb{S}$ creates one ciphertext matrix $\mathbf{C}_i$. Its rows are indexed by the leaves $\mathsf{ind} \in \{0,1\}^\ell$, and for each such row, it samples noise $\mathbf{E}_i[\mathsf{ind}] \leftarrow \overline{\mathcal{D}}^{2\tau}$ and computes

$$\mathbf{C}_i[\mathsf{ind}] := \mathbf{r}^{(i)}[\mathsf{ind}] \cdot B + \mathbf{r}^{(i+1)}[\mathsf{ind}] \cdot (\overline{\mathsf{ind}}_i \cdot \mathbf{g}^\top \ \mathsf{ind}_i \cdot \mathbf{g}^\top) + \mathbf{E}_i[\mathsf{ind}].$$

Here $\mathsf{ind}_i \in \{0,1\}$ denotes the $i$-th bit of the leaf index $\mathsf{ind}$, $\overline{\mathsf{ind}}_i := 1 - \mathsf{ind}_i$ denotes its complement, and $\overline{\mathcal{D}}$ is the $\mathsf{LacE}$ error distribution. The first term serves as the masking term with the public node key $B$ and the "secret" key $\mathbf{r}^{(i)}$. The second term programs the branch selected by the bit $\mathsf{ind}_i$. Collecting all levels gives

$$\mathsf{ct}_{\mathsf{lac}} := (\mathbf{C}_1, \ldots, \mathbf{C}_\ell),$$

and the encryption output is $(\mathbf{r}, \mathsf{ct}_{\mathsf{lac}})$.

**Digest.** Given an input vector $\mathbf{x} \in \mathcal{R}_q^\mu$, the digest algorithm places the input $\mathbf{x}$ at the leaves of the same depth-$\ell$ tree, namely

$$y_{\mathsf{ind}} := \mathbf{x}[\mathsf{ind}] \qquad \forall \mathsf{ind} \in \{0,1\}^\ell.$$

Then, for every internal node in the tree, it uses the same public node key $B$ to compute a digest value from its two child values, *i.e.*,

$$\mathsf{NodeDigest}_B(u_L, u_R) := (B_1 \ B_2) \begin{pmatrix} \mathbf{g}^{-1}(u_L) \\ \mathbf{g}^{-1}(u_R) \end{pmatrix}.$$

The digest algorithm then moves upward from the leaves to the root. Here $\{0,1\}^{<\ell} := \bigcup_{t=0}^{\ell-1} \{0,1\}^t$ denotes the set of all binary prefixes indexing internal nodes. For every level $t = \ell - 1, \ldots, 0$ and every prefix $\mathsf{pre} \in \{0,1\}^t$, it computes

$$y_{\mathsf{pre}} = \mathsf{NodeDigest}_B(y_{\mathsf{pre}\|0}, y_{\mathsf{pre}\|1}) \qquad \forall \mathsf{pre} \in \{0,1\}^{<\ell}.$$

After all levels are processed, the root value $d_{\mathbf{x}} := y_\epsilon$ is the $\mathsf{LacE}$ digest of $\mathbf{x}$.

**Evaluation.** Given $\mathsf{ct}_{\mathsf{lac}}$ and $\mathbf{x}$, $\mathbb{R}$ first recomputes the same intermediate values $y_{\mathsf{pre}}$ as in $\mathsf{LacE.Digest}(\mathbf{x})$. At this point, every ciphertext row $\mathbf{C}_i[\mathsf{ind}]$ can be paired with the two child values that appear on the root-to-leaf path for index $\mathsf{ind}$. Then, for each leaf index $\mathsf{ind} \in \{0,1\}^\ell$, $\mathbb{R}$ sums one contribution from every tree level:

$$\mathsf{LacE.Eval}(\mathsf{ct}_{\mathsf{lac}}, \mathbf{x})[\mathsf{ind}] := \sum_{i=1}^{\ell} \mathbf{C}_i[\mathsf{ind}] \cdot \begin{pmatrix} -\mathbf{g}^{-1}(y_{\mathsf{ind}_{:(i-1)}\|0}) \\ -\mathbf{g}^{-1}(y_{\mathsf{ind}_{:(i-1)}\|1}) \end{pmatrix}.$$

This sum telescopes along the path indexed by $\mathsf{ind}$. At level $i$, the public-key term reconstructs the parent digest value $y_{\mathsf{ind}_{:(i-1)}}$, while the selector term extracts the child value $y_{\mathsf{ind}_{:i}}$. Summing over all levels cancels the intermediate masks, leaving only the top and bottom terms:

$$\mathsf{LacE.Eval}(\mathsf{ct}_{\mathsf{lac}}, \mathbf{x})[\mathsf{ind}] \approx \mathbf{r}[\mathsf{ind}] \cdot d_{\mathbf{x}} - \mathbf{s}[\mathsf{ind}] \cdot \mathbf{x}[\mathsf{ind}].$$

Stacking these coordinates gives the vector relation

$$\mathsf{LacE.Eval}(\mathsf{ct}_{\mathsf{lac}}, \mathbf{x}) \approx \mathbf{r} \cdot d_{\mathbf{x}} - \mathbf{s} \odot \mathbf{x}.$$

This is exactly the cancellation pattern used in $\mathsf{TinyLabels}$, and Section 6.4 reuses the same interface for the extra root wrapper.

$\underline{\mathsf{LacE.Setup}(1^\lambda, 1^\mu) \to (\mathbf{b}_1, \mathbf{b}_2) \in \mathcal{R}_q^\tau \times \mathcal{R}_q^\tau}$

- PARAMETERS: $\tau := \lceil \log_g q \rceil$, $\mathbf{g} := (1, g, \ldots, g^{\tau-1})$.
- PROCEDURES:
  1. $\mathbf{b}_1, \mathbf{b}_2 \leftarrow \mathcal{R}_q^\tau$
  2. return $(\mathbf{b}_1, \mathbf{b}_2)$

$\underline{\mathsf{LacE.Enc}(\mathbf{s} \in \mathcal{R}_q^\mu) \to (\mathbf{r} \in \mathcal{R}_q^\mu, \mathsf{ct_{lac}})}$

- PROCEDURES:
  1. $\ell := \log_2 \mu$
  2. $\mathbf{r}^{(1)} := \mathbf{r} \leftarrow \mathcal{R}_q^\mu$, $\mathbf{r}^{(\ell+1)} := \mathbf{s}$, and $\mathbf{r}^{(2)}, \ldots, \mathbf{r}^{(\ell)} \leftarrow \mathcal{R}_q^\mu$
  3. For $i \in [\ell]$:
     (a) $\mathbf{E}_i \leftarrow \overline{\mathcal{D}}^{\mu \times 2\tau}$
     (b) For $\mathsf{ind} \in \{0,1\}^\ell$: $\mathbf{C}_i[\mathsf{ind}] := \mathbf{r}^{(i)}[\mathsf{ind}] \cdot (\mathbf{b}_1^\top \ \mathbf{b}_2^\top) + \mathbf{r}^{(i+1)}[\mathsf{ind}] \cdot (\overline{\mathsf{ind}}_i \cdot \mathbf{g}^\top \ \mathsf{ind}_i \cdot \mathbf{g}^\top) + \mathbf{E}_i[\mathsf{ind}]$
  4. return $(\mathbf{r}, \mathsf{ct_{lac}} := (\mathbf{C}_1, \ldots, \mathbf{C}_\ell))$

$\underline{\mathsf{LacE.Digest}(\mathbf{x} \in \mathcal{R}_q^\mu) \to d_\mathbf{x} \in \mathcal{R}_q}$

- PROCEDURES:
  1. $\ell := \log_2 \mu$ and $y_\mathsf{ind} := \mathbf{x}[\mathsf{ind}]$ for all $\mathsf{ind} \in \{0,1\}^\ell$
  2. For $t = \ell - 1, \ldots, 0$:
     (a) For $\mathsf{pre} \in \{0,1\}^t$: $y_\mathsf{pre} := (\mathbf{b}_1^\top \ \mathbf{b}_2^\top)\binom{-\mathbf{g}^{-1}(y_{\mathsf{pre}\|0})}{-\mathbf{g}^{-1}(y_{\mathsf{pre}\|1})}$
  3. return $d_\mathbf{x} := y_\epsilon$

$\underline{\mathsf{LacE.Eval}(\mathsf{ct_{lac}}, \mathbf{x} \in \mathcal{R}_q^\mu) \to \boldsymbol{\phi} \in \mathcal{R}_q^\mu}$

- PROCEDURES:
  1. $\ell := \log_2 \mu$ and parse $\mathsf{ct_{lac}} := (\mathbf{C}_1, \ldots, \mathbf{C}_\ell)$
  2. Compute $\{y_\mathsf{pre}\}_{\mathsf{pre} \in \{0,1\}^{\le \ell}}$ as in $\mathsf{LacE.Digest}(\mathbf{x})$
  3. For $\mathsf{ind} \in \{0,1\}^\ell$: $\boldsymbol{\phi}[\mathsf{ind}] := \sum_{i=1}^\ell \mathbf{C}_i[\mathsf{ind}] \cdot \binom{-\mathbf{g}^{-1}(y_{\mathsf{ind}_{:(i-1)}\|0})}{-\mathbf{g}^{-1}(y_{\mathsf{ind}_{:(i-1)}\|1})}$
  4. return $\boldsymbol{\phi}$

Figure 3: Original (full-gadget) LacE interface, before the one-digit truncation optimization.

### 5.3 One Leaf Path of Expand

**Two Nested Trees.** Changing our perspective, we can view the shrink/expand execution as two nested trees. The outer tree is the protocol-recursion tree shown in Figures 10 and 11: each level groups the current vector into blocks, shrinks each block to a digest representation, and recursively obtains authenticated shares for the shorter digest vector. Inside each block processed at a node of this protocol-recursion tree, the map $\mathsf{Digest} : \mathcal{R}_q^\mu \to \mathcal{R}_q$ computes one block digest using a binary tree over the $\mu$ entries of that block. This subsection freezes one node of the protocol-recursion tree and focuses on a single *root-to-leaf* path in its internal digest tree.

Fix the block $\mathbf{x} \in \mathcal{R}_q^\mu$ and assign each of its $\mu$ elements to a leaf of a complete binary tree. Each parent node value in this tree is obtained by applying a compression function $\mathsf{NodeDigest} : \mathcal{R}_q^2 \to \mathcal{R}_q$ to its children. $d_\mathbf{x}$ is the root.

**A Node Opening.** Consider an arbitrary node of this block's binary Digest tree. Let $(u_L, u_R) \in \mathcal{R}_q^2$ be its left and right child values, and let $y = \mathsf{NodeDigest}_B(u_L, u_R)$ be its parent value. Define the gadget lifted input

$$w(u_L, u_R) := \begin{pmatrix} -\mathbf{g}^{-1}(u_L) \\ -\mathbf{g}^{-1}(u_R) \end{pmatrix} \in \mathcal{R}_q^{2\tau},$$

where $\mathbf{g}^{-1} : \mathcal{R}_q \to \mathcal{R}_q^\tau$ is the decomposition gadget that returns the input in a "small" linear representation, so that reconstruction by $\mathbf{g}$ is linear and $u = \mathbf{g}(\mathbf{g}^{-1}(u))$. The decomposition gadget

is discussed in greater detail in Section 6.

The vector $w(u_L, u_R)$ is the gadget-lifted opening of the two child values. When this lifted vector is small in the sense of Definition 9, we call it a *bounded opening*. With the public matrix $B = (B_L\ B_R)$ and

$$\widehat{G}_L := (\mathbf{g}^\top\ 0), \qquad \widehat{G}_R := (0\ \mathbf{g}^\top),$$

the parent and each child satisfy

$$y = -Bw(u_L, u_R), \qquad u_b = -\widehat{G}_b w(u_L, u_R) \quad \text{for } b \in \{L, R\}.$$

The child values $u_L, u_R$ are arbitrary ring elements; the bounded-opening condition applies to the lifted gadget vector $w(u_L, u_R)$.

**One Leaf Path.** Fix an index $\mathsf{ind} \in [\mu]$. We focus on the algebra at this *single* leaf: the sender will prepare $\mathsf{LacE}$ and $\mathsf{LHE}$ ciphertexts that allow the receiver to generate the IT-MAC form for this leaf

$$m_{\mathsf{ind}} = s_{\mathsf{ind}} \cdot x_{\mathsf{ind}} + k_{\mathsf{ind}}.$$

Let $d_{\mathbf{x}} = \mathsf{Digest}(\mathbf{x}) \in \mathcal{R}_q$. Since the receiver knows $\mathbf{x}$, it can recompute this digest tree: the leaves are the entries of $\mathbf{x}$, each internal node is obtained by the node map above, the root is $d_{\mathbf{x}}$, and each internal node has the bounded opening $w(u_L, u_R)$. For the path ending at $x_{\mathsf{ind}}$, let

$$y_0, y_1, \ldots, y_\ell \qquad \text{with} \qquad y_0 = d_{\mathbf{x}}, \quad y_\ell = x_{\mathsf{ind}}$$

be the node values on the root-to-leaf path. For each $i = 1, \ldots, \ell$, let $b_i \in \{L, R\}$ be the branch direction and set

$$w_i := w(u_{i,L}, u_{i,R}) \in \mathcal{R}_q^{2\tau},$$

where $(u_{i,L}, u_{i,R})$ are the left and right children of $y_{i-1}$ and $u_{i,b_i} = y_i$. The value $w_i$ is not an additional secret object: it is the bounded opening of this node and is computed directly from the two child values in the receiver's digest tree. It has small norm in the bounded-opening sense defined above. The key point is that the same small-norm bounded opening $w_i$ represents both the parent and the selected child through public linear maps:

$$y_{i-1} = -Bw_i, \qquad y_i = -\widehat{G}_{b_i} w_i.$$

**Multiplicative Factors.** To generate the $\mathsf{LacE}$ materials $\mathsf{ct}_{\mathsf{lac}} = (\mathbf{C}_1, \ldots, \mathbf{C}_\ell)$, the offline sender first enumerates the *multiplicative factors* carried at each level of the tree, in particular

$$r^{(1)} := \widetilde{s}, \qquad r^{(2)}, \ldots, r^{(\ell)} \leftarrow \mathcal{R}_q, \qquad r^{(\ell+1)} := s_{\mathsf{ind}}.$$

Thus $r^{(1)} = \widetilde{s}$ is the starting multiplicative factor from the simplified authenticated digest label (in Equation (6)) on the root digest $y_0 = d_{\mathbf{x}}$. As explained below, after the $i$-th translation step is evaluated, the multiplicative factor becomes $r^{(i+1)}$ on the child value $y_i$. After the last step, $r^{(\ell+1)} = s_{\mathsf{ind}}$ is the outgoing multiplicative factor on the leaf $y_\ell = x_{\mathsf{ind}}$.

**Row Translation.** For the fixed leaf $\mathsf{ind}$, the selected row $\mathbf{C}_i[\mathsf{ind}]$ of the $i$-th $\mathsf{LacE}$ ciphertext $\mathbf{C}_i$ plays the role of a "reusable garbled gate" for the $i$-th translation step. $\mathbf{C}_i[\mathsf{ind}]$ is prepared in the offline phase and does not depend on the receiver's choice of $\mathbf{x}$.

Later, the receiver's bounded opening $w_i$ acts as a "key" that enables the translation operation: evaluating $\mathbf{C}_i[\mathsf{ind}]w_i$ converts the authenticated label on the parent value $y_{i-1}$, under multiplicative factor $r^{(i)}$, to the digest preimage child value $y_i$ in the digest tree, under multiplicative factor $r^{(i+1)}$, up to bounded noise. In our current simplification, the row $\mathbf{C}_i[\mathsf{ind}]$ is defined by

$$\mathbf{C}_i[\mathsf{ind}] := -r^{(i)}B + r^{(i+1)}\widehat{G}_{b_i} + \mathbf{E}_i[\mathsf{ind}] \in \mathcal{R}_q^{1\times 2\tau},$$

so $\mathbf{C}_i[\mathsf{ind}]w_i \in \mathcal{R}_q$.

**Path Telescope.** The selected row $\mathbf{C}_i[\mathsf{ind}]$ is generated with a small row-noise term $\mathbf{E}_i[\mathsf{ind}] \in \mathcal{R}_q^{1\times 2\tau}$ for security. This results in the noise being scaled by the bounded opening $w_i$, that is,

$$\mathbf{C}_i[\mathsf{ind}]w_i = r^{(i)}y_{i-1} - r^{(i+1)}y_i + \mathbf{E}_i[\mathsf{ind}]w_i.$$

We keep the factor $w_i$ and $\mathbf{E}_i[\mathsf{ind}]$ explicit because the scaled noise term $\mathbf{E}_i[\mathsf{ind}]w_i$ will be controlled through the size of the opening, as discussed below.

Summing these identities over the path gives the leaf entry of the $\mathsf{LacE}$ evaluation

$$\begin{aligned}\phi &:= \sum_{i=1}^{\ell} \mathbf{C}_i[\mathsf{ind}]w_i = r^{(1)}d_{\mathbf{x}} - r^{(\ell+1)}x_{\mathsf{ind}} + \sum_{i=1}^{\ell} \mathbf{E}_i[\mathsf{ind}]w_i \\ &= r^{(1)}d_{\mathbf{x}} - r^{(\ell+1)}x_{\mathsf{ind}} + \mathsf{noise} = \mathsf{LacE.Eval}(\mathsf{ct}_{\mathsf{lac}}, \mathbf{x})[\mathsf{ind}]\end{aligned}$$

All intermediate node terms cancel. The only surviving structured terms are the root term, the leaf term, and the accumulated row noise evaluated on the bounded openings.

**Digest-Term Cancellation.** The $\mathsf{LacE}$ value $\phi$ contains the digest term minus the leaf term and is computed with knowledge of the small-norm bounded openings $w_i$. The $\mathsf{LHE}$ portion gives the matching digest term plus the sender's additive mask. In the notation of the lower-level realization, define the fixed-leaf value

$$\theta := \mathsf{LHE.Dec}(\mathsf{sk}_{\mathsf{rt}}, \mathsf{ct}_{\mathrm{res}})[\mathsf{ind}] = r^{(1)}d_{\mathbf{x}} + k_{\mathsf{ind}} + \mathsf{noise}'.$$

At this fixed leaf, the $\mathsf{Expand}$ update is $\theta - \phi$. By subtracting the two parts, we obtain

$$\theta - \phi = (r^{(1)}d_{\mathbf{x}} + k_{\mathsf{ind}} + \mathsf{noise}') - (r^{(1)}d_{\mathbf{x}} - r^{(\ell+1)}x_{\mathsf{ind}} + \mathsf{noise}) = r^{(\ell+1)}x_{\mathsf{ind}} + k_{\mathsf{ind}} - \mathsf{noise}''.$$

After the denoising step (see Section 6.2), using $r^{(\ell+1)} = s_{\mathsf{ind}}$, the receiver obtains the leaf IT-MAC form

$$m_{\mathsf{ind}} = s_{\mathsf{ind}}x_{\mathsf{ind}} + k_{\mathsf{ind}}.$$

**Full Block Output.** Repeating this $\mathsf{LacE}$ path telescope and $\mathsf{LHE}$ correction for every entry of the block yields the full output IT-MAC form

$$\mathbf{m} = \mathbf{s} \odot \mathbf{x} + \mathbf{k}, \qquad \mathbf{s}, \mathbf{k}, \mathbf{m} \in \mathcal{R}_q^{\mu}.$$

Even when two entries share a prefix in the digest tree, they use separate multiplicative factors $r^{(i)}$ and separate selected rows $\mathbf{C}_i[\mathsf{ind}]$ for their root-to-leaf translation steps. What is shared is the digest-tree node values and their bounded openings, not the multiplicative factors or selected rows that transport the authenticated label. Thus the telescope for a leaf uses its own sequence of multiplicative factors $r^{(1)}, \ldots, r^{(\ell+1)}$ and selected rows $\mathbf{C}_i[\mathsf{ind}]$. In this simplified view, the collective translation data across the block has one row of an $\mathsf{LacE}$ ciphertext for each leaf-depth pair, hence $O(\mu \log \mu)$ rows in total.

## 5.4 Bounded Openings Enable Translation

As seen above, the point of the digest is not merely to compress a block to a root. For our digest, the translation only works for the input that actually produced the root. When the receiver computes $d_{\mathbf{x}}$, it also computes, for every edge on the path to a leaf $x_{\mathsf{ind}}$, a bounded opening $w_i$ satisfying

$$y_{i-1} = -Bw_i, \qquad y_i = -\widehat{G}_{b_i}w_i.$$

The equations alone are not enough. In the telescope, the error is $\sum_i \mathbf{E}_i[\mathsf{ind}]w_i$, so the row noise remains controlled only when the lifted opening is bounded. A large algebraic preimage may have the right formal shape, but it does not yield a usable noisy translation: the noise $\mathbf{E}_i[\mathsf{ind}]w_i$ is large when $w_i$ is large, and this destroys the payload $r^{(i+1)}y_i$.

Thus the receiver cannot take a root digest and freely decide which leaves it opens to. To translate the root's authenticated digest label down to $x_{\mathsf{ind}}$, it needs the bounded openings $w_i$ tied to the same digest computation that produced $d_{\mathbf{x}}$. If a receiver could translate the same root toward a different set of leaves, it would have found a second bounded opening $w_i'$. Under the SIS assumption, no PPT adversary can produce two distinct bounded openings $w_i, w_i'$ of one node digest $y_{i-1}$, except with negligible probability; see Theorems 5 and 6.

### 5.5 Composes the Pieces Together

Now we explain what are the two properties of the (local) Shrink/Expand identity that make our high-level recursive protocol succinct.

First, the LacE and LHE materials are reusable. The LacE ciphertexts $\mathbf{C}_i$ in $\mathsf{ct}_{\mathsf{lac}}$, and in particular their selected rows $\mathbf{C}_i[\mathsf{ind}]$, are fixed in the one-time setup and are independent of the receiver's later block $\mathbf{x}$, the digest values $y_i$, and the bounded openings $w_i$. Similarly, the LHE bridge ciphertext $\mathsf{ct}_{\mathsf{r}}$ is fixed in the one-time setup and can be reused with the digest value of any block at that level. The query input only selects bounded openings on a valid digest tree; the same materials can therefore be reused across all of the blocks in a given level of the protocol-recursion tree.

Second, the identity is self-similar. A recursive call produces authenticated shares for the decomposed digest inputs $\hat{\mathbf{d}}_{\mathbf{x}}$. After recombination, Expand turns each authenticated digest label into the block IT-MAC form $\mathbf{m} = \mathbf{s} \odot \mathbf{x} + \mathbf{k}$, which has the same form expected at the next level. Thus the protocol repeatedly shrinks blocks to digest inputs, obtains authenticated shares recursively, and expands those shares back to the original blocks.

The striking point is that these are not separate mechanisms. The same translation identity from TinyLabels moves the multiplicative factor down a digest path. THe reuse and self-similarity then turn this identity into the logarithmic chosen-input VOLE protocol.

### 5.6 The Complete Shrink/Expand Interface

The simplified interface stated in the beginning of this section assume that the input and output are one and $\mu$ ring elements, respectively. Yet, the actual recursive interface uses the same cancellation pattern, but the object passed between recursive levels is not the bare digest $d_{\mathbf{x}}$. For a block $\mathbf{x} \in \mathcal{R}_q^{\mu}$, the shrink step first computes

$$d_{\mathbf{x}} := \mathsf{LacE.Digest}(\mathbf{x}) \in \mathcal{R}_q.$$

For technical reasons described in Section 6, the next recursive level receives the decomposed digest representation

$$\hat{\mathbf{d}}_{\mathbf{x}} := \mathsf{decompose}(d_{\mathbf{x}}) = \mathsf{transpose}((\mathbf{g}^{-1}(d_{\mathbf{x}}) \bmod q_t)_{t\in[\rho]}) \in \mathcal{R}_q^{\tau\rho}.$$

After the recursive call returns, the parties recombine CRT coordinates and sum over gadget digits. The result is the authenticated digest label consumed by Expand:

$$(\mathbb{S} : k_d, \mathbb{R} : m_d) \qquad \text{with} \qquad m_d = \mathsf{sk_r}^{\top}\mathbf{g}^{-1}(d_{\mathbf{x}}) + k_d.$$

Given this authenticated digest label, together with $\mathbf{x}$, $d_{\mathbf{x}}$, and the reusable LacE/LHE material, Expand outputs the block IT-MAC form

$$(\mathbb{S} : \mathbf{k}, \mathbb{R} : \mathbf{m}) \qquad \text{with} \qquad \mathbf{m} = \mathbf{s} \odot \mathbf{x} + \mathbf{k}.$$

Thus the real interface replaces the single authenticated digest label $\widetilde{m} = \widetilde{s}d_{\mathbf{x}} + \widetilde{k}$ by decomposed label pieces that match the ciphertext modulus, gadget basis, and CRT coordinates used by the next recursive call. The row translation still performs the same cancellation proved above; the lower-level realization adds the representation changes, noise handling, and rounding needed to make the cancellation correct in the actual parameter domains. Section 6 gives this realization: CRT decomposition appears in Section 6.1, gadget decomposition and noise handling in Section 6.2, and digest randomization in Section 6.4. The formal protocol boxes are given in Figures 10 and 11; Section 7 then describes the concrete optimizations.

# 6 Lower-Level Realization

The overview and Section 5 give the clean recursive picture. This section restores the concrete machinery needed to realize it: CRT decomposition keeps the recursive interface in compatible domains, gadget decomposition keeps noisy evaluations correct, and root randomization hides the receiver's chosen input while preserving the same Shrink/Expand shape.

## 6.1 CRT-Decomposition for Type Mismatch

While the high-level recursive strategy is intuitive, realizing it requires overcoming subtle technical challenges. A critical issue arises from the mismatch between the plaintext and ciphertext spaces.

**Conversion between $\mathbb{Z}_p$ and $\mathcal{R}_p$.** We start by clarifying the type of the messages and resulting shares in CI-VOLE. For our intended applications such as ZK and GC, message space $\mathcal{M} = \mathbb{Z}_p$. However, the underlying LHE and LacE schemes operate over the ring $\mathcal{M} = \mathcal{R}_p$ as in TinyLabels. TinyLabels solves it by encoding a vector of messages $\mathbb{Z}_p^n$ into the encryption scheme plaintext $\mathcal{R}_p$ via the Chinese Remainder Theorem (CRT). After obtaining the authenticated shares in $\mathcal{R}_q$, it applies inverse CRT to reconstruct the shares in $\mathbb{Z}_p$. The general case of CI-VOLE over $\mathbb{Z}_p$ reduces to CI-VOLE over $\mathcal{R}_p$ via the wrapper in Figure 4. Henceforth, we treat all values as ring elements in $\mathcal{R}_q$ or $\mathcal{R}_p$.

**Modulus Mismatch.** There is an important modulus mismatch: the ciphertexts of the underlying LHE and LacE schemes operate over $\mathcal{C} = \mathcal{R}_q$ while the messages operate over $\mathcal{M} = \mathcal{R}_p$ with $q \gg p$. In other words, the digest and keys operate in a much larger ciphertext modulus $q$ than the message modulus $p$. When the parties try to input the digests and keys in $\mathcal{R}_q$ into the next level of shrink/expand, it simply fails because the mechanism expects inputs in $\mathcal{R}_p$.

The gap between the modulus $p$ and $q$ is essential for underlying RLWE-based LHE and LacE because these schemes inject small noise into the ciphertexts to ensure security. After evaluation on the ciphertext, the noise grows. Thus, these schemes encode the message in the higher bits in the ciphertext to leave enough room for noise growth in the lower bits, ensuring the correctness of decryption. Thus, we must have $q \gg p$ to prevent the noise from growing into the higher bits and causing decryption failure.

A naïve solution is to enlarge the next level's message modulus to be $p' = q$ and increase the ciphertext modulus to $q' \gg p' = q$ accordingly. However, with recursion, these blowups compound multiplicatively, resulting in ciphertext sizes exponential in the recursion depth and linear in $|\mathbf{x}|$.

Another issue with using different moduli is that each level of shrink/expand gadgets must use LacE and LHE with different parameters, forcing $\mathbb{S}$ to send different $(\mathsf{ct}_{\mathsf{lac}}, \mathsf{ct}_{\mathsf{r}})$ and introducing another logarithmic factor in the communication.

Hence, for the efficiency of our scheme, we want to keep the message modulus $p$ and ciphertext modulus $q$ constant across all levels of recursion.

$\mathsf{LogVOLE}_{\mathbb{Z}}.\mathsf{Offline}(\mathbb{S} : \Delta \in \mathbb{Z}_p,\ \mathbb{B} : w \in \mathbb{N})$
$\to (\mathbb{S} : u \in \mathsf{State}, \mathbb{R} : v \in \mathsf{State})$

- Functionality: Parties receive the state produced by $\mathsf{LogVOLE}_{\mathcal{R}}.\mathsf{Offline}$
- Parameters: See Section Section 12.1. Let $n$ be the degree of $\mathcal{R}_q$.
- Procedures:
  1. $\mathbb{B}$: $w' := \lceil w/n \rceil$; $\delta := \lfloor q/p \rfloor$
  2. $\mathbb{S}$: $s := \mathsf{Encode}(\mathsf{Repeat}(n, \Delta)) \cdot \delta \in \mathcal{R}_q$
  3. return $\mathsf{LogVOLE}_{\mathcal{R}}.\mathsf{Offline}(\mathbb{S} : s,\ \mathbb{B} : w')$

$\mathsf{LogVOLE}_{\mathbb{Z}}.\mathsf{Online}(\mathbb{S} : u \in \mathsf{State},\ \mathbb{R} : (v \in \mathsf{State}, \mathbf{x} \in \mathbb{Z}_p^w),$
$\mathbb{B} : \mathsf{id} = (\mathsf{sid}, \mathsf{mode}),\ \mathsf{mode} \in \{\mathsf{interactive}, \mathsf{non\text{-}interactive}\})$
$\to (\mathbb{S} : \mathbf{k} \in \mathbb{Z}_p^w, \mathbb{R} : \mathbf{m} \in \mathbb{Z}_p^w)$

- Procedures:
  1. $\mathbb{R}$: $\tilde{\mathbf{x}} := \mathsf{Encode}(\mathbf{x}) \in \mathcal{R}_q^{w'}$
  2. $(\mathbb{S} : \mathbf{k}, \mathbb{R} : \mathbf{m}) \leftarrow \mathsf{LogVOLE}_{\mathcal{R}}.\mathsf{Online}(\mathbb{S} : u,\ \mathbb{R} : (v, \tilde{\mathbf{x}}),\ \mathbb{B} : \mathsf{id})$
  3. return $(\mathbb{S} : \mathbf{k} := \mathsf{Encode}^{-1}(\mathbf{k}), \mathbb{R} : \mathbf{m} := \mathsf{Encode}^{-1}(\mathbf{m}))$

Note:
- $\mathsf{Encode}$ converts vectors in $\mathbb{Z}_p^n$ to $\mathcal{R}_p$ via CRT batching [BGH13, SV14], and $\mathsf{Encode}^{-1}$ is the inverse unbatching map; we implicitly canonically lift into $\mathcal{R}_q$ when needed.

Figure 4: Wrapper protocol $\mathsf{LogVOLE}_{\mathbb{Z}}$ reducing CI-VOLE over $\mathbb{Z}_p$ to CI-VOLE over $\mathcal{R}_q$.

**Our CRT-based Decomposition Solution.** The high-level idea is to decompose the digest and keys in $\mathcal{R}_q$ into smaller values in $\mathcal{R}_p$ using CRT decomposition[5]. Since CRT preserves the algebraic structure (*i.e.*, addition and multiplication) after conversion, the resulting values still maintain the desired algebraic relations (*i.e.*, $\mathsf{sk}_{\mathbf{x}} = \mathbf{d}_{\mathbf{x}} \cdot \mathsf{sk}_1 + \mathsf{sk}_k$).

Concretely, the parties sample $\rho$ NTT-friendly primes $q_1, \ldots, q_\rho$ and set the ciphertext modulus as $q = \prod_{i \in [\rho]} q_i$. Thanks to the underlying LacE and LHE, the parties can use different plaintext moduli $p = q_1, \ldots, q_\rho$ in different ciphertexts.

When handling a query, $\mathbb{R}$ CRT-decomposes $d_{\mathbf{x}}$ into

$$d_{\mathbf{x},i} = d_{\mathbf{x}} \bmod q_i \in \mathcal{R}_{q_i} \text{ for } i \in [\rho]$$

In other words, the decomposition applies to the coefficients of the ring elements. Similarly, $\mathbb{S}$ decomposes the keys $\mathsf{sk}_{\mathbf{x}}$ and $\mathsf{sk}_k$ into

$$\mathsf{sk}_{\mathbf{x},i} = \mathsf{sk}_{\mathbf{x}} \bmod q_i \in \mathcal{R}_{q_i} \text{ and } \mathsf{sk}_{k,i} = \mathsf{sk}_k \bmod q_i \in \mathcal{R}_{q_i} \text{ for } i \in [\rho]$$

respectively. Then, they feed these decomposed values into the next level of shrink/expand instance. Suppose $\mathbb{R}$ has obtained $\mathsf{sk}_{\mathbf{x},i} = \mathbf{d}_{\mathbf{x},i} \cdot \mathsf{sk}_{1,i} + \mathsf{sk}_{k,i}$ for $i \in [\rho]$ $\mathbb{R}$ efficiently reconstructs $\mathsf{sk}_{\mathbf{x}} = \mathbf{d}_{\mathbf{x}} \cdot \mathsf{sk}_1 + \mathbf{k}'_{\mathbf{x}}$ using CRT.

The remaining problem is how to enable $\mathbb{R}$ to obtain $\mathsf{sk}_{\mathbf{x},i}$ for $i \in [\rho]$. This is slightly different from CI-VOLE, where we assume there is only one global key in each instance.

A naïve solution is to instantiate $\rho$ independent ShrinkExpand instances, where the $i$-th instance handles queries in $\mathcal{R}_{q_i}$. Yet, it will also inflate the number of digests by a factor of $\rho$, which in turn increases the input number of the next level's ShrinkExpand by a factor of $\rho$ and causes a $\rho^2$-blowup in the ciphertext size.[6]

[5]Here, CRT decomposes coefficients into smaller magnitudes, whereas in TinyLabels, CRT maps integers to ring elements.

[6]To maintain the shrinking ratio of $\alpha$, each ShrinkExpand instance should process $\rho \cdot \alpha$ inputs. Together with $\rho$ instances, the total ciphertext count increases to $O(\rho^2 \cdot \alpha)$.

We propose an optimized approach using a single $\mathsf{ShrinkExpand}_{\mathcal{R}}$ instance, requiring only $O(\alpha\rho)$ ciphertexts for a shrinking factor $\alpha = w/w'$. With the element-wise multiplication functionality, the parties compute the following using $\mathsf{ShrinkExpand}$:

$$\begin{pmatrix} \mathbf{d}_{\mathbf{x},1} \\ \vdots \\ \mathbf{d}_{\mathbf{x},\rho} \end{pmatrix} \odot \begin{pmatrix} \mathsf{sk}_{1,1} \cdot \delta_1 \\ \vdots \\ \mathsf{sk}_{1,\rho} \cdot \delta_\rho \end{pmatrix} + \begin{pmatrix} \mathbf{k}_1 \\ \vdots \\ \mathbf{k}_\rho \end{pmatrix}$$

Here, $\mathbf{d}_{\mathbf{x},i}$ and $\mathsf{sk}_{1,i}$ are each duplicated $\alpha$ times, yielding output in $\mathcal{R}_q^{\alpha\rho}$. As is standard in BFV-like schemes, $\mathsf{sk}_{1,i}$ is multiplied by $\delta_i = q'/q_i$ to lift the message for noise tolerance. Upon receiving the output from $\mathsf{ShrinkExpand}$, the parties scale down the $i$-th chunk (of $\rho$ elements) by $\delta_i$ and round, effectively obtaining the result modulo $q_i$.

**Concrete Efficiency.** We emphasize that this decomposition has only small impact on practical performance. For efficiency, BFV-like schemes typically use CRT to represent large ciphertext words as smaller ($\leq$ 64-bit) machine words; thus, constructing $q$ as a product of small primes is standard practice. In our evaluation, we use a small $\rho = 4$.

**Cost Accounting.** The CRT layer changes the width of each recursive $\mathsf{ShrinkExpand}$ block to $\mu = \alpha\rho\tau$. For an input length $w = \tau\rho\alpha^\ell$, the offline recursion has $\ell = \log_\alpha(w/(\tau\rho)) = O(\log_\alpha w)$ levels. Summing the non-root setup over these levels gives

$$O(\log_\alpha w \cdot \alpha\rho\tau \cdot \log(\alpha\rho) \cdot |\mathcal{R}_q| + \log_\alpha w \cdot \lambda)$$

bits of communication. For computation, we count $\mathsf{Mult}_{\mathcal{R}}$ because full ring multiplications dominate integer operations, ring additions, and CRT bookkeeping. The matching offline computation is

$$O(\log_\alpha w \cdot \alpha\rho\tau \cdot \log(\alpha\rho))\, \mathsf{Mult}_{\mathcal{R}}.$$

Online, the only communication outside local recursion is the root interaction, which costs $O(\tau|\mathcal{R}_q|)$ bits. The online local work is

$$O(w)\, \mathsf{Mult}_{\mathcal{R}} + O(w\tau \log \mu)\, \mathsf{Mult}_{\mathcal{R}_q}^{\mathsf{NTT}} + O(n)\, \mathbb{Z}_q\text{-arithmetic}.$$

The end-to-end accounting in Section 12.1 folds these terms together with the root wrapper and the setup-reuse optimizations.

## 6.2 Gadget Decomposition for Noise Handling

Thus far, we have omitted the details of noise management in $\mathsf{LHE}$ schemes. While addressing noise introduces some complexity, the overall structure of our construction remains the same. We assume familiarity with standard lattice-based $\mathsf{LHE}$ schemes such as BGV or BFV.

The core issue arises in $\mathsf{ShrinkExpand}$. To enable recursion and reduce problem size, $\mathbb{R}$ computes $\mathsf{LHE.Enc}(\mathsf{sk}_{\mathbf{x}} \cdot d_{\mathbf{x}}, \mathbf{r} \cdot d_{\mathbf{x}})$ by evaluating $\mathsf{ct}_1 \cdot d_{\mathbf{x}}$. However, since $d_{\mathbf{x}}$ is a pseudo-random element in $\mathcal{R}_q$ with large coefficients, this multiplication amplifies the noise in $\mathsf{ct}_1$ beyond the decryption bound, destroying the plaintext messages in $\mathbf{r} \cdot d_{\mathbf{x}}$ and rendering the resulting ciphertext invalid.

**Gadget Decomposition.** To address this, $\mathsf{TinyLabels}$ employs the gadget decomposition technique. Consequently, our "compressed" decryption key is no longer a share of the digest multiplied by a global key, but rather a share of the inner product of the decomposed digests. We detail this below.

Following $\mathsf{TinyLabels}$, the $\mathsf{LHE}$ public parameters include a small integer base $g$, chosen such that the noise budget of $\mathsf{ct}_1$ tolerates multiplication by $g$. During $\mathsf{LHE}$ evaluation, $\mathbb{R}$ decomposes $d_{\mathbf{x}}$

into digits $b_1, \ldots, b_\tau$ where $\tau = \lceil \log_g q \rceil$, satisfying $d_{\mathbf{x}} = \sum_{i=1}^{\tau} b_i g^{i-1}$ and $\|b_i\|_\infty \leq g$. We denote this decomposition as $\mathbf{g}^{-1}(d_{\mathbf{x}}) = (b_1, \ldots, b_\tau)$ and the gadget vector as $\mathbf{g} = (1, g, \ldots, g^{\tau-1})$, such that $d_{\mathbf{x}} = \mathbf{g}^T \cdot \mathbf{g}^{-1}(d_{\mathbf{x}})$. Here, the inner product is defined as $\mathbf{a}^T \cdot \mathbf{b} = \sum_i \mathbf{a}[i] \cdot \mathbf{b}[i]$, and $\mathbf{a} \cdot \mathbf{b}^T$ denotes outer product, *i.e.*, $(\mathbf{a} \cdot \mathbf{b}^T)[i][j] = \mathbf{a}[i] \cdot \mathbf{b}[i]$.

$$\mathsf{ct}_{\mathsf{r}} = \mathbf{a} \cdot \mathsf{sk}_{\mathbf{x}}^T + \mathbf{r} \cdot \mathbf{g} + \mathbf{E} \in \mathcal{R}_q^{w \times \tau}$$

where $\mathbf{a}, \mathbf{r} \in \mathcal{R}_q^\tau$ and $\mathsf{sk}_{\mathbf{x}} \in \mathcal{R}_q^\tau$.

$\mathbb{R}$ then computes a $\mathsf{LHE}$ encryption of $\mathbf{r} \cdot d_{\mathbf{x}}$ row-by-row. For each $i \in [w]$:

$$\begin{aligned}
\mathsf{ct}_1[i] \cdot \mathbf{g}^{-1}(d_{\mathbf{x}}) &= (\mathbf{a}[i] \cdot \mathsf{sk}_{\mathbf{x}}^T + \mathbf{r}[i] \cdot \mathbf{g}^T + \mathbf{E}[i]) \cdot \mathbf{g}^{-1}(d_{\mathbf{x}}) \\
&= \mathbf{a}[i] \cdot (\mathsf{sk}_{\mathbf{x}}^T \cdot \mathbf{g}^{-1}(d_{\mathbf{x}})) + \mathbf{r}[i] \cdot (\mathbf{g}^T \cdot \mathbf{g}^{-1}(d_{\mathbf{x}})) + \mathbf{E}' \\
&= \mathbf{a}[i] \cdot (\mathsf{sk}_{\mathbf{x}}^T \cdot \mathbf{g}^{-1}(d_{\mathbf{x}})) + \mathbf{r}[i] \cdot d_{\mathbf{x}} + \mathbf{E}'
\end{aligned}$$

where $\mathsf{ct}_1[i]$ denotes the $i$-th row of $\mathsf{ct}_1$ (a vector of size $\tau$). Since $\|\mathbf{g}^{-1}(d_{\mathbf{x}})\|_\infty \leq g$, the error term $\mathbf{E}'$ remains small enough for correct decryption. Thus, $\mathbb{R}$ obtains an encryption of $\mathbf{r} \cdot d_{\mathbf{x}}$ under the key $\mathsf{sk}_{\mathbf{x}}^T \cdot \mathbf{g}^{-1}(d_{\mathbf{x}}) = \sum_{i \in [\tau]} \mathsf{sk}_{\mathbf{x}}[i] \cdot \mathbf{g}^{-1}(d_{\mathbf{x}})[i]$.

**Incorporating Gadget Decomposition into Shrink/Expand.** Although the decryption key is now an inner product, we can still adapt our previously proposed shrinking technique. Observe that if the parties obtain shares $\langle \mathsf{sk}_{\mathbf{x}}[i] \cdot \mathbf{g}^{-1}(d_{\mathbf{x}})[i] \rangle$ for each $i \in [\tau]$, they can locally sum them to recover the share $\langle \mathsf{sk}_{\mathbf{x}}^T \cdot \mathbf{g}^{-1}(d_{\mathbf{x}}) \rangle = \sum_{i \in [\tau]} \langle \mathsf{sk}_{\mathbf{x}}[i] \cdot \mathbf{g}^{-1}(d_{\mathbf{x}})[i] \rangle$ as the inner-product key. To achieve this, we extend the element-wise multiplication trick from Section 6.1 to generate these shares in parallel. Specifically, they invoke $\mathsf{ShrinkExpand}$ to compute the following terms, with $\alpha$-fold duplication for shrinking:

$$\begin{pmatrix} \vdots \\ \mathbf{g}^{-1}(\mathbf{d}_{\mathbf{x}})[i] \bmod q_j \\ \vdots \end{pmatrix} \odot \begin{pmatrix} \vdots \\ \mathsf{sk}_{\mathbf{x}}[i] \cdot \delta_j \\ \vdots \end{pmatrix} + \begin{pmatrix} \vdots \\ \mathbf{k}_{i,j} \\ \vdots \end{pmatrix} \text{ where } i \in [\tau], j \in [\rho]$$

Upon receiving the results from $\mathsf{ShrinkExpand}$, the parties perform rescaling and rounding, CRT-compose the results from different moduli, and sum the additive shares to obtain their inner-product decryption keys.

**Remark on Noise Accumulation.** We emphasize that at the end of each recursion level, the parties perform scale-and-round, as in $\mathsf{TinyLabels}$, to obtain *noise-free* shares of $\langle \mathsf{sk}_{\mathbf{x}}^T \cdot \mathbf{g}^{-1}(d_{\mathbf{x}}) \rangle$, which can then be used to derive more noise-free labels at the lower level. Consequently, no noise accumulates with the depth of recursion.

**Parameter Choice.** Setting $g \approx q^\epsilon$ is sufficient to control noise. Thus, we have $\tau = O(1/\epsilon)$. In our evaluation, we use $\tau = 2$. Section 7.1 proposes a further optimization to reduce the multiplicative cost impact from gadget decomposition from $\tau$ to $\tau - 1$.

### 6.3 Summary of Transition Between Levels

We summarize the transition between two $\mathsf{ShrinkExpand}$ levels and how to choose the proper parameters to ensure efficient recursion. Suppose the input and output sizes of the current level are $w$ and $w'$, respectively, and the parties aim to shrink the input size by a factor of $\alpha = w/w' > 1$ (*e.g.*, 2). Throughout this summary, for a digest $d \in \mathcal{R}_q$, we write

$$\mathsf{decompose}(d) := \mathsf{transpose}((\mathbf{g}^{-1}(d) \bmod q_t)_{t \in [\rho]}) \in \mathcal{R}_q^{\tau\rho}.$$

**Pre-Shrink.** The current $\mathsf{Shrink}$ handles a vector in $\mathcal{R}_p^w$.[7] For the $\mathsf{ShrinkExpand}$ at this level, the parties set the shrink instance input size to be $\mu := \alpha\tau\rho$. Then, to handle the input $\mathbf{x}$ of length $w$, they partition $\mathbf{x}$ into $b := w/\mu$ blocks of size $\mu$ each, denoted as $\mathbf{x}_1, \ldots, \mathbf{x}_b \in \mathcal{R}_q^\mu$.

**Shrink.** For each block, the local shrink procedure computes one digest $d_i := \mathsf{Shrink}(\mathbf{x}_i) \in \mathcal{R}_q$ for $i \in [b]$.

**Post-Shrink.** $\mathbb{R}$ locally decomposes each $d_i$ into $\tau$ digits $g^{-1}(d_i) \in \mathcal{R}_q^\tau$ and applies CRT to obtain

$$\hat{d}_i := \mathsf{decompose}(d_i) \in \mathcal{R}_q^{\tau\rho}.$$

Stacking these $\hat{d}_i$ gives the actual recursive input

$$\hat{\mathbf{d}} := (\hat{d}_1 \parallel \cdots \parallel \hat{d}_b) \in \mathcal{R}_q^{b\tau\rho} = \mathcal{R}_q^{w/\alpha}.$$

This is exactly why we set $\mu = \alpha\tau\rho$: one block of length $\mu$ shrinks to one digest, but then the two decomposition mechanisms inflate the digest to $\tau\rho$ $\mathcal{R}_p$ elements, so the length drops from $w$ to $w/\alpha$.

Then, $\hat{\mathbf{d}}$ is fed to the next $\mathsf{Shrink}$ instance at the next level (and is renamed as $\mathbf{x}_i$).

When $\mathbb{R}$ can no longer shrink the digests ($d_{\mathsf{rt}}$), the parties reach **the root level**. $\mathbb{R}$ randomizes this digest as $d'_{\mathsf{rt}}$ and sends it to $\mathbb{S}$, who then computes and returns the key $\mathsf{sk}_{\mathsf{rt}}$ to $\mathbb{R}$. In other words, the parties get

$$(\mathbb{S} : k_{\mathsf{rt}}, \mathbb{R} : \mathsf{sk}_{\mathsf{rt}}) \qquad \text{such that} \qquad \mathsf{sk}_{\mathsf{rt}} = \mathsf{sk}_{\mathsf{r}}{}^\top \cdot \mathbf{g}^{-1}(d_{\mathsf{rt}}) + k_{\mathsf{rt}}.$$

**Return from recursion.** The recursive call on $\hat{\mathbf{d}}$ returns, for each block $i \in [b]$,

$$(\mathbb{S} : \hat{k}_i, \mathbb{R} : \hat{m}_i) \in \mathcal{R}_p^{\tau\rho} \times \mathcal{R}_p^{\tau\rho}$$

such that for every gadget digit $\ell \in [\tau]$ and CRT modulus $j \in [\rho]$,

$$\hat{m}_i[\ell, j] = \mathsf{sk}_{\mathsf{r}}[\ell] \cdot \mathbf{g}^{-1}(d_i)[\ell] + \hat{k}_i[\ell, j] \bmod q_j$$

**Pre-Expand.** The parties then locally recombine the CRT coordinates for each $\ell$, obtaining

$$\mathbb{S} : k_i[\ell] \qquad \mathbb{R} : m_i[\ell] = \mathsf{sk}_{\mathsf{r}}[\ell] \cdot \mathbf{g}^{-1}(d_i)[\ell] + k_i[\ell]$$

Finally, summing these $\tau$ gadget digit shares gives the short digest-side relation needed by local expand:

$$\mathbb{R} : m_{d_i} = \sum_{\ell \in [\tau]} m_i[\ell] = \mathsf{sk}_{\mathsf{r}}{}^T \cdot \mathbf{g}^{-1}(d_i) + k_{d_i}, \quad \mathbb{S} : k_{d_i} := \sum_{\ell \in [\tau]} k_i[\ell]$$

**Expand.** When invoking the $\mathsf{Expand}$ procedures, $\mathbb{S}$ inputs the block $\mathbf{x}_i \in \mathcal{R}_q^\mu$ together with $\mathsf{sk}_{x_i} = m_{d_i}$ and the digest $d_i$, and $\mathbb{R}$ inputs $\mathsf{sk}_k = k_{d_i}$. $\mathsf{Expand}$ outputs

$$(\mathbb{S} : \tilde{\mathbf{k}}_i, \mathbb{R} : \tilde{\mathbf{m}}_i) \in \mathcal{R}_q^\mu \times \mathcal{R}_q^\mu \quad \text{such that} \quad \tilde{\mathbf{m}}_i = \mathbf{s} \odot \mathbf{x}_i + \tilde{\mathbf{k}}_i.$$

The global key $\mathbf{s}$ is set during the setup, and its value depends on the current level. At the base level, $\mathbf{s}_i = (\Delta, \ldots, \Delta)$, *i.e.*, the authentication key of the application duplicated $\mu$ times. For non-base levels, $\mathbf{s} = \mathsf{Repeat}((\mathsf{sk}_{r,i} \bmod q_j)_{i \in [\tau], j \in [\rho]}, \alpha)$ where $\mathsf{sk}_{r,i} \leftarrow \mathcal{R}_q$ is the decryption key for $\mathsf{LHE}$ ciphertexts.

[7] To be pedantic, at non-base levels, it would be a vector of elements in $\mathcal{R}_{q_1}, \mathcal{R}_{q_2}, \ldots, \mathcal{R}_{q_\rho}$ where $q_1 \approx \cdots \approx q_\rho \approx p$

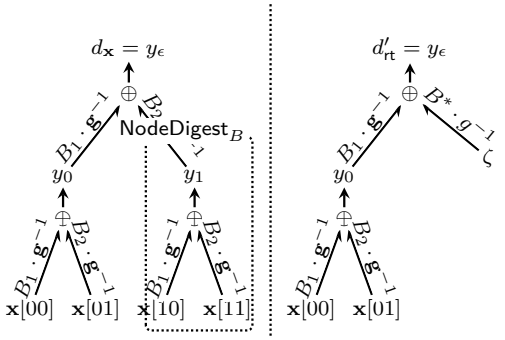


Figure 5: Digest computation trees. The left panel is the ordinary binary-tree recursion computing $d_{\mathbf{x}} = y_\epsilon$; the dotted box highlights one representative node-digest computation $\mathsf{NodeDigest}_B$. The right panel illustrates the randomized root computation: the left subtree is unchanged, the right subtree is replaced by an input $\zeta$, and the right incoming edge to the root is labeled $B^* \cdot g^{-1}$.

**Post-Expand.** Then, $\mathbb{S}$ and $\mathbb{R}$ input $\tilde{\mathbf{k}}_i$ and $\tilde{\mathbf{m}}_i$, respectively, to their local pre-expand gadget as the return of the recursion.

**Remark on Noise Accumulation.** We emphasize that at the end of the Expand procedures, the parties perform scale-and-round, as in TinyLabels, to obtain *noise-free* output, so noise does not accumulate as the recursion unwinds.

### 6.4 Wiring the Randomized Root Digest

The digest-security section shows that the receiver can publish a randomized root digest without revealing its input; see Section A.2. This subsection explains the protocol mechanics needed to use that randomized root while still expanding back to the ordinary root relation.

Recall that the ordinary recursion shrinks the receiver's blocks until it obtains a single root digest $d_{\mathsf{rt}} \in \mathcal{R}_q$. At the top we replace the ordinary binary node by the randomized node analyzed in Section A.2:

$$\mathsf{RandNodeDigest}^{\tau,m}_{B'}(u, \zeta) := B_1 \cdot \mathbf{g}^{-1}(u) + B^* \cdot \zeta,$$

where $B_1 \leftarrow \mathcal{R}_q^\tau, B^* \leftarrow \mathcal{R}_q^m$ and $B' = (B_1 \quad B^*)$.

**Embedding the randomized node.** We use $\mathsf{RandNodeDigest}_{\tau,m}$ only once, at the root of the digest tree. As in the standard construction, the receiver decomposes $d_{\mathsf{rt}}$ as $\hat{d}_{\mathsf{rt}} := \mathsf{decompose}(d_{\mathsf{rt}}) \in \mathcal{R}_q^{\tau\rho}$. We then use a top wrapper tree with $\mu = 2\tau\rho$ inputs. The $\tau\rho$ entries of $\hat{d}_{\mathsf{rt}}$ are placed at the leaves of the left half of this tree, and $\mathbb{R}$ evaluates the ordinary digest recursion on that left subtree using the usual node digest, bottom-up exactly as in the original construction. We denote the resulting subtree root by $y_{\mathsf{left}} \in \mathcal{R}_q$. Thus, $y_{\mathsf{left}}$ is exactly the value that the unmodified construction would pass as the left child of the root node.

The change is that we do not build the right subtree at all. Instead, the root is an instance of $\mathsf{RandNodeDigest}_{\tau,m}$. Concretely, $\mathbb{R}$ samples $\zeta \in \mathcal{R}_q^m$ such that $\|\zeta\|_\infty < g$. For our concrete parameters, Section A.2 shows that $m = 3$ suffices for hiding. Then, we define the new root digest as $y_\epsilon := \mathsf{RandNodeDigest}^{\tau,m}_{B'}(y_{\mathsf{left}}, \zeta)$. This is the randomized digest $d'_{\mathsf{rt}} = y_\epsilon$.

Then, the parties run the usual ping-pong protocol on $d'_{\mathsf{rt}}$ so that $\mathbb{R}$ obtains the short randomized root key $\mathsf{sk}'_{\mathsf{rt}} = \mathsf{sk}_{\mathsf{r}} \cdot g^{-1}(d'_{\mathsf{rt}}) + \mathsf{sk}'_k$.

**De-randomization via a modified shrink/expand.** To convert the key share for $d'_{\mathsf{rt}}$ back into one for $d_{\mathsf{rt}}$, we modify only the top level ciphertexts in $\mathsf{LacE}$ used by $\mathsf{ShrinkExpand}$, *i.e.*, $\mathbf{C}_1$. All lower levels remain unchanged. The ciphertext $\mathbf{C}_1[\mathsf{ind}]$ is changed to

$$\mathbf{C}_1[\mathsf{ind}] := \mathbf{r}^{(1)}[\mathsf{ind}] \cdot B' + \mathbf{r}^{(2)}[\mathsf{ind}] \cdot (\overline{\mathsf{ind}}_1 \cdot \mathbf{g}^\top \; 0^\top) + \mathbf{E}_1[\mathsf{ind}] \in \mathcal{R}_q^{\tau+m}.$$

Evaluation is unchanged except that the top level now pairs $\mathbf{C}_1[\mathsf{ind}]$ with $(\mathbf{g}^{-1}(y_{\mathsf{left}}) \,\|\, \zeta)$. Hence the public-key term reconstructs the new root value $y_\epsilon$, while the selector term still extracts only the left child $y_{\mathsf{left}}$. In particular, for the rows with $\mathsf{ind}_1 = 0$,

$$\mathbf{C}_1[\mathsf{ind}] \cdot \begin{pmatrix} -\mathbf{g}^{-1}(y_{\mathsf{left}}) \\ -\zeta \end{pmatrix} = -\mathbf{r}^{(1)}[\mathsf{ind}] \cdot y_\epsilon - \mathbf{r}^{(2)}[\mathsf{ind}] \cdot y_{\mathsf{left}} + \mathbf{E}'_1[\mathsf{ind}].$$

The second term is exactly the child value passed to the lower levels, so the same telescoping cancellation goes through as in ordinary $\mathsf{LacE}$: the $\mathbf{r}^{(2)}[\mathsf{ind}] \cdot y_{\mathsf{left}}$ term cancels with the first term coming from the left subtree, and the remaining top mask is $\mathbf{r}^{(1)}[\mathsf{ind}] \cdot y_\epsilon$. Therefore the modified expand procedure still returns authenticated shares of the lifted left block $\hat{d}_{\mathsf{rt}}$, from which the parties recombine the desired relation for $d_{\mathsf{rt}}$.

As in the usual $\mathsf{ShrinkExpand}$ gadget, $\mathbb{S}$ must additionally provide one fresh $\mathsf{LHE}$ encryption of the new top randomness $\mathbf{r}^{(1)}$.

**Optimizations.** First, $\mathbb{R}$ never evaluates the deleted right subtree, so all rows and procedures for indices with leading bit 1 can be skipped. This halves the setup communication and computation. Second, only $\mathbf{C}_1$ changes: the lower matrices $\mathbf{C}_2, \ldots, \mathbf{C}_\ell$ are identical to those of an ordinary recursion level and can therefore be reused, which saves setup communication. Hence, the only new setup material is the modified top matrix, the public matrix $B^*$, and the extra $\mathsf{LHE}$ encryption of $\mathbf{r}^{(1)}$.

# 7 Optimizations for Practical Efficiency

## 7.1 One-Digit Gadget Truncation

Recall that in Section 6.1, we explained that our construction requires gadget decomposition to contain the noise growth in $\mathsf{LacE}$ and $\mathsf{LHE}$ evaluations. The gadget decomposition is the main source of the $\tau$-factor overhead in the heaviest objects in the protocol, particularly in the $\mathsf{LacE}$ ciphertext $\mathsf{ct}_{\mathsf{lac}}$.

Here is a simple calculation: our $\mathsf{ShrinkExpand}$ construction with input size $\mu = \alpha \cdot \tau \cdot \rho$ has $\mu \log_2 \mu$ nodes in the laconic ciphertexts, and each node has $\tau$ ciphertexts for the gadget decomposition. Hence, there are $O(\tau^2 \log \tau)$ ciphertexts for the setup. In terms of computation, each $\mathsf{ShrinkExpand}$ requires $\mu \cdot O(\tau \log \tau)$ ring operations. Combining with the fact that processing a $\mathsf{CI\text{-}VOLE}$ query requires $2 \cdot w/\mu$ invocations of $\mathsf{ShrinkExpand}$, the computation overhead of a query is $w \cdot O(\tau \log \tau)$ ring operations.

With our truncation optimization, we can reduce the $\tau$ factor in overhead to $\tau - 1$, *i.e.*, replacing $\tau$ by $\tau - 1$ in the above cost calculations. Concretely, as we use $\tau = 2$ in our implementation, this optimization reduces the setup communication by $\approx 4\times$ and computation by $\approx 2\times$.

**At the high level** , our trick is to omit the least-significant gadget digit in the $\mathsf{LHE}$ and $\mathsf{LacE}$ evaluations. This introduces additional error in the resulting ciphertexts, but it can still be absorbed into the existing noise margin.

**Truncating LHE.** Recall that in $\mathsf{LHE}$, the base-$g$ decomposition is $d_{\mathbf{x}} = \sum_{i=1}^{\tau} b_i g^{i-1}$ with $\|b_i\|_\infty \le g$ and $\mathbf{g}^{-1}(d_{\mathbf{x}}) = (b_1, \ldots, b_\tau)$. Define $\mathsf{low}_g(d_{\mathbf{x}}) := b_1$ and $\mathsf{high}_g(d_{\mathbf{x}}) := d_{\mathbf{x}} - \mathsf{low}_g(d_{\mathbf{x}}) = \sum_{i=2}^{\tau} b_i g^{i-1}$, *i.e.*, $\mathsf{low}_g(d_{\mathbf{x}})$ is the least-significant gadget digit and $\mathsf{high}_g(d_{\mathbf{x}})$ is the composed value without $\mathsf{low}_g(d_{\mathbf{x}})$. Let $\mathbf{g}_{\mathsf{hi}}^{-1}(d_{\mathbf{x}}) := (b_2, \ldots, b_\tau)$ together with the truncated gadget vector $\mathbf{g}_{\mathsf{hi}} := (g, g^2, \ldots, g^{\tau-1})$.

In the $\mathsf{LHE}$ component, instead of using the full gadget vector $\mathbf{g}$ for the message $\mathbf{r}$, $\mathbb{S}$ omit the least-significant gadget column and publish $\mathsf{ct}_1^{\mathsf{hi}} \in \mathcal{R}_q^{w \times (\tau-1)}$ of the form

$$\mathsf{ct}_1^{\mathsf{hi}} := \mathsf{LHE.Enc}(\mathsf{sk}_{\mathbf{x}}, \mathbf{r}) = \mathbf{a} \cdot \mathsf{sk}_{\mathbf{x}}^T + \mathbf{r} \cdot \mathbf{g}_{\mathsf{hi}}^T + \mathbf{E},$$

so that the receiver computes, row-by-row,

$$\begin{aligned}
\mathsf{ct}_1^{\mathsf{hi}}[i] \cdot \mathbf{g}_{\mathsf{hi}}^{-1}(d_{\mathbf{x}}) &= (\mathbf{a}[i] \cdot \mathsf{sk}_{\mathbf{x}}^T + \mathbf{r}[i] \cdot \mathbf{g}_{\mathsf{hi}}^T + \mathbf{E}[i]) \cdot \mathbf{g}_{\mathsf{hi}}^{-1}(d_{\mathbf{x}}) \\
&= \mathbf{a}[i] \cdot (\mathsf{sk}_{\mathbf{x}}^T \cdot \mathbf{g}_{\mathsf{hi}}^{-1}(d_{\mathbf{x}})) + \mathbf{r}[i] \cdot (\mathbf{g}_{\mathsf{hi}}^T \mathbf{g}_{\mathsf{hi}}^{-1}(d_{\mathbf{x}})) + \mathbf{E}'_i \\
&= \mathbf{a}[i] \cdot (\mathsf{sk}_{\mathbf{x}}^T \cdot \mathbf{g}_{\mathsf{hi}}^{-1}(d_{\mathbf{x}})) + \mathbf{r}[i] \cdot \mathsf{high}_g(d_{\mathbf{x}}) + \mathbf{E}'_i,
\end{aligned}$$

where $\mathbf{E}'_i := \mathbf{E}[i] \cdot \mathbf{g}_{\mathsf{hi}}^{-1}(d_{\mathbf{x}})$ is covered by the same gadget-noise analysis as in Section 6.2. Observe that compared with the full evaluation (which has $\mathbf{r}[i] \cdot d_{\mathbf{x}}$), the only difference in the message is an *additional* error term $e_{\mathsf{drop},i} = \mathbf{r}[i] \cdot d_{\mathbf{x}} - \mathbf{r}[i] \cdot \mathsf{high}_g(d_{\mathbf{x}}) = \mathbf{r}[i] \cdot \mathsf{low}_g(d_{\mathbf{x}})$.

To make this error term small, $\mathbb{S}$ samples the laconic mask $\mathbf{r}$ from $\mathsf{LacE}$ error distribution $\overline{\mathcal{D}}$, as in Figure 16, so that the effect of dropping one digit can be absorbed into the existing noise margin. Note that $\|e_{\mathsf{drop},i}\|_\infty = \|\mathbf{r}[i] \cdot \mathsf{low}_g(d_{\mathbf{x}})\|_\infty \le n \cdot \|\mathbf{r}[i]\|_\infty \cdot \|\mathsf{low}_g(d_{\mathbf{x}})\|_\infty \le n \cdot g \cdot \max \overline{\mathcal{D}}$ because $\|\mathsf{low}_g(d_{\mathbf{x}})\|_\infty \le g$ and $\|\mathbf{r}[i]\|_\infty \le \max \overline{\mathcal{D}}$. This extra term therefore contributes an additive $ng \max \overline{\mathcal{D}}$ term to the correctness margin.

**Truncating LacE.** An analogous truncation applies to the LacE evaluation in Figure 16. Let $\ell := \log_2 w$ and write $\mathsf{ct}_{\mathsf{lac}}^{\mathsf{hi}} := (\mathbf{C}_0, \ldots, \mathbf{C}_{\ell-1})$. Given an input vector $\mathbf{x} \in \mathcal{R}_q^w$, $\mathsf{LacE}_{\mathsf{hi}}.\mathsf{Eval}$ first computes the intermediate digests $y_{\mathsf{pre}}$ as in $\mathsf{LacE}_{\mathsf{hi}}.\mathsf{Digest}$, and then for each leaf index $\mathsf{ind} \in \{0,1\}^\ell$ forms the internal-node summands

$$\mathbf{C}_i[\mathsf{ind}] \cdot \begin{pmatrix} -\mathbf{g}_{\mathsf{hi}}^{-1}(y_{\mathsf{ind}_{:i} \| 0}) \\ -\mathbf{g}_{\mathsf{hi}}^{-1}(y_{\mathsf{ind}_{:i} \| 1}) \end{pmatrix} \qquad \forall i \in [0, \ell - 2]$$

Unfolding one such term using the ciphertext structure (the gadget-indexed block in $\mathbf{C}_i[\mathsf{ind}]$ selects the $i$-th bit of $\mathsf{ind}$) yields the gadget factor

$$-\mathbf{g}_{\mathsf{hi}}^T \mathbf{g}_{\mathsf{hi}}^{-1}(y_{\mathsf{ind}_{:i} \| \mathsf{ind}_i}) = -\mathsf{high}_g(y_{\mathsf{ind}_{:i} \| \mathsf{ind}_i}) = -y_{\mathsf{ind}_{:i} \| \mathsf{ind}_i} + \mathsf{low}_g(y_{\mathsf{ind}_{:i} \| \mathsf{ind}_i}),$$

whereas the baseline (full-gadget) evaluation would contribute $-y_{\mathsf{ind}_{:i} \| \mathsf{ind}_i}$. Thus, relative to the baseline, truncation injects an additive discrepancy term

$$e_{\mathsf{drop}}^{(i)}(\mathsf{ind}) := \mathbf{r}^{(i+1)}[\mathsf{ind}] \cdot \mathsf{low}_g(y_{\mathsf{ind}_{:i} \| \mathsf{ind}_i}),$$

where $\mathbf{r}^{(i+1)}[\mathsf{ind}] \in \mathcal{R}_q$ is the corresponding mask at internal level $i+1$, sampled from $\overline{\mathcal{D}}$. Using $\|uv\|_\infty \le n \cdot \|u\|_\infty \cdot \|v\|_\infty$, $\|\mathbf{r}^{(i+1)}[\mathsf{ind}]\|_\infty \le \max \overline{\mathcal{D}}$, and $\|\mathsf{low}_g(\cdot)\|_\infty \le g$, each such contribution satisfies $\|e_{\mathsf{drop}}^{(i)}(\mathsf{ind})\|_\infty \le ng \max \overline{\mathcal{D}}$. An evaluation path has length at most $\ell - 1 = \log_2 w - 1$, so the total additional truncation error per output coordinate is bounded by

$$\|\textstyle\sum_{i=0}^{\ell-2} e_{\mathsf{drop}}^{(i)}(\mathsf{ind})\|_\infty \le ng(\max \overline{\mathcal{D}})(\ell - 1) \le ng(\max \overline{\mathcal{D}}) \log_2 \mu,$$

which is absorbed into the existing $\mathsf{LacE}$ noise budget.

**Security.** Most changes in this optimization can be understood as skipping the least-significant gadget digit in their evaluations, and thus do not affect the security. The security-relevant change is that $\mathsf{LacE}$ is now encrypted using masks $\{\mathbf{r}^{(i)}\}_{i\in[\ell]}$ from the error distribution $\overline{\mathcal{D}}$ instead of the uniform distribution in $\mathcal{R}_q$. A concern is that an adversary might be able to find a collision of the digests, allowing a malicious $\mathbb{R}$ to use two different inputs $\mathbf{x}$ and $\mathbf{x}'$ but with similar digests $d_{\mathbf{x}}$ and $d_{\mathbf{x}'}$ such that the high-digit part of the digest is the same and $(d_{\mathbf{x}} - d_{\mathbf{x}'})\mathbf{r}$ is small enough to preserve the message. Then, this $\mathbb{R}$ can locally evaluate $\mathsf{ShrinkExpand}$ for these two inputs and obtain the correlated randomness for both inputs using the same decryption key $\mathsf{sk}_{\mathbf{x}}$, and then solve for $\Delta$.

Fortunately, even with the short $\{\mathbf{r}^{(i)}\}$, our protocol still enjoys the same security guarantee as the full-gadget version. It is computationally hard for the adversary to find such collisions under a variant of SIS assumption, which can be reduced to the standard SIS assumption with a slight loss in the security parameter [DSH21]. In other words, for a fixed decryption key $\mathsf{sk}_{\mathbf{x}}$, $\mathbb{R}$ is still computationally bound to an input $\mathbf{x}$ as in the full-gadget version.

### 7.2 Golden-Seed Grinding

The ideas in the overview (Section 4 and Section 6) achieve an asymptotic communication complexity $O(\lambda^3)$. Here, we go one step further and introduce an optimization that can reduce the communication cost to $O(\lambda \log^2 |\mathbf{x}|)$ and enable us to use smaller RLWE parameters for better concrete performance.

**Background about avoiding $\mathsf{ct}_k$ transmission.** We first explain the RO optimization in $\mathsf{TinyLabels}$ used to avoid transmitting the large ciphertext $\mathsf{ct}_k$. To sample a random $\mathsf{ct}_k$ without transmitting it, both parties agree on a nonce $v$ and set $\mathsf{ct}_k[j] = H(v||j)$ for each $j \in [w']$ where $w'$ is the number of ring elements in $\mathsf{ct}_k$. The ciphertext has the form $\mathsf{ct}_k = \mathbf{a} \cdot \mathsf{sk}_k + \delta \cdot \mathsf{msg} + e$ as in BFV-like schemes, where $\mathbf{a}$ is the public random element, $\mathsf{sk}_k$ is the secret key, $\delta = q/p$ is the noise scaling factor, $\mathsf{msg}$ is the message, $e$ is the noise term, and $p$ and $q$ are the message and ciphertext moduli, respectively.

However, an important nuance is that $\mathsf{ct}_k$ is uniformly chosen, so $e$ is also uniformly distributed in $R_\delta$. Then, $\mathsf{ct}_k$ is added to $\mathsf{ct}_r \cdot d_{\mathbf{x}}$ and $\mathsf{ct}_{\mathsf{lac}}$ to produce the final ciphertext

$$\mathsf{ct}_{\mathsf{res}} = \mathsf{ct}_k + \mathsf{ct}_r \cdot d_{\mathbf{x}} + \mathsf{ct}_{\mathsf{lac}}$$

The issue is that if the noise in $\mathsf{ct}_{\mathsf{res}}$, *i.e.*,

$$\mathrm{noise}(\mathsf{ct}_{\mathsf{res}}) = \mathrm{noise}(\mathsf{ct}_k) + \mathrm{noise}(\mathsf{ct}_r) \cdot g^{-1}(d_{\mathbf{x}}) + \mathrm{noise}(\mathsf{ct}_{\mathsf{lac}})$$

is too large, decryption will fail, meaning $\mathbb{R}$ cannot obtain the correct share of $\mathbf{x} \cdot \Delta$. To resolve this issue, $\mathsf{TinyLabels}$ proposes a theoretical solution, where the parties estimate the noise bound $B$ of $\mathsf{ct}_r \cdot g^{-1}(d_{\mathbf{x}})$ and $\mathsf{ct}_{\mathsf{lac}}$ and ensure that there is an exponential gap $2^\lambda$ between $\delta$ and $B$, *i.e.*, $\delta > B \cdot 2^\lambda$, to ensure decryption correctness. Only if the noise in $\mathsf{ct}_k$ is in the "bad intervals"[8] that are very close to $\delta/2$, the noise from $\mathsf{ct}_r \cdot d_{\mathbf{x}} + \mathsf{ct}_{\mathsf{lac}} < B$ could nudge $\mathsf{ct}_{\mathsf{res}}$'s noise over $\delta/2$ and cause decryption failure, which happens with $< 2^{-\lambda}$ probability.

While it achieves good asymptotic communication by eliminating the transmission of $\mathsf{ct}_k$, it requires $\log q = \Theta(\lambda)$. It further inflates the ring degree $n = O(\lambda \log q) = O(\lambda^2)$ for security [ACC+19] and thus the ring element size to $O(n \log q) = O(\lambda^3)$, leading to larger communication and computation costs.

[8] more precisely, the bad intervals are $[-\delta/2 + B, -\delta/2) \cup (\delta/2 - B, \delta/2)$ over its entire noise interval $[-\delta/2, \delta/2)$

**TinyLabels's Rejection Sampling and Issues.** The goal now is to arrive at a moderate set of RLWE parameters for good concrete efficiency. Instead of having a construction entirely $\mathsf{ct}_k$-free, TinyLabels uses rejection sampling to strike a middle ground where for each *coefficient* in $\mathsf{ct}_k$, $\mathbb{S}$ only sends a few bits. Specifically, on top of a seed $\varsigma$ (serving as a nonce), the parties use a counter value $d_{i,j}$ to derive each coefficient of $\mathsf{ct}_k$ as $\mathsf{ct}_k[i,j] = H(\varsigma||i||j||d_{i,j})$. Notably, $\mathbb{S}$ must ensure $\mathsf{ct}_k[i,j]$'s noise is in the "good interval" $[-\delta/2 + B, \delta/2 - B]$. Hence, $\mathbb{S}$ would use its $\mathsf{sk}_k$ to decrypt the sampled $\mathsf{ct}_k[i,j]$ to learn the noise value. If the noise is too large, it increments $d_{i,j}$ and resamples $\mathsf{ct}_k[i,j]$ until its noise falls in the good interval. After the sampling, $\mathbb{S}$ additionally sends $(d_{i,j})_{i,j}$ to $\mathbb{R}$. Because correctness is guaranteed by resampling, the gap between $\delta$ and $B$ need not be $2^\lambda$ but rather a small constant 2. Each coefficient on average requires 2 sampling attempts, meaning sending $(d_{i,j})_{i,j}$ requires $\approx 2N$ bits where $N$ is the total number of coefficients in $\mathsf{ct}_k$.

While it achieves good concrete efficiency, sending $\approx 2N = O(|\mathbf{x}|)$ bits immediately defeats our purpose of achieving succinct communication.

**Our Golden-Seed Trick.** Here, we introduce our "golden-seed" grinding trick to achieve the best of both worlds: we eliminate the additive $\lambda$ overhead in $\log q$ while keeping the communication cost succinct. In hindsight, this trick is surprisingly straightforward and can be seen as a generalization of the rejection sampling trick.

Our idea is that instead of setting the gap between $\delta$ and $B$ to be either $2^\lambda$ or 2, we set it to be $2cN$ for a constant $c \geq 1$. Now, the parties derive $\mathsf{ct}_k$ using a global seed $\varsigma$ like before but without the per-coefficient rejection counters, *i.e.*, $\mathsf{ct}_k[i,j] = H(\varsigma||i||j) \in \mathbb{Z}_q$ where $q = 2cN \cdot Bp$. Now, because the gap is $2cN$, the probability that a single coefficient falls into the bad intervals is $P = 2B/(2cNB) = 1/(cN)$. Given that $\mathsf{ct}_k$ has $N$ coefficients in total, a seed yields all "good" coefficients with probability approximately $(1-P)^N \approx \exp(-PN) = \exp(-1/c)$. With a natural choice $c = 1$, the success probability is $\exp(-1) \approx 37\%$. In the unfortunate case that the seed is not "golden", *i.e.*, at least one coefficient falls into the bad intervals, $\mathbb{S}$ simply discards the seed and tries the next one. On average, $\mathbb{S}$ only needs to try $(1-P)^{-N} \approx \exp(1/c)$ seeds to find a golden one. For $c = 1$, $\mathbb{S}$ needs to try $\approx 2.718$ seeds.

The concrete sampling routine used by $\mathsf{LogVOLE}_{\mathcal{R}}$ is specified in Figure 6. It computes the public noise budget $B$, hashes candidate seeds from $(\mathsf{id}, d_{\mathsf{rt}}, k_{\mathsf{rt}})$, and rejects a candidate whenever one of the induced correction ciphertexts exceeds the bound.

**Cost Reduction.** The additional communication cost is $\lambda$ for sending the seed. But now the parties can choose a smaller $q$ no longer scaling with $2^\lambda$ but rather $q = O(\log |\mathbf{x}|)$ as $N \approx 2|\mathbf{x}|$ in our binary-tree-like recursion. Consequently, the ring degree $n = O(\lambda q) = O(\lambda \log^2 |\mathbf{x}|)$ is also smaller, resulting in a lower communication cost of $n \cdot q = O(\lambda \log^2 |\mathbf{x}|)$. The smaller ring elements also reduce the concrete computation cost.

**Integrating with Our Recursion Scheme.** The above idea focuses on optimizing TinyLabels. We further apply it to our recursion scheme, which is essentially a recursion tree where every node is a (TinyLabels-like) ShrinkExpand instance. $\mathbb{S}$ can sample a golden seed in the offline phase because the sampling trick only requires $\mathbb{S}$'s knowledge of $\mathsf{sk}_k$ and the public parameters. To be more specific, recall that $\mathbb{S}$ can sample a $\mathsf{sk}_k$ at the root level and then expand it recursively by decrypting $\mathsf{ct}_k$ with the $\mathsf{sk}_k$ of the parent level in offline phase. Hence, $\mathbb{S}$ can sample a single seed used for every ShrinkExpand instance's hashing in a LogVOLE invocation. If any hashing in a node results in a bad coefficient, $\mathbb{S}$ resamples the seed. The only caveat is that the gap between $\delta$ and $B$ becomes $2cN$, where $N$ is the total number of coefficients in the entire recursion tree.

$\mathsf{FindGoldenSeed}(\mathsf{id},\ k_{\mathsf{rt}} \in \mathcal{R}_q,\ d_{\mathsf{rt}} \in \mathcal{R}_q) \to \varsigma \in \{0,1\}^\lambda$

- PROCEDURES:
  1. $B \leftarrow \mathsf{EstimateNoiseBound}()$
  2. $\mathbf{k}'_1 := (k_{\mathsf{rt}})$
  3. $\mathsf{ctr} := 0$
  4. $\varsigma := H(\mathsf{id}, d_{\mathsf{rt}}, k_{\mathsf{rt}}, \mathsf{ctr}) \in \{0,1\}^\lambda$ and $\mathsf{ctr} := \mathsf{ctr} + 1$
  5. For $j \in [\ell]$:
     (a) If $j = \ell$: $\phi := \mathsf{Repeat}(\rho, p)$ and $\boldsymbol{\delta} := \mathsf{Repeat}(\alpha\tau, \mathsf{Repeat}(\rho, \lfloor q/p \rfloor))$;
     Else: $\phi := (q_t)_{t\in[\rho]}$ and $\boldsymbol{\delta} := \mathsf{Repeat}(\alpha\tau, (q/q_t)_{t\in[\rho]})$
     (b) For $i \in [\alpha^{j-1}]$:
        i. $\mathsf{ct}_{j,i} := H_{\mathsf{ct}}(\mathsf{id}, \varsigma, j, i, d_{\mathsf{rt}}) \in \mathcal{R}_q^{\alpha\tau\rho}$
        ii. $\tilde{\mathbf{k}}_{j,i} := \mathsf{LHE.Dec}(\mathbf{k}'_{j,i}, \mathsf{ct}_{j,i}) \in \mathcal{R}_q^{\alpha\tau\rho}$
        iii. $\mathbf{e}_{j,i} := \tilde{\mathbf{k}}_{j,i} \bmod \boldsymbol{\delta}$, coefficient-wise in $(-\boldsymbol{\delta}/2, \boldsymbol{\delta}/2]$
        iv. If $\|\mathbf{e}_{j,i}\|_\infty > B$: Goto Step 4
        v. $\mathbf{k}_{j,i} := \mathsf{Denoise}(\phi, \tilde{\mathbf{k}}_{j,i}) \in \mathcal{R}_q^{\alpha\tau\rho}$
     (c) $\mathbf{k}_j := \mathbf{k}_{j,1} \,\|\, \cdots \,\|\, \mathbf{k}_{j,\alpha^{j-1}} \in \mathcal{R}_q^{\alpha^j\tau\rho}$
     (d) If $j < \ell$: $\mathbf{k}'_{j+1} := \mathsf{Agg}(\mathsf{crt}((q_t)_{t\in[\rho]}, \mathbf{k}_j), \alpha^j\tau \to \alpha^j) \in \mathcal{R}_q^{\alpha^j}$
  6. return $\varsigma$

$\mathsf{EstimateNoiseBound}() \to B$

- FUNCTIONALITY: Derive an error bound $B$ tolerable in $\mathsf{ShrinkExpand}_{\mathcal{R}}$.
- PROCEDURES:
  1. $\tau := \lceil \log_g q \rceil$; $s_{\mathsf{LacE}} := s_{\mathsf{LHE}} := 8$.
  2. $\overline{s}_{\mathsf{LacE}} := 2s_{\mathsf{LacE}} \cdot g \cdot n \cdot \sqrt{2T\tau}$; $\overline{s}_{\mathsf{LHE}} := 2s_{\mathsf{LHE}} \cdot g \cdot n \cdot \sqrt{2T\tau}$.
  3. $B_{\mathsf{LacE}} := g \cdot \tau \cdot n \cdot \log_2(\mu) \cdot s_{\mathsf{LacE}} \cdot \sqrt{\lambda}$.
  4. $B_{\mathsf{LHE}} := (g \cdot \tau \cdot n \cdot s_{\mathsf{LHE}} + \overline{s}_{\mathsf{LHE}}) \cdot \sqrt{\lambda}$.
  5. $\max \overline{\chi}_{\mathsf{LacE}} := \sqrt{\lambda} \cdot \overline{s}_{\mathsf{LacE}}$.
  6. return $B := B_{\mathsf{LacE}} + B_{\mathsf{LHE}} + \max \overline{\chi}_{\mathsf{LacE}}$.

Figure 6: Golden-seed grinding helper used by $\mathsf{ShrinkExpand}_{\mathcal{R}}$ and $\mathsf{LogVOLE}_{\mathcal{R}}$.

**Security.** In the random-oracle model, the seed defines the same correction ciphertexts that would otherwise be sampled coefficient-wise, except that we condition on the same good-interval predicate used by TinyLabels's rejection sampling. This conditioning only exposes that the published ciphertexts are valid for correctness; it does not add leakage beyond the rejection information already present in TinyLabels. Since a golden seed is found with constant probability, the simulator can sample or program the random oracle with only constant expected overhead. The rest of the transcript is unchanged, so security continues to rely on the same RLWE, LacE, and LHE assumptions as the underlying TinyLabels-like ShrinkExpand protocol.

# 8 Generalized CI-VOLE Functionality

Figure 1 gives the basic semi-honest functionality used to state the core idea: after the sender fixes $\Delta$, the receiver chooses $\mathbf{x}$, and the functionality samples $\mathbf{k}$ and returns the correlated outputs. The full protocol needs a slightly more flexible API because different applications require different timing. In the non-interactive setting, the receiver may try candidate calls before deciding which one to release to the sender. In the interactive setting, the sender-side mask $\mathbf{k}$ may have to be available before the receiver's $\mathbf{x}$-dependent output is released. The generalized functionality in Figure 7 exposes exactly these two timing patterns while preserving the same underlying VOLE relation $\mathbf{m} = \mathbf{k} + \mathbf{x} \cdot \Delta$.

**Parameter:** A finite field $\mathcal{M}$.
**State:** A global tuple $(\Delta, w)$, initially $\perp$; a non-interactive table $T_{\mathsf{ni}}$ with entries $T_{\mathsf{ni}}[\mathsf{sid}] = (\mathbf{x}, \mathbf{k}, \mathsf{released})$ and default value $(\perp, \perp, \mathsf{false})$; and an interactive table $T_{\mathsf{int}}$ with entries $T_{\mathsf{int}}[\mathsf{sid}] = (\mathbf{x}, \mathbf{k}, \mathsf{released})$ and default value $(\perp, \perp, \mathsf{false})$, where $\mathsf{sid} \in \{0,1\}^*$, $\mathbf{x}, \mathbf{k} \in \mathcal{M}^w \cup \{\perp\}$, and $\mathsf{released} \in \{\mathsf{false}, \mathsf{true}\}$.

$\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$.set-delta($\mathbb{S} : \Delta \in \mathcal{M},\ w \in \mathbb{N}$):
- If the global tuple is unset:
  - Store $(\Delta, w)$ and send $w$ to $\mathbb{R}$.

$\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$.setx-ni($\mathbb{R} : \mathsf{sid},\ \mathbf{x} \in \mathcal{M}^w$):
- Let $T_{\mathsf{ni}}[\mathsf{sid}] = (\mathbf{x}', \mathbf{k}, \mathsf{released})$.
- If $(\Delta, w)$ is set and $\mathsf{released} = \mathsf{false}$:
  - Sample $\mathbf{k} \leftarrow \mathcal{M}^w$.
  - Set $T_{\mathsf{ni}}[\mathsf{sid}] := (\mathbf{x}, \mathbf{k}, \mathsf{false})$.
  - Compute $\mathbf{m} := \mathbf{k} + \mathbf{x} \cdot \Delta$ and send $\mathbf{m}$ to $\mathbb{R}$.

$\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$.release-ni($\mathbb{R} : \mathsf{sid}$):
- Let $T_{\mathsf{ni}}[\mathsf{sid}] = (\mathbf{x}, \mathbf{k}, \mathsf{released})$.
- If $(\Delta, w)$ is set, $\mathbf{x} \neq \perp$, and $\mathbf{k} \neq \perp$:
  - Set $T_{\mathsf{ni}}[\mathsf{sid}] := (\mathbf{x}, \mathbf{k}, \mathsf{true})$.
  - Send $\mathbf{k}$ to $\mathbb{S}$.

$\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$.setk-int($\mathbb{S} : \mathsf{sid},\ \mathbf{k}^\star \in \mathcal{M}^w$):
- Let $T_{\mathsf{int}}[\mathsf{sid}] = (\mathbf{x}, \mathbf{k}', \mathsf{released})$.
- If $(\Delta, w)$ is set and $\mathsf{released} = \mathsf{false}$:
  - If $\mathbb{S}$ is honest, sample $\mathbf{k} \leftarrow \mathcal{M}^w$; otherwise set $\mathbf{k} := \mathbf{k}^\star$.
  - Set $T_{\mathsf{int}}[\mathsf{sid}] := (\mathbf{x}, \mathbf{k}, \mathsf{false})$.
  - Send $\mathbf{k}$ to $\mathbb{S}$.

$\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$.setx-int($\mathbb{R} : \mathsf{sid},\ \mathbf{x} \in \mathcal{M}^w$):
- Let $T_{\mathsf{int}}[\mathsf{sid}] = (\mathbf{x}', \mathbf{k}, \mathsf{released})$.
- If $(\Delta, w)$ is set and $\mathbf{x}' = \perp$:
  - Set $T_{\mathsf{int}}[\mathsf{sid}] := (\mathbf{x}, \mathbf{k}, \mathsf{false})$.
  - Send x-set to $\mathbb{S}$.

$\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$.release-int($\mathbb{S} : \mathsf{sid}$):
- Let $T_{\mathsf{int}}[\mathsf{sid}] = (\mathbf{x}, \mathbf{k}, \mathsf{released})$.
- If $(\Delta, w)$ is set, $\mathbf{x} \neq \perp$, and $\mathbf{k} \neq \perp$:
  - Set $T_{\mathsf{int}}[\mathsf{sid}] := (\mathbf{x}, \mathbf{k}, \mathsf{true})$.
  - Compute $\mathbf{m} := \mathbf{k} + \mathbf{x} \cdot \Delta$.
  - Send $\mathbf{m}$ to $\mathbb{R}$.

Figure 7: Generalized CI-VOLE functionality. The two modes capture the timing patterns used by our applications: non-interactive calls let the receiver reroll before release, while interactive calls fix $\mathbf{x}$ and $\mathbf{k}$ separately before releasing $\mathbf{m}$.

# 9 The Construction

This section summarizes the core CI-VOLE protocols. Sections 5 to 7 give the lower-level mechanisms, and Sections B and E give the security proofs.

### 9.1 Core CI-VOLE over $\mathcal{R}_q$

The reusable one-time setup and query phases are given in Figures 10 and 11. At a high level, the one-time setup phase prepares the sender and receiver state, while the query phase generates correlations for chosen inputs. The root wrapper and shrink/expand procedures appear in Figures 8 and 9.

**Theorem 2** (Semi-honest summary)**.** *Assume that the Ring-*LWE *assumption holds for the parameter regime used by the underlying* LacE *and* LHE *schemes, and consider the programmable random-oracle model. Then the protocol in Figures 10 and 11 correctly realizes $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ over $\mathcal{R}_q$ and is secure against semi-honest adversaries.*

Section 10 states the malicious-security conditions for the variant without golden-seed optimization, and Section E gives the proof.

**Construction Details.**The lower-level mechanisms include the $\mathbb{Z}_p$-to-$\mathcal{R}_q$ wrapper, the CRT and gadget decompositions, the noise-management argument, and the digest properties used in the security proofs. Sections A, B, 6 and 7 give these details.

**Semi-Honest Security.** The values $\Delta$ held by $\mathbb{S}$ and $\mathbf{x}$ held by $\mathbb{R}$ are their private inputs. The semi-honest security proof shows that the transcript hides both values except through the prescribed outputs. Theorem 9 gives the proof.

**Wrapper over $\mathbb{Z}_p$.** Our intended applications (*e.g.*, ZKPs) work over $\mathbb{Z}_p$, so we batch field elements into ring slots, invoke the ring protocol, and unbatch the resulting shares. Figure 4 gives the wrapper, and Sections B and 6.1 give its analysis.

## 10 Malicious Security

The semi-honest construction in Figures 10 and 11 does not by itself specify the extra checks needed against active parties. This section records the protocol-level changes used by the malicious variant. The protocol ideas and relations are given here; Section E then describes the corresponding simulators, extraction routes, and hybrids.

### 10.1 Protocol-Level Changes

The malicious protocol is obtained by making the following local changes to the semi-honest protocol: we disable the golden-seed optimization, certify the sender's reusable setup packages, use a no-seed interactive online response, and optionally append a receiver-side random-linear check.

#### 10.1.1 Incompatible with Golden Seed Optimziation

Our malicious-security proof is for the variant that does not use the golden-seed optimization of Section 7.2. The semi-honest optimization calls $\mathsf{FindGoldenSeed}(\mathsf{id}, k'_{\mathsf{rt}}, d'_{\mathsf{rt}})$ so that the seed-derived correction ciphertexts avoid the bad noise intervals. Against a malicious sender, that would create an extra validity obligation: the receiver would need to know that the advertised seed was actually good for the hidden $k'_{\mathsf{rt}}$. We do not currently prove that obligation. Accordingly, the malicious theorem is stated for the no-golden variant below. In this variant, the sender does not run FindGoldenSeed, and the receiver performs no golden-seed verification.

This is a parameter tradeoff, not an additional protocol check. The malicious theorem is stated for parameters where arbitrary seed-derived or hash-derived correction ciphertexts are correct except with negligible probability, using the no-golden modulus gap described in Section 7.2. For the

$\mathsf{LogVOLE}_{\mathcal{R}}.\mathsf{SetupRoot}(\mathbb{S} : \mathsf{sk}_{\mathsf{r}} \in \mathcal{R}_q^{\tau}) \to (\mathbb{S} : (\mathbf{r}_{\mathsf{rt}}^{(1)}, \mathsf{sk}_{\mathsf{r,rt}}), \mathbb{R} : (\mathsf{ct}_{\mathsf{lac,rt}}, \mathsf{ct}_{\mathsf{r,rt}}, B^*))$

- Functionality: Prepare the root wrapper that converts a key for the randomized digest $d'_{\mathsf{rt}}$ into a key for the ordinary digest $d_{\mathsf{rt}}$.
- Parameters: Uses left width $\tau\rho$, randomizer width $m > \tau$, and the public LacE node key $(\mathbf{b}_1, \mathbf{b}_2)$.

(1) For $t \in [\rho]$: $\delta_t := q/q_t$; $\mathsf{sk}_{\mathsf{r}t} := \mathsf{sk}_{\mathsf{r}} \bmod q_t \in \mathcal{R}_q^{\tau}$
(2) $\mathbb{S}$: $\hat{\mathsf{sk}}_{\mathsf{r}} := \mathsf{transpose}((\mathsf{sk}_{\mathsf{r}t} \cdot \delta_t)_{t\in[\rho]}) \in \mathcal{R}_q^{\tau\rho}$
(3) $\mathbb{S}$: $(\mathbf{r}_{\mathsf{rt}}^{(2)}, \mathsf{ct}_{\mathsf{lacrt},L}) := \mathsf{LacE.Enc}(\hat{\mathsf{sk}}_{\mathsf{r}})$
(4) $\mathbb{S}$: $B^* \leftarrow \mathcal{R}_q^{1\times m}$, $B' := (\mathbf{b}_1^{\top}\ B^*) \in \mathcal{R}_q^{1\times(\tau+m)}$, and $\mathbf{r}_{\mathsf{rt}}^{(1)} \leftarrow \mathcal{R}_q^{\tau\rho}$
(5) $\mathbb{S}$: For each $\mathsf{ind} \in \{0,1\}^{\log_2(\tau\rho)}$, sample $\mathbf{E}_{\mathsf{rt}}^{\mathsf{top}}[\mathsf{ind}] \leftarrow \overline{\mathcal{D}}^{\tau+m}$ and set

$$\mathbf{C}_{\mathsf{rt}}^{\mathsf{top}}[\mathsf{ind}] := \mathbf{r}_{\mathsf{rt}}^{(1)}[\mathsf{ind}] \cdot B' - \mathbf{r}_{\mathsf{rt}}^{(2)}[\mathsf{ind}] \cdot (\mathbf{g}^{\top}\ 0^m) + \mathbf{E}_{\mathsf{rt}}^{\mathsf{top}}[\mathsf{ind}] \in \mathcal{R}_q^{\tau+m}.$$

(6) $\mathbb{S}$: $(\mathsf{ct}_{\mathsf{r,rt}}, \mathsf{sk}_{\mathsf{r,rt}}) := \mathsf{LHE.Enc}_1(\mathbf{r}_{\mathsf{rt}}^{(1)})$
(7) $\mathbb{S} \to \mathbb{R}$: $(\mathsf{ct}_{\mathsf{lac,rt}} := (\mathbf{C}_{\mathsf{rt}}^{\mathsf{top}}, \mathsf{ct}_{\mathsf{lacrt},L}), \mathsf{ct}_{\mathsf{r,rt}}, B^*)$
(8) return $(\mathbb{S} : (\mathbf{r}_{\mathsf{rt}}^{(1)}, \mathsf{sk}_{\mathsf{r,rt}}), \mathbb{R} : (\mathsf{ct}_{\mathsf{lac,rt}}, \mathsf{ct}_{\mathsf{r,rt}}, B^*))$

$\mathsf{LogVOLE}_{\mathcal{R}}.\mathsf{RandRootDigest}(\mathbb{R} : (d_{\mathsf{rt}} \in \mathcal{R}_q,\ \zeta_{\mathsf{rt}} \in \mathcal{R}_q^m)) \to \mathbb{R} : d'_{\mathsf{rt}}$

- Functionality: Randomize $d_{\mathsf{rt}}$ into $d'_{\mathsf{rt}}$.
- Assumption: $\|\zeta_{\mathsf{rt}}\|_\infty < g$.
- Notation: Uses the public matrix $B^*$ sampled in SetupRoot.

(1) $\mathbb{R}$: $\hat{d_{\mathsf{rt}}} := \mathsf{decompose}(d_{\mathsf{rt}}) \in \mathcal{R}_q^{\tau\rho}$
(2) $\mathbb{R}$: $y_{\mathsf{L}} := \mathsf{LacE.Digest}(\hat{d_{\mathsf{rt}}})$
(3) $\mathbb{R}$: $d'_{\mathsf{rt}} := (\mathbf{b}_1^{\top}\ B^*)\binom{\mathbf{g}^{-1}(y_{\mathsf{L}})}{\zeta_{\mathsf{rt}}}$
(4) return $\mathbb{R} : d'_{\mathsf{rt}}$

$\mathsf{LogVOLE}_{\mathcal{R}}.\mathsf{RootDigest}(\mathbb{R} : \mathbf{x} \in \mathcal{R}_q^{w'},\ \mathbb{B} : j) \to \mathbb{R} : d_{\mathsf{rt}} \in \mathcal{R}_q$

(1) If $w' = \alpha\tau\rho$: return $\mathbb{R} : \mathsf{ShrinkExpand}_{\mathcal{R}}.\mathsf{Shrink}(\mathbf{x})$
(2) $\mathbb{B}$: $b := w'/(\alpha\tau\rho)$
(3) $\mathbb{R}$: $(\mathbf{x}_i)_{i\in[b]} := \mathsf{Partition}(\mathbf{x}, b) \in \mathcal{R}_q^{b\times\alpha\tau\rho}$
(4) $\mathbb{R}$: $\mathbf{d} := (\mathsf{ShrinkExpand}_{\mathcal{R}}.\mathsf{Shrink}(\mathbf{x}_i))_{i\in[b]} \in \mathcal{R}_q^{b}$
(5) $\mathbb{R}$: $\hat{\mathbf{d}} := ((\mathbf{g}^{-1}(\mathbf{d}_i) \bmod q_t)_{t\in[\rho]})_{i\in[b]} \in \mathcal{R}_q^{b\rho\tau}$
(6) $\mathbb{R}$: $\hat{\mathbf{d}} := (\mathsf{transpose}(\hat{\mathbf{d}}_i \in \mathcal{R}_q^{\rho\tau}) \in \mathcal{R}_q^{\tau\rho})_{i\in[b]}$
(7) return $\mathsf{LogVOLE}_{\mathcal{R}}.\mathsf{RootDigest}(\mathbb{R} : \hat{\mathbf{d}},\ \mathbb{B} : j-1)$

$\mathsf{LogVOLE}_{\mathcal{R}}.\mathsf{DerandRoot}\big(\mathbb{S} : (\mathbf{r}_{\mathsf{rt}}^{(1)}, k'_{\mathsf{rt}} \in \mathcal{R}_q),$
$\mathbb{R} : ((\mathsf{ct}_{\mathsf{lac,rt}}, \mathsf{ct}_{\mathsf{r,rt}}, B^*), \mathsf{sk}_{\mathsf{rt}}{}' \in \mathcal{R}_q,$
$d_{\mathsf{rt}} \in \mathcal{R}_q, d'_{\mathsf{rt}} \in \mathcal{R}_q, \zeta_{\mathsf{rt}} \in \mathcal{R}_q^m),\ \mathbb{B} : (\mathsf{id}, \varsigma)\big)$
$\to (\mathbb{S} : k_{\mathsf{rt}}, \mathbb{R} : \mathsf{sk}_{\mathsf{rt}})$

- Functionality: Convert a key share for $d'_{\mathsf{rt}}$ into one for $d_{\mathsf{rt}}$.

(1) $\mathbb{R}$: $\hat{d_{\mathsf{rt}}} := \mathsf{decompose}(d_{\mathsf{rt}}) \in \mathcal{R}_q^{\tau\rho}$
(2) $\mathbb{R}$: Parse $\mathsf{ct}_{\mathsf{lac,rt}} = (\mathbf{C}_{\mathsf{rt}}^{\mathsf{top}}, \mathsf{ct}_{\mathsf{lacrt},L})$; $y_{\mathsf{L}} := \mathsf{LacE.Digest}(\hat{d_{\mathsf{rt}}})$
(3) $\mathbb{B}$: $\mathsf{ct}_{\mathsf{k,rt}} := H_{\mathsf{ct}}(\mathsf{id}, \varsigma, \mathsf{rt}, 0) \in \mathcal{R}_q^{\tau\rho}$
(4) $\mathbb{R}$: $\mathsf{ct}_{\mathsf{res,rt}} := \mathsf{ct}_{\mathsf{r,rt}} \cdot \mathbf{g}^{-1}(d'_{\mathsf{rt}}) + \mathsf{ct}_{\mathsf{k,rt}}$
(5) $\mathbb{R}$: $\boldsymbol{\phi}_{\mathsf{rt}} := \mathbf{C}_{\mathsf{rt}}^{\mathsf{top}} \cdot \binom{\mathbf{g}^{-1}(y_{\mathsf{L}})}{\zeta_{\mathsf{rt}}} + \mathsf{LacE.Eval}(\mathsf{ct}_{\mathsf{lacrt},L}, \hat{d_{\mathsf{rt}}})$
(6) $\mathbb{R}$: $\tilde{\mathbf{m}}_{\mathsf{rt}} := \mathsf{LHE.Dec}(\mathsf{sk}_{\mathsf{rt}}{}', \mathsf{ct}_{\mathsf{res,rt}}) - \boldsymbol{\phi}_{\mathsf{rt}}$
(7) $\mathbb{S}$: $\tilde{\mathbf{k}}_{\mathsf{rt}} := \mathsf{LHE.Dec}(k'_{\mathsf{rt}}, \mathsf{ct}_{\mathsf{k,rt}})$
(8) $\mathbb{S}$: $\hat{\mathbf{k}} := \mathsf{Denoise}((q_t)_{t\in[\rho]}, \tilde{\mathbf{k}}_{\mathsf{rt}})$; $\mathbb{R}$: $\hat{\mathbf{m}} := \mathsf{Denoise}((q_t)_{t\in[\rho]}, \tilde{\mathbf{m}}_{\mathsf{rt}})$
(9) $\mathbb{S}$: $k_{\mathsf{rt}} := \mathsf{Agg}(\mathsf{crt}((q_t)_{t\in[\rho]}, \hat{\mathbf{k}}), \tau \to 1)$
(10) $\mathbb{R}$: $\mathsf{sk}_{\mathsf{rt}} := \mathsf{Agg}(\mathsf{crt}((q_t)_{t\in[\rho]}, \hat{\mathbf{m}}), \tau \to 1)$
(11) return $(\mathbb{S} : k_{\mathsf{rt}}, \mathbb{R} : \mathsf{sk}_{\mathsf{rt}})$

Figure 8: Randomized root-digest construction for $\mathsf{LogVOLE}_{\mathcal{R}}$.

$\mathsf{ShrinkExpand.Offline}(\mathbb{S} : \mathbf{s} \in \mathcal{R}_q^\mu) \to (\mathbb{S} : (r, \mathsf{sk_r}), \mathbb{R} : (\mathsf{ct_{lac}}, \mathsf{ct_r}))$

- Functionality: $\mathbb{S}$ receives $(r, \mathsf{sk_r})$ and $\mathbb{R}$ receives $(\mathsf{ct_{lac}}, \mathsf{ct_r})$.
- Parameters: $\alpha, \mu, g, \tau$, where $\tau = \lceil \log_g q \rceil$ and $\mu = \alpha\rho\tau$.
- Procedures:
  1. $\mathbb{S}$: $(r, \mathsf{ct_{lac}}) := \mathsf{LacE.Enc}(\mathbf{s})$
  2. $\mathbb{S}$: $(\mathsf{ct_r}, \mathsf{sk_r}) := \mathsf{LHE.Enc}_1(r)$
  3. $\mathbb{S}$: send $(\mathsf{ct_{lac}}, \mathsf{ct_r})$ to $\mathbb{R}$

$\mathsf{ShrinkExpand.Shrink}(\mathbf{x} \in \mathcal{R}_q^\mu) \to d_{\mathsf{rt}} \in \mathcal{R}_q$

- Procedures: The party computes $d_{\mathsf{rt}} := \mathsf{LacE.Digest}(\mathbf{x})$

$\mathsf{ShrinkExpand.Expand}(\mathbb{S} : (r, k'_{\mathsf{rt}} \in \mathcal{R}_q),$
$\mathbb{R} : ((\mathsf{ct_{lac}}, \mathsf{ct_r}), \mathsf{sk_{rt}} \in \mathcal{R}_q, d_{\mathsf{rt}} \in \mathcal{R}_q, \mathbf{x} \in \mathcal{R}_q^\mu),$
$\mathbb{B} : (\phi, \mathsf{id}, \varsigma \in \{0,1\}^\lambda, j, i)) \to (\mathbb{S} : \mathbf{k} \in \mathcal{R}_q^\mu, \mathbb{R} : \mathbf{m} \in \mathcal{R}_q^\mu)$

- Assumption: $\mathsf{sk_{rt}} = \mathsf{sk_r}^T \cdot \mathbf{g}^{-1}(d_{\mathsf{rt}}) + k'_{\mathsf{rt}}$.
- Procedures:
  1. $\mathbb{B}$: $\mathsf{ct_k} := H_{\mathsf{ct}}(\mathsf{id}, \varsigma, j, i)$
  2. $\mathbb{R}$: $\mathsf{ct_{res}} := \mathsf{ct_r} \cdot \mathbf{g}^{-1}(d_{\mathsf{rt}}) + \mathsf{ct_k}$
  3. $\mathbb{R}$: $\tilde{\mathbf{m}} := \mathsf{LHE.Dec}(\mathsf{sk_{rt}}, \mathsf{ct_{res}}) - \mathsf{LacE.Eval}(\mathsf{ct_{lac}}, \mathbf{x})$
  4. $\mathbb{S}$: $\tilde{\mathbf{k}} := \mathsf{LHE.Dec}(k'_{\mathsf{rt}}, \mathsf{ct_k})$
  5. return $(\mathbb{R} : \mathbf{m} := \mathsf{Denoise}(\phi, \tilde{\mathbf{m}}), \mathbb{S} : \mathbf{k} := \mathsf{Denoise}(\phi, \tilde{\mathbf{k}}))$

Note:

- Optional optimization: omit the least-significant gadget digit by replacing the width $\tau$ with $\tau' = \tau - 1$ throughout, $\mathbf{g}^{-1}$ with $\mathbf{g}_{\mathsf{hi}}^{-1}$, and $\mathsf{LHE}$ and $\mathsf{LacE}$ with $\mathsf{LHE_{hi}}$ and $\mathsf{LacE_{hi}}$, respectively. Section 7.1 and figs. 15 and 16 give the details.

Figure 9: Full $\mathsf{ShrinkExpand}_{\mathcal{R}}$ protocol.

$\mathsf{LogVOLE}_{\mathcal{R}}.\mathsf{Offline}(\mathbb{S} : s \in \mathcal{R}_q,\ \mathbb{B} : w \in \mathbb{N})$
$\to (\mathbb{S} : u \in \mathsf{State}, \mathbb{R} : v \in \mathsf{State})$

- Functionality: $\mathbb{S}$ receives state $u = (\mathsf{sk_{r,rt}}, (\mathbf{r}_{\mathsf{rt}}^{(1)}, u'))$ and $\mathbb{R}$ receives state $v = ((\mathsf{ct_{lac,rt}}, \mathsf{ct_{r,rt}}, B^*), v')$.
- Parameters: See Section 12.1.
- Assumption: $w = \tau\rho\alpha^\ell$ for some $\ell \in \mathbb{N}_{\geq 1}$.
- Notation: $\mathsf{ShrinkExpand}_{\mathcal{R}}.\mathsf{Offline}_{\mathsf{sk_r}}$ is the same setup as $\mathsf{ShrinkExpand}_{\mathcal{R}}.\mathsf{Offline}$, except that the $\mathsf{LHE.Enc}_1$ state is fixed to the retained key $\mathsf{sk_r}$.
- Procedures:
  1. $\mathbb{S}$: $\mathbf{s}_{\mathsf{leaf}} := \mathsf{Repeat}(\alpha\tau\rho, s) \in \mathcal{R}_q^{\alpha\tau\rho}$
  2. $(\mathbb{S} : (r_{\mathsf{leaf}}, \mathsf{sk_r}), \mathbb{R} : (\mathsf{ct_{lac,leaf}}, \mathsf{ct_{r,leaf}})) \leftarrow \mathsf{ShrinkExpand}_{\mathcal{R}}.\mathsf{Offline}(\mathbb{S} : \mathbf{s}_{\mathsf{leaf}})$
  3. If $\ell = 1$: $\mathbb{S}$: $r_{\mathsf{int}} := \perp$; $\mathbb{R}$: $(\mathsf{ct_{lac,int}}, \mathsf{ct_{r,int}}) := \perp$
  4. Else:
     (a) For $j \in [\rho]$: $\delta_j := q/q_j$
     (b) For $j \in [\rho]$: $\mathsf{sk}_{\mathsf{r}j} := \mathsf{sk_r} \bmod q_j \in \mathcal{R}_q^\tau$ with canonical lifting.
     (c) $\mathbb{S}$: $\mathbf{s}_{\mathsf{int}} := \mathsf{Repeat}(\alpha, \mathsf{transpose}((\mathsf{sk}_{\mathsf{r}j} \cdot \delta_j)_{j\in[\rho]} \in \mathcal{R}_q^{\rho\tau}) \in \mathcal{R}_q^{\tau\rho})$
     (d) $(\mathbb{S} : (r_{\mathsf{int}}, \mathsf{sk_r}), \mathbb{R} : (\mathsf{ct_{lac,int}}, \mathsf{ct_{r,int}})) \leftarrow \mathsf{ShrinkExpand}_{\mathcal{R}}.\mathsf{Offline}(\mathbb{S} : \mathbf{s}_{\mathsf{int}})$
  5. $\mathbb{S}$: $u' := (r_{\mathsf{leaf}}, r_{\mathsf{int}})$; $\mathbb{R}$: $v' := ((\mathsf{ct_{lac,leaf}}, \mathsf{ct_{r,leaf}}), (\mathsf{ct_{lac,int}}, \mathsf{ct_{r,int}}))$
  6. $(\mathbb{S} : (\mathbf{r}_{\mathsf{rt}}^{(1)}, \mathsf{sk_{r,rt}}), \mathbb{R} : (\mathsf{ct_{lac,rt}}, \mathsf{ct_{r,rt}}, B^*)) \leftarrow \mathrm{SetupRoot}(\mathbb{S} : \mathsf{sk_r})$ ▷ see Figure 8.
  7. return $(\mathbb{S} : (\mathsf{sk_{r,rt}}, (\mathbf{r}_{\mathsf{rt}}^{(1)}, u')), \mathbb{R} : ((\mathsf{ct_{lac,rt}}, \mathsf{ct_{r,rt}}, B^*), v'))$
- The subscript $\mathsf{leaf}$ denotes the level-$\ell$ package generated from the sender input $s = \Delta$. The subscript $\mathsf{int}$ denotes the package shared by internal levels $1, \ldots, \ell - 1$; it is $\perp$ when $\ell = 1$.

Figure 10: Offline phase of $\mathsf{LogVOLE}_{\mathcal{R}}$.

$\mathsf{LogVOLE}_{\mathcal{R}}.\mathsf{Online}(\mathbb{S} : u \in \mathsf{State},\ \mathbb{R} : (v \in \mathsf{State}, \mathbf{x} \in \mathcal{R}_q^w),$
$\mathbb{B} : \mathsf{id} = (\mathsf{sid}, \mathsf{mode}),\ \mathsf{mode} \in \{\mathsf{interactive}, \mathsf{non\text{-}interactive}\})$
$\to (\mathbb{S} : \mathbf{k} \in \mathcal{R}_q^w, \mathbb{R} : \mathbf{m} \in \mathcal{R}_q^w)$

- PARAMETERS: See Section 12.1.
- ASSUMPTION: $w = \tau\rho\alpha^\ell$ for some $\ell \in \mathbb{N}_{\geq 1}$.
- PROCEDURES:
  1. $\mathbb{S}$: Parse $u = (\mathsf{sk}_{\mathsf{r,rt}}, (\mathbf{r}_{\mathsf{rt}}^{(1)}, u'))$
  2. $\mathbb{R}$: Parse $v = ((\mathsf{ct}_{\mathsf{lac,rt}}, \mathsf{ct}_{\mathsf{r,rt}}, B^*), v')$
  3. $\mathbb{R}$: $\zeta_{\mathsf{rt}} \leftarrow \{\zeta \in \mathcal{R}_q^m : \|\zeta\|_\infty < g\}$
  4. $\mathbb{R}$: $d_{\mathsf{rt}} := \mathsf{RootDigest}(\mathbb{R} : \mathbf{x},\ \mathbb{B} : \ell)$
  5. $\mathbb{R}$: $d'_{\mathsf{rt}} := \mathsf{RandRootDigest}(\mathbb{R} : (d_{\mathsf{rt}}, \zeta_{\mathsf{rt}}))$ ▷ see Figure 8.
  6. $\mathbb{R} \to \mathbb{S}$: $d'_{\mathsf{rt}}$
  7. If $\mathsf{mode} = \mathsf{interactive}$:
     (a) $\mathbb{S}$: $k'_{\mathsf{rt}} \leftarrow \mathcal{R}_q$
     (b) $\mathbb{S}$: $\mathsf{sk}_{\mathsf{rt}}{}' := \mathsf{sk}_{\mathsf{r,rt}}^\top \cdot \mathbf{g}^{-1}(d'_{\mathsf{rt}}) + k'_{\mathsf{rt}}$
     (c) $\mathbb{S}$: $\varsigma := \mathsf{FindGoldenSeed}(\mathsf{id}, k'_{\mathsf{rt}}, d'_{\mathsf{rt}})$
     (d) $\mathbb{S} \to \mathbb{R}$ : $(\varsigma, \mathsf{sk}_{\mathsf{rt}}{}')$
     Else:
     (a) $\mathbb{B}$: $(\mathsf{sk}_{\mathsf{rt}}{}', \varsigma) := H(\mathsf{id}, d'_{\mathsf{rt}}) \in \mathcal{R}_q \times \{0,1\}^\lambda$
     (b) $\mathbb{S}$: $k'_{\mathsf{rt}} := -\mathsf{sk}_{\mathsf{r,rt}}{}^\top \cdot \mathbf{g}^{-1}(d'_{\mathsf{rt}}) + \mathsf{sk}_{\mathsf{rt}}{}' \in \mathcal{R}_q$
  8. $(\mathbb{S} : k_{\mathsf{rt}}, \mathbb{R} : \mathsf{sk}_{\mathsf{rt}}) := \mathsf{DerandRoot}(\mathbb{S} : (\mathbf{r}_{\mathsf{rt}}^{(1)}, k'_{\mathsf{rt}}),\ \mathbb{R} : ((\mathsf{ct}_{\mathsf{lac,rt}}, \mathsf{ct}_{\mathsf{r,rt}}, B^*), \mathsf{sk}_{\mathsf{rt}}{}', d_{\mathsf{rt}}, d'_{\mathsf{rt}}, \zeta_{\mathsf{rt}}),\ \mathbb{B} : (\mathsf{id}, \varsigma))$
  9. $(\mathbb{S} : \mathbf{k}, \mathbb{R} : \mathbf{m}) := \mathsf{OnlineRec}(\mathbb{S} : (u', k_{\mathsf{rt}}),\ \mathbb{R} : (v', \mathsf{sk}_{\mathsf{rt}}, \mathbf{x}),\ \mathbb{B} : (\mathsf{id}, \varsigma, \ell, \ell))$
  10. return $(\mathbb{S} : (\mathbf{k}_i)_{i\in[w]}, \mathbb{R} : (\mathbf{m}_i)_{i\in[w]})$
- PRIVATE $\mathsf{OnlineRec}(\mathbb{S} : (u \in \mathsf{State}, k_{\mathsf{rt}}),\ \mathbb{R} : (v \in \mathsf{State}, \mathsf{sk}_{\mathsf{rt}}, \mathbf{x} \in \mathcal{R}_q^{w'}),$
$\mathbb{B} : (\mathsf{id}, \varsigma, j, \ell)) \to (\mathbb{S} : \mathbf{k} \in \mathcal{R}_q^{w'}, \mathbb{R} : \mathbf{m} \in \mathcal{R}_q^{w'})$

  1. $\mathbb{S}$: Parse $u = (r_{\mathsf{leaf}}, r_{\mathsf{int}})$
  2. $\mathbb{R}$: Parse $v = ((\mathsf{ct}_{\mathsf{lac,leaf}}, \mathsf{ct}_{\mathsf{r,leaf}}), (\mathsf{ct}_{\mathsf{lac,int}}, \mathsf{ct}_{\mathsf{r,int}}))$
  3. If $j = \ell$: $\mathbb{B}$: $(r, \mathsf{ct}_{\mathsf{lac}}, \mathsf{ct}_{\mathsf{r}}) := (r_{\mathsf{leaf}}, \mathsf{ct}_{\mathsf{lac,leaf}}, \mathsf{ct}_{\mathsf{r,leaf}})$
     Else: $\mathbb{B}$: $(r, \mathsf{ct}_{\mathsf{lac}}, \mathsf{ct}_{\mathsf{r}}) := (r_{\mathsf{int}}, \mathsf{ct}_{\mathsf{lac,int}}, \mathsf{ct}_{\mathsf{r,int}})$
  4. $\mathbb{B}$: $b := w'/(\alpha\tau\rho)$
  5. $\mathbb{R}$: $(\mathbf{x}_i)_{i\in[b]} := \mathsf{Partition}(\mathbf{x}, b) \in \mathcal{R}_q^{b\times\alpha\tau\rho}$
  6. $\mathbb{R}$: $\mathbf{d} := (\mathsf{ShrinkExpand}_{\mathcal{R}}.\mathsf{Shrink}(\mathbf{x}_i))_{i\in[b]} \in \mathcal{R}_q^b$
  7. If $j = 1$: $\mathbb{B}$: $b = 1$; $\mathbb{S}$: $\mathbf{k}_1 := k_{\mathsf{rt}}$; $\mathbb{R}$: $\mathsf{sk}_{\mathbf{d},1} := \mathsf{sk}_{\mathsf{rt}}$ Else: ▷ $1 < j \leq \ell$
     (a) $\mathbb{R}$: $\hat{\mathbf{d}} := ((\mathbf{g}^{-1}(\mathbf{d}_i) \bmod q_t)_{t\in[\rho]})_{i\in[b]} \in \mathcal{R}_q^{b\rho\tau}$
     (b) $\mathbb{R}$: $\hat{\mathbf{d}} := (\mathsf{transpose}(\hat{\mathbf{d}}_i \in \mathcal{R}_q^{\rho\tau}) \in \mathcal{R}_q^{\tau\rho})_{i\in[b]}$
     (c) $(\mathbb{S} : \mathbf{k}, \mathbb{R} : \mathbf{m}) \leftarrow \mathsf{OnlineRec}(\mathbb{S} : (u, k_{\mathsf{rt}}),\ \mathbb{R} : (v, \mathsf{sk}_{\mathsf{rt}}, \hat{\mathbf{d}}),\ \mathbb{B} : (\mathsf{id}, \varsigma, j-1, \ell))$
     (d) $\mathbb{S}$: $\mathbf{k} := \mathsf{Agg}(\mathsf{crt}((q_t)_{t\in[\rho]}, \mathbf{k}), b\tau \to b)$
     (e) $\mathbb{R}$: $\mathsf{sk}_{\mathbf{d}} := \mathsf{Agg}(\mathsf{crt}((q_t)_{t\in[\rho]}, \mathbf{m}), b\tau \to b)$
  8. If $j = \ell$: $\mathbb{B}$: $\phi := \mathsf{Repeat}(\rho, p)$ Else: $\mathbb{B}$: $\phi := (q_h)_{h\in[\rho]}$
  9. For $i \in [b]$: $(\mathbb{S} : \mathbf{k}_i, \mathbb{R} : \mathbf{m}_i) := \mathsf{ShrinkExpand}_{\mathcal{R}}.\mathsf{Expand}(\mathbb{S} : (r, \mathbf{k}_i), \mathbb{R} : ((\mathsf{ct}_{\mathsf{lac}}, \mathsf{ct}_{\mathsf{r}}), \mathsf{sk}_{\mathbf{d},i}, \mathbf{d}_i, \mathbf{x}_i), \mathbb{B} : (\phi, \mathsf{id}, \varsigma, j, i))$
  10. return $(\mathbb{S} : \mathbf{k} := (\mathbf{k}_i)_{i\in[b]}, \mathbb{R} : \mathbf{m} := (\mathbf{m}_i)_{i\in[b]})$
- In OnlineRec, $j$ is the current recursion-tree level: $j = \ell$ is the leaf level, and $j = 1$ is the root level where the recursive call stops and uses the root-wrapper output.

Figure 11: Online phase of $\mathsf{LogVOLE}_{\mathcal{R}}$.

concrete parameter regime used in our evaluation, we estimate that moving to this no-golden regime would increase both computation and communication by about a factor of two. This overhead does not change the asymptotic or application-level message of the construction, but it matters for our reported concrete numbers: the implementation and evaluation therefore focus on the optimized semi-honest protocol with golden-seed grinding. Supporting the golden-seed optimization against malicious senders is left as a separate extension; it would require either a public seed-validity test, a narrow proof of seed goodness, or a proof that bad seeds are otherwise simulatable.

### 10.1.2 Online Responses and Correction Labels

The two modes obtain their online response differently. In interactive mode, the sender sends a fresh root response directly. In non-interactive mode, the response is derived from an online random oracle $H_{\mathsf{on}}$.

Concretely, for a session identifier $\mathsf{id}$, the interactive sender samples

$$k'_{\mathsf{rt}} \leftarrow R_q,$$

sets

$$\mathsf{sk_{rt}}' := \mathsf{sk}_{\mathsf{r},\mathsf{rt}}^{\top}\mathbf{g}^{-1}(d'_{\mathsf{rt}}) + k'_{\mathsf{rt}},$$

and sends $\mathsf{sk_{rt}}'$. In the non-interactive mode, the pair $(\mathsf{sk_{rt}}', \varsigma)$ is derived from the online random oracle as in the semi-honest syntax, and therefore may depend on the receiver's digest:

$$H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}}),$$

and the sender defines

$$k'_{\mathsf{rt}} := \mathsf{sk_{rt}}' - \mathsf{sk}_{\mathsf{r},\mathsf{rt}}^{\top}\mathbf{g}^{-1}(d'_{\mathsf{rt}}).$$

The root correction ciphertext $\mathsf{ct}_{\mathsf{k},\mathsf{rt}}$ and each ordinary correction ciphertext $\mathsf{ct}_{\mathsf{k}\,j,i}$ are hash-derived as usual. The only change from the golden-seed variant is that the malicious protocol does not grind over seeds to obtain unusually good noise. In the interactive mode the parties derive the label seed

$$\varsigma := H_{\mathsf{seed}}(\mathsf{id}, \mathsf{sk_{rt}}') \in \{0,1\}^{\lambda}$$

and the hash labels are

$$H_{\mathsf{ct}}(\mathsf{id}, \varsigma, \mathsf{rt}, 0), \qquad H_{\mathsf{ct}}(\mathsf{id}, \varsigma, j, i),$$

so the online response determines the public label namespace without being used directly as a bit-string seed. For a corrupted receiver, the fresh $\mathsf{sk_{rt}}'$ is not available before the sender's online message, so the final $H_{\mathsf{ct}}$ labels remain unset until after receiver-input extraction. For a corrupted sender, once the setup proof extracts $\mathsf{sk}_{\mathsf{r},\mathsf{rt}}$, the received $\mathsf{sk_{rt}}'$ simply defines the effective root mask

$$k'_{\mathsf{rt}} := \mathsf{sk_{rt}}' - \mathsf{sk}_{\mathsf{r},\mathsf{rt}}^{\top}\mathbf{g}^{-1}(d'_{\mathsf{rt}}).$$

No online NIZK is needed: in the interactive malicious-sender functionality, the sender is allowed to determine the effective mask induced by its online response.

In the non-interactive mode the corresponding labels are transcript-bound, e.g.,

$$H_{\mathsf{ct}}(\mathsf{id}, d'_{\mathsf{rt}}, \varsigma, \mathsf{rt}, 0), \qquad H_{\mathsf{ct}}(\mathsf{id}, d'_{\mathsf{rt}}, \varsigma, j, i).$$

Since $\varsigma$ is itself derived from $H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}})$, the dependence on $d'_{\mathsf{rt}}$ is already implicit; we write it explicitly in the malicious proof to make the programming point unambiguous. The compact algorithm boxes suppress these fixed oracle labels.

### 10.1.3 Sender-Side Certified Setup

To reuse one public key across sessions, the sender publishes a certified setup for a fixed $\Delta$. Its consistency proofs let the receiver check that the setup is well formed without revealing $\Delta$. After verifying these proofs, receivers can use the same public key with domain-separated identifiers.

In the semi-honest offline phase, the sender publishes the recursive shrink/expand ciphertext packages

$$(\mathsf{ct}_{\mathsf{lac}}{}^{(h)}, \mathsf{ct}_{\mathsf{r}}{}^{(h)})_{h\in L}$$

and the widened root package

$$(\mathsf{ct}_{\mathsf{lac,rt}}, \mathsf{ct}_{\mathsf{r,rt}}, B^*).$$

In the malicious protocol, each such package is accompanied by a NIZK proof of knowledge that it was honestly generated from one hidden sender state. It is useful to view both cases through one interface: a setup package is a public payload of arity $t \in \{2, 4\}$, together with a proof for the corresponding bounded linear setup relation. The ordinary shrink/expand payload has arity 2, namely $(\mathsf{ct}_{\mathsf{lac}}{}^{(h)}, \mathsf{ct}_{\mathsf{r}}{}^{(h)})$. Writing $\mathsf{ct}_{\mathsf{lac,rt}} = (\mathbf{C}^{\mathsf{top}}_{\mathsf{rt}}, \mathsf{ct}_{\mathsf{lacrt},L})$, the root payload has arity 4, namely $(\mathbf{C}^{\mathsf{top}}_{\mathsf{rt}}, \mathsf{ct}_{\mathsf{lacrt},L}, \mathsf{ct}_{\mathsf{r,rt}}, B^*)$. For ordinary level $h$, the statement has the form

$$\mathsf{st}^{(h)}_{\mathsf{setup}} := (\mathsf{pp}^{(h)}, \mathsf{ct}_{\mathsf{lac}}{}^{(h)}, \mathsf{ct}_{\mathsf{r}}{}^{(h)}), \qquad \pi^{(h)}_{\mathsf{setup}}.$$

For the widened root wrapper, the statement has the form

$$\mathsf{st}^{\mathsf{rt}}_{\mathsf{setup}} := (\mathsf{pp}^{\mathsf{rt}}, \mathsf{ct}_{\mathsf{lac,rt}}, \mathsf{ct}_{\mathsf{r,rt}}, B^*), \qquad \pi^{\mathsf{rt}}_{\mathsf{setup}}.$$

The intended relation binds each public ciphertext package to one hidden carried-mask system, including the $\Delta$-dependent data encoded by the sender's setup.

For one ordinary shrink/expand block of width $\mu$, let $\ell := \log_2 \mu$, let $s \in R_q^\mu$ be the outgoing block secret, and write the receiver's public setup package as

$$\mathsf{st}_{\mathsf{setup}} := (\mathsf{pp}, \mathsf{ct}_{\mathsf{lac}}, \mathsf{ct}_{\mathsf{r}}).$$

The public parameters $\mathsf{pp}$ include the ring and bridge parameters together with explicit coefficient bounds

$$\Gamma_{\mathsf{ring}}, \quad \Gamma_{\mathsf{br}}, \quad \Gamma_{\mathsf{row}}.$$

Here $\Gamma_{\mathsf{ring}}$ is the canonical bound used to represent arbitrary ring elements in the proof system, while the other bounds are the actual correctness bounds for the bridge-opening noise and LacE row-noise terms. The concrete values are determined by the $\mathsf{LHE}/\mathsf{LacE}$ parameter choices and the denoising analysis of Section 6.2; the malicious statement below treats them as part of $\mathsf{pp}$. The witness is

$$w_{\mathsf{setup}} = \Big(r, \mathsf{sk}_{\mathsf{br}}, s, \{a_{i,j}\}_{i\in\{0,\dots,\ell\},\, j\in[\mu]}, \{E_{i,j}\}_{i\in\{0,\dots,\ell-1\},\, j\in[\mu]}, \eta_{\mathsf{ct}_{\mathsf{r}}}\Big).$$

Here $r \in R_q^\mu$ is the root carried-mask vector, $\mathsf{sk}_{\mathsf{br}}$ and $\eta_{\mathsf{ct}_{\mathsf{r}}}$ open the bridge ciphertext $\mathsf{ct}_{\mathsf{r}}$, the values $a_{i,j}$ are the carried masks along the path to leaf $j$, and $E_{i,j}$ are the row-noise terms. The setup relation $R_{\mathsf{setup}}$ checks the following conditions. First, $\mathsf{ct}_{\mathsf{r}}$ is a valid bridge ciphertext for $r$, expressed as the bridge's exact modular opening equation

$$M_{\mathsf{ct}_{\mathsf{r}}} \begin{pmatrix} r \\ \mathsf{sk}_{\mathsf{br}} \\ \eta_{\mathsf{ct}_{\mathsf{r}}} \end{pmatrix} = u_{\mathsf{ct}_{\mathsf{r}}} \pmod q.$$

Second, the LacE rows are tied to one carried-mask system. Writing

$$\mathsf{ct}_{\mathsf{lac}j} = (C_{0,j}, \ldots, C_{\ell-1,j}), \qquad C_{i,j} \in R_q^{2m},$$

the relation requires

$$a_{0,j} = r_j, \qquad a_{\ell,j} = s_j,$$

and, for each depth $i$,

$$C_{i,j} = a_{i,j} \cdot (A_0 \ A_1) + a_{i+1,j} \cdot \widehat{G}_{i,j} + E_{i,j},$$

where $\widehat{G}_{i,j}$ is the branch-dependent gadget selector on the path to leaf $j$. Third, all witness coordinates satisfy their stated bounds under canonical coefficient embedding:

$$\|r_j\|_\infty, \|s_j\|_\infty, \|a_{i,j}\|_\infty, \|\mathsf{sk}_{\mathsf{br}}\|_\infty \le \Gamma_{\mathsf{ring}} \quad \text{for all } i, j,$$
$$\|\eta_{\mathsf{ct}_\mathsf{r}}\|_\infty \le \Gamma_{\mathsf{br}}, \qquad \|E_{i,j}\|_\infty \le \Gamma_{\mathsf{row}} \quad \text{for all } i, j.$$

For vector or matrix witnesses, $\|\cdot\|_\infty$ denotes the maximum coefficient norm over all entries.

The reason this is the right statement is the telescope it enforces. For a root-to-leaf path with node values $Y = (y_0, \ldots, y_\ell)$, where $y_0 = d_x$ and $y_\ell = x_j$, let

$$z_i := \begin{pmatrix} -g^{-1}(y_{i,L}) \\ -g^{-1}(y_{i,R}) \end{pmatrix} \in R_q^{2m}.$$

Using the digest algebra,

$$y_i = (A_0 \ A_1) z_i, \qquad \widehat{G}_{i,j} z_i = -y_{i+1}.$$

Multiplying the row equation by $z_i$ and summing over the path gives

$$\sum_{i=0}^{\ell-1} C_{i,j} z_i = r_j d_x - s_j x_j + \sum_{i=0}^{\ell-1} E_{i,j} z_i.$$

Thus the setup proof certifies exactly the reusable translation package: the public rows are not just syntactically well formed, but are bound to one root mask, one outgoing block secret, and one noisy carried-mask telescope.

The widened root wrapper is certified by the same kind of relation, with the dimensions and public data of SetupRoot from Figure 8. We include $B^*$ explicitly in the public statement, since it is sampled by the sender and later used by the receiver to form the randomized root digest:

$$\mathsf{st}^{\mathsf{rt}}_{\mathsf{setup}} = (\mathsf{pp}^{\mathsf{rt}}, \mathsf{ct}_{\mathsf{lac,rt}}, \mathsf{ct}_{\mathsf{r,rt}}, B^*), \qquad B^* \in R_q^{1\times m}.$$

Writing $\mathsf{ct}_{\mathsf{lac,rt}} = (\mathbf{C}^{\mathsf{top}}_{\mathsf{rt}}, \mathsf{ct}_{\mathsf{lacrt},L})$, the root witness contains

$$w^{\mathsf{rt}}_{\mathsf{setup}} = (\hat{\mathsf{sk}}_\mathsf{r}, \mathbf{r}^{(1)}_{\mathsf{rt}}, \mathbf{r}^{(2)}_{\mathsf{rt}}, \mathsf{sk}_{\mathsf{r,rt}}, \{E^{\mathsf{top}}_{\mathsf{rt}}[\mathsf{ind}]\}_{\mathsf{ind}}, \eta_{\mathsf{lac,rt}}, \eta_{\mathsf{r,rt}}).$$

Here $\hat{\mathsf{sk}}_\mathsf{r} \in R_q^{\tau\rho}$ is the CRT-expanded root secret, $\mathbf{r}^{(2)}_{\mathsf{rt}}$ and $\eta_{\mathsf{lac,rt}}$ open $\mathsf{ct}_{\mathsf{lacrt},L}$ as an $\mathsf{LacE}$-encryption of $\hat{\mathsf{sk}}_\mathsf{r}$, and $\mathsf{sk}_{\mathsf{r,rt}}$ and $\eta_{\mathsf{r,rt}}$ open $\mathsf{ct}_{\mathsf{r,rt}}$ as an $\mathsf{LHE.Enc}_1$-ciphertext of $\mathbf{r}^{(1)}_{\mathsf{rt}}$. The relation also computes

$$B' := (\mathbf{b}_1^\top \ B^*) \in R_q^{1\times(\tau+m)}$$

from the public $B^*$ and requires, for every root row index $\mathsf{ind}$,

$$\mathbf{C}^{\mathsf{top}}_{\mathsf{rt}}[\mathsf{ind}] = \mathbf{r}^{(1)}_{\mathsf{rt}}[\mathsf{ind}] \cdot B' - \mathbf{r}^{(2)}_{\mathsf{rt}}[\mathsf{ind}] \cdot (\mathbf{g}^\top \ 0^m) + E^{\mathsf{top}}_{\mathsf{rt}}[\mathsf{ind}].$$

All root witness coordinates satisfy the same style of coefficient bounds as above: ordinary ring-valued witnesses are bounded by $\Gamma_{\mathsf{ring}}$, while the encryption-opening noises and the top-row errors are bounded by the corresponding bridge and row-noise bounds in $\mathsf{pp}^{\mathsf{rt}}$. Thus the root proof binds the public root package to the hidden root state $(\mathbf{r}^{(1)}_{\mathsf{rt}}, \mathsf{sk}_{\mathsf{r,rt}})$ used in the online root response.

### 10.1.4 Interactive Online Response

In the interactive online mode, after receiving the receiver's randomized root digest $d'_{\mathsf{rt}}$, the sender returns only $\mathsf{sk_{rt}}'$, plus the optional check challenge in the random-linear-check variant. There is no online NIZK and no golden-seed claim. The $H_{\mathsf{ct}}$ labels are keyed by $(\mathsf{id}, H_{\mathsf{seed}}(\mathsf{id}, \mathsf{sk_{rt}}'))$, so $\mathsf{sk_{rt}}'$ is the public value that separates this online call from other calls. For an honest sender, $\mathsf{sk_{rt}}'$ has the form

$$\mathsf{sk_{rt}}' = \mathsf{sk}_{\mathsf{r,rt}}^{\top}\mathbf{g}^{-1}(d'_{\mathsf{rt}}) + k'_{\mathsf{rt}} \qquad \text{for uniform } k'_{\mathsf{rt}} \in \mathcal{R}_q.$$

For a malicious sender, once the setup proof has extracted $\mathsf{sk}_{\mathsf{r,rt}}$, the same equation is read as the definition of the effective root-side mask $k'_{\mathsf{rt}}$. The root-side affine relation

$$m_{d_x}^{\mathsf{root}} = r \cdot d_x + k_{d_x}^{\mathsf{root}}$$

then holds with the same root carried-mask vector $r$ fixed by the setup proof.

**Instantiating the sender proofs.** The sender-side setup statement is a structured statement about short witnesses to exact modular linear relations. For setup, after flattening ring elements into coefficient vectors over $\mathbb{Z}_q$, the algebraic part has the form

$$M_{\mathsf{setup}} w_{\mathsf{setup}} = u_{\mathsf{setup}} \pmod q.$$

The rows of $M_{\mathsf{setup}}$ encode the bridge-opening equations, the endpoint equations for the carried-mask chains, and the row-programming equations for the public LacE ciphertext rows. The proof also enforces the concrete bounds $\Gamma_{\mathsf{ring}}, \Gamma_{\mathsf{br}}, \Gamma_{\mathsf{row}}$ on the corresponding witness coordinates.

We instantiate these proofs using the Fiat–Shamir NIZK argument-of-knowledge interface from Section 3.8, using lattice proofs for bounded solutions to exact modular linear equations in the style of Bootle–Lyubashevsky–Seiler and the later LNP framework [BLS19, LNP22]. For any public modulus $q$, public matrix $M$, public target $u$, and public coordinate-wise bound vector $\Gamma$, the proof system gives a proof $\pi$ for the predicate $R_{\mathsf{lin}}(M, u, \Gamma; w)$, which checks $Mw = u \pmod q$ and $\|w_t\|_\infty \le \Gamma_t$ for every bounded coordinate $t$. The malicious theorem pays a union bound over all setup-package proofs. We apply this interface once to each arity-2 ordinary setup statement and once to the arity-4 widened root setup statement, using the bounds in $\mathsf{pp}^{(h)}$ or $\mathsf{pp}^{\mathsf{rt}}$. This baseline certifies the modular decomposition directly; batching across leaves, recursion levels, or repeated bridge instances can be added later without changing the malicious protocol interface.

### 10.1.5 Receiver-Input Extraction

For a malicious receiver, the simulator must recover one effective input for the final CI-VOLE output. We consider two extraction routes.

**Digest-knowledge extractability assumption.** We propose the digest-knowledge extractability assumption in Section E.4.1 as a knowledge-extractability assumption, analogous in spirit to the AGM. It builds on the bounded digest binding property established in Section A.4 under the SIS assumption: a malicious receiver cannot produce two distinct bounded openings of the same digest, except with negligible probability. Under this assumption, an extractor can recover the admissible opening underlying the receiver's successful use of a digest from the receiver's computation. Applying this extractor recursively below the public root digest yields an effective receiver input. This route adds no receiver-side protocol steps, computation, or communication; the certified sender setup remains required.

**Random-linear check.** Alternatively, we append the random-linear-combination check of Section 10.1.6. This route sacrifices a few final output entries for masking and adds a consistency check. That section gives the construction and the rewinding argument for extracting the effective input.

### 10.1.6 Optional Receiver-Side Random-Linear Check

We describe the random-linear-combination check using standard correlation-checking techniques [KOS15, YWL+20, Roy22]. This check is used only to bind and extract the receiver's effective input; it does not change the shrink/expand core.

In this variant, the check challenge is carried alongside the online response. In interactive mode, the sender samples an independent $\rho_{\mathsf{chk}}$ and sends

$$(\mathsf{sk}_{\mathsf{rt}}{}', \rho_{\mathsf{chk}}).$$

In non-interactive mode, the online random oracle instead returns

$$(\mathsf{sk}_{\mathsf{rt}}{}', \varsigma, \rho_{\mathsf{chk}}) := H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}}).$$

The coefficients for the final check are derived from $\rho_{\mathsf{chk}}$. Thus the interactive check uses an honestly sampled statistical challenge, while the non-interactive version relies on the usual computational Fiat–Shamir/random-oracle treatment.

We describe the adaptor for an application output of length $w$. The protocol is run for a total length

$$W \geq w + \tau_{\mathsf{chk}},$$

where the first $w$ coordinates are the application output and the next $\tau_{\mathsf{chk}}$ coordinates are sacrificed for checking. If the underlying recursive protocol only supports particular widths, $W$ is the next supported padded width and any remaining coordinates are discarded. Thus the final VOLE relation is

$$m_i = \Delta \cdot x_i + k_i \qquad i \in [W],$$

over $\mathcal{R}_q$. This is a check on the final output relation, not on the internal digest tree, gadget decomposition, or CRT details. The receiver holds $x, m$, the sender holds $\Delta, k$, and the random linear combinations below are taken over the first $w$ application coordinates. If an application prefers the top-level field formulation, the same wrapper can instead be applied after the final field projection.

Let $T$ be the challenge-free prefix for the check. In the interactive mode, this is all protocol messages up to and including the receiver obtaining $\mathsf{sk}_{\mathsf{rt}}{}'$, but not the check challenge. In the non-interactive mode, this is the transcript up to the online random-oracle query $d'_{\mathsf{rt}}$, together with fixed first two answer components $(\mathsf{sk}_{\mathsf{rt}}{}', \varsigma)$; each probe answers the same query with these fixed components and a fresh challenge field. The challenge seed $\rho_{\mathsf{chk}}$ is sampled by the sender in interactive mode, and is the third component of $H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}})$ in non-interactive mode. The parties derive from $\rho_{\mathsf{chk}}$ independent coefficient vectors

$$\chi^{(t)} = (\chi_1^{(t)}, \ldots, \chi_w^{(t)}) \in \mathcal{R}_q^w, \qquad t \in [\tau_{\mathsf{chk}}].$$

For each $t$, let $j_t := w + t$ denote the corresponding sacrificed coordinate. This coordinate itself supplies the scalar check-VOLE correlation: the sender's mask is

$$y_t := k_{j_t},$$

while the receiver's check input and output are

$$u_t := x_{j_t}, \qquad z_t := m_{j_t}.$$

By the final VOLE relation,

$$z_t = y_t + u_t \cdot \Delta.$$

The application ignores these sacrificed coordinates after the check. The adaptor then runs the following check.

1. The receiver computes
$$a_t := \sum_{i\in[w]} \chi_i^{(t)} x_i$$
and sends $u'_t := a_t + u_t$ to the sender.
2. The sender computes
$$c_t := \sum_{i\in[w]} \chi_i^{(t)} k_i, \qquad Y_t := y_t + u'_t \cdot \Delta, \qquad V_t := c_t + Y_t,$$
and sends $h_t := H_{\mathsf{chk}}(T, t, V_t)$ to the receiver.
3. The receiver computes
$$b_t := \sum_{i\in[w]} \chi_i^{(t)} m_i, \qquad W_t := b_t + z_t,$$
and accepts only if $h_t = H_{\mathsf{chk}}(T, t, W_t)$ for every $t$.

Correctness is the identity
$$V_t = \sum_i \chi_i^{(t)} k_i + \Big(\sum_i \chi_i^{(t)} x_i + u_t\Big)\Delta + y_t = \sum_i \chi_i^{(t)} m_i + z_t = W_t.$$

The hash hides the sender's checked value, while the scalar check-VOLE masks the receiver's random linear view from the sender. No separate check-VOLE source is needed; the check correlations are ordinary output coordinates that are reserved for this purpose. For soundness parameter $\lambda$, choose $\tau_{\mathsf{chk}} = \lceil \lambda/\beta_{\mathsf{chk}} \rceil$, where $\beta_{\mathsf{chk}}$ is the per-check soundness in bits of the chosen linear hash family over $\mathcal{R}_q^w$. For example, if one works after a field projection to a field of size $2^b$, then fully random coefficients give $\beta_{\mathsf{chk}} = b$; for a polynomial/power hash over an output of length $w$, the usual almost-universality loss reduces this by about $\lceil \log_2 w \rceil$. Thus a check domain with 128 bits of per-check soundness needs one repetition for 128-bit soundness, whereas smaller checked domains require enough independent repetitions that the accumulated check entropy reaches $\lambda$.

For extraction, the extractor repeatedly rewinds the receiver to the point after its challenge-free transcript $T$ is fixed, keeping the same $\mathsf{sk_{rt}}'$ and issuing a fresh independently sampled set of random coefficients for each check. In the non-interactive mode this means reprogramming only the challenge field of the online random-oracle answer. In an accepting rewind, the receiver message gives the affine equation
$$u'_t = \sum_{i\in[w]} \chi_i^{(t)} x_i + x_{j_t}.$$

The sacrificed entries are fixed by the prefix just like the application entries. The extractor collects enough such accepting responses to obtain a full-rank system in the unknowns
$$(x_1, \ldots, x_w, x_{j_1}, \ldots, x_{j_{\tau_{\mathsf{chk}}}}).$$

Solving this system recovers the application part of the effective input. The formal extraction argument is carried out in the corrupted-receiver hybrid proof of Theorem 11.

The extraction proof also needs a binding condition. The check alone would not rule out a receiver that somehow prepares two complete VOLE explanations for the same public transcript, one for $x$ and one for $x' \neq x$, and answers consistently with either one. The random-linear-check proof does not use digest binding to exclude this pathology. It uses a protocol-specific binding fact: after the sender-side ciphertexts have been simulated, the receiver's transcript is independent of $\Delta$, while two distinct final-output VOLE explanations under the same sender mask give a nontrivial equation

involving $\Delta$. Thus a receiver that can keep both explanations alive would effectively predict a fresh authenticated relation involving $\Delta$, which occurs only with negligible probability in the full proof.

The proof section treats the prefix $T$, the hash family, the number of rewinds, and the $\Delta$-independence condition as explicit parameters of the extraction lemma.

# 11 IT-MAC-based ZKP

Our ZK application uses LogVOLE to remove the linear derandomization step from VOLE-based IT-MAC proofs. Prior IT-MAC-based ZKP follow the paradigm that they first obtain random VOLE tuples and then send a correction vector for the prover's actual execution trace. Here, the prover instead uses LogVOLE to obtain authenticated shares directly on its extended witness.

**Standard IT-MACs for Arithmetic Verification Circuits.** CI-VOLE can be straight-forwardly use the standard IT-MAC checking [YSWW21] to prove that its wire values satisfy each arithmetic gate of the circuit. The prover forms the extended witness $\mathbf{x} = (x_1, \ldots, x_N)$ and obtains labels

$$m_j = k_j + x_j\Delta \qquad j \in [N].$$

Addition and output checks are affine. For a multiplication gate $x_a x_b = x_c$, define

$$\epsilon := x_a x_b - x_c.$$

With

$$U := k_a k_b + k_c\Delta, \quad V_2 := m_a m_b, \quad V_1 := m_c - x_a m_b - x_b m_a,$$

the verifier equation is

$$U = V_2 + V_1\Delta + \epsilon\Delta^2.$$

Thus an honest gate has $\epsilon = 0$, and a bad gate leaves a quadratic error term. Random batching compresses all multiplication, addition, and output checks into one equation of the form

$$\widehat{U} = \widehat{V}_2 + \widehat{V}_1\Delta.$$

The prover commits to $(\widehat{V}_1, \widehat{V}_2)$ before the verifier opens $\Delta$, so it cannot repair this equation after seeing the MAC key.

The complete, self-contained protocol and proof sketch are given in Section F.

**Knowledge soundness by binding the witness.** The standard IT-MAC-based ZKP paradigm achieves soundness because of the randomness of $\Delta$ chosen by the verifier and LogVOLE's privacy of $\Delta$. In fact, LogVOLE can further guarantee knowledge soundness, *i.e.*, our proof system guarantees that the prover knows a valid witness. It requires an extractor that recovers a valid extended witness $\mathbf{x}$ from a prover that produces accepting proofs with sufficiently high probability. To this end, we can use either receiver-input extraction approach described in Section 10.1.5: the random-linear check of Section 10.1.6, or the digest-knowledge extractability assumption of Section E.4.1, which imposes no extra protocol steps. Section F applies the extraction argument to the opened ZK protocol.

Table 3: Comparison of (Concretely Efficient) IT-MAC-based ZK schemes in terms of extended witness size $|\mathbf{x}|$.

| Scheme | Functionalities Used | Assumptions | Total Communication | Computation |
|---|---|---|---|---|
| Pure-VOLE-based[#] | VOLE + De-random. | *e.g.*, LPN[!] | $O(\|\mathbf{x}\|)$ | $O(\|\mathbf{x}\|)$ Field ops |
| AntMan [WYY+22][†] | VOLE + AHE | *e.g.*, LPN + RLWE[!] | $O_\lambda(\|\mathbf{x}\|^{3/4})$ | $O(\|\mathbf{x}\|/n)$ $\mathsf{Mult}_\mathcal{R}$ + $O(\|\mathbf{x}\|)$ Field ops |
| **Ours** | CI-VOLE | RLWE | $O(\lambda(\log\|\mathbf{x}\|+\sigma)^2)$ | $O(\|\mathbf{x}\|/n)$ $\mathsf{Mult}_\mathcal{R}$ |

- $\lambda$ and $\sigma$ are the computational and statistical security parameters, respectively.
- Pure-VOLE-based schemes, including [WYKW21, BMRS21, DIO21b, YSWW21, DILO22], require derandomization and thus incur linear communication.
- These schemes can use VOLE in a block-box way, but we assume they adopt the (most efficient and popular) LPN-based VOLE and/or RLWE-based AHE protocols as in their implementation.

### 11.1 Upgrading to Malicious Verifier Security

In some specific ZK application, it is desirable to the protoocl secure against a malicious verifier, rather than a semi-honest one. In this case, the prover needs to check the LogVOLE setup is well formed. We describe two ways to perform this check.

**Reusable public-key setup against malicious verifier.** Here the verifier is the LogVOLE sender. For a reusable public key, the prover verifies the sender-side consistency proofs of Section 10.1.3 before using the setup.

**Opened fresh-setup against malicious verifier.** Alternatively, the verifier generates a fresh semi-honest LogVOLE setup for each proof session, including the golden-seed optimization. After the prover commits to the proof values that must be fixed before learning $\Delta$, the verifier reveals $\Delta$ and all setup coins, including encryption randomness, noise, and seeds. The prover recomputes the prover-visible setup and opens its committed proof values only if the check succeeds. This replaces sender-side NIZKs with a direct setup check and retains the optimized semi-honest protocol. Revealing $\Delta$ makes this setup single use.

Below is the formal statement about our LogVOLE-based ZK protocol, adopting the open fresh-setup against malicious verifier.

**Theorem 3** (ZK from opened LogVOLE setup)**.** *Assume the semi-honest security of the fresh LogVOLE execution, the binding of the commitment scheme, and the soundness of the random-linear-check adaptor of Section 10.1.6. Then the protocol in Figure 17 is a designated-verifier zero-knowledge proof with soundness error* $\mathsf{negl}(\lambda) + O(1/p)$ *per independent field check. For an extended witness* $\mathbf{x}$ *over a field with* $\log p = O(\sigma)$*, the online communication from LogVOLE is* $O(\lambda(\log|\mathbf{x}|+\sigma)^2)$*, plus* $O(1)$ *field elements and the small random-linear-check overhead.*

## 12 Performance and Evaluation

### 12.1 Cost Analysis

In this section, we analyze the cost of our optimized, interactive construction LogVOLE. The accounting uses the lower-level realization in Section 6 and the ShrinkExpand construction in Figure 9.

**Basic Parameters.** We count the interactive protocol with the golden-seed optimization, whose ring-level setup and query appear in Figures 8, 10 and 11. Let $N := |\mathbf{x}|$ be the number of requested $\mathbb{Z}_p$-labels, and let $m_{\mathsf{msg}} := \log p$ be the bit length of a $\mathbb{Z}_p$-label. Let $w := \lceil N/n \rceil$ be the number of ring elements after CRT batching. $|\mathcal{R}_q| = n \log q$ is a ring element's size. Golden-seed grinding gives $\log q = O(m_{\mathsf{msg}} + \log N)$; with $n = O(\lambda \log q)$, we have $|\mathcal{R}_q| = O(\lambda(m_{\mathsf{msg}} + \log N)^2)$ bits.

We use $\tau$ for the gadget width. Set $\mu := \alpha\rho\tau$ as the batch width of one non-root $\mathsf{ShrinkExpand}$ block: each block shrinks by $\alpha$, carries $\rho$ CRT residues, and uses $\tau$ gadget digits. Note that $w = \tau\rho\alpha^{\ell}$ for $\ell = \log_{\alpha}(w/(\tau\rho))$. Also, the constants $\alpha, \rho, \tau$ do *not* depend on $N$.

**(One-time) Setup Communication.** Setup communication is the material sent once before any query. The non-root shrink/expand setup consists of the offline $\mathsf{ShrinkExpand}_{\mathcal{R}}$ materials with

$$|\mathsf{ct}_{\mathsf{r}}| = O(\mu\tau) \qquad \text{and} \qquad |\mathsf{ct}_{\mathsf{lac}}| = O(\mu\tau\log\mu)$$

ring elements, and hence costs $O(\mu\tau\log\mu\,|\mathcal{R}_q|)$ bits. This cost does not scale with $\ell$: internal nodes in our recursion tree use the same $\mathsf{LHE}$ key $\mathsf{sk}$ (see Section 4.2), so one $(\mathsf{ct}_{\mathsf{lac}}, \mathsf{ct}_{\mathsf{r}})$ state supports every internal level. Adding the cost of the randomized-root wrapper with randomizer width $m_{\mathsf{rt}}$, the setup cost is:

$$O(\alpha\rho\tau^2\log(\alpha\rho\tau)\cdot|\mathcal{R}_q| + (\rho\tau^2 + \tau m_{\mathsf{rt}})\log(\rho\tau + m_{\mathsf{rt}})\cdot|\mathcal{R}_q|).$$

For fixed $\alpha, \rho, \tau, m_{\mathsf{rt}}$, this setup cost is $O(|\mathcal{R}_q|) = O(\lambda(m_{\mathsf{msg}} + \log N)^2)$ bits.

**Per-query Communication.** For each query, the parties only need to exchange the randomized root digest and the root key. In the interactive mode, $\mathbb{R}$ sends $d'_{\mathsf{rt}} \in \mathcal{R}_q$, and $\mathbb{S}$ replies with $\mathsf{sk}_{\mathsf{rt}}{}' \in \mathcal{R}_q$ and a $\lambda$-bit golden seed, for $2|\mathcal{R}_q| + \lambda = O(|\mathcal{R}_q|) = O(\lambda(m_{\mathsf{msg}} + \log N)^2)$ bits. For the non-interactive mode, $\mathbb{S}$ need not send $\mathsf{sk}_{\mathsf{rt}}{}'$, halving the communication cost.

**(One-time) Setup Computation.** We count computation in $\mathsf{Mult}_{\mathcal{R}}$, since ring multiplications are the dominant operations. The setup cost comes from generating the two sets of $\mathsf{ShrinkExpand}$ material (one for the leaf level, one for the internal levels). It costs $O(\mu\tau\log\mu)$ ring operations. In addition, the root-randomization wrapper contributes the same fixed-width setup term as in the communication count, giving the total cost

$$O(\alpha\rho\tau^2\log(\alpha\rho\tau) + (\rho\tau^2 + \tau m_{\mathsf{rt}})\log(\rho\tau + m_{\mathsf{rt}}))\,\mathsf{Mult}_{\mathcal{R}}.$$

**Per-query Computation.** When a query walks down the recursion tree, it incurs $w + w/\alpha + w/\alpha^2 + \cdots = O(w)$ total inputs to $\mathsf{ShrinkExpand}$ instances. Moreover, the $\mathsf{LacE}$ evaluation for each ring element follows a path of length $\log\mu$, and each path step has $\tau$ gadget digits. Thus, it touches $O(w\tau\log\mu)$ ring elements. Observe that we can reuse the NTT representation to save computation: Since $\mathsf{ct}_{\mathsf{lac}}$ is fixed, the parties can store it in the NTT form, so the query can reuse that representation. Notice that a full $\mathsf{Mult}_{\mathcal{R}}$ includes the expensive NTT/INTT transforms, whereas $\mathsf{Mult}^{\not{\mathsf{NTT}}}_{\mathcal{R}_q}$ only requires element-wise multiplication with NTT reuse. In our parameter regime, $\tau\log\mu < \log n$, so these $O(\tau\log\mu)$ $\mathsf{Mult}^{\not{\mathsf{NTT}}}_{\mathcal{R}_q}$ operations are cheaper than an ordinary $\mathsf{Mult}_{\mathcal{R}}$ (with $O(\log n)$ overhead). In addition to the $\mathsf{ShrinkExpand}$ evaluation, the remaining $\mathsf{LHE}/\mathsf{LacE}$ evaluations, denoising, and CRT/gadget recombination contribute another $O(w)$ $\mathsf{Mult}_{\mathcal{R}}$, giving the query cost

$$O(w)\,\mathsf{Mult}_{\mathcal{R}} + O(w\tau\log\mu)\,\mathsf{Mult}^{\not{\mathsf{NTT}}}_{\mathcal{R}_q}.$$

We keep the $\mathsf{Mult}^{\not{\mathsf{NTT}}}_{\mathcal{R}_q}$ term explicit for concrete accounting, but under $\tau\log\mu < \log n$ it is absorbed by the $O(w)\,\mathsf{Mult}_{\mathcal{R}}$ term. Thus the per-query cost is $O(N/n)$ ring operations. Sections 6.1 and 6.2 give the CRT/gadget details.

## 12.2 Evaluation

**Parameter Setting.** We evaluate two instantiations of $\mathsf{LogVOLE}$, targeting $\lambda = 128$ computational security and $\sigma = 40$ statistical security. Both use one-digit gadget truncation, $\alpha = 2$, and $\tau = 2$. The interactive instantiation uses the golden-seed optimization, ring degree $n = 8192$, and $\rho = 4$ CRT

Table 4: Concrete communication for the two parameter settings.

| Setting | Setup | Per-query payload | | |
|---|---|---|---|---|
| | | Staged | Non-Staged | Non-interactive |
| Goldenseed | 39.6 MB | 451 kB | 226 kB | — |
| No Goldenseed | 397.4 MB | 1704 kB | 852 kB | 852 kB |

**Staged**: chosen-input VOLE where the sender obtains its masks before the receiver fixes its input. **Non-Staged**: chosen-input VOLE without this timing requirement, using the digest-derived root key to omit one ring element; this suffices for applications such as ZKP. **Non-interactive**: public-key CI-VOLE with one receiver message and no sender reply. — denotes an unsupported mode.

primes of 55 bits each, giving $\log_2 q \approx 220$, $g = 2^{110}$, and $\mu = \alpha\rho\tau = 16$. It uses a 54-bit plaintext prime $p$.[9] The instantiation without golden-seed grinding uses $n = 16384$, $\rho = 8$ CRT primes of 52 bits each, $\log_2 q \approx 416$, $g = 2^{208}$, and $\mu = 32$, with a 51-bit plaintext prime $p$. This setting supports the non-interactive transformation in Section 4.4, without the golden-seed optimization. We call these settings *Interactive (Optimized with Goldenseed trick)* and *Non-interactive* throughout the evaluation. The benchmarks time the arithmetic of the two settings; the non-interactive communication costs below apply the public-key transformation to the setting without golden-seed grinding.

**Implementation and Hardware.** Our C++ implementation builds on TinyLabels [DLL25] and Microsoft SEAL [Mic23]. The experiments use Georgia Tech's PACE cluster. The parties run in separate processes and communicate over loopback TCP. The pure-CPU evaluations use Intel Xeon 6972P processors. We assign disjoint sets of 1, 2, 4, 8, or 16 physical cores to each party. The CPU build uses GCC 14.2.0, SEAL 4.1.1, and Intel HEXL 1.2.5 with AVX-512 enabled. The GPU runs place both parties on one NVIDIA H200 with 140 GiB of device memory, with 4 cores per party on an Intel Xeon Platinum 8562Y+ host. The GPU build uses SEAL 4.1.1 without HEXL, CUDA 13.3, and driver 595.71.05. Online GPU throughput uses inputs and outputs resident on the device.[10]

Our online timings measure the first query or proof after a fresh setup. Later executions that reuse the setup (*i.e.*, warmed executions) can achieve 50–100% higher online throughput.

**Communication Costs.** Table 4 reports the concrete setup and query costs, including the non-staged variant described in Section 4.4. Throughout the paper, communication sizes use decimal units: 1 kB $= 10^3$ bytes, 1 MB $= 10^6$ bytes, and 1 GB $= 10^9$ bytes. Our query sizes are rounded up to whole kB.

**Break-even Point of LogVOLE's Setup Cost vs. random-VOLE's Derandomization.** Random VOLE followed by derandomization sends one field element per tuple, *i.e.*, $N\lceil \log_2 p \rceil$ bits for $N$ tuples, in addition to their setup cost of generating the random VOLEs. LogVOLE's setup plus query communication is smaller than this correction payload once $N$ exceeds approximately 5.90 million with golden-seed optimization, or 62.5 million without it, using the 54- and 51-bit fields, respectively.

**$\mathbb{Z}_p$-Correlation Throughput.** Table 5 gives online and end-to-end correlation throughput for both settings on CPU and GPU. At 16 CPU cores per party, online throughput is 13.87 million correlations/s with golden-seed grinding and 10.29 million without it; end-to-end throughput is 11.17 and 5.08 million correlations/s. On the H200, online throughput is 532.30 and 285.71 million correlations/s; end-to-end throughput is 326.88 and 115.71 million correlations/s.

[9] IT-MAC-based ZKP can achieve $\sigma$-bit statistical soundness over a field of size $p > 2^\sigma$. Prior work [YSWW21, LHLA25a] mainly uses the 61-bit Mersenne prime $p = 2^{61} - 1$ because it permits efficient modular arithmetic.

[10] GPU parties share one H200 and use 4 disjoint physical host cores each; CPU parties use disjoint physical cores on one NUMA node. Online timing ends when both parties finish.

Table 5: Throughput in millions/s: CI-VOLE correlations, ZKP multiplication gates, and garbled input labels.

| Hardware | Correlations | | ZKP | | Garbled input labels | |
|---|---|---|---|---|---|---|
| | Online | End-to-end | Online | End-to-end | Online | End-to-end |
| **Interactive (Optimized with Goldenseed trick)** | | | | | | |
| CPU, 1 core | 1.40 | 1.37 | 1.33 | 1.28 | 0.39 | 0.38 |
| CPU, 2 cores | 2.65 | 2.54 | 2.19 | 2.06 | 0.87 | 0.84 |
| CPU, 4 cores | 5.06 | 4.66 | 3.93 | 3.60 | 1.62 | 1.52 |
| CPU, 8 cores | 6.23 | 5.64 | 6.61 | 5.74 | 2.45 | 2.22 |
| CPU, 16 cores | 13.87 | 11.17 | 12.95 | 9.95 | 4.26 | 3.62 |
| GPU | 532.30 | 326.88 | 451.73 | 280.33 | 136.69 | 93.40 |
| **Non-interactive / without Goldenseed Optimization** | | | | | | |
| CPU, 1 core | 0.98 | 0.89 | 0.95 | 0.85 | 0.32 | 0.29 |
| CPU, 2 cores | 1.94 | 1.62 | 1.87 | 1.54 | 0.61 | 0.51 |
| CPU, 4 cores | 3.94 | 2.83 | 3.76 | 2.66 | 1.18 | 0.86 |
| CPU, 8 cores | 6.79 | 4.05 | 6.58 | 3.83 | 1.97 | 1.22 |
| CPU, 16 cores | 10.29 | 5.08 | 9.99 | 4.78 | 2.89 | 1.53 |
| GPU | 285.71 | 115.71 | 275.28 | 111.29 | 82.08 | 36.51 |

CPU core counts are physical cores per party. GPU uses one H200 and 4 physical host cores per party. All listed rates include loopback TCP, without network emulation. Each ZKP proof checks 33,038,208 multiplication gates; each garbled input workload produces 11,184,810 labels.

**ZK Multiplication Throughput.** The ZKP columns of Table 5 report multiplication-gate throughput for the semi-honest verifier protocol, instantiated with LogVOLE. Following JesseQ [LHLA25a], we use a multiplication-chain workload; the benchmark checks 33,038,208 multiplication gates per proof.[11] With 16 CPU cores per party, online throughput is 12.95 and 9.99 million gates/s for the interactive and non-interactive settings, respectively; end-to-end throughput is 9.95 and 4.78 million gates/s. The H200 achieves 451.73 and 275.28 million gates/s online, or 280.33 and 111.29 million gates/s end-to-end. Figure 17 gives the concrete ZKP protocol.

**Performance Gain with Various Bandwidths.** Figure 12 projects the online speedup over a ZK scheme that derandomizes random VOLE tuples. To favor the derandomization approach, we count only its transmission time and omit its computation. For $G = 33{,}038{,}208$ multiplication gates, its payload is $G\lceil \log_2 p \rceil$ bits, using the matching 54- or 51-bit field. This counts one correction per gate, as in [YSWW21, LHLA25a].[12] When counting the communication time for LogVOLE, we use the non-staged optimization in Section 4.4: the interactive query contains one ring element and the golden seed. Including the small ZK check messages gives 226 kB for the interactive setting and 853 kB for the non-interactive setting. The projection excludes round-trip latency. At 100 Mbps, the projected speedups are 6.94× and 4.99× on CPU, and 195.7× and 89.5× on GPU, in the same setting order. At 10 Mbps, they are 65.3×, 42.2×, 704.0×, and 210.2×, respectively.

**Matrix Multiplication.** Table 6 shows the separate $1024 \times 1024$ matrix-multiplication benchmark. We use one thread per party, as in prior works [Set20, YSWW21, LHLA25a, LHLA25b]. The proof checks the $n^2$ inner products of the matrix product.

**Garbled Input Compression.** Table 7 compares setup and per-query communication with TinyLabels.

The garbled-input-label columns of Table 5 report throughput for generating garbled input labels with (interactive) staged $\mathsf{LogVOLE}_{\mathcal{R}}$. We concatenate three $\mathbb{Z}_p$-labels into each garbled input

[11] Addition gates require no additional labels; their consistency checks use local additions over labels.

[12] Some protocols exploit branching or layered circuits to reduce communication [WYY+22, YHH+23, LHLA25a]. The projection uses the arbitrary-circuit cost; LogVOLE can independently benefit from such optimizations.

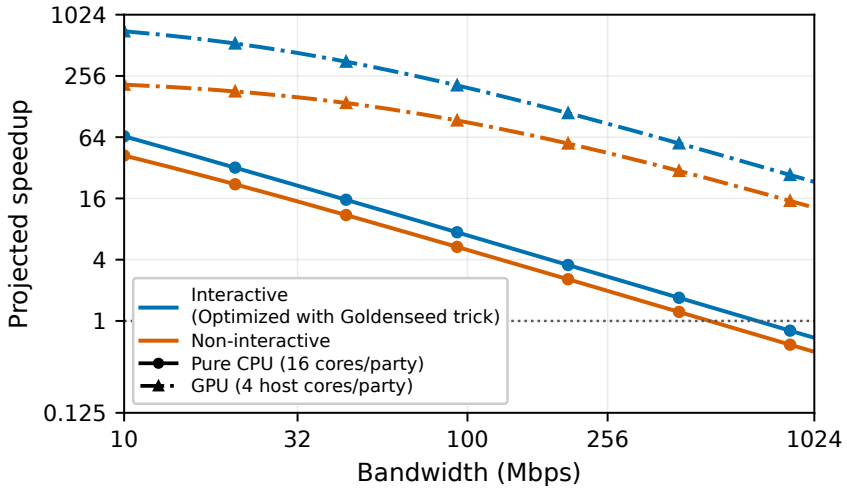


Figure 12: Projected online ZKP speedup over VOLE derandomization under various bandwidths.

label to meet $\lambda = 128$ computational security in both parameter settings. At 16 CPU cores per party, online throughput is 4.26 and 2.89 million garbled labels/s with and without golden-seed grinding. On the H200, it is 136.69 and 82.08 million labels/s.

## 13 Discussion

**Committed VOLE.** LogVOLE can be viewed as the *dual* of committed VOLE [SLY+25]: committed VOLE commits to $\mathbf{x}$ during one-time setup and reuses it across different $\Delta$, whereas LogVOLE commits to $\Delta$ during setup and reuses it for different $\mathbf{x}$. Extending LogVOLE to support committed VOLE with sublinear total communication is an interesting direction for future work.

**ZKP.** An open direction is to combine LogVOLE with VOLE-based ZK execution, branching, and memory access [JKY26, YHH+23, YHH+24, HHK+24, YH24] to reduce computation while retaining succinct communication. Another direction is to extend our designated-verifier ZKP to a group of mutually untrusted verifiers, potentially using a distributed trusted setup [BGM17, GKM+18, NMM+25]. Further applications include proofs of LLM inference and neural-network analysis [LWQ+24, DNY+26, WYX+21, HCL+24].

**Secure computation.** LogVOLE may reduce communication in privacy-preserving machine learning [NC23, NC21, NCW+21, KNMC26] through garbled-circuit input-label delivery and chosen-input correlation generation [LMSP21]. Quantifying the end-to-end savings remains future work.

Table 6: Online and total costs for proving $1024 \times 1024$ matrix multiplication with a public product.

| **Scheme** | **Network** | **Online** | **Communication** | |
|---|---|---|---|---|
| | | **runtime (s)** | **Online** | **Total** |
| *zk-SNARK* | | | | |
| Spartan [Set20] | Unspecified | $\geq 5000$ | $\leq 100$ kB | $\leq 100$ kB |
| Virgo [ZXZS19] | 500 Mbps | 357 | 221 kB | 221 kB |
| *IT-MAC-based* | | | | |
| Wolverine [WYKW21] | Localhost | 1627 | 30.7 GB* | — |
| Mac'n'Cheese [BMRS21] | Localhost | 2684 | 23.3 GB* | — |
| QuickSilver [YSWW21] | 500 Mbps | 10 | 22.7 MB* | $\approx$ 29.9 MB† |
| JesseQ [LHLA25a] | 500 Mbps | 7 | 15.1 MB* | $\approx$ 27.2 MB† |
| **Ours** | 500 Mbps | 4.24 | 226 kB | $\approx$ 39.8 MB |

For a fair comparison with prior work, our experiment uses a **single thread per party**. As in prior work, we do not use Freivalds' algorithm; using it can reduce our runtime to <1 s. Our matrix benchmark uses the golden-seed optimization with the communication optimization in the non-staged variant. Total communication includes all preprocessing for one proof with a fresh setup, without amortization. Prior runtimes are as reported in their papers. * Reported online communication is scaled by 55/61 from 61-bit fields to accommodate LogVOLE's 55-bit field size. † Total communication is estimated from the cited protocols and public OT/VOLE backend counts [EMP21, LHLA25b], including circuit preprocessing. The field-size related communication are scaled accordingly. They are cost estimates and not measurements of 55-bit implementations. — denotes unavailable totals.

Table 7: Communication cost for garbled-input compression.

| **Scheme** | **Setup** | **Per-Query** $N = 699{,}050$ | **Per-Query** $N = 11{,}184{,}810$ |
|---|---|---|---|
| **Ours** | 39.6 MB (55.6×) | 451 kB (1.2×) | 451 kB (18.6×) |
| TinyLabels [DLL25] | 2.2 GB* | 524 kB | 8.39 MB |

- Our row uses the golden-seed optimized version with staged CI-VOLE. • The setup cost marked with * is the concrete cost reported in the TinyLabels paper.
- For TinyLabels, per-query communication follows its concrete $6N$-bit cost estimation.
- The factors in the **Ours** row are the cost reductions relative to TinyLabels costs.

## References


[ABI+23] Benny Applebaum, Amos Beimel, Yuval Ishai, Eyal Kushilevitz, Tianren Liu, and Vinod Vaikuntanathan. Succinct computational secret sharing. In *55th ACM STOC*, pages 1553–1566. ACM Press, June 2023.

[ACC+19] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption standard. Cryptology ePrint Archive, Report 2019/939, 2019.

[AIKW13] Benny Applebaum, Yuval Ishai, Eyal Kushilevitz, and Brent Waters. Encoding functions with constant online rate or how to compress garbled circuits keys. In *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 166–184. Springer, Berlin, Heidelberg, August 2013.

[AMR25] Damiano Abram, Giulio Malavolta, and Lawrence Roy. Succinct oblivious tensor evaluation and applications: Adaptively-secure laconic function evaluation and trapdoor hashing for all circuits. In *57th ACM STOC*, pages 1875–1886. ACM Press, June 2025.

[ARS24] Damiano Abram, Lawrence Roy, and Peter Scholl. Succinct homomorphic secret sharing. In *EUROCRYPT 2024, Part VI*, volume 14656 of *LNCS*, pages 301–330. Springer, Cham, May 2024.

[BBC+24] Maxime Bombar, Dung Bui, Geoffroy Couteau, Alain Couvreur, Clément Ducros, and Sacha Servan-Schreiber. FOLEAGE: $\mathbb{F}_4$OLE-based multi-party computation for boolean circuits. In *ASIACRYPT 2024, Part VI*, volume 15489 of *LNCS*, pages 69–101. Springer, Singapore, December 2024.

[BBMHS22] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, and Peter Scholl. Moz$\mathbb{Z}_{2^k}$arella: Efficient vector-OLE and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. In *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 329–358. Springer, Cham, August 2022.

[BCC+23] Dung Bui, Haotian Chu, Geoffroy Couteau, Xiao Wang, Chenkai Weng, Kang Yang, and Yu Yu. An efficient ZK compiler from SIMD circuits to general circuits. Cryptology ePrint Archive, Report 2023/1610, 2023.

[BCCD23] Maxime Bombar, Geoffroy Couteau, Alain Couvreur, and Clément Ducros. Correlated pseudorandomness from the hardness of quasi-abelian decoding. In *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 567–601. Springer, Cham, August 2023.

[BCCT11] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. Cryptology ePrint Archive, Report 2011/443, 2011.

[BCG+19a] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In *ACM CCS 2019*, pages 291–308. ACM Press, November 2019.

[BCG+19b] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Cham, August 2019.

[BCG+20a] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Correlated pseudorandom functions from variable-density LPN. In *61st FOCS*, pages 1069–1080. IEEE Computer Society Press, November 2020.

[BCG+20b] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-LPN. In *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 387–416. Springer, Cham, August 2020.

[BCG+22] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. Correlated pseudorandomness from expand-accumulate codes. In *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 603–633. Springer, Cham, August 2022.

[BCG+23] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. Oblivious transfer with constant computational overhead. In *EUROCRYPT 2023, Part I*, volume 14004 of *LNCS*, pages 271–302. Springer, Cham, April 2023.

[BCGI18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In *ACM CCS 2018*, pages 896–912. ACM Press, October 2018.

[BCM+24] Dung Bui, Geoffroy Couteau, Pierre Meyer, Alain Passelègue, and Mahshid Riahinia. Fast public-key silent OT and more from constrained Naor-Reingold. In *EUROCRYPT 2024, Part VI*, volume 14656 of *LNCS*, pages 88–118. Springer, Cham, May 2024.

[BCM+25] Lennart Braun, Geoffroy Couteau, Kelsey Melissaris, Mahshid Riahinia, and Elahe Sadeghi. Fast pseudorandom correlation functions from sparse LPN. In *ASIACRYPT 2025, Part VII*, volume 16251 of *LNCS*, pages 439–473. Springer, Singapore, December 2025.

[BDSS25] Elette Boyle, Lalita Devadas, and Sacha Servan-Schreiber. Non-interactive distributed point functions. In *PKC 2025, Part I*, volume 15674 of *LNCS*, pages 3–35. Springer, Cham, May 2025.

[BGH13] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in LWE-based homomorphic encryption. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 1–13. Springer, Berlin, Heidelberg, February / March 2013.

[BGH+23] Gabrielle Beck, Aarushi Goel, Aditya Hegde, Abhishek Jain, Zhengzhong Jin, and Gabriel Kaptchuk. Scalable multiparty garbling. In *ACM CCS 2023*, pages 2158–2172. ACM Press, November 2023.

[BGH+25] Elette Boyle, Niv Gilboa, Matan Hamilis, Yuval Ishai, and Yaxin Tu. Improved constructions for distributed multi-point functions. In *2025 IEEE Symposium on Security and Privacy*, pages 2414–2432. IEEE Computer Society Press, May 2025.

[BGIK22] Elette Boyle, Niv Gilboa, Yuval Ishai, and Victor I. Kolobov. Programmable distributed point functions. In *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 121–151. Springer, Cham, August 2022.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-SNARK parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

[BL25] Katharina Boudgoust and Oleksandra Lapiha. Leftover hash lemma(s) over cyclotomic rings. Cryptology ePrint Archive, Report 2025/1080, 2025.

[BLMR13] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 410–428. Springer, Berlin, Heidelberg, August 2013.

[BLS19] Jonathan Bootle, Vadim Lyubashevsky, and Gregor Seiler. Algebraic techniques for short(er) exact lattice-based zero-knowledge proofs. In *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 176–202. Springer, 2019.

[BMRS21] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 92–122, Virtual Event, August 2021. Springer, Cham.

[CD08] Ran Canetti and Ronny Ramzi Dakdouk. Extractable perfectly one-way functions. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 449–460. Springer, Berlin, Heidelberg, July 2008.

[CDH+25] Geoffroy Couteau, Lalita Devadas, Aditya Hegde, Abhishek Jain, and Sacha Servan-Schreiber. Multi-key homomorphic secret sharing. In *EUROCRYPT 2025, Part V*, volume 15605 of *LNCS*, pages 3–33. Springer, Cham, May 2025.

[CHHK25] Geoffroy Couteau, Carmit Hazay, Aditya Hegde, and Naman Kumar. Breaking the $1/\lambda$-rate barrier for arithmetic garbling. In *EUROCRYPT 2025, Part VI*, volume 15606 of *LNCS*, pages 182–213. Springer, Cham, May 2025.

[CRR21] Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In *CRYPTO 2021, Part III*, volume 12827 of *LNCS*, pages 502–534, Virtual Event, August 2021. Springer, Cham.

[DILO22] Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Improving line-point zero knowledge: Two multiplications for the price of one. In *ACM CCS 2022*, pages 829–841. ACM Press, November 2022.

[DIO21a] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. In *ITC 2021*, volume 199 of *LIPIcs*, pages 5:1–5:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[DIO21b] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. In Stefano Tessaro, editor, *ITC 2021*, volume 199 of *LIPIcs*, pages 5:1–5:24. Schloss Dagstuhl, July 2021.

[DKL+23a] Nico Döttling, Dimitris Kolonelos, Russell W. F. Lai, Chuanwei Lin, Giulio Malavolta, and Ahmadreza Rahimi. Efficient laconic cryptography from learning with errors. In *EUROCRYPT 2023, Part III*, volume 14006 of *LNCS*, pages 417–446. Springer, Cham, April 2023.

[DKL+23b] Nico Döttling, Dimitris Kolonelos, Russell W. F. Lai, Chuanwei Lin, Giulio Malavolta, and Ahmadreza Rahimi. Efficient laconic cryptography from learning with errors. Cryptology ePrint Archive, Report 2023/404, 2023.

[DLL25] Marian Dietz, Hanjun Li, and Huijia Lin. TinyLabels: How to compress garbled circuit input labels, efficiently. In *EUROCRYPT 2025, Part VI*, volume 15606 of *LNCS*, pages 245–274. Springer, Cham, May 2025.

[DNY+26] Sankha Das, Lucien K. L. Ng, Yibin Yang, Vladimir Kolesnikov, and Teodora Baluta. Azkaban: Zero-knowledge abstract analysis for neural networks. In *ACM CCS 2026*. ACM, 2026.

[DSH21] Anaëlle Le Dévéhat, Hiroki Shizuya, and Shingo Hasegawa. On the higher-bit version of approximate inhomogeneous short integer solution problem. In Mauro Conti, Marc Stevens, and Stephan Krenn, editors, *CANS 21*, volume 13099 of *LNCS*, pages 253–272. Springer, Cham, December 2021.

[DSH22] Anaëlle Le Dévéhat, Hiroki Shizuya, and Shingo Hasegawa. On the higher bit version of approximate inhomogeneous short integer solution problem. Cryptology ePrint Archive, Report 2022/1144, 2022.

[EMP21] EMP-toolkit. emp-zk: Public implementation of QuickSilver, 2021. Source revision a4e8489, VOLE-backend communication audit.

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Cham, August 2018.

[GHJY26] Vipul Goyal, David Heath, Abhishek Jain, and Yibin Yang. Suffix-invariant programmable PRFs and applications to stacked garbling. In *CRYPTO 2026*, volume 16807 of *LNCS*, pages 625–657, Cham, Switzerland, 2026. Springer.

[GKM+18] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-SNARKs. In *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 698–728. Springer, Cham, August 2018.

[GLH+26] Xiaojie Guo, Hanlin Liu, Zhicong Huang, Hongrui Cui, Wenhao Zhang, Cheng Hong, Xiao Wang, Kang Yang, and Yu Yu. Dory: Streaming PCG with small memory. In *2026 IEEE Symposium on Security and Privacy*, pages 1902–1920. IEEE Computer Society Press, May 2026.

[GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled RAM from one-way functions. In *47th ACM STOC*, pages 449–458. ACM Press, June 2015.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[Gol01] Oded Goldreich. *The Foundations of Cryptography - Volume 1: Basic Techniques*. Cambridge University Press, 2001.

[GOS18] Sanjam Garg, Rafail Ostrovsky, and Akshayaram Srinivasan. Adaptive garbled RAM from laconic oblivious transfer. In *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 515–544. Springer, Cham, August 2018.

[GS18] Sanjam Garg and Akshayaram Srinivasan. Adaptively secure garbling with near optimal online complexity. In *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 535–565. Springer, Cham, April / May 2018.

[GTRS25] Tianyao Gu, Afonso Tinoco, Sri Harish G. Rajan, and Elaine Shi. PicoGRAM: Practical garbled RAM from decisional Diffie-Hellman. In *CRYPTO 2025, Part VIII*, volume 16007 of *LNCS*, pages 247–281. Springer, Cham, August 2025.

[HCL+24] Meng Hao, Hanxiao Chen, Hongwei Li, Chenkai Weng, Yuan Zhang, Haomiao Yang, and Tianwei Zhang. Scalable zero-knowledge proofs for non-linear functions in machine learning. In *USENIX Security 2024*. USENIX Association, August 2024.

[Hea24] David Heath. Efficient arithmetic in garbled circuits. In *EUROCRYPT 2024, Part V*, volume 14655 of *LNCS*, pages 3–31. Springer, Cham, May 2024.

[HHK+22] Abida Haque, David Heath, Vladimir Kolesnikov, Steve Lu, Rafail Ostrovsky, and Akash Shah. Garbled circuits with sublinear evaluator. In *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 37–64. Springer, Cham, May / June 2022.

[HHK+24] Carmit Hazay, David Heath, Vladimir Kolesnikov, Muthuramakrishnan Venkitasubramaniam, and Yibin Yang. LogRobin++: Optimizing proofs of disjunctive statements in VOLE-based ZK. In *ASIACRYPT 2024, Part V*, volume 15488 of *LNCS*, pages 367–401. Springer, Singapore, December 2024.

[HK20] David Heath and Vladimir Kolesnikov. Stacked garbling - garbled circuit proportional to longest execution path. In *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 763–792. Springer, Cham, August 2020.

[HK21a] David Heath and Vladimir Kolesnikov. LogStack: Stacked garbling with $O(b \log b)$ computation. In *EUROCRYPT 2021, Part III*, volume 12698 of *LNCS*, pages 3–32. Springer, Cham, October 2021.

[HK21b] David Heath and Vladimir Kolesnikov. One hot garbling. In *ACM CCS 2021*, pages 574–593. ACM Press, November 2021.

[HKN24] David Heath, Vladimir Kolesnikov, and Lucien K. L. Ng. Garbled circuit lookup tables with logarithmic number of ciphertexts. In *EUROCRYPT 2024, Part V*, volume 14655 of *LNCS*, pages 185–215. Springer, Cham, May 2024.

[HKN+25] David Heath, Vladimir Kolesnikov, Varun Narayanan, Rafail Ostrovsky, and Akash Shah. Multiparty garbling from OT with linear scaling and RAM support. In

*CRYPTO 2025, Part IV*, volume 16003 of *LNCS*, pages 556–588. Springer, Cham, August 2025.

[HKO22] David Heath, Vladimir Kolesnikov, and Rafail Ostrovsky. EpiGRAM: Practical garbled RAM. In *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 3–33. Springer, Cham, May / June 2022.

[HKP21] David Heath, Vladimir Kolesnikov, and Stanislav Peceny. Garbling, stacked and staggered - faster k-out-of-n garbled function evaluation. In *ASIACRYPT 2021, Part II*, volume 13091 of *LNCS*, pages 245–274. Springer, Cham, December 2021.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *ACM CCS 2013*, pages 955–966. ACM Press, November 2013.

[JKY26] Sunghyeon Jo, Vladimir Kolesnikov, and Yibin Yang. ZEE200: Zero knowledge for everything and everyone @ 200 KHz. Cryptology ePrint Archive, Paper 2026/828, 2026.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *24th ACM STOC*, pages 723–732. ACM Press, May 1992.

[KNMC26] Andes Y. L. Kei, Lucien K. L. Ng, Jack P. K. Ma, and Sherman S. M. Chow. Sort, sweep, mirror: Batch private interval lookup with logarithmic cost. In *2026 IEEE Symposium on Security and Privacy*, pages 1937–1950. IEEE Computer Society Press, May 2026.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, Berlin, Heidelberg, August 2015.

[KPR+25] Vladimir Kolesnikov, Stanislav Peceny, Rahul Rachuri, Srinivasan Raghuraman, Peter Rindal, and Harshal Shah. Block-accumulate codes: Accelerated linear codes for PCGs and ZK. Cryptology ePrint Archive, Report 2025/1828, 2025.

[KPRR25] Vladimir Kolesnikov, Stanislav Peceny, Srinivasan Raghuraman, and Peter Rindal. Stationary syndrome decoding for improved PCGs. In *CRYPTO 2025, Part I*, volume 16000 of *LNCS*, pages 284–317. Springer, Cham, August 2025.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Berlin, Heidelberg, July 2008.

[LHLA25a] Mengling Liu, Yang Heng, Xingye Lu, and Man Ho Au. Jesseq: Efficient zero-knowledge proofs for circuits over any field. In *2025 IEEE Symposium on Security and Privacy*, pages 3402–3420. IEEE, 2025.

[LHLA25b] Mengling Liu, Yang Heng, Xingye Lu, and Man Ho Au. JesseQ: Public implementation, 2025. Source revision 7a26597, JQv2 VOLE backend and polynomial helper.

[LLLH26] Can Liu, Lenny Liu, Ning Luo, and David Heath. Improved garbled RAM via garbled merge. In *ACM CCS 2026*. ACM, 2026.

[LLX+26] Zhe Li, Hongqing Liu, Chaoping Xing, Yizhou Yao, and Chen Yuan. Faster pseudorandom correlation generators via Walsh-Hadamard transform. In *CRYPTO 2026, Part VIII*, volume 16807 of *LNCS*, pages 430–461. Springer, 2026.

[LLZ25] Wei-Kai Lin, Zhenghao Lu, and Hong-Sheng Zhou. Efficient garbled pseudorandom functions and lookup tables from minimal assumption. In *TCC 2025, Part I*, volume 16268 of *LNCS*, pages 64–96. Springer, Cham, December 2025.

[LMSP21] Ryan Lehmkuhl, Pratyush Mishra, Akshayaram Srinivasan, and Raluca Ada Popa. Muse: Secure inference resilient to malicious clients. In *USENIX Security 2021*, pages 2201–2218. USENIX Association, August 2021.

[LNP22] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plançon. Lattice-based zero-knowledge proofs and applications: Shorter, simpler, and more general. In *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 71–101. Springer, 2022.

[LP11] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In Aggelos Kiayias, editor, *CT-RSA 2011*, volume 6558 of *LNCS*, pages 319–339. Springer, Berlin, Heidelberg, February 2011.

[LWQ+24] Tao Lu, Haoyu Wang, Wenjie Qu, Zonghui Wang, Jinye He, Tianyang Tao, Wenzhi Chen, and Jiaheng Zhang. An efficient and extensible zero-knowledge proof framework for neural networks. Cryptology ePrint Archive, Report 2024/703, 2024.

[LXY+25] Zhe Li, Chaoping Xing, Yizhou Yao, Chen Yuan, and Mengmeng Zhou. Succinct line-point zero-knowledge arguments from homomorphic secret sharing. Cryptology ePrint Archive, Report 2025/1866, 2025.

[LXYY25] Zhe Li, Chaoping Xing, Yizhou Yao, and Chen Yuan. Efficient pseudorandom correlation generators for any finite field. In *EUROCRYPT 2025, Part V*, volume 15605 of *LNCS*, pages 145–175. Springer, Cham, May 2025.

[Mic23] Microsoft SEAL. Microsoft SEAL (release 4.1). `https://github.com/Microsoft/SEAL`, January 2023. Microsoft Research, Redmond, WA.

[MORS25] Pierre Meyer, Claudio Orlandi, Lawrence Roy, and Peter Scholl. Silent circuit relinearisation: Sublinear-size (boolean and arithmetic) garbled circuits from DCR. In *CRYPTO 2025, Part IV*, volume 16003 of *LNCS*, pages 426–458. Springer, Cham, August 2025.

[NC21] Lucien K. L. Ng and Sherman S. M. Chow. GForce: GPU-friendly oblivious and rapid neural network inference. In *USENIX Security 2021*, pages 2147–2164. USENIX Association, August 2021.

[NC23] Lucien K. L. Ng and Sherman S. M. Chow. SoK: Cryptographic neural-network computation. In *2023 IEEE Symposium on Security and Privacy*, pages 497–514. IEEE Computer Society Press, May 2023.

[NCW+21] Lucien K. L. Ng, Sherman S. M. Chow, Anna P. Y. Woo, Donald P. H. Wong, and Yongjun Zhao. Goten: GPU-outsourcing trusted execution of neural network training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 14876–14883, 2021.

[NK25] Lucien K. L. Ng and Vladimir Kolesnikov. Toss: Garbled PIR from table-only stacking. In *ACM CCS 2025*, pages 4634–4648. ACM Press, October 2025.

[NMM+25] Lucien K. L. Ng, Pedro Moreno-Sanchez, Mohsen Minaei, Panagiotis Chatzigiannis, Adithya Bhat, and Duc V. Le. Lite-PoT: Practical powers-of-tau setup ceremony. In *ACM CCS 2025*, pages 2114–2128. ACM Press, October 2025.

[OSY21] Claudio Orlandi, Peter Scholl, and Sophia Yakoubov. The rise of paillier: Homomorphic secret sharing and public-key silent OT. In *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 678–708. Springer, Cham, October 2021.

[Roy22] Lawrence Roy. SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 657–687. Springer, Cham, August 2022.

[RR21] Mike Rosulek and Lawrence Roy. Three halves make a whole? Beating the half-gates lower bound for garbled circuits. In *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 94–124, Virtual Event, August 2021. Springer, Cham.

[RRT23] Srinivasan Raghuraman, Peter Rindal, and Titouan Tanguy. Expand-convolute codes for pseudorandom correlation generators from LPN. In *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 602–632. Springer, Cham, August 2023.

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 704–737. Springer, Cham, August 2020.

[SGRR19] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-OLE: Improved constructions and implementation. In *ACM CCS 2019*, pages 1055–1072. ACM Press, November 2019.

[SLY+25] Yunqing Sun, Hanlin Liu, Kang Yang, Yu Yu, Xiao Wang, and Chenkai Weng. Committed vector oblivious linear evaluation and its applications. In *ACM CCS 2025*, pages 3635–3648. ACM Press, October 2025.

[SV14] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *DCC*, 71(1):57–81, 2014.

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy*, pages 1074–1091. IEEE Computer Society Press, May 2021.

[WYX+21] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In *USENIX Security 2021*, pages 501–518. USENIX Association, August 2021.

[WYY+22] Chenkai Weng, Kang Yang, Zhaomin Yang, Xiang Xie, and Xiao Wang. AntMan: Interactive zero-knowledge proofs with sublinear communication. In *ACM CCS 2022*, pages 2901–2914. ACM Press, November 2022.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[YH24] Yibin Yang and David Heath. Two shuffles make a RAM: Improved constant overhead zero knowledge RAM. In *USENIX Security 2024*, pages 1435–1452. USENIX Association, August 2024.

[YHH+23] Yibin Yang, David Heath, Carmit Hazay, Vladimir Kolesnikov, and Muthuramakrishnan Venkitasubramaniam. Batchman and robin: Batched and non-batched branching for interactive ZK. In *ACM CCS 2023*, pages 1452–1466. ACM Press, November 2023.

[YHH+24] Yibin Yang, David Heath, Carmit Hazay, Vladimir Kolesnikov, and Muthuramakrishnan Venkitasubramaniam. Tight ZK CPU: Batched ZK branching with cost proportional to evaluated instruction. In *ACM CCS 2024*, pages 3095–3109. ACM Press, October 2024.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In *ACM CCS 2021*, pages 2986–3001. ACM Press, November 2021.

[YWL+20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In *ACM CCS 2020*, pages 1607–1626. ACM Press, November 2020.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Berlin, Heidelberg, April 2015.

[ZXZS19] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. Cryptology ePrint Archive, Report 2019/1482, 2019.

# Appendices

# A Digest Properties

## A.1 Roadmap

This section argues for the digest-side properties used later by both security arguments. Recall that our protocol uses the digest in two places. First, the receiver reveals one randomized public root digest. Second, the translate/expand algebra uses the lifted opening input below that digest. Accordingly, the security proof needs three properties from the digest layer. The first is a *root hiding* statement showing that the randomized root node can be revealed without leaking the receiver's input. The second is a *binding* statement showing that bounded openings are computationally unique. The third is a *smudging* statement showing that large openings are statistically useless in the $\mathsf{ct}_{\mathsf{lac}}$-row evaluations.

These properties are used in different places. The semi-honest proof cites the randomized-root hiding lemma from this section, and then carries out a TinyLabel-style recursive simulation. Then, for malicious security, we establish an extraction argument for the receiver's input based on the bounded-opening uniqueness theorems and the smudging lemma.

## A.2 Hiding of the randomized root digest

The lower-level realization in Section 6.4 introduces a special randomized node at the root of the digest tree:

$$\mathsf{RandNodeDigest}^{\tau,m}_{B'}(u,\zeta) := B_1 \cdot \mathbf{g}^{-1}(u) + B^* \cdot \zeta,$$

where $B_1 \leftarrow \mathcal{R}_q^\tau$, $B^* \leftarrow \mathcal{R}_q^m$, and $B' = (B_1 \quad B^*)$. The construction section uses this definition algebraically to wire the randomized root into the top $\mathsf{Shrink}/\mathsf{Expand}$ step. Here we record the digest-security fact: for a slightly wider randomizer dimension $m$, this published root is statistically close to uniform.

The intuition is the Ajtai-commitment viewpoint also used in laconic encryption [DKL+23b]: the digest is an affine map with a message part and a randomizer part. Here the message part is $B_1 \cdot \mathbf{g}^{-1}(u)$, while $B^* \cdot \zeta$ supplies the entropy that statistically hides it. The only extra point is that this affine map lives over the ring $\mathcal{R}_q$, so we instantiate the ring leftover hash lemma of Boudgoust and Lapiha [BL25].

We isolate the external statement because it is linear: the theorem below bounds $B^*\zeta$, and the lemma transfers that bound through the fixed affine shift $B_1 \cdot \mathbf{g}^{-1}(u)$.

**Lemma 7** (Parameter choice for $\mathsf{RandNodeDigest}_{\tau,m}$)**.** *For $\zeta \in \mathcal{R}_q^m$ sampled coefficientwise uniformly from a centered interval of size $G = 2\eta + 1 \geq g$, if*

$$m = \tau + \Theta\left(\frac{\sigma + \log n}{\log g}\right),$$

*then for every fixed $u \in \mathcal{R}_q$,*

$$\mathsf{SD}\big((B^*, \mathsf{RandNodeDigest}^{\tau,m}_{B'}(u,\zeta)), (B^*, U(\mathcal{R}_q))\big) \leq 2^{-\sigma}.$$

**Theorem 4** (Ring-LHL randomizer bound)**.** *Let $\zeta \in \mathcal{R}_q^m$ be sampled coefficientwise uniformly from a centered interval of size $G = 2\eta + 1 \geq g$, and let $A \leftarrow \mathcal{R}_q^m$. For $u \leftarrow \mathcal{R}_q$,*

$$\mathsf{SD}\big((A\zeta, A), (u, A)\big) \leq \frac{1}{2}\sqrt{\left(\frac{q}{g^m} + 1\right)^n - 1}.$$

*Consequently, if* $m = \tau + s$ *and*

$$s \geq \frac{2\sigma + \log_2 n + O(1)}{\log_2 g},$$

*then*

$$\mathsf{SD}\big((A\zeta, A), (u, A)\big) \leq 2^{-\sigma}.$$

Here $\mathsf{SD}(x, y)$ denotes the statistical distance between two distributions $x$ and $y$.

*Proof.* We use Boudgoust and Lapiha's bounded-uniform corollary for one hash output [BL25, Corollary 4]; it is the needed specialization of their Theorem 4, with Lemma 28 giving the collision bound and Lemma 4 the $\mathcal{R}_q$-ideal norm count. Since $G = 2\eta + 1 \geq g$, it gives

$$\mathsf{SD}\big((A\zeta, A), (u, A)\big) \leq \frac{1}{2}\sqrt{\left(\frac{q}{G^m} + 1\right)^n - 1} \leq \frac{1}{2}\sqrt{\left(\frac{q}{g^m} + 1\right)^n - 1}.$$

Let $m = \tau + s$ and $\tau = \lceil \log_g q \rceil$, so $q/g^m \leq g^{-s}$. For $g^{-s} \leq 1/n$, the elementary inequality $(1 + t)^n \leq 1 + 2nt$ implies

$$\mathsf{SD}\big((A\zeta, A), (u, A)\big) \leq \sqrt{n} \cdot g^{-s/2}.$$

The stated lower bound on $s$ makes this at most $2^{-\sigma}$. □

*Proof of Lemma 7.* Fix $u \in \mathcal{R}_q$ and set $c := B_1 \cdot \mathbf{g}^{-1}(u)$. Then

$$\mathsf{RandNodeDigest}_{B'}^{\tau,m}(u, \zeta) = c + B^* \cdot \zeta.$$

For each fixed $B^*$, translation by $c$ preserves uniformity and statistical distance. Thus, for $v \leftarrow \mathcal{R}_q$,

$$\mathsf{SD}\big((B^*, \mathsf{RandNodeDigest}_{B'}^{\tau,m}(u, \zeta)), (B^*, v)\big) = \mathsf{SD}\big((B^*, B^*\zeta), (B^*, v)\big).$$

This is the distribution in Theorem 4 with $A = B^*$. The lemma's choice of $m$ supplies the required slack $s = m - \tau$, giving the claimed $2^{-\sigma}$ bound. □

**Concrete instantiation.** Under the parameter choices of Section 12, we have $n = 8192 = 2^{13}$, $\log_2 q = 220$, $g = 2^{110}$, and $\sigma = 40$. Therefore

$$\tau = \lceil \log_g q \rceil = \lceil 220/110 \rceil = 2.$$

Taking $s = 1$ gives $m = \tau + 1 = 3$, and the bound from Theorem 4 becomes

$$\mathsf{SD}\big((A\zeta, A), (u, A)\big) \leq \sqrt{n} \cdot g^{-1/2} = 2^{13/2} \cdot 2^{-55} = 2^{-48.5} < 2^{-40}.$$

So for our implementation parameters, three randomizer ring elements already suffice.

## A.3 Digest Openings and Norm Bounds

We begin by recalling the definition of digests in Section 5.3. Using the node-key notation of Section 5.2, the digest of one ordered child pair is

$$\mathsf{NodeDigest}_B(u_L, u_R) := B\begin{pmatrix}\mathbf{g}^{-1}(u_L) \\ \mathbf{g}^{-1}(u_R)\end{pmatrix}.$$

For the translate/expand algebra, we use the signed lifted input

$$w(u_L, u_R) := \begin{pmatrix} -\mathbf{g}^{-1}(u_L) \\ -\mathbf{g}^{-1}(u_R) \end{pmatrix} \in \mathcal{R}_q^{2\tau}.$$

Then

$$d = -B\, w(u_L, u_R), \qquad u_L = -(\mathbf{g}^\top \ 0) w(u_L, u_R), \qquad u_R = -(0 \ \mathbf{g}^\top) w(u_L, u_R).$$

This is the input later paired with the selected $\mathsf{ct}_{\mathsf{lac}}$ row.

All norms in this section are coefficient norms; $\|\cdot\|_2$ denotes the coefficient Euclidean norm. We define the smudging threshold

$$\beta := \frac{q}{s},$$

where $s$ is the Gaussian parameter of the row-noise terms in the $\mathsf{ct}_{\mathsf{lac}}$ rows.

**Definition 9** (Bounded node opening)**.** *For a digest $d \in \mathcal{R}_q$, an ordered child pair $(u_L, u_R)$ is a* bounded opening *of $d$ if*

$$d = \mathsf{NodeDigest}_B(u_L, u_R) \qquad \textit{and} \qquad \|w(u_L, u_R)\|_2 \le \beta.$$

## A.4 Binding of bounded openings

In this subsection, we only consider openings bounded by $\beta$. We show that a PPT adversary cannot produce two different short inputs for one node digest unless it solves SIS. The next subsection explains why inputs outside this regime are harmless for our construction.

**Theorem 5** (Bounded binding from SIS)**.** *Assume $\mathsf{SIS}_{q,2\tau,2\beta}$ is hard for uniformly random $B \in \mathcal{R}_q^{1\times 2\tau}$. For every PPT adversary $\mathcal{A}$,*

$$\begin{aligned} \Pr_{B,\mathcal{A}} \big[ & (d, u_L, u_R, u'_L, u'_R) \leftarrow \mathcal{A}(B), \\ & (u_L, u_R) \neq (u'_L, u'_R), \\ & d = \mathsf{NodeDigest}_B(u_L, u_R) = \mathsf{NodeDigest}_B(u'_L, u'_R), \\ & \|w(u_L, u_R)\|_2 \le \beta \ \wedge \ \|w(u'_L, u'_R)\|_2 \le \beta \big] \le \mathsf{negl}(\lambda), \end{aligned}$$

*where the probability is over $B \leftarrow U(\mathcal{R}_q^{1\times 2\tau})$ and the randomness of $\mathcal{A}$.*

*Proof.* Suppose the event occurs. Set

$$z := w(u_L, u_R) - w(u'_L, u'_R).$$

The two openings are distinct, so the recovery equations above give $z \neq 0$. Since both open the same digest,

$$Bz = Bw(u_L, u_R) - Bw(u'_L, u'_R) = -d + d = 0.$$

The triangle inequality gives

$$\|z\|_2 \le \|w(u_L, u_R)\|_2 + \|w(u'_L, u'_R)\|_2 \le 2\beta.$$

Thus a successful adversary yields a nonzero SIS solution for $B$ at radius $2\beta$, contradicting the assumption. □

We also need the same statement for the full digest tree.

**Definition 10** (Bounded opening tree)**.** *Fix a depth $\ell \geq 0$. A labeled full binary tree*

$$\mathcal{T} = (d_{\mathsf{pre}})_{\mathsf{pre}\in\{0,1\}^{\leq \ell}}$$

*is a* bounded opening tree *for the root digest $d_\epsilon \in \mathcal{R}_q$ if for every internal node $\mathsf{pre} \in \{0,1\}^{<\ell}$, the ordered child pair*

$$(d_{\mathsf{pre}\|0}, d_{\mathsf{pre}\|1})$$

*is a bounded opening of $d_{\mathsf{pre}}$ in the sense of Definition 9. Two bounded opening trees are distinct if they differ at some node label.*

**Theorem 6** (Bounded tree binding)**.** *Assume $\mathsf{SIS}_{q,2\tau,2\beta}$ is hard for uniformly random $B \in \mathcal{R}_q^{1\times 2\tau}$. For every $\ell \geq 0$, no PPT adversary outputs $d \in \mathcal{R}_q$ and two distinct depth-$\ell$ bounded opening trees of $d$, except with negligible probability.*

*Proof.* The case $\ell = 0$ is immediate. For $\ell \geq 1$, let

$$\mathcal{T} = (d_{\mathsf{pre}})_{\mathsf{pre}\in\{0,1\}^{\leq \ell}} \qquad \text{and} \qquad \mathcal{T}' = (d'_{\mathsf{pre}})_{\mathsf{pre}\in\{0,1\}^{\leq \ell}}$$

be distinct bounded opening trees with $d_\epsilon = d'_\epsilon = d$. Choose a minimum-depth internal node $\mathsf{pre}$ whose children differ:

$$(d_{\mathsf{pre}\|0}, d_{\mathsf{pre}\|1}) \neq (d'_{\mathsf{pre}\|0}, d'_{\mathsf{pre}\|1}).$$

Minimality gives $d_{\mathsf{pre}} = d'_{\mathsf{pre}}$. The two child pairs are therefore distinct bounded openings of the same digest, contradicting Theorem 5. □

### A.5 Smudging of large openings

Recall the $\mathsf{LacE}$ ciphertext notation from Section 6. Write $\mathsf{ct}_{\mathsf{lac}} = (\mathbf{C}_1, \ldots, \mathbf{C}_\ell)$. Fix $i \in [\ell]$ and $\mathsf{ind} \in \{0,1\}^\ell$. For the selected row, write

$$C := \mathbf{C}_i[\mathsf{ind}] = C^\circ + \mathbf{E}_i[\mathsf{ind}],$$

where

$$C^\circ := \mathbf{r}^{(i)}[\mathsf{ind}] \cdot B + \mathbf{r}^{(i+1)}[\mathsf{ind}] \cdot (\overline{\mathsf{ind}}_i \cdot \mathbf{g}^\top \ \mathsf{ind}_i \cdot \mathbf{g}^\top).$$

Here $\mathsf{ind}_i$ is the $i$-th bit of $\mathsf{ind}$, $\overline{\mathsf{ind}}_i = 1 - \mathsf{ind}_i$, and the entries of $\mathbf{E}_i[\mathsf{ind}]$ are sampled independently from a discrete Gaussian of parameter $s$. For a fixed opening input $w$, $C^\circ w$ is the meaningful message part and $\mathbf{E}_i[\mathsf{ind}]w$ is the smudge noise.

The threshold $\beta = q/s$ is the point at which the HE-standard dual estimate gives $\exp(-\pi)$-scale distinguishing advantage for one row evaluation. This is the target advantage used in the lattice-cost accounting of [ACC+19, Section 2.1.2.c]; the repetitions needed for the fixed success probability are already included in those cost estimates.

**Theorem 7** (Dual smudging estimate)**.** *([LP11, Section 4],[ACC+19, Section 2.1.2.c]) Let $\mathbf{e} \in \mathcal{R}_q^D$ be sampled coefficient-wise from the Gaussian error distribution with parameter $s$, and let $\mathbf{v} \in \mathcal{R}_q^D$ be fixed. Set $\alpha = s/q$. The distinguishing advantage between $\mathbf{e}^T \cdot \mathbf{v}$ and $U(\mathcal{R}_q)$ is at most*

$$\exp\Big(-\pi(\|\mathbf{v}\|_2 \alpha)^2\Big).$$

*In particular, if $\|\mathbf{v}\|_2 \geq q/s$, then this advantage is at most $\exp(-\pi)$.*

**Lemma 8** (Smudging lemma)**.** *Fix $i \in [\ell]$, $\mathsf{ind} \in \{0,1\}^{\ell}$, and*

$$C := \mathbf{C}_i[\mathsf{ind}] = C^{\circ} + \mathbf{E}_i[\mathsf{ind}].$$

*For every fixed $w \in \mathcal{R}_q^{2\tau}$, the distinguishing advantage between $Cw$ and $U(\mathcal{R}_q)$ is at most*

$$\exp\Big(-\pi(\|w\|_2 s/q)^2\Big).$$

*If $\|w\|_2 \geq \beta$, this advantage is at most* $\exp(-\pi)$.

*Proof.* Condition on $C^{\circ}$. The noisy part $\mathbf{E}_i[\mathsf{ind}]w$ has the form $\mathbf{e}^T \cdot \mathbf{v}$ with $\|\mathbf{v}\|_2 = \|w\|_2$. Applying Theorem 7 gives the stated bound, and adding the fixed term $C^{\circ}w$ preserves statistical distance from uniform. □

**Corollary 1** (Digest-usefulness dichotomy)**.** *Assume the SIS premise of Theorem 5. Fix a digest $d$ with one bounded opening. Except with negligible probability, no PPT adversary can output a distinct bounded opening of $d$. For every opening input $w \in \mathcal{R}_q^{2\tau}$ with $\|w\|_2 \geq \beta$, and every selected row $C = C^{\circ} + \mathbf{E}_i[\mathsf{ind}]$, the distinguishing advantage between $Cw$ and $U(\mathcal{R}_q)$ is at most* $\exp(-\pi)$.

# B Formalization and Semi-honest Security

In this section, we prove semi-honest security for the CI-VOLE construction built from the $\mathbb{Z}_p$-level wrapper $\mathsf{LogVOLE}_{\mathbb{Z}}$ and the ring-level core protocol $\mathsf{LogVOLE}_{\mathcal{R}}$. The wrapper $\mathsf{LogVOLE}_{\mathbb{Z}}$ from Figure 4 reduces CI-VOLE over $\mathbb{Z}_p$ to CI-VOLE over $\mathcal{R}_q$, and the core ring protocol $\mathsf{LogVOLE}_{\mathcal{R}}$ is described in Figures 10 and 11. Our focus here is the core semi-honest argument for the construction from Sections 4 and 6, excluding the malicious-security analysis in Section 10 and the optimizations in Section 7.

## B.1 Correctness

**Interface to the generalized functionality.** $\mathsf{LogVOLE}_{\mathbb{Z}}$ and $\mathsf{LogVOLE}_{\mathcal{R}}$ do not expose separate public methods named set-delta, setx-ni, release-ni, setk-int, setx-int or release-int. Instead, the functionality interface from Figure 7 is realized through the two protocol entry points of $\mathsf{LogVOLE}_{\mathbb{Z}}$. The call $\mathsf{LogVOLE}_{\mathbb{Z}}.\mathsf{Offline}(\mathbb{S} : \Delta, \mathbb{B} : w)$ realizes $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}.\mathsf{set\text{-}delta}(\mathbb{S} : \Delta, w)$. The online call $\mathsf{LogVOLE}_{\mathbb{Z}}.\mathsf{Online}(\mathbb{B} : \mathsf{id} = (\mathsf{sid}, \mathsf{mode}))$ is then interpreted according to the mode tag. In the interactive mode, one online invocation on $(\mathsf{sid}, \mathbf{x})$ follows the timing of setk-int, setx-int, and release-int: the sender-side share is fixed before the receiver output is released, and the final output pair again satisfies the same VOLE relation. In the non-interactive mode, one online invocation on receiver input $(\mathsf{sid}, \mathbf{x})$ computes the pair $(\mathbf{k}, \mathbf{m})$ corresponding to the functionality entries created by setx-ni and later released by release-ni; the protocol returns both shares locally, even though the functionality conceptually delays the sender's output until release.

**Theorem 8** (Correctness of the core CI-VOLE protocol)**.** *Let $\Pi$ be the CI-VOLE protocol obtained by combining the wrapper $\mathsf{LogVOLE}_{\mathbb{Z}}$ from Figure 4 with the core ring protocol $\mathsf{LogVOLE}_{\mathcal{R}}$ from Figures 10 and 11, excluding the optimization from Section 7. Then $\Pi$ realizes $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ in both the interactive and non-interactive settings of Figure 7. Concretely, for $w \leq 2^{\lambda}$, every session identifier* sid*, every mode* $\mathsf{mode} \in \{\mathsf{interactive}, \mathsf{non\text{-}interactive}\}$*, and every $\Delta \in \mathbb{Z}_p$, $\mathbf{x} \in \mathbb{Z}_p^w$,*

$$\Pr[(\mathbf{k}, \mathbf{m}) \leftarrow \Pi(\mathbb{S} : \Delta, \mathbb{R} : (\mathsf{sid}, \mathbf{x}), \mathbb{B} : \mathsf{mode}) : \mathbf{m} = \mathbf{x} \cdot \Delta + \mathbf{k}] = 1.$$

*Proof sketch.* The detailed proof first treats the interactive mode. It starts from the actual input $\mathbf{x}$, writes the encoded ring input as $\mathsf{Encode}(\mathbf{x})$, and then describes the recursive input vector processed at each level. The base case is the root level, where $\mathsf{DerandRoot}$ converts the short relation for the randomized root digest into the short relation for the ordinary root digest. From that point, the proof uses a per-level noisy expand lemma and induction on the recursion levels: if the incoming short relation at one level is correct, then the local expand step introduces only bounded LacE, LHE, and masking noise, and the denoising margin removes that noise exactly, so the caller recovers the correct short relation for the next level. Iterating this argument reaches the encoded leaf input, where the leaf denoising step yields the exact ring-level VOLE relation, and $\mathsf{Encode}^{-1}$ then recovers the final $\mathbb{Z}_p$-level output relation. The non-interactive mode follows immediately, because after it establishes the same randomized-root short relation, the rest of the argument is identical. See Section B.3 for the full proof. □

## B.2 Semi-honest proof of CI-VOLE

**Definition 11** (Semi-honest security for a corrupted sender)**.** *Let* $\Pi$ *be the protocol realizing* $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$*. Let input length* $w$ *and number of queries* $N$ *with* $w, N \le \mathsf{poly}(\lambda)$*. Sender input* $\Delta$*, and receiver input* $\{\mathbf{x}_i \in \mathbb{Z}_p^w\}_{i\in[N]}$*. Also, there is a known sequence of invocations of* $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ *operations*

$$\mathsf{seq} = (\mathsf{call}_i \in \{\mathsf{set\text{-}delta}, \mathsf{setx\text{-}ni}, \mathsf{release\text{-}ni}, \mathsf{setk\text{-}int}, \mathsf{setx\text{-}int}, \mathsf{release\text{-}int}\})_i.$$

*This shared information is bundled as* $\mathsf{parm} = (1^\lambda, w, N, \mathsf{seq})$*. Let* $\mathsf{out}_{\mathbb{S}}$ *denote the ordered stream of outputs delivered to the corrupted sender by the ideal functionality while the sequence* $\mathsf{seq}$ *is executed. Thus* $\mathsf{out}_{\mathbb{S}}$ *records both the value and the time at which each sender output is released: in the non-interactive mode, the sender output appears only at* $\mathsf{release\text{-}ni}$*, while in the interactive mode it appears at* $\mathsf{setk\text{-}int}$*. Let* $\mathsf{out}_{\mathbb{R}}^{\mathsf{real}}$ *denote the ordered stream of outputs delivered to the honest* $\mathbb{R}$ *in the real execution of* $\Pi$*, and let* $\mathsf{out}_{\mathbb{R}}^{\mathsf{ideal}}$ *denote the ordered stream of outputs delivered to the honest* $\mathbb{R}$ *by* $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ *while the sequence* $\mathsf{seq}$ *is executed.*

*Define the real and simulated corrupted-sender games*

$$\mathsf{Game}_{\mathbb{S}}^{\mathsf{real}}(\mathsf{parm}, \Delta, \{\mathbf{x}_i\}_{i\in[N]}) \text{ and } \mathsf{Game}_{\mathbb{S}}^{\mathsf{sim}}(\mathsf{parm}, \Delta, \mathsf{out}_{\mathbb{S}})$$

*as follows.*

- $\mathsf{Game}_{\mathbb{S}}^{\mathsf{real}}$*: run the real protocol on* $\mathbb{S}$*'s setup* $\Pi.\mathsf{set\text{-}delta}(\Delta, w)$*. Then the parties invoke the functionalities according to* $\mathsf{seq}$ *where* $\mathbb{R}$ *inputs* $\{\mathbf{x}_i\}_{i\in[N]}$*. Then the game outputs*

$$(\mathsf{view}_{\mathbb{S}}^{\mathsf{real}}, \mathsf{out}_{\mathbb{R}}^{\mathsf{real}}).$$

- $\mathsf{Game}_{\mathbb{S}}^{\mathsf{sim}}$*: let*

$$\mathsf{view}_{\mathbb{S}}^{\mathsf{sim}} \leftarrow \mathsf{Sim}_{\mathbb{S}}(\mathsf{parm}, \Delta, \mathsf{out}_{\mathbb{S}}).$$

*Then the game outputs*

$$(\mathsf{view}_{\mathbb{S}}^{\mathsf{sim}}, \mathsf{out}_{\mathbb{R}}^{\mathsf{ideal}}).$$

*We say that* $\Pi$ *is* semi-honestly secure against a corrupted sender *if there exists a PPT simulator* $\mathsf{Sim}_{\mathbb{S}}$ *such that*

$$\left\{\mathsf{Game}_{\mathbb{S}}^{\mathsf{real}}(\mathsf{parm}, \Delta, \{\mathbf{x}_i\}_{i\in[N]})\right\}_\lambda \approx_c^{2^\lambda} \left\{\mathsf{Game}_{\mathbb{S}}^{\mathsf{sim}}(\mathsf{parm}, \Delta, \mathsf{out}_{\mathbb{S}})\right\}_\lambda$$

**Definition 12** (Semi-honest security for a corrupted receiver)**.** *Let* $\Pi$ *be the protocol realizing* $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ *and the parameters are defined as in Def. 11. Let* $\mathsf{out}_{\mathbb{R}}$ *denote the outputs delivered to the corrupted receiver by the ideal functionality as the invocation sequence is executed. Again,* $\mathsf{out}_{\mathbb{R}}$ *is an ordered stream: in the non-interactive mode it records the output delivered at* setx-ni*, whereas in the interactive mode it records the output delivered only at* release-int*. Let* $\mathsf{out}_{\mathbb{S}}^{\mathsf{real}}$ *denote the ordered stream of outputs delivered to the honest* $\mathbb{S}$ *in the real execution of* $\Pi$*, and let* $\mathsf{out}_{\mathbb{S}}^{\mathsf{ideal}}$ *denote the ordered stream of outputs delivered to the honest* $\mathbb{S}$ *by* $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ *while the sequence* seq *is executed. Define the real and simulated corrupted-receiver games* $\mathsf{Game}_{\mathbb{R}}^{\mathsf{real}}(\mathsf{parm}, \Delta, \{\mathbf{x}_i\}_{i\in[N]})$ *and* $\mathsf{Game}_{\mathbb{R}}^{\mathsf{sim}}(\mathsf{parm}, \{\mathbf{x}_i\}_{i\in[N]}, \mathsf{out}_{\mathbb{R}})$ *as follows.*

- $\mathsf{Game}_{\mathbb{R}}^{\mathsf{real}}$ *is run as in* $\mathsf{Game}_{\mathbb{S}}^{\mathsf{real}}$ *of Def. 11, but outputs*

$$(\mathsf{view}_{\mathbb{R}}^{\mathsf{real}}, \mathsf{out}_{\mathbb{S}}^{\mathsf{real}}).$$

- $\mathsf{Game}_{\mathbb{R}}^{\mathsf{sim}}$*: let*

$$\mathsf{view}_{\mathbb{R}}^{\mathsf{sim}} \leftarrow \mathsf{Sim}_{\mathbb{R}}(\mathsf{parm}, \{\mathbf{x}_i\}_{i\in[N]}, \mathsf{out}_{\mathbb{R}}).$$

  *Then the game outputs*

$$(\mathsf{view}_{\mathbb{R}}^{\mathsf{sim}}, \mathsf{out}_{\mathbb{S}}^{\mathsf{ideal}}).$$

*We say that* $\Pi$ *is* semi-honestly secure against a corrupted receiver *if there exists a PPT simulator* $\mathsf{Sim}_{\mathbb{R}}$ *such that*

$$\{\mathsf{Game}_{\mathbb{R}}^{\mathsf{real}}(\mathsf{parm}, \Delta, \{\mathbf{x}_i\}_{i\in[N]})\}_\lambda \approx_c^{2^\lambda} \{\mathsf{Game}_{\mathbb{R}}^{\mathsf{sim}}(\mathsf{parm}, \{\mathbf{x}_i\}_{i\in[N]}, \mathsf{out}_{\mathbb{R}})\}_\lambda.$$

**Theorem 9** (Semi-honest security of the core protocol)**.** *Assume the* LacE *and* LHE *instantiations used by the protocol are* $2^\lambda$*-simulation secure with the leakage stated in Sections 3.6.2 and 3.6.4, and assume the randomized root digest is statistically hiding as in Lemma 7. Let* $\Pi$ *be the* CI-VOLE *protocol obtained by combining the wrapper* $\mathsf{LogVOLE}_{\mathbb{Z}}$ *from Figure 4 with the core ring protocol* $\mathsf{LogVOLE}_{\mathcal{R}}$ *from Figures 10 and 11, excluding the optimization from Section 7.* $\Pi$ *is semi-honestly secure against corrupted senders and receivers (Def. 11 and 12).*

*Proof sketch.* For a corrupted $\mathbb{S}$, the simulator reproduces the offline view exactly, replaces the randomized root digest by a uniform ring element using Lemma 7 and theorem 4, and then programs the session-specific $H$- and $H_{\mathsf{ct}}$-queries so that the sender's local computation is consistent with the released sender output in both modes. A short hybrid argument shows that the only changes are the digest replacement, the programmed correction ciphertexts, and a negligible pre-query abort event.

For a corrupted $\mathbb{R}$, the simulator first replaces the receiver-visible setup ciphertexts, then programs the online oracle points from the leaves upward, and finally simulates the root response. At each level, the next target relation is obtained from the previous one by the same deterministic recombination map used in the protocol. The corresponding hybrid argument first programs the random oracle with valid ciphertexts and then replaces the level-$t$ LacE and LHE setup ciphertexts, together with the corresponding correction ciphertexts (used in programming the random oracle), by their simulated counterparts, first at the leaf level, then at each intermediate level, and finally at the root. The full proof appears in Appendix B.4.

□

## B.3 Proof of Theorem 8

*Proof.* We first prove the statement for mode = interactive. We then explain why the proof for mode = non-interactive follows immediately from the same argument.

**Recursive input vectors.** We begin from the actual wrapper input

$$\mathbf{x} = (x_1, \ldots, x_w) \in \mathbb{Z}_p^w.$$

$\mathsf{LogVOLE}_{\mathbb{Z}}.\mathsf{Online}$ first applies the batching map $\mathsf{Encode}$ and obtains the encoded ring input

$$\tilde{\mathbf{x}}^{(\ell)} := \mathsf{Encode}(\mathbf{x}) \in \mathcal{R}_q^{w'}, \qquad w' = \lceil w/n \rceil.$$

This is the vector given to the ring-level protocol $\mathsf{LogVOLE}_{\mathcal{R}}.\mathsf{Online}$. Since $\mathsf{LogVOLE}_{\mathcal{R}}$ identifies the leaf level with recursion-tree level $\ell$, it is convenient to denote by $\mathbf{x}^{(t)}$ the vector processed by $\mathsf{OnlineRec}$ at actual level $t \in [\ell]$, with the convention

$$\mathbf{x}^{(\ell)} := \tilde{\mathbf{x}}^{(\ell)} = \mathsf{Encode}(\mathbf{x}).$$

Now fix some level $t \in \{2, \ldots, \ell\}$, and suppose $\mathbf{x}^{(t)} \in \mathcal{R}_q^{w^{(t)}}$ is the current input of $\mathsf{OnlineRec}$. The protocol transforms $\mathbf{x}^{(t)}$ into the next higher recursive input $\mathbf{x}^{(t-1)}$ as follows.

First, it partitions $\mathbf{x}^{(t)}$ into

$$b^{(t)} := \frac{w^{(t)}}{\alpha\tau\rho}$$

blocks

$$\mathbf{x}_1^{(t)}, \ldots, \mathbf{x}_{b^{(t)}}^{(t)} \in \mathcal{R}_q^{\alpha\tau\rho}.$$

For each block, it computes the shrink digest

$$d_i^{(t)} := \mathsf{Shrink}(\mathbf{x}_i^{(t)}) \in \mathcal{R}_q.$$

Next, it applies gadget decomposition to each digest:

$$\mathbf{g}^{-1}(d_i^{(t)}) \in \mathcal{R}_q^{\tau}.$$

Each of those $\tau$ ring elements is then reduced componentwise modulo each CRT modulus $q_h$, $h \in [\rho]$. This gives the CRT-expanded digit vector

$$\left((\mathbf{g}^{-1}(d_i^{(t)})[u] \bmod q_h)_{h\in[\rho]}\right)_{u\in[\tau]} \in \mathcal{R}_q^{\tau\rho}.$$

Finally, the protocol transposes this $\rho\tau$-layout into the $\tau\rho$-layout used by the next recursive call. Denoting the resulting vector by $\hat{d}_i^{(t-1)} \in \mathcal{R}_q^{\tau\rho}$, the input at the next higher level is

$$\mathbf{x}^{(t-1)} := (\hat{d}_1^{(t-1)}, \ldots, \hat{d}_{b^{(t)}}^{(t-1)}) \in \mathcal{R}_q^{b^{(t)}\tau\rho}.$$

Thus $\mathbf{x}^{(t-1)}$ is obtained from $\mathbf{x}^{(t)}$ by four deterministic steps:
1. partition into blocks of width $\alpha\tau\rho$,
2. shrink each block to one digest in $\mathcal{R}_q$,
3. apply gadget decomposition to each digest,
4. write those digits in CRT coordinates and transpose them into the recursive format.

Iterating this description from $t = \ell$ down to $t = 1$ defines every input vector

$$\mathbf{x}^{(\ell)}, \mathbf{x}^{(\ell-1)}, \ldots, \mathbf{x}^{(1)}.$$

The root level $t = 1$ is the base level of the recursive proof, because it is the stopping case of $\mathsf{OnlineRec}$: at that level no further recursive call is made, and the procedure uses the short root relation returned by $\mathsf{DerandRoot}$.

**Induction statement.** For $t \in [\ell]$, let $\mathcal{I}_t$ be the following statement:
$\mathcal{I}_t$**.** When $\mathsf{OnlineRec}$ is called on input vector $\mathbf{x}^{(t)}$ together with the correct incoming short authenticated relation for that level, the procedure returns exact authenticated shares of the entire vector $\mathbf{x}^{(t)}$. Moreover, if $t < \ell$, then after the caller above applies CRT recombination and gadget aggregation to that returned value, it recovers the exact short digest-side relation required to expand the blocks of $\mathbf{x}^{(t+1)}$.

We prove $\mathcal{I}_t$ for all $t \in [\ell]$ by induction on $t$, starting from the root $t = 1$ and proceeding toward the leaf $t = \ell$.

**A local expand lemma.** We first isolate the algebra used at one level.

**Lemma 9** (Per-level expand identity)**.** *Fix a level $t \in [\ell]$ and one block index $i \in [b^{(t)}]$. Suppose the incoming short digest-side relation for that block is correct, namely*

$$\mathsf{sk}_{d_i}^{(t)} = \mathsf{sk}_{\mathsf{r}}{}^{(t)\top} \mathbf{g}^{-1}(d_i^{(t)}) + \mathbf{k}_i^{(t)},$$

*where $\mathbf{k}_i^{(t)}$ is the sender's short share and $\mathsf{sk}_{d_i}^{(t)}$ is the receiver's matching short share. Then the local expand step for $\mathbf{x}_i^{(t)}$ outputs noisy values $\tilde{\mathbf{k}}_i^{(t)}, \tilde{\mathbf{m}}_i^{(t)} \in \mathcal{R}_q^{\alpha\tau\rho}$ such that*

$$\tilde{\mathbf{m}}_i^{(t)} - \tilde{\mathbf{k}}_i^{(t)} = \mathbf{s}_i^{(t)} \odot \mathbf{x}_i^{(t)} + \mathbf{e}^{(t,i)},$$

*with $\|\mathbf{e}^{(t,i)}\|_\infty \le B$, where*

$$B := B_{\mathsf{LacE}} + B_{\mathsf{LHE}} + \max \overline{\chi}_{\mathsf{LacE}}.$$

*If the relevant scale factor is larger than $2B$, then the denoising step removes $\mathbf{e}^{(t,i)}$ exactly.*

*Proof.* Fix the block $\mathbf{x}_i^{(t)}$, its digest $d_i^{(t)}$, the local LacE ciphertext $\mathsf{ct}_{\mathsf{lac}}$, and the local ciphertext $\mathsf{ct}_{\mathsf{r}}$. Write $\mathbf{r}_i^{(t)} \in \mathcal{R}_q^{\alpha\tau\rho}$ for the hidden LacE randomness vector used in that offline package, and write $\mathbf{s}_i^{(t)}$ for the sender's underlying block secret. By the $\mathsf{LacE}$ interface, evaluating $\mathsf{ct}_{\mathsf{lac}}$ on $\mathbf{x}_i^{(t)}$ yields

$$\mathsf{LacE.Eval}(\mathsf{ct}_{\mathsf{lac}}, \mathbf{x}_i^{(t)}) = \mathbf{r}_i^{(t)} \cdot d_i^{(t)} - \mathbf{s}_i^{(t)} \odot \mathbf{x}_i^{(t)} + \mathbf{e}_{\mathsf{LacE}}^{(t,i)}, \tag{7}$$

where $\mathbf{e}_{\mathsf{LacE}}^{(t,i)}$ is the local LacE evaluation error.

Next, the receiver forms

$$\mathsf{ct}_{\mathrm{res}} := \mathsf{ct}_{\mathsf{r}} \cdot \mathbf{g}^{-1}(d_i^{(t)}) + \mathsf{ct}_{\mathsf{k}}.$$

Because $\mathsf{ct}_{\mathsf{r}}$ is an LHE encryption of the bridge vector and the incoming short relation is $\mathsf{sk}_{d_i}^{(t)} = \mathsf{sk}_{\mathsf{r}}{}^{(t)\top} \mathbf{g}^{-1}(d_i^{(t)}) + \mathbf{k}_i^{(t)}$, LHE correctness gives

$$\mathsf{LHE.Dec}(\mathsf{sk}_{d_i}^{(t)}, \mathsf{ct}_{\mathrm{res}}) = \mathbf{r}_i^{(t)} \cdot d_i^{(t)} + \tilde{\mathbf{k}}_i^{(t)} + \mathbf{e}_{\mathsf{LHE}}^{(t,i)} + \mathbf{e}_{\mathsf{mask}}^{(t,i)}, \tag{8}$$

where $\tilde{\mathbf{k}}_i^{(t)} = \mathsf{LHE.Dec}(\mathbf{k}_i^{(t)}, \mathsf{ct}_{\mathsf{k}})$ is the sender's local noisy share, $\mathbf{e}_{\mathsf{LHE}}^{(t,i)}$ is the LHE decryption error, and $\mathbf{e}_{\mathsf{mask}}^{(t,i)}$ is the masking term hidden in $\mathsf{ct}_{\mathsf{k}}$.

The receiver then subtracts the LacE evaluation from the LHE output. Combining (7) and (8), we obtain

$$\begin{aligned} \tilde{\mathbf{m}}_i^{(t)} &:= \mathsf{LHE.Dec}(\mathsf{sk}_{d_i}^{(t)}, \mathsf{ct}_{\mathrm{res}}) - \mathsf{LacE.Eval}(\mathsf{ct}_{\mathsf{lac}}, \mathbf{x}_i^{(t)}) \\ &= \mathbf{s}_i^{(t)} \odot \mathbf{x}_i^{(t)} + \tilde{\mathbf{k}}_i^{(t)} + \mathbf{e}_{\mathsf{LacE}}^{(t,i)} + \mathbf{e}_{\mathsf{LHE}}^{(t,i)} + \mathbf{e}_{\mathsf{mask}}^{(t,i)}. \end{aligned} \tag{9}$$

Thus

$$\tilde{\mathbf{m}}_i^{(t)} - \tilde{\mathbf{k}}_i^{(t)} = \mathbf{s}_i^{(t)} \odot \mathbf{x}_i^{(t)} + \mathbf{e}^{(t,i)},$$

where

$$\mathbf{e}^{(t,i)} := \mathbf{e}_{\mathsf{LacE}}^{(t,i)} + \mathbf{e}_{\mathsf{LHE}}^{(t,i)} + \mathbf{e}_{\mathsf{mask}}^{(t,i)}.$$

Now let $B_{\mathsf{LacE}}$ be a uniform bound on the infinity norm of the LacE evaluation noise, let $B_{\mathsf{LHE}}$ be a uniform bound on the infinity norm of the LHE decryption noise, and let $\overline{\chi}_{\mathsf{LacE}}$ be the masking-noise distribution used to hide the LacE-side leakage. Define

$$B := B_{\mathsf{LacE}} + B_{\mathsf{LHE}} + \max \overline{\chi}_{\mathsf{LacE}}.$$

Then $\|\mathbf{e}^{(t,i)}\|_\infty \le B$.

It remains to justify exact denoising. At an internal level, the vector $\tilde{\mathbf{m}}_i^{(t)}$ is organized in CRT coordinates, so each relevant entry is interpreted modulo one CRT modulus $q_h$, and the scale gap is $\delta_h = q/q_h$. At the leaf level, the relevant scale gap is $\delta = \lfloor q/p \rfloor$. In either case, if the chosen scale factor is strictly larger than $2B$, then every coordinate lies strictly within half of the target rounding interval around the intended clean value. Therefore the denoising map recovers that clean value uniquely. This is exactly the claimed statement. □

**Base case:** $t = 1$ **(the root level).** In the interactive mode, the receiver computes the deterministic root digest $d_{\mathsf{rt}}$, randomizes it to $d'_{\mathsf{rt}}$, and sends $d'_{\mathsf{rt}}$ to the sender. The sender samples $k'_{\mathsf{rt}} \leftarrow \mathcal{R}_q$, computes

$$\mathsf{sk_{rt}}' := \mathsf{sk}_{\mathsf{r,rt}}^\top \mathbf{g}^{-1}(d'_{\mathsf{rt}}) + k'_{\mathsf{rt}},$$

and sends $(\varsigma, \mathsf{sk_{rt}}')$ to the receiver. We now open the algebra in $\mathsf{DerandRoot}$ and verify that it indeed removes the randomization. Recall,

$$\hat{d}_{\mathsf{rt}} := \mathsf{decompose}(d_{\mathsf{rt}}) \in \mathcal{R}_q^{\tau\rho} \qquad \text{and} \qquad y_{\mathsf{L}} := \mathsf{LacE.Digest}(\hat{d}_{\mathsf{rt}}).$$

By the definition of the randomized root digest in Figure 8,

$$d'_{\mathsf{rt}} = (\mathbf{b}_1^\top \; B^*) \begin{pmatrix} \mathbf{g}^{-1}(y_{\mathsf{L}}) \\ \zeta_{\mathsf{rt}} \end{pmatrix}.$$

The root-wrapper LacE ciphertext $\mathsf{ct}_{\mathsf{lac,rt}}$ is generated from the vector

$$\hat{\mathsf{sk}}_{\mathsf{r}} := \mathsf{transpose}\big((\mathsf{sk}_{\mathsf{r}t} \cdot \delta_t)_{t\in[\rho]}\big) \in \mathcal{R}_q^{\tau\rho},$$

and its row formula is

$$\mathbf{C}_{\mathsf{rt}}^{\mathsf{top}}[\mathsf{ind}] = \mathbf{r}_{\mathsf{rt}}^{(1)}[\mathsf{ind}] \cdot (\mathbf{b}_1^\top \; B^*) - \mathbf{r}_{\mathsf{rt}}^{(2)}[\mathsf{ind}] \cdot (\mathbf{g}^\top \; 0^m) + \mathbf{E}_{\mathsf{rt}}^{\mathsf{top}}[\mathsf{ind}].$$

Therefore, evaluating that LacE object on $\binom{\mathbf{g}^{-1}(y_{\mathsf{L}})}{\zeta_{\mathsf{rt}}}$ gives

$$\mathbf{C}_{\mathsf{rt}}^{\mathsf{top}} \cdot \begin{pmatrix} \mathbf{g}^{-1}(y_{\mathsf{L}}) \\ \zeta_{\mathsf{rt}} \end{pmatrix} = \mathbf{r}_{\mathsf{rt}}^{(1)} \cdot d'_{\mathsf{rt}} - \mathbf{r}_{\mathsf{rt}}^{(2)} \cdot y_{\mathsf{L}} + \mathbf{e}_{\mathsf{rt}}^{\mathsf{top}}, \tag{10}$$

where $\mathbf{e}_{\mathsf{rt}}^{\mathsf{top}}$ is the top-row masking error.

Next, $\mathsf{ct}_{\mathsf{lacrt},L}$ is an ordinary $\mathsf{LacE}$ ciphertext for $\hat{\mathsf{sk}}_{\mathsf{r}}$. Therefore the standard $\mathsf{LacE}$ interface applied to $\hat{d}_{\mathsf{rt}}$ gives

$$\mathsf{LacE.Eval}(\mathsf{ct}_{\mathsf{lacrt},L}, \hat{d}_{\mathsf{rt}}) = \mathbf{r}_{\mathsf{rt}}^{(2)} \cdot y_{\mathsf{L}} - \hat{\mathsf{sk}}_{\mathsf{r}} \odot \hat{d}_{\mathsf{rt}} + \mathbf{e}_{\mathsf{rt}}^{\mathsf{sub}}, \tag{11}$$

where $\mathbf{e}_{\mathsf{rt}}^{\mathsf{sub}}$ is the lower-subtree LacE evaluation error.

The receiver adds (10) and (11) when it computes $\boldsymbol{\phi}_{\mathsf{rt}}$. Hence

$$\boldsymbol{\phi}_{\mathsf{rt}} = \mathbf{r}_{\mathsf{rt}}^{(1)} \cdot d'_{\mathsf{rt}} - \hat{\mathsf{sk}}_{\mathsf{r}} \odot \hat{d}_{\mathsf{rt}} + \mathbf{e}_{\mathsf{rt}}^{\mathsf{lac}}, \tag{12}$$

where

$$\mathbf{e}_{\mathsf{rt}}^{\mathsf{lac}} := \mathbf{e}_{\mathsf{rt}}^{\mathsf{top}} + \mathbf{e}_{\mathsf{rt}}^{\mathsf{sub}}.$$

The $\mathbf{r}_{\mathsf{rt}}^{(2)} \cdot y_{\mathsf{L}}$ terms cancel at this step.

On the other hand, the root ciphertext $\mathsf{ct}_{\mathsf{r},\mathsf{rt}}$ is an LHE encryption of $\mathbf{r}_{\mathsf{rt}}^{(1)}$, so after the receiver forms

$$\mathsf{ct}_{\mathsf{res},\mathsf{rt}} := \mathsf{ct}_{\mathsf{r},\mathsf{rt}} \cdot \mathbf{g}^{-1}(d'_{\mathsf{rt}}) + \mathsf{ct}_{\mathsf{k},\mathsf{rt}},$$

LHE correctness implies

$$\mathsf{LHE.Dec}(\mathsf{sk}_{\mathsf{rt}}{}', \mathsf{ct}_{\mathsf{res},\mathsf{rt}}) = \mathbf{r}_{\mathsf{rt}}^{(1)} \cdot d'_{\mathsf{rt}} + \tilde{\mathbf{k}}'_{\mathsf{rt}} + \mathbf{e}_{\mathsf{rt}}^{\mathsf{lhe}}, \tag{13}$$

where $\tilde{\mathbf{k}}'_{\mathsf{rt}} := \mathsf{LHE.Dec}(k'_{\mathsf{rt}}, \mathsf{ct}_{\mathsf{k},\mathsf{rt}})$ is the sender's root noisy share and $\mathbf{e}_{\mathsf{rt}}^{\mathsf{lhe}}$ is the root LHE decryption error.

Subtracting (12) from (13), the $\mathbf{r}_{\mathsf{rt}}^{(1)} \cdot d'_{\mathsf{rt}}$ term cancels, and we obtain

$$\tilde{\mathbf{m}}_{\mathsf{rt}} = \hat{\mathsf{sk}}_{\mathsf{r}} \odot \hat{d}_{\mathsf{rt}} + \tilde{\mathbf{k}}'_{\mathsf{rt}} + \mathbf{e}_{\mathsf{rt}}^{\mathsf{root}}, \tag{14}$$

where $\mathbf{e}_{\mathsf{rt}}^{\mathsf{root}} := \mathbf{e}_{\mathsf{rt}}^{\mathsf{lhe}} - \mathbf{e}_{\mathsf{rt}}^{\mathsf{lac}}$ is the total root-wrapper error term.

Equation (14) is exactly the noisy authenticated relation for the CRT/gadget-expanded vector $\hat{d}_{\mathsf{rt}}$, with MAC vector $\hat{\mathsf{sk}}_{\mathsf{r}}$. The denoising step in $\mathsf{DerandRoot}$ uses the CRT scale factors $\delta_h = q/q_h$, so the same margin argument as in Lemma 9 removes $\mathbf{e}_{\mathsf{rt}}^{\mathsf{root}}$ coordinatewise. Thus, after denoising, the parties hold exact authenticated shares of $\hat{d}_{\mathsf{rt}}$ under the MAC vector $\hat{\mathsf{sk}}_{\mathsf{r}}$. CRT recombination then inverts the reduction modulo $q_1, \ldots, q_\rho$ for each gadget digit, and gadget aggregation reconstructs the original root digest $d_{\mathsf{rt}}$. By the definition of $\hat{\mathsf{sk}}_{\mathsf{r}}$, this recombination turns $\hat{\mathsf{sk}}_{\mathsf{r}} \odot \hat{d}_{\mathsf{rt}}$ into the affine form $\mathsf{sk}_{\mathsf{r},\mathsf{rt}}^{\top}\mathbf{g}^{-1}(d_{\mathsf{rt}})$. Therefore (14) yields the clean short root relation

$$\mathsf{sk}_{\mathsf{rt}} = \mathsf{sk}_{\mathsf{r},\mathsf{rt}}^{\top}\mathbf{g}^{-1}(d_{\mathsf{rt}}) + k_{\mathsf{rt}}.$$

This is precisely the incoming short relation required by the root call to $\mathsf{OnlineRec}$.

At the root level the algorithm does not recurse further. The root input has one block, so it sets $(\mathbf{k}_1, \mathsf{sk}_{\mathbf{d},1}) := (k_{\mathsf{rt}}, \mathsf{sk}_{\mathsf{rt}})$ and expands that block using the scalar digest $d_1^{(1)}$. Applying Lemma 9 to this root block gives the noisy identity (9). Since the root level is internal, the denoising is performed with the CRT scale factors $\delta_h = q/q_h$, and the parameter choice ensures $\delta_h > 2B$ for every $h \in [\rho]$, where $B$ bounds the infinity norm of every local error term. Hence denoising is exact blockwise. Concatenating the blocks yields exact authenticated shares of $\mathbf{x}^{(1)}$, and the caller above would recover the exact short digest-side relation required by the next level. Thus $\mathcal{I}_1$ holds.

**Induction hypothesis.** Fix some $t \in \{1, \ldots, \ell - 1\}$. Assume $\mathcal{I}_t$: whenever $\mathsf{OnlineRec}$ is called on $\mathbf{x}^{(t)}$ with the correct incoming short relation, it returns exact authenticated shares of $\mathbf{x}^{(t)}$, and after CRT recombination and gadget aggregation the caller recovers the exact short digest-side relation needed one level below.

**Induction step.** We prove $\mathcal{I}_{t+1}$. Consider a call to $\mathsf{OnlineRec}$ on the level-$(t+1)$ vector $\mathbf{x}^{(t+1)}$, and assume the incoming short relation at that level is correct.

The algorithm first partitions $\mathbf{x}^{(t+1)}$ into blocks $\mathbf{x}_1^{(t+1)}, \ldots, \mathbf{x}_{b^{(t+1)}}^{(t+1)}$, computes the block digests $d_i^{(t+1)} = \mathsf{Shrink}(\mathbf{x}_i^{(t+1)})$, applies gadget decomposition, writes the result in CRT coordinates, and

transposes the coordinates to form the higher-level recursive input $\mathbf{x}^{(t)}$. This is exactly the deterministic transformation described at the beginning of the proof.

The protocol now makes the recursive call on $\mathbf{x}^{(t)}$. By the induction hypothesis $\mathcal{I}_t$, this call returns exact authenticated shares of $\mathbf{x}^{(t)}$. We now open the caller-side post-processing explicitly. Because each $\hat{d}_i^{(t)}$ was formed by listing the CRT coordinates of $\mathbf{g}^{-1}(d_i^{(t+1)})$, the returned value for block $i$ consists of authenticated shares of those very CRT coordinates. Applying CRT recombination first inverts the reduction modulo $q_1, \ldots, q_\rho$ and therefore reconstructs the gadget digits of $d_i^{(t+1)}$. Applying gadget aggregation next multiplies those digits by the gadget basis $(1, g, \ldots, g^{\tau-1})$ and sums them, thereby reconstructing the original digest $d_i^{(t+1)}$. Consequently the caller recovers the exact short digest-side relation

$$\mathsf{sk}_{d_i}^{(t+1)} = \mathsf{sk}_{\mathsf{r}}{}^{(t+1)\top}\mathbf{g}^{-1}(d_i^{(t+1)}) + \mathbf{k}_i^{(t+1)}$$

for each block $i$.

Each block $\mathbf{x}_i^{(t+1)}$ is then expanded using that exact short relation. Applying Lemma 9 to every block shows that the receiver's local output has the form

$$\tilde{\mathbf{m}}_i^{(t+1)} = \mathbf{s}_i^{(t+1)} \odot \mathbf{x}_i^{(t+1)} + \tilde{\mathbf{k}}_i^{(t+1)} + \mathbf{e}^{(t+1,i)}.$$

If $t+1 < \ell$, then this is still an internal level, so the relevant scale factors are again the CRT factors $\delta_h = q/q_h$, all larger than $2B$. Hence denoising removes $\mathbf{e}^{(t+1,i)}$ exactly for every block. The concatenation of the denoised blocks therefore yields exact authenticated shares of the whole vector $\mathbf{x}^{(t+1)}$. This proves $\mathcal{I}_{t+1}$.

**Conclusion at the encoded leaf input.** By induction, $\mathcal{I}_t$ holds for every level $t \in [\ell]$, and in particular for the leaf level $t = \ell$. At that level,

$$\mathbf{x}^{(\ell)} = \tilde{\mathbf{x}}^{(\ell)} = \mathsf{Encode}(\mathbf{x}).$$

Unlike the internal levels, the leaf package is generated from the repeated $\Delta$-vector and targets the final VOLE relation rather than a digest-side relation. Thus the clean block value in (9) is now $\mathsf{Repeat}(\alpha\tau\rho, \Delta) \odot \mathbf{x}_i^{(\ell)}$, and the relevant scale factor is the leaf factor $\delta = \lfloor q/p \rfloor$. Since $\delta > 2B$, the final denoising step is exact. Therefore the leaf-level output of the ring protocol is the exact encoded relation

$$\mathbf{m} = \tilde{\mathbf{x}}^{(\ell)} \cdot \Delta + \mathbf{k}.$$

**The final $\mathsf{Encode}^{-1}$ step.** The wrapper $\mathsf{LogVOLE}_{\mathbb{Z}}.\mathsf{Online}$ applies $\mathsf{Encode}^{-1}$ to the ring outputs of both parties. By definition, $\mathsf{Encode}^{-1}$ is the inverse of the batching map $\mathsf{Encode}$, so it recovers the coordinate representation in $\mathbb{Z}_p^w$ without changing the affine relation. Applying $\mathsf{Encode}^{-1}$ to the exact encoded relation above therefore gives

$$\mathbf{m} = \mathbf{x} \cdot \Delta + \mathbf{k}$$

coordinatewise in $\mathbb{Z}_p^w$. This proves Theorem 8 for the interactive mode.

**Remark on the non-interactive mode.** The non-interactive case follows immediately from the same proof. The only difference in Figure 11 is that the pair $(\mathsf{sk}_{\mathsf{rt}}{}', \varsigma)$ is derived as

$$(\mathsf{sk}_{\mathsf{rt}}{}', \varsigma) := H(\mathsf{id}, d'_{\mathsf{rt}}),$$

and the sender then defines

$$k'_{\mathsf{rt}} := -\mathsf{sk}_{\mathsf{r},\mathsf{rt}}^{\top}\mathbf{g}^{-1}(d'_{\mathsf{rt}}) + \mathsf{sk}_{\mathsf{rt}}{}'.$$

Hence the same root relation

$$\mathsf{sk_{rt}}' = \mathsf{sk}_{\mathsf{r,rt}}^{\top}\mathbf{g}^{-1}(d'_{\mathsf{rt}}) + k'_{\mathsf{rt}}$$

holds exactly. This is the only property of $(\mathsf{sk_{rt}}', \varsigma)$ used anywhere in the interactive proof. After that point, DerandRoot, all recursive input vectors $\mathbf{x}^{(t)}$, every application of Lemma 9, and the final $\mathsf{Encode}^{-1}$ step are identical. Therefore the non-interactive mode also outputs $\mathbf{m} = \mathbf{x} \cdot \Delta + \mathbf{k}$ with probability 1. □

### B.4 Proof of Theorem 9 (Semi-honest Security of the Core Protocol)

*Proof.* Below, we treat the two-corruption cases separately. The corrupt-sender case proves Definition 11, and the corrupt-receiver case is the analogous simulation statement on the receiver side. Together they imply the theorem.

**Case 1: $\mathbb{S}$ is corrupt.** We first isolate the parts of the corrupted sender's view whose simulation is not immediate.

In the offline phase, the corrupted $\mathbb{S}$ sees only its input $(\Delta, w)$, its random tape, and the sender state produced by the local computation. It receives no message from $\mathbb{R}$ in $\mathsf{LogVOLE}_{\mathbb{Z}}$.Offline, $\mathsf{LogVOLE}_{\mathcal{R}}$.Offline, $\mathsf{ShrinkExpand}_{\mathcal{R}}$.Offline, or SetupRoot. Hence, the simulation of the offline view is straight-forward.

In the online phase, in both the interactive and non-interactive mode, the only message received by $\mathbb{S}$ is the randomized root digest $d'_{\mathsf{rt}}$. By Lemma 7 and theorem 4, this digest is statistically close to a uniform element of $\mathcal{R}_q$. After $\mathbb{S}$ receives $d'_{\mathsf{rt}}$, the rest of its view is determined by its tape, its offline state, and the oracle interfaces $H$ and $H_{\mathsf{ct}}$. The sender side oracle calls are:

- $\mathsf{FindGoldenSeed}(\mathsf{id}, k'_{\mathsf{rt}}, d'_{\mathsf{rt}})$ in the interactive mode;
- $H(\mathsf{id}, d'_{\mathsf{rt}})$ in the non-interactive mode;
- $H_{\mathsf{ct}}(\mathsf{id}, \varsigma, \mathsf{rt}, 0)$ inside DerandRoot;
- $H_{\mathsf{ct}}(\mathsf{id}, \varsigma, j, i)$ inside every call to $\mathsf{ShrinkExpand}_{\mathcal{R}}$.Expand.

The ideal output is the vector $\mathbf{k}$, released at setk-int in the interactive mode and at release-ni in the non-interactive mode. Thus the simulation task is to replace $d'_{\mathsf{rt}}$ by a uniform ring element and to program $H$ and $H_{\mathsf{ct}}$ so that $\mathbb{S}$'s local computation remains consistent with the released $\mathbf{k}$.

**Simulator $\mathsf{Sim}_{\mathbb{S}}$.** We describe the simulator below.

1. $\mathsf{Sim}_{\mathbb{S}}$ samples a uniformly random tape $R^*$ for the corrupted $\mathbb{S}$ and uses it throughout.
2. $\mathsf{Sim}_{\mathbb{S}}$ maintains two lazy tables:
   - a table $Q_H$ for queries to $H$;
   - a table $Q_{\mathsf{ct}}$ for queries to $H_{\mathsf{ct}}$.

   On a repeated query, the simulator returns the recorded answer. On a fresh query, it records and returns a fresh uniform answer of the appropriate type. Both the corrupted $\mathbb{S}$ and the simulator may query these tables.
3. On the offline call $\mathsf{set\text{-}delta}(\mathbb{S} : \Delta, w) \in \mathsf{seq}$, $\mathsf{Sim}_{\mathbb{S}}$ invokes the ideal functionality on $(\Delta, w)$ and runs $\mathbb{S}$'s offline algorithm on tape $R^*$. This reproduces the view of corrupt sender correctly. $\mathsf{Sim}_{\mathbb{S}}$ records the randomness from $R^*$ corresponding to the $\mathsf{LHE.Enc}_1()$ secret key produced at the root wrapper, i.e., $\mathsf{sk}_{\mathsf{r,rt}}$ from SetupRoot. Looking ahead, this will be useful in programming the random oracles in the non-interactive online mode.

For each online call ($\mathsf{setk\text{-}int}(\mathbb{S} : \mathsf{sid})$) (in interactive mode) or ($\mathsf{release\text{-}ni}(\mathbb{S} : \mathsf{sid})$) (in non-interactive mode): The simulator begins by sampling

$$\widehat{d}'_{\mathsf{rt}} \xleftarrow{\$} \mathcal{R}_q$$

and sends $\widehat{d}'_{\mathsf{rt}}$ to the corrupted $\mathbb{S}$ as the public randomized root digest for that session, after programming the relevant oracle points for that session as described below.

**Interactive Mode.** Consider a session with ideal call $\mathsf{setk\text{-}int}(\mathbb{S} : \mathsf{sid})$.

1. $\mathsf{Sim}_{\mathbb{S}}$ records the randomness from $R^*$ corresponding to the sampled root key share $\mathbf{k}'_{\mathsf{rt}}$.
2. $\mathsf{Sim}_{\mathbb{S}}$ invokes the ideal functionality on $\mathsf{setk\text{-}int}(\mathbb{S} : \mathsf{sid})$ and obtains the ideal output $\mathbf{k} \in \mathbb{Z}_p^w$.
3. $\mathsf{Sim}_{\mathbb{S}}$ next emulates $\mathsf{FindGoldenSeed}$. A seed is rejected not because a ciphertext is syntactically malformed, but because for some designated correction ciphertext the sampled decryption-noise vector has a coefficient in the bad interval $[-\delta/2, -\delta/2 + B) \cup (\delta/2 - B, \delta/2)$. Equivalently, the corresponding bound test $\|\mathbf{e}_{j,i}\|_\infty \leq B$ fails. By the golden-seed analysis in Section 7, acceptance occurs with constant probability, so the simulator can reproduce the same rejection pattern with constant expected overhead.
4. Concretely, $\mathsf{Sim}_{\mathbb{S}}$ programs the $H$-queries of $\mathsf{FindGoldenSeed}$ one counter value at a time. For a rejected counter $q$, it samples $\varsigma_q$ uniformly at random and writes

$$Q_H[(\mathsf{sid}, \widehat{d}'_{\mathsf{rt}}, k'_{\mathsf{rt}}, q)] := \varsigma_q$$

   and programs at least one designated point of $Q_{\mathsf{ct}}$ under $\varsigma_q$ so that the corresponding $\mathbb{S}$-side decryption violates the bound test. For the first accepting counter $q^\star$, it samples $\varsigma$ uniformly at random and it writes

$$Q_H[(\mathsf{sid}, \widehat{d}'_{\mathsf{rt}}, k'_{\mathsf{rt}}, q^\star)] := \varsigma$$

   and programs all designated $H_{\mathsf{ct}}$-points under $\varsigma$ so that every bound test succeeds (this programming is described below). Thus the corrupted $\mathbb{S}$ sees the same control flow as in the real execution: some number of rejected seeds followed by one accepted seed.
5. Once $\varsigma$ is fixed, $\mathsf{Sim}_{\mathbb{S}}$ programs the remaining correction-ciphertext points explicitly. For the root point

$$H_{\mathsf{ct}}(\mathsf{sid}, \varsigma, \mathsf{rt}, 0)$$

   it first fixes the target root key $k_{\mathsf{rt}} \in \mathcal{R}_q$. It then defines

$$\hat{\mathbf{k}}_{\mathsf{rt}} := \mathsf{decompose}(k_{\mathsf{rt}}) \in \mathcal{R}_q^{\tau\rho}.$$

   Next, for each $t \in [\rho]$, it samples a noise vector $\mathbf{e}^{\mathsf{ct}}_{\mathsf{rt},t}$ coefficient-wise from the good interval $[-\delta_t/2 + B, \delta_t/2 - B]$, where $\delta_t := q/q_t$, and sets

$$\tilde{\mathbf{k}}_{\mathsf{rt}} := \mathsf{transpose}((\delta_t \cdot \hat{\mathbf{k}}_{\mathsf{rt},t} + \mathbf{e}^{\mathsf{ct}}_{\mathsf{rt},t})_{t\in[\rho]}) \in \mathcal{R}_q^{\tau\rho}.$$

   Finally it sets

$$Q_{\mathsf{ct}}[(\mathsf{sid}, \varsigma, \mathsf{rt}, 0)] := \mathsf{ct}_{\mathsf{k,rt}},$$

   where $\mathsf{ct}_{\mathsf{k,rt}}$ is sampled from the same distribution as an honest $\mathsf{LHE}$-offset ciphertext under the randomized root key $\mathbf{k}'_{\mathsf{rt}}$, conditioned on decrypting to the prescribed value $\tilde{\mathbf{k}}_{\mathsf{rt}}$. Thus the real sender-side decryption under $k'_{\mathsf{rt}}$ returns

$$\mathsf{LHE.Dec}(k'_{\mathsf{rt}}, \mathsf{ct}_{\mathsf{k,rt}}) = \tilde{\mathbf{k}}_{\mathsf{rt}}$$

   with $\mathsf{Denoise}((q_t)_{t\in[\rho]}, \tilde{\mathbf{k}}_{\mathsf{rt}})$ equal to $\hat{\mathbf{k}}_{\mathsf{rt}}$, and hence

$$\mathsf{Agg}(\mathsf{crt}((q_t)_{t\in[\rho]}, \hat{\mathbf{k}}_{\mathsf{rt}}), \tau \to 1) = \mathbf{k}_{\mathsf{rt}}.$$

6. $\mathsf{Sim}_{\mathbb{S}}$ then proceeds level by level through the recursion. For each recursion level $t \in [\ell]$ and block $i \in [\alpha^{t-1}]$, it first fixes the compact target key that should emerge after the caller applies $\mathsf{Agg}(\mathsf{crt}((q_h)_{h\in[\rho]}, \cdot))$ to the output of the corresponding call to $\mathsf{ShrinkExpand}_{\mathcal{R}}.\mathsf{Expand}$. Write this compact target as $\mathbf{k}^{\star}_{t,i}$. For internal levels, $\mathbf{k}^{\star}_{t,i} \in \mathcal{R}_q^{\alpha}$; at the leaf level, the $\mathbf{k}^{\star}_{t,i}$ blocks are chosen so that $\mathsf{Encode}^{-1}$ applied to their concatenation is exactly the ideal output $\mathbf{k}$. The simulator then defines the expanded target

$$\hat{\mathbf{k}}_{t,i} := \mathsf{decompose}(\mathbf{k}^{\star}_{t,i}) \in \mathcal{R}_q^{\alpha\tau\rho}.$$

Next, for each CRT component $h \in [\rho]$, it samples a noise vector $\mathbf{e}^{\mathsf{ct}}_{t,i,h}$ coefficient-wise from the good interval $[-\delta_h/2 + B, \delta_h/2 - B]$, where $\delta_h := q/q_h$, and sets

$$\tilde{\mathbf{k}}_{t,i} := \mathsf{transpose}\big((\delta_h \cdot \hat{\mathbf{k}}_{t,i,h} + \mathbf{e}^{\mathsf{ct}}_{t,i,h})_{h\in[\rho]}\big) \in \mathcal{R}_q^{\alpha\tau\rho}.$$

It then programs

$$H_{\mathsf{ct}}(\mathsf{sid}, \varsigma, t, i)$$

by setting

$$Q_{\mathsf{ct}}[(\mathsf{sid}, \varsigma, t, i)] := \mathsf{ct}_{\mathsf{k},t,i},$$

and writing $\mathsf{parent\text{-}key}_{t,i}$ for the sender key used by the corresponding real call to $\mathsf{ShrinkExpand}_{\mathcal{R}}.\mathsf{Expand}$. Thus where $\mathsf{ct}_{\mathsf{k},t,i}$ is an honest-form LHE-offset ciphertext, sampled relative to the sender key that the real call $\mathsf{ShrinkExpand}_{\mathcal{R}}.\mathsf{Expand}(\mathbb{S} : (r, \mathsf{parent\text{-}key}_{t,i}), \ldots)$ would use for that block and conditioned on decrypting to the prescribed value $\tilde{\mathbf{k}}_{t,i}$. Concretely:

- at the first level below the root, $\mathsf{parent\text{-}key}_{t,i} = \mathbf{k}_{\mathsf{rt}}$, namely the root key returned by $\mathsf{DerandRoot}$;
- at every deeper level $t > 1$, if $a_t(i) := \lceil i/\alpha \rceil$ denotes the ancestor block index of $i$ at level $t-1$, then $\mathsf{parent\text{-}key}_{t,i} = \mathbf{k}^{\star}_{t-1,a_t(i)}$, namely the compact key fixed for that ancestor block after the caller's $\mathsf{Agg}(\mathsf{crt}(\cdot))$ step;
- at the leaf level $t = \ell$, the chosen compact targets $\mathbf{k}^{\star}_{t,i}$ are the leaf blocks that decode to the final ideal output $\mathbf{k}$.

The ciphertext is programmed so that the real sender-side decryption returns

$$\mathsf{LHE.Dec}(\mathsf{parent\text{-}key}_{t,i}, \mathsf{ct}_{\mathsf{k},t,i}) = \tilde{\mathbf{k}}_{t,i},$$

and hence satisfies

$$\mathsf{Denoise}(\phi_t, \tilde{\mathbf{k}}_{t,i}) = \hat{\mathbf{k}}_{t,i},$$

where $\phi_t = \mathsf{Repeat}(\rho, p)$ at the leaf level and $\phi_t = (q_h)_{h\in[\rho]}$ at every internal level. After the real CRT recombination and aggregation steps, these programmed values induce the compact target $\mathbf{k}^{\star}_{t,i}$. These compact targets are exactly the next-level recursive keys at internal levels and decode to the ideal output $\mathbf{k}$ at the leaves.

7. If any $H$- or $H_{\mathsf{ct}}$-query point that $\mathsf{Sim}_{\mathbb{S}}$ needs to program has already been assigned in $Q_H$ or $Q_{\mathsf{ct}}$, then $\mathsf{Sim}_{\mathbb{S}}$ aborts.

**Non-interactive Mode.** The non-interactive simulation is the same, except that the protocol does not run $\mathsf{FindGoldenSeed}$. Instead, $\mathsf{Sim}_{\mathbb{S}}$ programs the online $H$-query directly.

1. $\mathsf{Sim}_{\mathbb{S}}$ first programs

$$Q_H[(\mathsf{sid}, \widehat{d}'_{\mathsf{rt}})] := (\mathsf{sk_{rt}}', \varsigma)$$

and thereby fixes $\mathbb{S}$'s root response. This determines the local value

$$\mathbf{k}'_{\mathsf{rt}} := -\mathsf{sk}^{\top}_{\mathsf{r},\mathsf{rt}}\mathbf{g}^{-1}(\widehat{d}'_{\mathsf{rt}}) + \mathsf{sk_{rt}}'$$

exactly as in the real protocol.

2. With $(\mathsf{sk_{rt}}', \varsigma)$ fixed, $\mathsf{Sim}_{\mathbb{S}}$ programs the same root point and the same root-to-leaf $H_{\mathsf{ct}}$-family exactly as in the interactive case, with the same conditioned-noise rule, the same $\mathsf{parent\text{-}key}_{t,i}$ notation, and the same abort condition on pre-programmed query points. The only difference is that there is no FindGoldenSeed loop: the seed is taken directly from $Q_H[(\mathsf{sid}, \widehat{d}'_{\mathsf{rt}})]$.

This completes the simulator construction for the corrupt-$\mathbb{S}$ case.

**Hybrid argument for corrupt $\mathbb{S}$.** We now prove that, for the simulator $\mathsf{Sim}_{\mathbb{S}}$ above, $\mathsf{Game}^{\mathsf{real}}_{\mathbb{S}}$ is computationally indistinguishable from $\mathsf{Game}^{\mathsf{sim}}_{\mathbb{S}}$. Let $\mathcal{H}_i$ denote the $i$-th hybrid, and let $\mathcal{D}_i$ denote the joint distribution of the corrupted $\mathbb{S}$-view and the honest $\mathbb{R}$-output when $\mathsf{Sim}_{\mathbb{S}}$ runs $\mathcal{H}_i$.

- $\mathcal{H}_0$: the honest $\mathbb{R}$ runs $\Pi$ on actual inputs, and the corrupted $\mathbb{S}$ interacts with it in the real execution. Thus
$$\mathcal{D}_0 \equiv \mathsf{Game}^{\mathsf{real}}_{\mathbb{R}}.$$
- $\mathcal{H}_1$: this hybrid is the same as $\mathcal{H}_0$, except that all answers to $H$ and $H_{\mathsf{ct}}$ are generated lazily from the tables $Q_H$ and $Q_{\mathsf{ct}}$ as in Step 2 of $\mathsf{Sim}_{\mathbb{S}}$. Since the real execution also samples these answers independently and returns the same answer on repeated queries,
$$\mathcal{D}_1 \equiv \mathcal{D}_0.$$
- $\mathcal{H}_2$: this hybrid is the same as $\mathcal{H}_1$, except that in each session it samples the target vector $\mathbf{k}$ uniformly at random and then programs the oracle points exactly as in the interactive and non-interactive branches of $\mathsf{Sim}_{\mathbb{S}}$. We compare the two views directly. In the interactive mode, $\mathsf{Sim}_{\mathbb{S}}$ first emulates FindGoldenSeed by programming the candidate $H$-answers so that rejected counters produce the same bad-noise event used in the real bound test, and the first accepting counter produces the same success event. Since the real procedure also scans counters until the first seed whose designated correction ciphertexts satisfy the same bound, this emulation distribution is unchanged. Conditioned on an accepting seed, each programmed point $Q_{\mathsf{ct}}[(\mathsf{sid}, \varsigma, \mathsf{rt}, 0)]$ and $Q_{\mathsf{ct}}[(\mathsf{sid}, \varsigma, t, i)]$ is sampled so that decryption under the known sender key returns scaled target plaintext plus a noise vector drawn coefficient-wise from the same good interval as in the real execution. Thus, from the corrupted $\mathbb{S}$'s perspective, the programmed oracle answers have the same distribution as the honest correction ciphertexts conditioned on seed acceptance. The same argument applies in the non-interactive mode, except that the accepting seed is fixed directly by $Q_H[(\mathsf{sid}, \widehat{d}'_{\mathsf{rt}})]$ rather than by a counter search. Finally, the simulation aborts only if a point that must be programmed has already been fixed in $Q_H$ or $Q_{\mathsf{ct}}$. This is a bad event consisting of a pre-query to one of the session-specific programmed points, and its probability is negligible by the polynomial query bound and the randomness of $\varsigma$. Hence
$$\mathcal{D}_2 \approx_c \mathcal{D}_1.$$
- $\mathcal{H}_3$: this hybrid is the same as $\mathcal{H}_2$, except that the randomized root digest $d'_{\mathsf{rt}}$ is replaced by a uniform element of $\mathcal{R}_q$. By Lemma 7 and theorem 4,
$$\mathcal{D}_3 \approx_c \mathcal{D}_2.$$
- $\mathcal{H}_4$: this hybrid is the same as $\mathcal{H}_3$, except that the vector $\mathbf{k}$ is no longer sampled internally by the hybrid. Instead, in the interactive mode it is taken from the ideal call setk-int, and in the non-interactive mode it is taken from the ideal call release-ni. In $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$, this vector is uniform in both modes. Therefore
$$\mathcal{D}_4 \equiv \mathcal{D}_3.$$

By construction, $\mathcal{H}_4$ is exactly $\mathsf{Game}^{\mathsf{sim}}_{\mathbb{S}}$.

Combining the hybrid transitions gives

$$\mathsf{Game}_{\mathbb{S}}^{\mathsf{real}} \approx_c \mathsf{Game}_{\mathbb{S}}^{\mathsf{sim}},$$

which proves the corrupt-$\mathbb{S}$ case.

**Case 2: $\mathbb{R}$ is corrupt.**

We first isolate the parts of the corrupted receiver's view whose simulation is not immediate.

In the offline phase, the corrupted $\mathbb{R}$ holds the public width $w$ and receives three sender-generated ciphertext collections:

- the leaf ciphertexts
$$(\mathsf{ct}_{\mathsf{lac,leaf}}, \mathsf{ct}_{\mathsf{r,leaf}})$$
from the call to $\mathsf{ShrinkExpand}_{\mathcal{R}}$.Offline on the leaf input vector in $\mathsf{LogVOLE}_{\mathcal{R}}$.Offline;
- when $\ell > 1$, the internal ciphertexts $(\mathsf{ct}_{\mathsf{lac,int}}, \mathsf{ct}_{\mathsf{r,int}})$ from the call to $\mathsf{ShrinkExpand}_{\mathcal{R}}$.Offline on the internal recursion input in $\mathsf{LogVOLE}_{\mathcal{R}}$.Offline;
- the root ciphertexts $(\mathsf{ct}_{\mathsf{lac,rt}}, \mathsf{ct}_{\mathsf{r,rt}}, B^*)$ from SetupRoot.

We simulate the three LacE-ciphertexts using $\mathsf{LacE.Sim}_1$ from Section 3.6.2, and we simulate the three $\mathsf{LHE.Enc}_1$-ciphertexts using $\mathsf{LHE.Sim}_1$ from Section 3.6.4. No other offline message from $\mathbb{S}$ depends on $\mathbb{R}$'s future input.

In the online phase, the relevant calls are $(\mathsf{setx\text{-}int}, \mathsf{release\text{-}int})$ in the interactive mode and setx-ni in the non-interactive mode. The simulator receives the corrupted $\mathbb{R}$'s input $\mathbf{x}$ from the call sequence and obtains the corresponding ideal output $\mathbf{m}$ from the ideal functionality. The seed $\varsigma$ can be simulated by sampling a uniform element of $\{0,1\}^\lambda$. The non-trivial step is to program the random-oracle ciphertexts so that the receiver-side decryptions are consistent with the target output $\mathbf{m}$. Informally, the simulator works from the leaf level upward. It first encodes $\mathbf{m}$, uses the leaf LacE- and LHE-simulators to manufacture receiver-side decryption targets for the leaf blocks, programs the corresponding $H_{\mathsf{ct}}$-values, and then uses the resulting simulated key vector as the target input for the next recursion level. Repeating this step level by level eventually reaches the root. The root response is then sent directly in the interactive mode and is installed as the output of the online random-oracle query in the non-interactive mode.

**Simulator $\mathsf{Sim}_{\mathbb{R}}$.**

1. $\mathsf{Sim}_{\mathbb{R}}$ samples a uniformly random tape $R^*$ for the corrupted $\mathbb{R}$ and uses it throughout.
2. $\mathsf{Sim}_{\mathbb{R}}$ maintains two lazy tables:
   - a table $Q_H$ for queries to $H$;
   - a table $Q_{\mathsf{ct}}$ for queries to $H_{\mathsf{ct}}$.

   On a repeated query, it returns the recorded answer. On a fresh query, it records and returns a fresh uniform answer of the appropriate type. Both the corrupted $\mathbb{R}$ and the simulator may query these tables.
3. In the offline phase, $\mathsf{Sim}_{\mathbb{R}}$ simulates the three sender-generated ciphertext collections as follows.
   - It first runs the corresponding setup algorithms for all offline ciphertext families: LacE.Setup and LHE.Setup for the leaf width, LacE.Setup and LHE.Setup for the internal width when $\ell > 1$, and LacE.Setup and LHE.Setup for the root width. These public parameters are exactly the ones used by the subsequent simulator calls.
   - For the leaf ciphertexts, it runs $\mathsf{LacE.Sim}_1$ to obtain a simulated ciphertext $\widetilde{\mathsf{ct}}_{\mathsf{lac,leaf}}$ together with its simulator state $\mathsf{st}^{\mathsf{lac}}_{\mathsf{leaf}}$, and runs $\mathsf{LHE.Sim}_1$ to obtain $(\widetilde{\mathsf{ct}}_{\mathsf{r,leaf}}, \mathsf{st}^{\mathsf{lhe}}_{\mathsf{leaf}})$.
   - When $\ell > 1$, it does the same for the internal ciphertexts, obtaining $\widetilde{\mathsf{ct}}_{\mathsf{lac,int}}$, $\mathsf{st}^{\mathsf{lac}}_{\mathsf{int}}$, $\widetilde{\mathsf{ct}}_{\mathsf{r,int}}$, and $\mathsf{st}^{\mathsf{lhe}}_{\mathsf{int}}$.
   - For the root ciphertexts, it runs $\mathsf{LacE.Sim}_1$ to obtain $\widetilde{\mathsf{ct}}_{\mathsf{lac,rt}}$ with state $\mathsf{st}^{\mathsf{lac}}_{\mathsf{rt}}$, runs $\mathsf{LHE.Sim}_1$ to obtain $\widetilde{\mathsf{ct}}_{\mathsf{r,rt}}$ with state $\mathsf{st}^{\mathsf{lhe}}_{\mathsf{rt}}$, and samples $B^*$ exactly as in the real protocol.

It sends these simulated ciphertexts to the corrupted $\mathbb{R}$.

4. For a call pair $(\mathsf{setx\text{-}int}(\mathbb{R} : \mathsf{sid}, \mathbf{x}), \mathsf{release\text{-}int}(\mathbb{S} : \mathsf{sid}))$ in the interactive mode, or a call $\mathsf{setx\text{-}ni}(\mathbb{R} : \mathsf{sid}, \mathbf{x})$ in the non-interactive mode, $\mathsf{Sim}_{\mathbb{R}}$ does the following.
   (a) It invokes the corresponding ideal functionality call(s) and obtains the ideal receiver output $\mathbf{m}$ for that session.
   (b) It samples $\varsigma \xleftarrow{\$} \{0,1\}^{\lambda}$.
   (c) It starts from the leaf level. First it computes the encoded target and input:

$$\mathbf{m}' := \mathsf{Encode}(\mathbf{m}), \qquad \widetilde{\mathbf{x}} := \mathsf{Encode}(\mathbf{x}).$$

   Let $b := w/(\alpha\tau\rho)$. It then partitions both $\mathbf{m}'$ and $\widetilde{\mathbf{x}}$ into $b$ leaf blocks of size $\alpha\tau\rho$, writing these as $(\mathbf{m}'_{\ell,i})_{i\in[b]}$ and $(\widetilde{\mathbf{x}}_{\ell,i})_{i\in[b]}$. For each leaf block, it computes the corresponding laconic digest

$$d_{\ell,i} := \mathsf{LacE.Digest}(\widetilde{\mathbf{x}}_{\ell,i}),$$

   and then evaluates the simulated leaf $\mathsf{LacE}$-ciphertext on the corrupted $\mathbb{R}$'s encoded input block:

$$\mathbf{v}_{\ell,i} := \mathsf{LacE.Eval}(\widetilde{\mathsf{ct}}_{\mathsf{lac,leaf}}, \widetilde{\mathbf{x}}_{\ell,i}).$$

   It then invokes $\mathsf{LacE.Sim}_2$ on $\mathsf{st}^{\mathsf{lac}}_{\mathsf{leaf}}$ and $\widetilde{\mathbf{x}}_{\ell,i}$ to obtain the corresponding simulated leakage error vector, denoted $\mathbf{e}^{\mathsf{lac}}_{\ell,i}$, and samples an independent noise vector $\overline{\mathbf{e}}_{\ell,i} \xleftarrow{\$} \overline{\chi}^{\alpha\tau\rho}_{\mathsf{LacE}}$. The simulator defines the leaf response target

$$\mathrm{res}_{\ell,i} := \mathbf{v}_{\ell,i} - \mathbf{e}^{\mathsf{lac}}_{\ell,i} + \mathbf{m}'_{\ell,i} - \overline{\mathbf{e}}_{\ell,i}.$$

   Next it invokes

$$(\widetilde{\mathsf{ct}}_{2,\ell,i}, \mathsf{st}^{\mathsf{lhe}}_{\ell,i}) := \mathsf{LHE.Sim}_2(\mathsf{st}^{\mathsf{lhe}}_{\mathsf{leaf}})$$

   to obtain the simulated second ciphertext and its per-query state for that block. Finally, it samples an auxiliary error vector $\mathbf{e}^{\mathsf{ct}}_{\ell,i}$ with $\|\mathbf{e}^{\mathsf{ct}}_{\ell,i}\|_\infty < \delta/2 - B$ and programs

$$Q_{\mathsf{ct}}[(\mathsf{sid}, \varsigma, \ell, i)] := \widetilde{\mathsf{ct}}_{2,\ell,i} + \mathbf{e}^{\mathsf{ct}}_{\ell,i}.$$

   These programmed leaf ciphertexts ensure that the receiver-side denoising returns the desired leaf block of $\mathbf{m}'$.
   (d) It then invokes

$$\widetilde{\mathsf{sk}}_{\ell,i} := \mathsf{LHE.Sim}_3(\mathsf{st}^{\mathsf{lhe}}_{\ell,i}, \mathrm{res}_{\ell,i}, d_{\ell,i})$$

   to obtain the simulated key for leaf block $i$. Writing $\widetilde{\mathsf{sk}}_\ell := (\widetilde{\mathsf{sk}}_{\ell,i})_{i\in[b]}$, the simulator also defines the previous-level input vector

$$\mathbf{x}^{(\ell-1)} := \left(\mathsf{transpose}(((\mathbf{g}^{-1}(d_{\ell,i}) \bmod q_t)_{t\in[\rho]}))\right)_{i\in[b]},$$

   namely the vector obtained from the leaf digests by gadget decomposition, CRT reduction modulo $(q_t)_{t\in[\rho]}$, and blockwise transpose, as in $\mathsf{OnlineRec}$. The simulator applies the inverse of the caller's $\mathsf{Agg}(\mathsf{crt}((q_t)_{t\in[\rho]}, \cdot), b\tau \to b)$ map, namely the coordinate-wise decomposition, reduction modulo $(q_t)_{t\in[\rho]}$, and blockwise transpose used in $\mathsf{OnlineRec}$, to obtain the previous recursion-level target vector $\boldsymbol{m}^{(\ell-1)}$. The pair $(\mathbf{x}^{(\ell-1)}, \boldsymbol{m}^{(\ell-1)})$ becomes the input/target pair for the next recursion level. The simulator repeats the same procedure at each intermediate level $t = \ell - 1, \ldots, 2$, now using the internal $\mathsf{LacE}$-state $\mathsf{st}^{\mathsf{lac}}_{\mathsf{int}}$ and the internal $\mathsf{LHE}$-state $\mathsf{st}^{\mathsf{lhe}}_{\mathsf{int}}$.

(e) At the root, the simulator starts from the target root key $\mathsf{sk}_{\mathsf{rt}} \in \mathcal{R}_q$. It defines

$$\hat{\mathbf{m}}_{\mathsf{rt}} := \mathsf{decompose}(\mathsf{sk}_{\mathsf{rt}}) \in \mathcal{R}_q^{\tau\rho}$$

and fixes a noisy lift $\widetilde{\mathbf{m}}_{\mathsf{rt}}$ satisfying

$$\mathsf{Denoise}((q_t)_{t\in[\rho]}, \widetilde{\mathbf{m}}_{\mathsf{rt}}) = \hat{\mathbf{m}}_{\mathsf{rt}}.$$

It then computes the root LacE term

$$\Phi_{\mathsf{rt}} := \widetilde{\mathbf{C}}_{\mathsf{rt}}^{\mathsf{top}} \begin{pmatrix} \mathbf{g}^{-1}(y_{\mathsf{L}}) \\ \zeta_{\mathsf{rt}} \end{pmatrix} + \mathsf{LacE.Eval}(\widetilde{\mathsf{ct}}_{\mathsf{lac,rt}}, \hat{d}_{\mathsf{rt}}),$$

where $\hat{d}_{\mathsf{rt}} := \mathsf{decompose}(d_{\mathsf{rt}})$ and $y_{\mathsf{L}} := \mathsf{LacE.Digest}(\hat{d}_{\mathsf{rt}})$ ($\mathsf{Sim}_{\mathbb{R}}$ can compute this as it knows the encoded input vector $\tilde{\mathbf{x}}$). It then invokes $\mathsf{LacE.Sim}_2$ on $\mathsf{st}_{\mathsf{rt}}^{\mathsf{lac}}$ and $\hat{d}_{\mathsf{rt}}$ to obtain the corresponding simulated leakage error vector $\mathbf{e}_{\mathsf{rt}}^{\mathsf{lac}}$, samples an independent noise vector $\overline{\mathbf{e}}_{\mathsf{rt}} \xleftarrow{\$} \overline{\chi}_{\mathsf{LacE}}^{\tau\rho}$, and sets the target pre-subtraction decryption value to

$$\mathrm{res}_{\mathsf{rt}} := \Phi_{\mathsf{rt}} - \mathbf{e}_{\mathsf{rt}}^{\mathsf{lac}} + \widetilde{\mathbf{m}}_{\mathsf{rt}} - \overline{\mathbf{e}}_{\mathsf{rt}}.$$

Next it invokes

$$(\widetilde{\mathsf{ct}}_{2,\mathsf{rt}}, \mathsf{st}_{\mathsf{rt}}^{\mathsf{lhe}}) := \mathsf{LHE.Sim}_2(\mathsf{st}_{\mathsf{rt}}^{\mathsf{lhe}})$$

and then

$$\mathsf{sk}_{\mathsf{rt}}{}' := \mathsf{LHE.Sim}_3(\mathsf{st}_{\mathsf{rt}}^{\mathsf{lhe}}, \mathrm{res}_{\mathsf{rt}}, d'_{\mathsf{rt}}).$$

Finally it programs

$$Q_{\mathsf{ct}}[(\mathsf{sid}, \varsigma, \mathsf{rt}, 0)] := \widetilde{\mathsf{ct}}_{2,\mathsf{rt}} + \mathbf{e}_{\mathsf{rt}}^{\mathsf{ct}}$$

for an auxiliary noise vector $\mathbf{e}_{\mathsf{rt}}^{\mathsf{ct}}$ in the good interval. With this programming, $\mathsf{DerandRoot}$ returns $\widetilde{\mathbf{m}}_{\mathsf{rt}}$ before denoising and therefore outputs the prescribed root key $\mathsf{sk}_{\mathsf{rt}}$.

(f) In the interactive mode, $\mathsf{Sim}_{\mathbb{R}}$ sends $(\varsigma, \mathsf{sk}_{\mathsf{rt}}{}')$ to the corrupted $\mathbb{R}$. In the non-interactive mode, it installs $(\mathsf{sk}_{\mathsf{rt}}{}', \varsigma)$ as the output of the random-oracle query $H(\mathsf{sid}, d'_{\mathsf{rt}})$.

(g) If any $H$- or $H_{\mathsf{ct}}$-query point that $\mathsf{Sim}_{\mathbb{R}}$ needs to program has already been assigned in $Q_H$ or $Q_{\mathsf{ct}}$, then $\mathsf{Sim}_{\mathbb{R}}$ aborts.

**Hybrid argument for corrupt $\mathbb{R}$.** We now prove that, for the simulator $\mathsf{Sim}_{\mathbb{R}}$ above, $\mathsf{Game}_{\mathbb{R}}^{\mathsf{real}}$ is computationally indistinguishable from $\mathsf{Game}_{\mathbb{R}}^{\mathsf{sim}}$. For each hybrid $\mathcal{H}_i$ below, let

$$\mathcal{D}_i := (\mathsf{view}_{\mathbb{R}}, \mathsf{out}_{\mathbb{S}})$$

denote the joint distribution of the corrupted $\mathbb{R}$-view and the honest $\mathbb{S}$-output.

- $\mathcal{H}_0$: the honest $\mathbb{S}$ runs $\Pi$ on input $\Delta$, and the corrupted $\mathbb{R}$ interacts with it in the real execution. Thus
$$\mathcal{D}_0 \equiv \mathsf{Game}_{\mathbb{R}}^{\mathsf{real}}.$$
- $\mathcal{H}_1$: this hybrid is the same as $\mathcal{H}_0$, except that all answers to $H$ and $H_{\mathsf{ct}}$ are generated lazily from the tables $Q_H$ and $Q_{\mathsf{ct}}$. Since the real execution samples the same answers independently and reuses them on repeated queries,
$$\mathcal{D}_1 \equiv \mathcal{D}_0.$$

- $\mathcal{H}_2$: this hybrid is the same as $\mathcal{H}_1$, except that it programs the session-specific oracle points exactly as described by $\mathsf{Sim}_{\mathbb{R}}$, but using the actual ciphertexts. In the interactive mode, the hybrid emulates the same seed-selection rule as the real execution: rejected seeds are precisely those for which some programmed correction ciphertext produces a bad-noise event, and the first accepted seed is one for which every relevant ciphertext lies in the good interval. In the non-interactive mode, the same condition is enforced directly on the programmed seed. Since the programmed correction ciphertexts have the same ciphertext-plus-good-noise form as in the real execution, the resulting joint distribution is unchanged except for the abort event that some programmed point has already been queried. This pre-query event is negligible by the polynomial query bound and the randomness of $\varsigma$. Hence
$$\mathcal{D}_2 \approx_c \mathcal{D}_1.$$
- $\mathcal{H}_3$: this hybrid is the same as $\mathcal{H}_2$, except that at the leaf level it samples the encoded leaf target $\mathbf{m}'_\ell$ uniformly at random and programs the $H_{\mathsf{ct}}$-points with the leaf nonces $(\mathsf{sid}, \varsigma, \ell, i)$ so that receiver-side denoising returns these chosen leaf blocks. This preserves the joint distribution, because at the leaf level the clean relation is affine: choosing the sender blocks $\mathbf{k}_\ell$ uniformly or choosing the receiver blocks $\mathbf{m}'_\ell$ uniformly determines the other side by the same affine relation. Thus
$$\mathcal{D}_3 \equiv \mathcal{D}_2.$$
- $\mathcal{H}_4$: this hybrid is the same as $\mathcal{H}_3$, except that it replaces the leaf-level $\mathsf{LacE}$ (resp. $\mathsf{LHE}$) setup ciphertext by the output of $\mathsf{LacE.Sim}_1$ (resp. $\mathsf{LHE.Sim}_1$), replaces the leaf-level $\mathsf{LHE}$ correction ciphertexts by the outputs of $\mathsf{LHE.Sim}_2$, programs the leaf-level $H_{\mathsf{ct}}$-points as in $\mathsf{Sim}_{\mathbb{R}}$, and then defines the next recursion target $\boldsymbol{m}^{(\ell-1)}$ from the leaf $\mathsf{LHE.Sim}_3$ outputs exactly as in $\mathsf{Sim}_{\mathbb{R}}$. Adaptive security of $\mathsf{LacE}$ and $\mathsf{LHE}$ implies
$$\mathcal{D}_4 \approx_c \mathcal{D}_3.$$
- $\mathcal{H}_5$: for each intermediate level $t = \ell - 1, \ldots, 2$, this hybrid replaces the level-$t$ $\mathsf{LacE}$ setup ciphertext by $\mathsf{LacE.Sim}_1$, replaces the root $\mathsf{LHE}$ setup ciphertext by $\mathsf{LHE.Sim}_1$, replaces the level-$t$ $\mathsf{LHE}$ correction ciphertexts by $\mathsf{LHE.Sim}_2$, programs the corresponding $H_{\mathsf{ct}}$-points as in $\mathsf{Sim}_{\mathbb{R}}$, and uses the $\mathsf{LHE.Sim}_3$-outputs to define the next target vector. Adaptive security of $\mathsf{LacE}$ and $\mathsf{LHE}$ at each level implies
$$\mathcal{D}_5 \approx_c \mathcal{D}_4.$$
- $\mathcal{H}_6$: this hybrid is the same as $\mathcal{H}_5$, except that it replaces the root $\mathsf{LacE}$ setup ciphertext by $\mathsf{LacE.Sim}_1$, replaces the root $\mathsf{LHE}$ setup ciphertext by $\mathsf{LHE.Sim}_1$, replaces the root $\mathsf{LHE}$ correction ciphertexts by the outputs of $\mathsf{LHE.Sim}_2$, programs the root oracle points as in $\mathsf{Sim}_{\mathbb{R}}$, and obtains the root response from $\mathsf{LHE.Sim}_3$. Adaptive security of the root $\mathsf{LacE}$ and $\mathsf{LHE}$ setup objects implies
$$\mathcal{D}_6 \approx_c \mathcal{D}_5.$$
- $\mathcal{H}_7$: once the root step has been simulated, the hybrid no longer uses the sender input $\Delta$. It therefore replaces the internal target vector by the sender-output stream obtained from $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$: in the interactive mode this is the output returned at $\mathsf{setk\text{-}int}$, and in the non-interactive mode it is the output returned at $\mathsf{release\text{-}ni}$. Since the preceding hybrids already made the joint distribution depend only on this target sender-output stream and the simulated receiver view, we have
$$\mathcal{D}_7 \equiv \mathcal{D}_6.$$
By construction, $\mathcal{H}_7$ is exactly $\mathsf{Game}^{\mathsf{sim}}_{\mathbb{R}}$.

Combining the hybrid transitions gives

$$\mathsf{Game}_{\mathbb{R}}^{\mathsf{real}} \approx_c \mathsf{Game}_{\mathbb{R}}^{\mathsf{sim}},$$

which proves the corrupt-$\mathbb{R}$ case.

□

**Extension: one-digit gadget truncation.** The proof that the same strategy applies to the one-digit gadget truncation optimization (Section 7.1) is deferred to Section D.

# C Correctness of Truncation Optimization

We prove correctness of the one-digit gadget truncation from Section 7.1. The variant is specified by the truncated $\mathsf{ShrinkExpand}_{\mathcal{R}}$ protocol in Figure 14, the truncated $\mathsf{LHE}$ interface in Figure 15, and the truncated $\mathsf{LacE}$ interface in Figure 16. Let $\tau' = \tau - 1$, and fix recursive index $h$. In this variant, $\mathbb{S}$ publishes $\mathsf{ct}_1^{(h),\mathsf{hi}} \in \mathcal{R}_q^{\mu\times\tau'}$ and $\mathsf{ct}_{\mathsf{lac}}{}^{(h),\mathsf{hi}}$ using masks $\mathbf{r}^{(h)}$ sampled from the $\mathsf{LacE}$ error distribution $\overline{\mathcal{D}}$, and $\mathbb{R}$ replaces $\mathbf{g}^{-1}(\cdot)$ by $\mathbf{g}_{\mathsf{hi}}^{-1}(\cdot)$ in the gadget-heavy steps.

**Reading roadmap.** Figure 13 summarizes the reading order and prerequisites for Sections C and D. The near-collision discussion in Section D.1 first uses the $\mathsf{LacE}$ identity of Lemma 11; its lemmas then enter the composed proof in Section D.4. Sections D.2 and D.3 establish the primitive simulations separately.

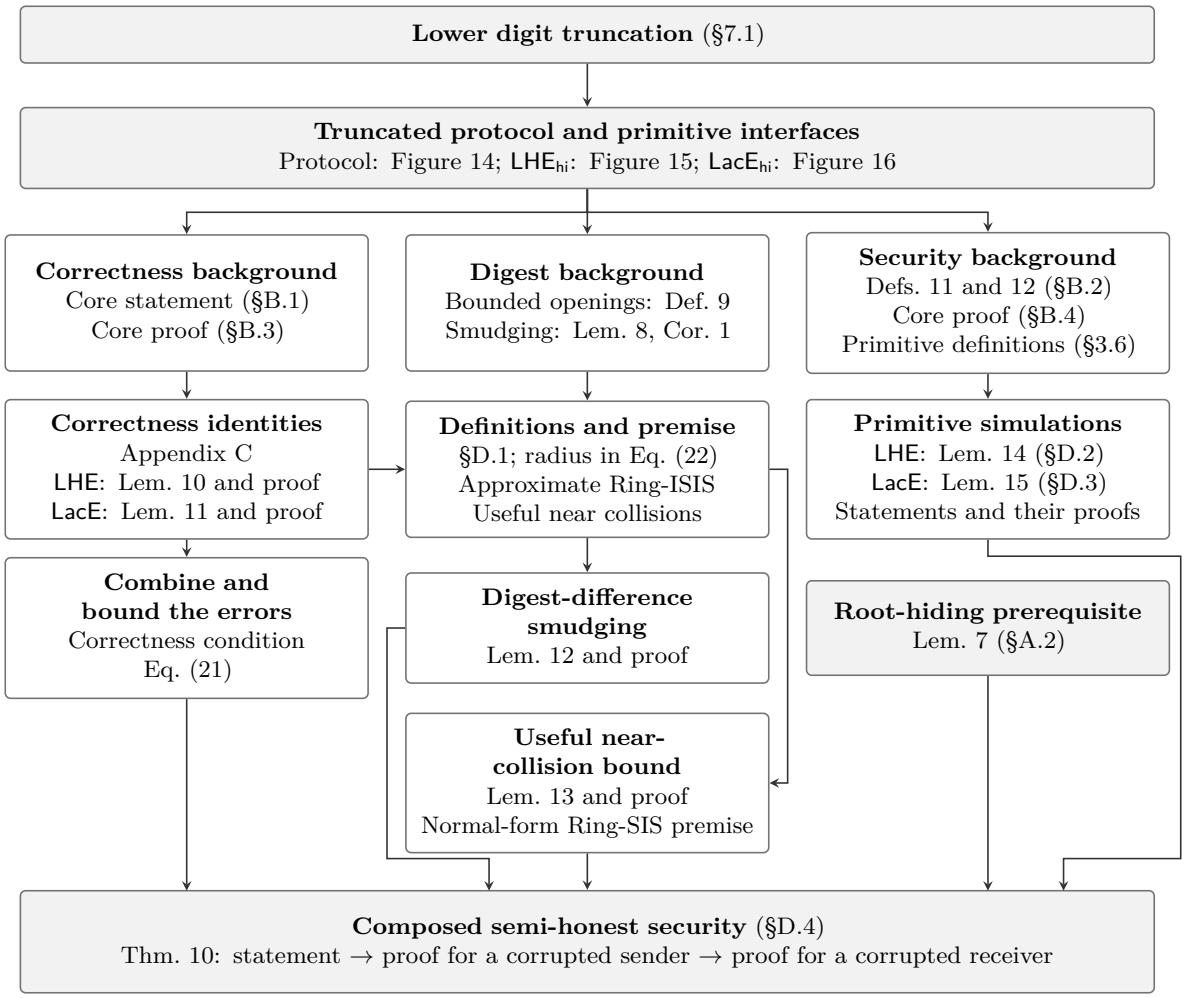


Figure 13: Reading roadmap for Appendices C and D. Arrows lead from prerequisite definitions and core proofs to the ingredients used in the composed security proof. Each cited lemma's proof follows its statement.

$\mathsf{ShrinkExpand}_{\mathsf{hi}}.\mathrm{Offline}(\mathbb{S} : \mathbf{s} \in \mathcal{R}_q^{\mu}) \to (\mathbb{S} : (r, \mathsf{sk_r}^{\mathsf{hi}}), \mathbb{R} : (\mathsf{ct_{lac}}^{\mathsf{hi}}, \mathsf{ct_r}^{\mathsf{hi}}))$

- PARAMETERS: $\alpha, \mu, g, \tau$, where $\tau = \lceil \log_g q \rceil$ and $\tau' := \tau - 1$.
- PROCEDURES:
  1. $\mathbb{S}$: $(r, \mathsf{ct_{lac}}^{\mathsf{hi}}) := \mathsf{LacE}_{\mathsf{hi}}.\mathsf{Enc}(\mathbf{s})$
  2. $\mathbb{S}$: $(\mathsf{ct_r}^{\mathsf{hi}}, \mathsf{sk_r}^{\mathsf{hi}}) := \mathsf{LHE}_{\mathsf{hi}}.\mathsf{Enc}_1(r)$
  3. $\mathbb{S}$: send $(\mathsf{ct_{lac}}^{\mathsf{hi}}, \mathsf{ct_r}^{\mathsf{hi}})$ to $\mathbb{R}$

$\mathsf{ShrinkExpand}_{\mathsf{hi}}.\mathrm{Shrink}(\mathbf{x} \in \mathcal{R}_q^{\mu}) \to d_{\mathbf{x}}^{\mathsf{hi}} \in \mathcal{R}_q$

- PROCEDURES: return $d_{\mathbf{x}}^{\mathsf{hi}} := \mathsf{LacE}_{\mathsf{hi}}.\mathsf{Digest}(\mathbf{x})$

$\mathsf{ShrinkExpand}_{\mathsf{hi}}.\mathrm{Expand}(\mathbb{S} : (r, \tilde{\mathbf{k}}' \in \mathcal{R}_q),$
$\mathbb{R} : ((\mathsf{ct_{lac}}^{\mathsf{hi}}, \mathsf{ct_r}^{\mathsf{hi}}), \mathsf{sk_x} \in \mathcal{R}_q, d_{\mathbf{x}}^{\mathsf{hi}} \in \mathcal{R}_q, \mathbf{x} \in \mathcal{R}_q^{\mu}),$
$\mathbb{B} : (\phi, \mathsf{id}, \varsigma \in \{0,1\}^{\lambda}, d_{\mathsf{rt}}, j, i)) \to (\mathbb{S} : \mathbf{k} \in \mathcal{R}_q^{\mu}, \mathbb{R} : \mathbf{m} \in \mathcal{R}_q^{\mu})$

- ASSUMPTION: $\mathsf{sk_x} = (\mathsf{sk_r}^{\mathsf{hi}})^T \cdot \mathbf{g}_{\mathsf{hi}}^{-1}(d_{\mathbf{x}}^{\mathsf{hi}}) + \tilde{\mathbf{k}}'$.
- PROCEDURES:
  1. $\mathbb{B}$: $\mathsf{ct_k} := H_{\mathsf{ct}}(\mathsf{id}, \varsigma, j, i, d_{\mathsf{rt}})$
  2. $\mathbb{R}$: $\mathsf{ct_{res}} := \mathsf{ct_r}^{\mathsf{hi}} \cdot \mathbf{g}_{\mathsf{hi}}^{-1}(d_{\mathbf{x}}^{\mathsf{hi}}) + \mathsf{ct_k}$
  3. $\mathbb{R}$: $\tilde{\mathbf{m}} := \mathsf{LHE}_{\mathsf{hi}}.\mathsf{Dec}(\mathsf{sk_x}, \mathsf{ct_{res}}) - \mathsf{LacE}_{\mathsf{hi}}.\mathsf{Eval}(\mathsf{ct_{lac}}^{\mathsf{hi}}, \mathbf{x})$
  4. $\mathbb{S}$: $\tilde{\mathbf{k}} := \mathsf{LHE}_{\mathsf{hi}}.\mathsf{Dec}(\tilde{\mathbf{k}}', \mathsf{ct_k})$
  5. return $(\mathbb{R} : \mathbf{m} := \mathsf{Denoise}(\phi, \tilde{\mathbf{m}}), \mathbb{S} : \mathbf{k} := \mathsf{Denoise}(\phi, \tilde{\mathbf{k}}))$

Figure 14: One-digit gadget-truncated variant of $\mathsf{ShrinkExpand}_{\mathcal{R}}$.

$\mathsf{LHE}_{\mathsf{hi}}.\mathsf{Setup}(1^{\lambda}, 1^w) \to \mathbf{a} \in \mathcal{R}_q^w$

- PARAMETERS: $\tau := \lceil \log_g q \rceil$, $\tau' := \tau - 1$, $\mathbf{g}_{\mathsf{hi}} := (g, g^2, \ldots, g^{\tau-1})$.
- PROCEDURES:
  1. $\mathbf{a} \leftarrow \mathcal{R}_q^w$
  2. return $\mathbf{a}$

$\mathsf{LHE}_{\mathsf{hi}}.\mathsf{Enc}_1(\boldsymbol{m}_1 \in \mathcal{R}_q^w) \to (\mathsf{ct}_1 \in \mathcal{R}_q^{w \times \tau'}, \mathsf{st}_1 \in \mathcal{R}_q^{\tau'})$

- PROCEDURES:
  1. $\mathsf{st}_1 := \mathbf{s}_1 \leftarrow \mathcal{R}_q^{\tau'}$ and $\mathbf{E}_1 \leftarrow \overline{\mathcal{D}}^{w \times \tau'}$
  2. $\mathsf{ct}_1 := \mathbf{a} \cdot \mathbf{s}_1^T + \boldsymbol{m}_1 \cdot \mathbf{g}_{\mathsf{hi}}^T + \mathbf{E}_1$
  3. return $(\mathsf{ct}_1, \mathsf{st}_1)$

$\mathsf{LHE}_{\mathsf{hi}}.\mathsf{Enc}_2(\boldsymbol{m}_2 \in \mathcal{R}_q^w) \to (\mathsf{ct}_2 \in \mathcal{R}_q^w, \mathsf{st}_2 \in \mathcal{R}_q)$

- PROCEDURES:
  1. $\mathsf{st}_2 := s_2 \leftarrow \mathcal{R}_q$ and $\overline{\mathbf{E}}_2 \leftarrow \overline{\mathcal{D}}^w$
  2. $\mathsf{ct}_2 := \mathbf{a} \cdot s_2 + \boldsymbol{m}_2 + \overline{\mathbf{E}}_2$
  3. return $(\mathsf{ct}_2, \mathsf{st}_2)$

$\mathsf{LHE}_{\mathsf{hi}}.\mathsf{KeyGen}(\mathsf{st}_1 \in \mathcal{R}_q^{\tau'}, \mathsf{st}_2 \in \mathcal{R}_q,\ y \in \mathcal{R}_q) \to \mathsf{sk}_y \in \mathcal{R}_q$

- PROCEDURES:
  1. Parse $\mathsf{st}_1 = \mathbf{s}_1$ and $\mathsf{st}_2 = s_2$
  2. return $\mathsf{sk}_y := \mathbf{s}_1^T \cdot \mathbf{g}_{\mathsf{hi}}^{-1}(y) + s_2$

$\mathsf{LHE}_{\mathsf{hi}}.\mathsf{Dec}(\mathsf{sk}_y \in \mathcal{R}_q,\ \mathsf{ct}_1 \in \mathcal{R}_q^{w \times \tau'},\ \mathsf{ct}_2 \in \mathcal{R}_q^w,\ y \in \mathcal{R}_q) \to \boldsymbol{m}_{\mathsf{res}} \in \mathcal{R}_q^w$

- PROCEDURES:
  1. $\mathsf{ct_{res}} := \mathsf{ct}_1 \cdot \mathbf{g}_{\mathsf{hi}}^{-1}(y) + \mathsf{ct}_2$
  2. return $\boldsymbol{m}_{\mathsf{res}} := \mathsf{ct_{res}} - \mathbf{a} \cdot \mathsf{sk}_y$

Figure 15: One-digit gadget-truncated LHE, denoted $\mathsf{LHE}_{\mathsf{hi}}$.

**Proof strategy.** The full-gadget correctness proof has two canceling products: the evaluated LHE ciphertext contains $\mathbf{r}^{(h)} \cdot d$, while the LacE evaluation contains the same term with the opposite sign.

$\underline{\mathsf{LacE_{hi}.Setup}(1^\lambda, 1^w) \to (\mathbf{b}_0, \mathbf{b}_1) \in \mathcal{R}_q^\tau \times \mathcal{R}_q^\tau}$

- PARAMETERS: $\tau := \lceil \log_g q \rceil$, $\tau' := \tau - 1$, $\tau_{\mathsf{leaf}} := \lceil \log_g p \rceil$, $\mathbf{g}_{\mathsf{hi}} := (g, g^2, \ldots, g^{\tau-1})$, $\mathbf{g}_{\mathsf{leaf}} := (1, g, \ldots, g^{\tau_{\mathsf{leaf}}-1})$.
- PROCEDURES:
  1. $\mathbf{b}_0, \mathbf{b}_1 \leftarrow \mathcal{R}_q^\tau$
  2. For $b \in \{0,1\}$: $\mathbf{b}_b^{\mathsf{hi}} := (\mathbf{b}_b[2], \ldots, \mathbf{b}_b[\tau])$ and $\mathbf{b}_b^{\mathsf{leaf}} := (\mathbf{b}_b[1], \ldots, \mathbf{b}_b[\tau_{\mathsf{leaf}}])$
  3. return $(\mathbf{b}_0, \mathbf{b}_1)$

$\underline{\mathsf{LacE_{hi}.Enc}(\mathbf{s} \in \mathcal{R}_q^w) \to (\mathbf{r} \in \mathcal{R}_q^w, \mathsf{ct_{lac}}^{\mathsf{hi}})}$

- PROCEDURES:
  1. $\ell := \log_2 w$
  2. $\mathbf{r}^{(0)} := \mathbf{r} \leftarrow \overline{\mathcal{D}}^w$, $\mathbf{r}^{(\ell)} := \mathbf{s}$, and $\mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(\ell-1)} \leftarrow \overline{\mathcal{D}}^w$
  3. For $i \in [0, \ell-2]$:
     (a) $\mathbf{E}_i \leftarrow \overline{\mathcal{D}}^{w \times 2\tau'}$
     (b) For $\mathsf{ind} \in \{0,1\}^\ell$:
        i. $\mathbf{C}_i[\mathsf{ind}] := \mathbf{r}^{(i)}[\mathsf{ind}] \cdot ((\mathbf{b}_0^{\mathsf{hi}})^T \ (\mathbf{b}_1^{\mathsf{hi}})^T) + \mathbf{r}^{(i+1)}[\mathsf{ind}] \cdot (\overline{\mathsf{ind}}_i \cdot \mathbf{g}_{\mathsf{hi}}^T \ \mathsf{ind}_i \cdot \mathbf{g}_{\mathsf{hi}}^T) + \mathbf{E}_i[\mathsf{ind}]$
  4. $\mathbf{E}_{\ell-1} \leftarrow \overline{\mathcal{D}}^{w \times 2\tau_{\mathsf{leaf}}}$
  5. For $\mathsf{ind} \in \{0,1\}^\ell$:
     (a) $\mathbf{C}_{\ell-1}[\mathsf{ind}] := \mathbf{r}^{(\ell-1)}[\mathsf{ind}] \cdot ((\mathbf{b}_0^{\mathsf{leaf}})^T \ (\mathbf{b}_1^{\mathsf{leaf}})^T) + \mathbf{s}[\mathsf{ind}] \cdot (\overline{\mathsf{ind}}_{\ell-1} \cdot \mathbf{g}_{\mathsf{leaf}}^T \ \mathsf{ind}_{\ell-1} \cdot \mathbf{g}_{\mathsf{leaf}}^T) + \mathbf{E}_{\ell-1}[\mathsf{ind}]$
  6. return $(\mathbf{r}, \mathsf{ct_{lac}}^{\mathsf{hi}} := (\mathbf{C}_0, \ldots, \mathbf{C}_{\ell-1}))$

$\underline{\mathsf{LacE_{hi}.Digest}(\mathbf{x} \in \mathcal{R}_q^w) \to d_{\mathbf{x}}^{\mathsf{hi}} \in \mathcal{R}_q}$

- PROCEDURES:
  1. $\ell := \log_2 w$ and $y_{\mathsf{ind}} := \mathbf{x}[\mathsf{ind}]$ for all $\mathsf{ind} \in \{0,1\}^\ell$
  2. For $\mathsf{ind} \in \{0,1\}^\ell$: $\mathbf{z}[\mathsf{ind}] := \mathbf{g}_{\mathsf{leaf}}^{-1}(y_{\mathsf{ind}})$
  3. For $\mathsf{pre} \in \{0,1\}^{\ell-1}$: $y_{\mathsf{pre}} := ((\mathbf{b}_0^{\mathsf{leaf}})^T \ (\mathbf{b}_1^{\mathsf{leaf}})^T)\binom{-\mathbf{z}[\mathsf{pre}\|0]}{-\mathbf{z}[\mathsf{pre}\|1]}$
  4. For $t = \ell-2, \ldots, 0$:
     (a) For $\mathsf{pre} \in \{0,1\}^t$: $y_{\mathsf{pre}} := ((\mathbf{b}_0^{\mathsf{hi}})^T \ (\mathbf{b}_1^{\mathsf{hi}})^T)\binom{-\mathbf{g}_{\mathsf{hi}}^{-1}(y_{\mathsf{pre}\|0})}{-\mathbf{g}_{\mathsf{hi}}^{-1}(y_{\mathsf{pre}\|1})}$
  5. return $d_{\mathbf{x}}^{\mathsf{hi}} := y_\epsilon$

$\underline{\mathsf{LacE_{hi}.Eval}(\mathsf{ct_{lac}}^{\mathsf{hi}}, \mathbf{x} \in \mathcal{R}_q^w) \to \boldsymbol{\phi}_{\mathsf{hi}} \in \mathcal{R}_q^w}$

- PROCEDURES:
  1. $\ell := \log_2 w$ and parse $\mathsf{ct_{lac}}^{\mathsf{hi}} := (\mathbf{C}_0, \ldots, \mathbf{C}_{\ell-1})$
  2. Compute $\{\mathbf{z}[\mathsf{ind}]\}_{\mathsf{ind} \in \{0,1\}^\ell}$ and $\{y_{\mathsf{pre}}\}_{\mathsf{pre} \in \{0,1\}^{\le \ell}}$ as in $\mathsf{LacE_{hi}.Digest}(\mathbf{x})$
  3. For $\mathsf{ind} \in \{0,1\}^\ell$: $\boldsymbol{\phi}_{\mathsf{hi}}[\mathsf{ind}] := \sum_{i=0}^{\ell-2} \mathbf{C}_i[\mathsf{ind}] \cdot \binom{-\mathbf{g}_{\mathsf{hi}}^{-1}(y_{\mathsf{ind}_{:i}\|0})}{-\mathbf{g}_{\mathsf{hi}}^{-1}(y_{\mathsf{ind}_{:i}\|1})} + \mathbf{C}_{\ell-1}[\mathsf{ind}] \cdot \binom{-\mathbf{z}[\mathsf{ind}_{:\ell-1}\|0]}{-\mathbf{z}[\mathsf{ind}_{:\ell-1}\|1]}$
  4. return $\boldsymbol{\phi}_{\mathsf{hi}}$

Figure 16: One-digit gadget-truncated $\mathsf{LacE}$, denoted $\mathsf{LacE_{hi}}$.

Truncation replaces each internal gadget reconstruction by $\mathsf{high}_g(d) = d - \mathsf{low}_g(d)$. We isolate the new low-digit errors in two identities, one for the evaluated $\mathsf{LHE}$ ciphertext and one for the $\mathsf{LacE}$ path telescope, and then add their norms to the usual rounding margin as in $\mathsf{LHE}$ and $\mathsf{LacE}$ noise removal.

We first recall the receiver's truncated expand ciphertext for a fixed chunk $j \in [w^{(h)'}]$. This gives the $\mathsf{LHE}$ identity that will be combined with the $\mathsf{LacE}$ path identity below.

$$\mathsf{ct}_{\mathsf{res}}^{(h,j),\mathsf{hi}} := \mathsf{ct}_1^{(h),\mathsf{hi}} \cdot \mathbf{g}_{\mathsf{hi}}^{-1}(\mathbf{d}^{(h-1)}[j]) + \mathsf{ct}_2^{(h,j)}. \tag{15}$$

The corresponding truncated key is

$$\mathsf{sk}_{\mathbf{x}}^{(h),\mathsf{hi}}[j] := \mathsf{sk}_1^{(h),\mathsf{hi}} \cdot \mathbf{g}_{\mathsf{hi}}^{-1}(\mathbf{d}^{(h-1)}[j]) + \tilde{\mathbf{k}}^{(h)}[j],$$

as required by Figure 14. Thus the evaluated $\mathsf{LHE}$ ciphertext decrypts to the high-digit product

$$\mathsf{LHE.Dec}(\mathsf{sk}_{\mathbf{x}}^{(h),\mathsf{hi}}[j], \mathsf{ct}_{\mathsf{res}}^{(h,j),\mathsf{hi}}) = \mathbf{r}^{(h)} \cdot \mathsf{high}_g(\mathbf{d}^{(h-1)}[j]) + \tilde{\mathbf{k}}^{(h-1,j)} + \overline{\boldsymbol{e}} + \boldsymbol{e}_{\mathsf{LHE}}^{(h,j)}, \tag{16}$$

where $\boldsymbol{e}_{\mathsf{LHE}}^{(h,j)}$ is the usual $\mathsf{LHE}$ decryption noise.

The next lemma shows that the $\mathsf{LHE}$ error caused by omitting the low digit is bounded by $ng \max \overline{\mathcal{D}}$. This is the first new term in the rounding condition.

**Lemma 10** (Truncated $\mathsf{LHE}$ product)**.** *Fix recursive index $h$, row $t \in [\mu]$, and digest $d \in \mathcal{R}_q$. Write*

$$\mathsf{ct}_1^{(h),\mathsf{hi}} = \mathbf{a} \cdot (\mathsf{sk}_1^{(h),\mathsf{hi}})^\top + \mathbf{r}^{(h)} \cdot \mathbf{g}_{\mathsf{hi}}^\top + \mathbf{E}_1.$$

*Then*

$$\mathsf{ct}_1^{(h),\mathsf{hi}}[t] \cdot \mathbf{g}_{\mathsf{hi}}^{-1}(d) = \mathbf{a}[t] \cdot ((\mathsf{sk}_1^{(h),\mathsf{hi}})^\top \mathbf{g}_{\mathsf{hi}}^{-1}(d)) + \mathbf{r}^{(h)}[t] \cdot d - e_{\mathsf{LHE},\mathsf{drop}}(t) + e_{\mathsf{LHE}}(t),$$

*where*

$$e_{\mathsf{LHE},\mathsf{drop}}(t) := \mathbf{r}^{(h)}[t] \cdot \mathsf{low}_g(d), \qquad e_{\mathsf{LHE}}(t) := \mathbf{E}_1[t] \cdot \mathbf{g}_{\mathsf{hi}}^{-1}(d).$$

*Moreover,*

$$\|e_{\mathsf{LHE},\mathsf{drop}}(t)\|_\infty \le ng \max \overline{\mathcal{D}},$$

*and*

$$\|e_{\mathsf{LHE}}(t)\|_\infty \le \tau' ng \max_{u \in [\tau']} \|\mathbf{E}_1[t][u]\|_\infty .$$

*Proof.* The only algebraic change is that the truncated gadget reconstructs the high-digit part:

$$\mathbf{g}_{\mathsf{hi}}^\top \mathbf{g}_{\mathsf{hi}}^{-1}(d) = \mathsf{high}_g(d) = d - \mathsf{low}_g(d).$$

Substituting this identity into the row product gives

$$\mathsf{ct}_1^{(h),\mathsf{hi}}[t] \cdot \mathbf{g}_{\mathsf{hi}}^{-1}(d) = \mathbf{a}[t] \cdot ((\mathsf{sk}_1^{(h),\mathsf{hi}})^\top \mathbf{g}_{\mathsf{hi}}^{-1}(d)) + \mathbf{r}^{(h)}[t] \cdot (d - \mathsf{low}_g(d)) + \mathbf{E}_1[t] \cdot \mathbf{g}_{\mathsf{hi}}^{-1}(d).$$

The stated bounds follow from

$$\|uv\|_\infty \le n \,\|\, u \,\|_\infty \|v\|_\infty,$$

using $\|\mathsf{low}_g(d)\|_\infty \le g$ and $\|\mathbf{g}_{\mathsf{hi}}^{-1}(d)[u]\|_\infty \le g$ for all $u \in [\tau']$. □

The next lemma shows that the $\mathsf{LacE}$ error caused by omitting low digits along the digest path is bounded by $ng(\max \overline{\mathcal{D}}) \log_2 \mu$. This is the second new term in the rounding condition.

**Lemma 11** (Truncated $\mathsf{LacE}$ path identity)**.** *Fix recursive index $h$ and chunk $j$. Set*

$$\mathbf{x} := \tilde{\mathbf{x}}_j^{(h-1)}, \qquad \mathsf{ct}_{\mathsf{lac}}{}^{\mathsf{hi}} := \mathsf{ct}_{\mathsf{lac}}{}^{(h),\mathsf{hi}}, \qquad d := \mathbf{d}^{(h-1)}[j] = \mathsf{LacE}_{\mathsf{hi}}.\mathsf{Digest}(\mathbf{x}),$$

*and let $\ell := \log_2 \mu$. For every leaf $\mathsf{ind} \in \{0,1\}^\ell$,*

$$\mathsf{LacE}_{\mathsf{hi}}.\mathsf{Eval}(\mathsf{ct}_{\mathsf{lac}}{}^{\mathsf{hi}}, \mathbf{x})[\mathsf{ind}] = \mathbf{r}^{(h)}[\mathsf{ind}] \cdot d - \mathbf{s}^{(h-1)}[\mathsf{ind}] \cdot \mathbf{x}[\mathsf{ind}] + e_{\mathsf{drop}}(\mathsf{ind}) + e_{\mathsf{LacE}}(\mathsf{ind}), \tag{17}$$

*where $e_{\mathsf{LacE}}(\mathsf{ind})$ is the usual $\mathsf{LacE}$ evaluation noise and*

$$e_{\mathsf{drop}}(\mathsf{ind}) := \sum_{i=0}^{\ell-2} \mathbf{r}^{(i+1)}[\mathsf{ind}] \cdot \mathsf{low}_g(y_{\mathsf{ind}_{:i} \| \mathsf{ind}_i}).$$

*The truncation term satisfies*

$$\|e_{\mathsf{drop}}(\mathsf{ind})\|_\infty \le ng(\max \overline{\mathcal{D}}) \log_2 \mu.$$

*Proof.* Let $y_{\mathsf{pre}}$ be the digest tree computed from $\mathbf{x}$ as in Figure 16; hence $y_\epsilon = d$ and $y_{\mathsf{ind}} = \mathbf{x}[\mathsf{ind}]$ at the leaves. The encryption samples intermediate masks $\mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(\ell-1)} \in \mathcal{R}_q^\mu$ from $\overline{\mathcal{D}}$, with

$$\mathbf{r}^{(0)} := \mathbf{r}^{(h)} \qquad \text{and} \qquad \mathbf{r}^{(\ell)} := \mathbf{s}^{(h-1)}.$$

For digest-tree depth $i \in \{0, \ldots, \ell-2\}$, define

$$\mathbf{z}_i(\mathsf{ind}) := \begin{pmatrix} -\mathbf{g}_{\mathsf{hi}}^{-1}(y_{\mathsf{ind}_{:i}\|0}) \\ -\mathbf{g}_{\mathsf{hi}}^{-1}(y_{\mathsf{ind}_{:i}\|1}) \end{pmatrix}.$$

The row $C := \mathbf{C}_i[\mathsf{ind}]$ splits as $C = C^\circ + \mathbf{E}_i[\mathsf{ind}]$, where

$$C^\circ := \mathbf{r}^{(i)}[\mathsf{ind}] \cdot ((\mathbf{b}_0^{\mathsf{hi}})^\top \, (\mathbf{b}_1^{\mathsf{hi}})^\top) + \mathbf{r}^{(i+1)}[\mathsf{ind}] \cdot (\overline{\mathsf{ind}}_i \cdot \mathbf{g}_{\mathsf{hi}}^\top \, \mathsf{ind}_i \cdot \mathbf{g}_{\mathsf{hi}}^\top).$$

The digest recursion gives

$$((\mathbf{b}_0^{\mathsf{hi}})^\top \, (\mathbf{b}_1^{\mathsf{hi}})^\top)\mathbf{z}_i(\mathsf{ind}) = y_{\mathsf{ind}_{:i}}.$$

The selected gadget block gives

$$(\overline{\mathsf{ind}}_i \mathbf{g}_{\mathsf{hi}}^\top \, \mathsf{ind}_i \mathbf{g}_{\mathsf{hi}}^\top)\mathbf{z}_i(\mathsf{ind}) = -y_{\mathsf{ind}_{:i}\|\mathsf{ind}_i} + \mathsf{low}_g(y_{\mathsf{ind}_{:i}\|\mathsf{ind}_i}).$$

Therefore

$$\begin{aligned} \mathbf{C}_i[\mathsf{ind}] \cdot \mathbf{z}_i(\mathsf{ind}) = \mathbf{r}^{(i)}[\mathsf{ind}] \cdot y_{\mathsf{ind}_{:i}} - \mathbf{r}^{(i+1)}[\mathsf{ind}] \cdot y_{\mathsf{ind}_{:i}\|\mathsf{ind}_i} \\ + \mathbf{r}^{(i+1)}[\mathsf{ind}] \cdot \mathsf{low}_g(y_{\mathsf{ind}_{:i}\|\mathsf{ind}_i}) + \mathbf{E}_i[\mathsf{ind}] \cdot \mathbf{z}_i(\mathsf{ind}). \end{aligned} \tag{18}$$

Summing (18) over $i = 0, \ldots, \ell-2$ telescopes the first two terms:

$$\sum_{i=0}^{\ell-2} \mathbf{C}_i[\mathsf{ind}] \cdot \mathbf{z}_i(\mathsf{ind}) = \mathbf{r}^{(h)}[\mathsf{ind}] \cdot d - \mathbf{r}^{(\ell-1)}[\mathsf{ind}] \cdot y_{\mathsf{ind}_{:\ell-1}} + e_{\mathsf{drop}}(\mathsf{ind}) + \sum_{i=0}^{\ell-2} \mathbf{E}_i[\mathsf{ind}] \cdot \mathbf{z}_i(\mathsf{ind}). \tag{19}$$

The leaf-adjacent layer is not truncated by $\mathsf{low}_g$; it uses $\tau_{\mathsf{leaf}} = \lceil \log_g p \rceil$ and $\mathbf{g}_{\mathsf{leaf}} = (1, g, \ldots, g^{\tau_{\mathsf{leaf}}-1})$. With $\mathbf{z}[\mathsf{ind}] := \mathbf{g}_{\mathsf{leaf}}^{-1}(\mathbf{x}[\mathsf{ind}])$, define

$$\mathbf{z}_{\ell-1}(\mathsf{ind}) := \begin{pmatrix} -\mathbf{z}[\mathsf{ind}_{:\ell-1}\|0] \\ -\mathbf{z}[\mathsf{ind}_{:\ell-1}\|1] \end{pmatrix}.$$

For the leaf row,

$$\mathbf{C}_{\ell-1}[\mathsf{ind}] \cdot \mathbf{z}_{\ell-1}(\mathsf{ind}) = \mathbf{r}^{(\ell-1)}[\mathsf{ind}] \cdot y_{\mathsf{ind}_{:\ell-1}} - \mathbf{s}^{(h-1)}[\mathsf{ind}] \cdot \mathbf{x}[\mathsf{ind}] + \mathbf{E}_{\ell-1}[\mathsf{ind}] \cdot \mathbf{z}_{\ell-1}(\mathsf{ind}). \tag{20}$$

Adding (19) and (20) gives (17), with $e_{\mathsf{LacE}}(\mathsf{ind})$ collecting the Gaussian row-noise terms.

Finally, $e_{\mathsf{drop}}(\mathsf{ind})$ has at most $\ell - 1$ summands. Since each mask is sampled from $\overline{\mathcal{D}}$, and since $\|\mathsf{low}_g(\cdot)\|_\infty \le g$,

$$\|e_{\mathsf{drop}}(\mathsf{ind})\|_\infty \le \sum_{i=0}^{\ell-2} \|\mathbf{r}^{(i+1)}[\mathsf{ind}] \cdot \mathsf{low}_g(y_{\mathsf{ind}_{:i}\|\mathsf{ind}_i})\|_\infty \le (\ell-1)ng \max \overline{\mathcal{D}} \le ng(\max \overline{\mathcal{D}}) \log_2 \mu.$$

□

**Combining the identities.** For chunk $j$, the receiver subtracts the $\mathsf{LacE}$ evaluation from the evaluated $\mathsf{LHE}$ ciphertext. Applying (16) and Lemma 11 element-wise gives the following identity. Write $d_j := \mathbf{d}^{(h-1)}[j]$ and $\mathbf{x}_j := \tilde{\mathbf{x}}_j^{(h-1)}$. Then

$$\begin{aligned}\tilde{\mathbf{m}}^{(h-1,j),\mathsf{hi}} &:= \mathsf{LHE.Dec}\big(\mathsf{sk}_{\mathbf{x}}^{(h),\mathsf{hi}}[j], \mathsf{ct}_{\mathsf{res}}^{(h,j),\mathsf{hi}}\big) - \mathsf{LacE}_{\mathsf{hi}}.\mathsf{Eval}(\mathsf{ct}_{\mathsf{lac}}{}^{(h),\mathsf{hi}}, \mathbf{x}_j) \\ &= \tilde{\mathbf{k}}^{(h-1,j)} + \mathbf{x}_j \cdot \mathbf{s}^{(h-1)} + \overline{\boldsymbol{e}} + \boldsymbol{e}_{\mathsf{LHE}}^{(h,j)} - \boldsymbol{e}_{\mathsf{LacE}}^{(h,j)} - \mathbf{r}^{(h)} \cdot \mathsf{low}_g(d_j) - \boldsymbol{e}_{\mathsf{drop}}^{(h,j)}.\end{aligned}$$

The terms $\boldsymbol{e}_{\mathsf{LHE}}^{(h,j)}$, $\boldsymbol{e}_{\mathsf{LacE}}^{(h,j)}$, and $\overline{\boldsymbol{e}}$ are the existing errors. The final two terms are new: one $\mathsf{LHE}$ low-digit product and one $\mathsf{LacE}$ path sum. By Lemma 10 and Lemma 11,

$$\|\mathbf{r}^{(h)} \cdot \mathsf{low}_g(d_j)\|_\infty \le ng \max \overline{\mathcal{D}},$$

and

$$\|\boldsymbol{e}_{\mathsf{drop}}^{(h,j)}\|_\infty \le ng(\max \overline{\mathcal{D}}) \log_2 \mu.$$

Thus each output entry differs from the clean value by at most

$$B_{\mathsf{LacE}} + B_{\mathsf{LHE}} + \max \overline{\chi}_{\mathsf{LacE}} + ng(\max \overline{\mathcal{D}})(1 + \log_2 \mu).$$

Correctness follows whenever every rounding gap satisfies, for $j \in [\rho]$,

$$\delta_j > 2\big(B_{\mathsf{LacE}} + B_{\mathsf{LHE}} + \max \overline{\chi}_{\mathsf{LacE}} + ng(\max \overline{\mathcal{D}})(1 + \log_2 \mu)\big). \tag{21}$$

# D Security Analysis for Truncation Optimization

We write $\Pi_{\mathsf{hi}}$ for the protocol obtained by replacing $\mathsf{ShrinkExpand}_{\mathcal{R}}$, $\mathsf{LacE}$, and $\mathsf{LHE}$ with the truncated variants in Figures 14 to 16. Let $\tau' = \tau - 1$. The proof below follows the same order as the $\mathsf{TinyLabels}$ proof for laconic encryption in Section B.3: first isolate the primitive-level hybrids that replace rows by Ring-LWE challenges, and then plug those hybrids into the recursive semi-honest simulator.

The reading order and prerequisites for this appendix and the correctness proofs in Section C appear in Figure 13.

## D.1 Near-collision binding

**Node-digest notation.** The abstract row $A$ in the near-collision premise is the public row that implements one local node digest in Section 6. In the baseline notation there, setup samples $(\mathbf{b}_1, \mathbf{b}_2) \in \mathcal{R}_q^\tau \times \mathcal{R}_q^\tau$, writes

$$B := (B_1\ B_2) := (\mathbf{b}_1^\top\ \mathbf{b}_2^\top),$$

and defines

$$\mathsf{NodeDigest}_B(u_L, u_R) := (B_1\ B_2)\begin{pmatrix}\mathbf{g}^{-1}(u_L) \\ \mathbf{g}^{-1}(u_R)\end{pmatrix}.$$

Thus $B \in \mathcal{R}_q^{1\times 2\tau}$. For a single ordered pair of child values, define the local opening vector

$$w_B(u_L, u_R) := \begin{pmatrix}\mathbf{g}^{-1}(u_L) \\ \mathbf{g}^{-1}(u_R)\end{pmatrix},$$

so that

$$d = \mathsf{NodeDigest}_B(u_L, u_R) = B\, w_B(u_L, u_R).$$

In the generic notation below this is the case $A := B$, $m = 2\tau$, and $w := w_B(u_L, u_R)$. The minus signs used by the algorithmic conventions in Figures 3 and 16 are absorbed into the opening vector, not into the public row $A$. Accordingly, for the truncated internal nodes of $\mathsf{LacE}_{\mathsf{hi}}$, the same notation uses

$$A := ((\mathbf{b}_0^{\mathsf{hi}})^T\ (\mathbf{b}_1^{\mathsf{hi}})^T), \qquad m := 2\tau', \qquad w(u_L, u_R) := \begin{pmatrix} -\mathbf{g}_{\mathsf{hi}}^{-1}(u_L) \\ -\mathbf{g}_{\mathsf{hi}}^{-1}(u_R) \end{pmatrix}.$$

The leaf layer uses the analogous full leaf-gadget row

$$A := ((\mathbf{b}_0^{\mathsf{leaf}})^T\ (\mathbf{b}_1^{\mathsf{leaf}})^T), \qquad m := 2\tau_{\mathsf{leaf}}, \qquad w(u_L, u_R) := \begin{pmatrix} -\mathbf{g}_{\mathsf{leaf}}^{-1}(u_L) \\ -\mathbf{g}_{\mathsf{leaf}}^{-1}(u_R) \end{pmatrix}.$$

The near-collision premise is stated once for this generic local digest row $A$, and the tree proof instantiates it for each local digest occurrence.

**Security concerns about the $d \cdot r$ term.** For a local LacE block with digest $d = d_{\mathbf{x}} = Aw$, the telescoping identity in Section 6 gives, in the full-gadget case,

$$\mathsf{LacE.Eval}(\mathsf{ct}_{\mathsf{lac}}, \mathbf{x})[\mathsf{ind}] \approx \mathbf{r}[\mathsf{ind}] \cdot d - \mathbf{s}[\mathsf{ind}] \cdot \mathbf{x}[\mathsf{ind}].$$

The truncated version has the same leading term. More precisely, by Equation (17),

$$\mathsf{LacE}_{\mathsf{hi}}.\mathsf{Eval}({\mathsf{ct}_{\mathsf{lac}}}^{\mathsf{hi}}, \mathbf{x})[\mathsf{ind}] = \mathbf{r}[\mathsf{ind}] \cdot d - \mathbf{s}[\mathsf{ind}] \cdot \mathbf{x}[\mathsf{ind}] + e_{\mathsf{drop}}(\mathsf{ind}) + e_{\mathsf{LacE}}(\mathsf{ind}),$$

where $e_{\mathsf{drop}}$ is the known dropped-digit contribution and $e_{\mathsf{LacE}}$ is the usual LacE evaluation noise. Thus the digest-indexed correction material for digest $d$ is designed to cancel the top term $\mathbf{r}[\mathsf{ind}] \cdot d$.

Now suppose the receiver keeps the correction/key material associated with $d = Aw$, but evaluates a second bounded opening $w'$ with digest $d' = Aw'$. Suppressing the already-budgeted truncation and Gaussian errors, let $\mathbf{x}$ and $\mathbf{x}'$ denote the leaf values induced by $w$ and $w'$, respectively. The two secret-dependent message terms are, explicitly, $\mathbf{s}[\mathsf{ind}] \cdot \mathbf{x}[\mathsf{ind}]$ for the intended leaf and $\mathbf{s}[\mathsf{ind}] \cdot \mathbf{x}'[\mathsf{ind}]$ for the substituted leaf. If $\tilde{\mathbf{k}}_d[\mathsf{ind}]$ denotes the fixed local sender share/key offset supplied by the surrounding correction layer, then the intended opening gives the local receiver share

$$\tilde{\mathbf{m}}_{d,\mathbf{x}}[\mathsf{ind}] := (\tilde{\mathbf{k}}_d[\mathsf{ind}] + \mathbf{r}[\mathsf{ind}] \cdot d) - (\mathbf{r}[\mathsf{ind}] \cdot d - \mathbf{s}[\mathsf{ind}] \cdot \mathbf{x}[\mathsf{ind}]) = \tilde{\mathbf{k}}_d[\mathsf{ind}] + \mathbf{s}[\mathsf{ind}] \cdot \mathbf{x}[\mathsf{ind}].$$

The second opening gives instead

$$\begin{aligned} \tilde{\mathbf{m}}^{\mathsf{sub}}_{d,\mathbf{x}'}[\mathsf{ind}] &:= (\tilde{\mathbf{k}}_d[\mathsf{ind}] + \mathbf{r}[\mathsf{ind}] \cdot d) - (\mathbf{r}[\mathsf{ind}] \cdot d' - \mathbf{s}[\mathsf{ind}] \cdot \mathbf{x}'[\mathsf{ind}]) \\ &= \tilde{\mathbf{k}}_d[\mathsf{ind}] + \mathbf{s}[\mathsf{ind}] \cdot \mathbf{x}'[\mathsf{ind}] + \mathbf{r}[\mathsf{ind}] \cdot (d - d'). \end{aligned}$$

Therefore an exact digest collision is stronger than necessary for an attack: a near collision can also be useful if the extra term $\mathbf{r}[\mathsf{ind}] \cdot (d - d')$ stays inside the rounding/noise budget, since the receiver then obtains the same form of local message term $\tilde{\mathbf{k}}_d[\mathsf{ind}] + \mathbf{s}[\mathsf{ind}] \cdot \mathbf{x}'[\mathsf{ind}]$, but under the reused digest-$d$ sender offset. The consequence is that the digest-indexed correction for one opening can be reused as if it were a correction for another opening. Comparing the intended opening with the second one gives

$$\tilde{\mathbf{m}}^{\mathsf{sub}}_{d,\mathbf{x}'}[\mathsf{ind}] - \tilde{\mathbf{m}}_{d,\mathbf{x}}[\mathsf{ind}] = \mathbf{s}[\mathsf{ind}] \cdot (\mathbf{x}'[\mathsf{ind}] - \mathbf{x}[\mathsf{ind}]) + \mathbf{r}[\mathsf{ind}] \cdot (d - d').$$

If the near-collision term is small enough to be absorbed by the decoding margin, the receiver learns a noisy linear equation in the sender secret $\mathbf{s}[\mathsf{ind}]$. Concretely, write

$$a_{\mathsf{ind}} := \mathbf{x}'[\mathsf{ind}] - \mathbf{x}[\mathsf{ind}], \qquad \eta_{\mathsf{ind}} := \mathbf{r}[\mathsf{ind}] \cdot (d - d')$$

after folding the already-budgeted LacE/LHE and dropped-digit errors into $\eta_{\mathsf{ind}}$. The observed difference has the form

$$\Delta m_{\mathsf{ind}} := \tilde{\mathbf{m}}^{\mathsf{sub}}_{d,\mathbf{x}'}[\mathsf{ind}] - \tilde{\mathbf{m}}_{d,\mathbf{x}}[\mathsf{ind}] = \mathbf{s}[\mathsf{ind}] \cdot a_{\mathsf{ind}} + \eta_{\mathsf{ind}}.$$

If $\eta_{\mathsf{ind}}$ lies within half of the relevant rounding gap, then denoising removes it exactly:

$$\mathsf{Denoise}(\phi, \Delta m_{\mathsf{ind}}) = \mathsf{Denoise}(\phi, \mathbf{s}[\mathsf{ind}] \cdot a_{\mathsf{ind}}).$$

At the leaf level, let $\bar{a}_{\mathsf{ind}} := \bar{x}'_{\mathsf{ind}} - \bar{x}_{\mathsf{ind}} \in \mathbb{Z}_p$ be the plaintext difference represented by $a_{\mathsf{ind}}$. In any CRT slot where $\bar{a}_{\mathsf{ind}}$ is invertible, this gives the explicit recovery formula

$$\bar{s}_{\mathsf{ind}} = \lceil \Delta m_{\mathsf{ind}} \rfloor_{\lceil p/q \rfloor} \cdot \bar{a}^{-1}_{\mathsf{ind}} \pmod{p},$$

where $\bar{s}_{\mathsf{ind}}$ denotes the decoded sender secret in that slot. With the opposite subtraction order one uses the opposite denominator, so the sign is immaterial. At recursive levels the same calculation is applied componentwise with the corresponding CRT scale factor $q_t/q$: one denoises the difference to obtain $\mathbf{s}[\mathsf{ind}] \cdot a_{\mathsf{ind}}$ modulo the component, and then inverts the chosen unit component of $a_{\mathsf{ind}}$. More generally, several near-colliding openings yield equations

$$\mathsf{Denoise}(\phi, \Delta m^{(t)}_{\mathsf{ind}}) = \mathbf{s}[\mathsf{ind}] \cdot a^{(t)}_{\mathsf{ind}}$$

in the decoded domain; if the chosen differences have full rank, solving this linear system reveals $\mathbf{s}[\mathsf{ind}]$. This breaks the sender-secret privacy that the LacE/LHE correction is supposed to protect.

**Proof strategy.** The truncation-specific binding issue is that a malicious receiver does not need an exact second opening of the digest. It may try to find two bounded openings $w, w'$ with digest values $d = Aw$ and $d' = Aw'$ such that the short-mask discrepancy

$$(d - d') \cdot r$$

is still small enough to pass through rounding and preserve the message. If $d - d'$ is large, multiplication by the Gaussian mask $r \leftarrow \overline{\mathcal{D}}$ smudges the discrepancy. By Lemma 8, $(d - d') \cdot r$ has at most $\exp(-\pi)$-scale distinguishing advantage from uniform once $\|d - d'\|_2 \geq q/s_{\mathsf{mask}}$. The concrete security calculation below accounts for the cost of finding such short dual vectors. If $d - d'$ is small enough to avoid smudging, then $z := w - w'$ gives an approximate SIS relation

$$Az = 0 + e \pmod{q}, \qquad e := d - d',$$

with both $z$ and $e$ short. Thus every useful second opening either loses its signal by smudging or yields a short approximate SIS witness, which we embed into normal-form SIS for $[A \mid -1]$.

**Useful near-collision premise.** The attack specific to the truncated protocol is not an exact digest collision. It is a near collision that leaves the masking term small. If a receiver uses a digest-indexed key for $d = Aw$ while evaluating another bounded opening $w'$, then the mismatch contains

$$(d - d') \cdot r, \qquad d' := Aw',$$

where $r \leftarrow \overline{\mathcal{D}}$ is one of the short masks used in the truncated rows. Let $s_{\mathsf{mask}}$ be the Gaussian parameter of these masks and define the smudging radius

$$\alpha_{\mathsf{use}} := \frac{q}{s_{\mathsf{mask}}}. \tag{22}$$

This is the digest-difference analogue of the bounded-opening threshold $\beta$ from Corollary 1: it marks the point beyond which the relevant product with fresh Gaussian noise has advantage at most $\exp(-\pi)$. The radius is not a correctness margin; it separates openings lost to smudging from openings that must be ruled out computationally.

**Lemma 12** (Digest-difference smudging)**.** *Let $d = Aw$ and $d' = Aw'$ be two digest values for a selected truncated row, and let $r \leftarrow \overline{\mathcal{D}}$ be the corresponding mask. If $\|d - d'\|_2 > \alpha_{\mathsf{use}}$, then $(d - d') \cdot r$ has at most $\exp(-\pi)$-scale distinguishing advantage from uniform, up to the statistical distance between the untruncated and truncated Gaussian experiments.*

*Proof.* By the same smudging estimate used in Lemma 8, if $\|d - d'\|_2 > \alpha_{\mathsf{use}}$, then $(d - d') \cdot r$ has at most $\exp(-\pi)$-scale distinguishing advantage from uniform in the untruncated Gaussian experiment. Replacing the untruncated Gaussian by its truncated version changes the experiment only by the statistical distance in Lemma 2. Such an opening cannot preserve the message. □

Thus the only collisions that can remain useful are bounded openings whose digest difference has norm at most $\alpha_{\mathsf{use}}$.

We state the hardness premise in the usual approximate ISIS form, where $y$ is the target and $e$ is the tolerated approximation error [DSH22]. For parameters $q, m, \beta, \alpha$ and target $y \in R_q$, let $\mathsf{ApproxRISIS}_{q,m,\beta,\alpha}(y)$ be the problem of, given $A \leftarrow R_q^{1\times m}$, finding $z \in R^m \setminus \{0\}$ and $e \in R$ such that

$$Az = y + e \pmod q, \qquad \|z\|_2 \le \beta, \qquad \|e\|_2 \le \alpha.$$

**Lemma 13** (Useful near-collision bound)**.** *For opening bound $\beta$, let $\mathsf{NearCol}_{\beta,\alpha}(A)$ be the event that a PPT algorithm outputs $w, w' \in R^m$ satisfying*

$$w \ne w', \qquad \|w\|_2, \|w'\|_2 \le \beta, \qquad \|Aw - Aw'\|_2 \le \alpha.$$

*Let*

$$A^\star := [\, A \mid -1 \,] \in R_q^{1\times(m+1)}$$

*and*

$$B_{\mathsf{near}}(\beta, \alpha) := \sqrt{(2\beta)^2 + \alpha^2},$$

*so the coefficient embedding gives the integer SIS family*

$$\mathsf{SIS}_{n,n(m+1),q,\, B_{\mathsf{near}}(\beta,\alpha)}$$

*for matrices of the form $[\mathrm{Emb}(A) \mid -I_n]$. If this SIS family is hard at bound $B_{\mathsf{near}}(\beta, \alpha_{\mathsf{use}})$, then*

$$\Pr_A[\mathsf{NearCol}_{\beta,\alpha_{\mathsf{use}}}(A)] \le \mathsf{negl}(\lambda).$$

*Proof.* Any such near collision gives a solution to the homogeneous instance

$$\mathsf{ApproxRISIS}_{q,m,2\beta,\alpha}(0).$$

Indeed, setting $z := w - w'$ and $e := Aw - Aw'$ gives $z \neq 0$, $\|z\|_2 \leq 2\beta$, $\|e\|_2 \leq \alpha$, and

$$Az = 0 + e \pmod{q}.$$

Equivalently, $(z, e)$ is an exact normal-form Ring-SIS solution for the augmented row $A^\star$, since $A^\star(z, e) = Az - e = 0 \pmod{q}$, at norm $B_{\mathsf{near}}(\beta, \alpha)$. Under coefficient embedding this is the integer SIS instance stated in the lemma. Therefore hardness at $B_{\mathsf{near}}(\beta, \alpha_{\mathsf{use}})$ rules out $\mathsf{NearCol}_{\beta,\alpha_{\mathsf{use}}}(A)$ except with negligible probability. □

**Concrete Security Evaluation.** We now instantiate the useful near-collision premise for the parameter set used in Section 12. There we take $\lambda = 128$, $n = 8192 = 2^{13}$, $\log_2 q = 220$. This parameter set conforms to the HE standard [ACC+19], which shows that it achieves 130 bits of security against an adversary solving the SIS problem. The near-collision search problem relaxes the SIS bound by a factor of $\sqrt{5}$, which slightly reduces the estimated cost to ≈128 bits. The detailed security estimate follows.

We first set the useful radius for near-collision vectors. This is the per-sample distinguishing scale targeted by the dual attack in the HE standard [ACC+19], not a negligible-bias threshold. The Gaussian parameter is $s = 8$, in the convention where the ordinary standard deviation is

$$\sigma = \frac{s}{\sqrt{2\pi}} \approx 3.2.$$

The dual-attack formula uses the LWE noise rate

$$\alpha_{\mathsf{HES}} = \frac{s}{q} = \frac{\sqrt{2\pi}\sigma}{q}.$$

For a dual relation vector $v$, the distinguishing advantage is estimated as [ACC+19, Section 2.1.2.c]

$$\exp\left(-\pi(\alpha_{\mathsf{HES}}\|v\|_2)^2\right).$$

At $\|v\|_2 = q/s$, this advantage is $e^{-\pi}$. This is the per-sample distinguishing advantage from one short dual vector, not the final security advantage. The HE-standard tables normalize attack costs to a fixed success probability, namely 51% [ACC+19, Section 2.1.2.e]; the BKZ work and repetitions needed to amplify the per-sample advantage to that probability are included in the reported cost. Therefore the near-collision check uses the same per-sample scale and then estimates the cost of finding such a relation under the HES BKZ.sieve model:

$$\alpha_{\mathsf{HES}}\|v\|_2 \approx 1, \qquad \|v\|_2 \approx \frac{1}{\alpha_{\mathsf{HES}}} = \frac{q}{s} = \frac{q}{\sqrt{2\pi}\sigma}.$$

Therefore the per-sample distinguishing radius is

$$\frac{q}{s} = \frac{q}{\sqrt{2\pi}\sigma} = \frac{q}{8} = 0.125q.$$

We use the slightly more conservative rounded value

$$\Gamma_{\mathsf{nc}} := 0.114\, q.$$

We instantiate Lemma 13 with $\beta = \alpha = \Gamma_{\mathsf{nc}}$. The augmented normal-form SIS witness has two opening-difference coordinates and one digest-error coordinate, so its Euclidean target radius is

$$B_{\mathsf{near}}(\Gamma_{\mathsf{nc}}, \Gamma_{\mathsf{nc}}) = \sqrt{(2\Gamma_{\mathsf{nc}})^2 + \Gamma_{\mathsf{nc}}^2} = \sqrt{5}\,\Gamma_{\mathsf{nc}} = 0.254911749435\,q.$$

Since $\log_2 q = 220$, this gives

$$\log_2 B_{\mathsf{near}} = 220 + \log_2(0.254911749435) = 218.028069777.$$

We now compute the cost of finding such a vector using the BKZ.sieve model as in [ACC+19, Sections 2.1.2 and 2.1.5]. The augmented row $[\mathrm{Emb}(A) \mid -I_n]$ has determinant $q^n$ and ambient dimension $3n$. For a dropped lattice dimension $d \in [n+1, 3n]$, the GSA prediction is

$$\log_2 \|v\|_2 \approx d \log_2 \delta_0(b) + \frac{n \log_2 q}{d},$$

where the Chen root-Hermite factor used in [ACC+19] is

$$\log_2 \delta_0(b) = \frac{\log_2(b/(2\pi e)) + \log_2(\pi b)/b}{2(b-1)}.$$

For each $d$, we solve for the smallest block size $b$ that reaches the target norm, then evaluate the HES sieve cost

$$\log_2 \mathrm{Cost}_{\mathsf{sieve}}(b, d) = 0.292b + 16.4 + \log_2(8d).$$

Minimizing over $d \in [8193, 24576]$ and rounding the BKZ block size upward, the optimum is $d = 16294$ and $b = 326$. The resulting security exponent is therefore

$$\lambda_{\mathsf{near}} = 0.292 \cdot 326 + 16.4 + \log_2(8 \cdot 16294) = 128.584053193.$$

Here $\lambda_{\mathsf{near}}$ is the usual "security bits" value: the estimated attack cost is $2^{\lambda_{\mathsf{near}}}$ elementary operations under the HES BKZ.sieve model. Thus the concrete augmented-SIS near-collision search is above the 128-bit target.

## D.2 Truncated LHE simulation

**Lemma 14** (Simulation security of $\mathsf{LHE}_{\mathsf{hi}}$)**.** *Assume the Ring-LWE premise used for* $\mathsf{LHE}$ *holds for width* $\tau'$*. Then* $\mathsf{LHE}_{\mathsf{hi}}$ *is simulation secure for the affine relation*

$$\boldsymbol{m}_{\mathsf{res}}^{(t)} = \boldsymbol{m}_1 \cdot \mathsf{high}_g(y^{(t)}) + \boldsymbol{m}_2^{(t)}$$

*with the same number of adaptive key queries as in the full-gadget proof of Section B.3.*

*Proof.* Fix messages $\boldsymbol{m}_1$, $\{\boldsymbol{m}_2^{(t)}\}_{t\in[T]}$, and queries $\{y^{(t)}\}_{t\in[T]}$. Let $\mathcal{D}_0$ be the real $\mathsf{LHE}_{\mathsf{hi}}$ view:

$$\mathsf{ct}_1 = \mathbf{a}\mathbf{s}_1^T + \boldsymbol{m}_1 \mathbf{g}_{\mathsf{hi}}^T + \mathbf{E}_1, \qquad \mathsf{ct}_2^{(t)} = \mathbf{a}s_2^{(t)} + \boldsymbol{m}_2^{(t)} + \mathbf{E}_2^{(t)},$$

together with keys $\mathsf{sk}_{y^{(t)}} = \mathbf{s}_1^T \mathbf{g}_{\mathsf{hi}}^{-1}(y^{(t)}) + s_2^{(t)}$. For $i = 0, \ldots, \tau'$, define a hybrid $\mathcal{D}_i$ in which the first $i$ columns of $\mathsf{ct}_1 - \boldsymbol{m}_1 \mathbf{g}_{\mathsf{hi}}^T$ are replaced by uniform Ring-LWE challenge columns, while the remaining columns are real. The transition $\mathcal{D}_{i-1} \approx_c \mathcal{D}_i$ is exactly the Ring-LWE row replacement used in the full $\mathsf{LHE}$ proof, with one fewer gadget column.

After all $\tau'$ columns of $\mathsf{ct}_1$ are uniform, define hybrids $\mathcal{D}_{\tau'+t}$, $t = 1, \ldots, T$, that replace $\mathsf{ct}_2^{(t)} - \boldsymbol{m}_2^{(t)}$ with uniform samples, one query at a time. Again each transition is a Ring-LWE replacement. In the final hybrid all ciphertext components are uniform subject only to the decryption equations

$$\mathsf{ct}_1 \mathbf{g}_{\mathsf{hi}}^{-1}(y^{(t)}) + \mathsf{ct}_2^{(t)} - \mathsf{ask}_{y^{(t)}} = \boldsymbol{m}_{\mathsf{res}}^{(t)} + \text{noise}.$$

Those equations depend on $y^{(t)}$ through $\mathbf{g}_{\mathsf{hi}}^T \mathbf{g}_{\mathsf{hi}}^{-1}(y^{(t)}) = \mathsf{high}_g(y^{(t)})$, so the final distribution is sampled by the usual $\mathsf{LHE}$ simulator with $\tau'$ columns and residuals $\boldsymbol{m}_{\mathsf{res}}^{(t)}$. A union bound over the $\tau' + T$ row replacements gives the claim. □

### D.3 Truncated LacE simulation

**Lemma 15** (Simulation security of $\mathsf{LacE}_{\mathsf{hi}}$)**.** *Assume adaptive error-leakage Ring-LWE for the secret and error distributions used in Figure 16, with leakage matrices whose entries are bounded by the truncated gadget digits. Then* $\mathsf{LacE}_{\mathsf{hi}}$ *is $T$-query simulation secure under $\overline{\chi}_{\mathsf{LacE}}$-masked evaluation-error leakage.*

*Proof.* Fix public parameters $(\mathbf{b}_0, \mathbf{b}_1)$, a message vector $\mathbf{s}$, and query vectors $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}$. For every query, compute the same digest tree as $\mathsf{LacE}_{\mathsf{hi}}.\mathsf{Digest}$. Thus each internal level uses vectors

$$\begin{pmatrix} -\mathbf{g}_{\mathsf{hi}}^{-1}(y_{\mathsf{pre}\|0}^{(t)}) \\ -\mathbf{g}_{\mathsf{hi}}^{-1}(y_{\mathsf{pre}\|1}^{(t)}) \end{pmatrix},$$

and the leaf level uses $\left(\begin{smallmatrix} -\mathbf{g}_{\mathsf{leaf}}^{-1}(x_{\mathsf{pre}\|0}^{(t)}) \\ -\mathbf{g}_{\mathsf{leaf}}^{-1}(x_{\mathsf{pre}\|1}^{(t)}) \end{smallmatrix}\right)$. All entries are bounded by the appropriate gadget base, so the $T$ leakage rows lie in the allowed leakage family.

Let $\mathcal{D}_0'$ be the real ciphertext distribution with the truncated Gaussians from Figure 16, and let $\mathcal{D}_0$ be the same experiment with untruncated Gaussians. By Lemma 2, $\mathcal{D}_0' \approx_s \mathcal{D}_0$.

Now order all laconic ciphertext rows

$$(i, \mathsf{ind}) \in \{0, \ldots, \ell - 1\} \times \{0,1\}^\ell.$$

For internal rows $i < \ell - 1$,

$$\mathbf{C}_i[\mathsf{ind}] = \mathbf{r}^{(i)}[\mathsf{ind}] \cdot ((\mathbf{b}_0^{\mathsf{hi}})^T \ (\mathbf{b}_1^{\mathsf{hi}})^T) + \mathbf{r}^{(i+1)}[\mathsf{ind}] \cdot (\overline{\mathsf{ind}}_i \mathbf{g}_{\mathsf{hi}}^T \ \mathsf{ind}_i \mathbf{g}_{\mathsf{hi}}^T) + \mathbf{E}_i[\mathsf{ind}],$$

and the leaf row has the analogous expression with $\mathbf{b}_b^{\mathsf{leaf}}$, $\mathbf{g}_{\mathsf{leaf}}$, and $\mathbf{s}[\mathsf{ind}]$. For $j = 0, \ldots, N$, where $N = \ell 2^\ell$, define $\mathcal{D}_j$ so that the first $j$ rows in this order are replaced by uniform row vectors in the appropriate ring module, while all later rows are real.

Consider the transition $\mathcal{D}_{j-1}$ to $\mathcal{D}_j$ for a row $(i, \mathsf{ind})$. Condition on all masks except the row secret $\mathbf{r}^{(i)}[\mathsf{ind}]$ and on all ciphertext rows not being replaced in this step. The visible row is an error-leakage Ring-LWE sample with public vector $((\mathbf{b}_0^{\mathsf{hi}})^T \ (\mathbf{b}_1^{\mathsf{hi}})^T)$ for internal levels, or $((\mathbf{b}_0^{\mathsf{leaf}})^T \ (\mathbf{b}_1^{\mathsf{leaf}})^T)$ at the leaf. The term containing the next mask, namely $\mathbf{r}^{(i+1)}[\mathsf{ind}] \cdot (\overline{\mathsf{ind}}_i \mathbf{g}_{\mathsf{hi}}^T \ \mathsf{ind}_i \mathbf{g}_{\mathsf{hi}}^T)$, is a fixed offset under this conditioning, and adding a fixed offset to a uniform row leaves it uniform. The leakage revealed by the $T$ evaluations is precisely the linear map that sends the row error $\mathbf{E}_i[\mathsf{ind}]$ to its contribution to the evaluation errors, plus the fresh hiding noise from $\overline{\chi}_{\mathsf{LacE}}$. Hence $\mathcal{D}_{j-1} \approx_c \mathcal{D}_j$ by adaptive error-leakage Ring-LWE.

After $N$ replacements, every laconic ciphertext row is uniform and the masked error-leakage values are generated independently from the allowed leakage distribution. This final hybrid is exactly the output of $\mathsf{LacE}_{\mathsf{hi}}.\mathsf{Sim}$ on the public parameters and the query vectors. Summing the $N$ negligible gaps proves the lemma. □

The preceding lemma is a fixed-query hiding statement, exactly as in TinyLabels's proof of LEnc: the databases are fixed inputs to the simulator, and possible digest collisions among those databases are not ruled out or needed. The change $\mathbf{r}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(\ell-1)} \leftarrow \overline{\mathcal{D}}$ is handled there, because these values are the Ring-LWE row secrets in the laconic ciphertext hybrids. The near-collision premise is a different statement: it says that one digest-indexed decryption key cannot be useful for two distinct bounded openings whose digest difference remains below the smudging radius. That premise is therefore used in the receiver-view hybrids of Theorem 10, before applying the fixed-query $\mathsf{LacE}_{\mathsf{hi}}$ simulator.

## D.4 Composed truncated protocol

**Theorem 10** (Semi-honest security of the truncated protocol)**.** *Assume the premises of Lemmas 14 and 15, the root-digest hiding condition of Lemma 7, the normal-form Ring-SIS hardness required by Lemma 13, and the truncated correctness relation Equation* (17) *under the rounding condition of Equation* (21)*. Then* $\Pi_{\mathsf{hi}}$ *securely realizes* $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ *in the random-oracle model against static semi-honest corruptions.*

*Proof.* We prove the two corruption cases separately. In both cases the simulator uses a lazy random-oracle table $Q$: a fresh, non-programmed query receives a uniform ciphertext-shaped answer, while programmed digest-indexed queries receive the ciphertext specified by the simulator.

**Corrupt sender.** The simulator $\mathsf{Sim}^{\mathsf{hi}}_{\mathbb{S}}$ follows the sender-side simulator from Section B.3, with every full-gadget object replaced by its truncated counterpart. It samples the sender's ideal output share, samples the root digest as $\widehat{d}^{\mathsf{hi}}_{\mathsf{rt}} \leftarrow \mathsf{high}_g(U_{\mathcal{R}_q})$, and programs the designated oracle points indexed by $\widehat{d}^{\mathsf{hi}}_{\mathsf{rt}}$ so that the recursive $\mathsf{LHE}_{\mathsf{hi}}$ decryptions produce the target sender share at the leaves.

Let $\mathcal{D}^S_R$ and $\mathcal{D}^S_I$ be the real and ideal distributions for the corrupt sender. Define hybrids:

*Hybrid* $\mathcal{H}^S_0$*.* This is the real execution of $\Pi_{\mathsf{hi}}$, so $\mathcal{D}^S_R \equiv \mathcal{D}^S_0$.

*Hybrid* $\mathcal{H}^S_1$*.* Replace the real randomized root digest by $\widehat{d}^{\mathsf{hi}}_{\mathsf{rt}} \leftarrow \mathsf{high}_g(U_{\mathcal{R}_q})$. This is statistically close to $\mathcal{H}^S_0$ by Lemma 7 and post-processing by $\mathsf{high}_g$.

*Hybrid* $\mathcal{H}^S_2$*.* Answer the random oracle lazily from $Q$, except for designated points that will later be programmed. This is identically distributed to $\mathcal{H}^S_1$ until a programmed point is queried before it is programmed; that event has probability at most the number of designated points divided by the ciphertext space size, hence negligible.

*Hybrid* $\mathcal{H}^S_3$*.* Program the designated points with simulated $\mathsf{LHE}_{\mathsf{hi}}$ correction ciphertexts and matching keys, level by level, using the residual plaintexts dictated by the recursive expand equations. For each level and block, the replacement of the real bridge/correction view by the simulated one is justified by Lemma 14. Since the number of levels and blocks is polynomial, $\mathcal{D}^S_2 \approx_c \mathcal{D}^S_3$.

*Hybrid* $\mathcal{H}^S_4$*.* Replace the honest receiver's output by the output of $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ using the sender share selected by the simulator. Equations (17) and (21) show that the programmed truncated execution decodes the same CI-VOLE relation, including the dropped low-digit terms, so $\mathcal{D}^S_3 \approx_c \mathcal{D}^S_4$. The last hybrid is exactly the ideal execution, and therefore $\mathcal{D}^S_R \approx_c \mathcal{D}^S_I$.

**Corrupt receiver.** The simulator $\mathsf{Sim}^{\mathsf{hi}}_{\mathbb{R}}$ is given the receiver's input $\mathbf{x}$ and ideal output $\mathbf{m}$. It computes all truncated digest values honestly from $\mathbf{x}$, runs the recursive partitioning used by the protocol, and simulates each local block as follows. First it invokes $\mathsf{LacE}_{\mathsf{hi}}.\mathsf{Sim}$ on the block query vectors to obtain the laconic ciphertext and fake evaluation errors. It then samples the hiding noise from $\overline{\chi}_{\mathsf{LacE}}$, sets the desired residual plaintext for the local $\mathsf{LHE}_{\mathsf{hi}}$ call so that subtracting the

simulated LacE evaluation yields the target noisy expand identity, and invokes $\mathsf{LHE_{hi}.Sim}$. The resulting correction ciphertexts are installed at the corresponding digest-indexed oracle points.

Let $\mathcal{D}_R^R$ and $\mathcal{D}_I^R$ be the real and ideal distributions for the corrupt receiver, augmented with the output of an arbitrary polynomial-time local post-processing algorithm applied to the receiver's view. This augmentation is without loss of generality, since indistinguishability is closed under efficient post-processing.

*Hybrid* $\mathcal{H}_0^R$. This is the real execution, so $\mathcal{D}_R^R \equiv \mathcal{D}_0^R$.

*Hybrid* $\mathcal{H}_1^R$. Replace the random oracle by the lazy table $Q$. The distribution is identical to $\mathcal{H}_0^R$.

*Hybrid* $\mathcal{H}_2^R$. This hybrid is the same as $\mathcal{H}_1^R$, except that it aborts on the following event $\mathsf{Bad_{use}}$. For some recursive node, the post-processor outputs two distinct protocol-useful openings $w, w'$ whose digest difference is small enough to preserve the local message. By Lemma 12, such openings must satisfy $\|Aw - Aw'\|_2 \le \alpha_{\mathsf{use}}$. If both openings are bounded, set $z := w - w'$ and

$$e := Aw - Aw'.$$

Then the digest row $A$ at that node satisfies

$$z \neq 0, \qquad \|w\|_2, \|w'\|_2 \le \beta, \qquad \|e\|_2 \le \alpha_{\mathsf{use}}, \qquad Az = 0 + e \pmod q.$$

This post-processing gives a solver for $\mathsf{NearCol}_{\beta,\alpha_{\mathsf{use}}}(A)$, equivalently for the homogeneous approximate Ring-ISIS instance with target 0. Let $N_{\mathsf{dig}}$ be the number of digest rows instantiated in the execution, and let $\mathsf{Bounded}$ denote the subevent that both openings satisfy the bound. By Lemma 13,

$$\Pr[\mathsf{Bad_{use}} \wedge \mathsf{Bounded}] \le N_{\mathsf{dig}} \cdot \Pr_A[\mathsf{NearCol}_{\beta,\alpha_{\mathsf{use}}}(A)] \le \mathsf{negl}(\lambda),$$

under normal-form Ring-SIS at norm bound

$$B_{\mathsf{near}}(\beta, \alpha_{\mathsf{use}}) = \sqrt{(2\beta)^2 + \alpha_{\mathsf{use}}^2},$$

equivalently under the coefficient embedding parameter

$$\mathsf{SIS}_{n,n(m+1),q,\, B_{\mathsf{near}}(\beta,\alpha_{\mathsf{use}})}$$

for matrices of the form $[\mathrm{Emb}(A) \mid -I_n]$. If at least one opening is not bounded, then Corollary 1 says the out-of-bound contribution is smudged and cannot be protocol-useful. Hence $\Pr[\mathsf{Bad_{use}}] \le \mathsf{negl}(\lambda)$, and $\mathcal{D}_1^R \approx_c \mathcal{D}_2^R$.

Conditioned on $\neg\mathsf{Bad_{use}}$, every programmed truncated digest value has a unique protocol-useful opening. Consequently, when the simulator supplies query vectors to the fixed-query $\mathsf{LacE_{hi}}$ and $\mathsf{LHE_{hi}}$ simulators below, these are the only useful queries associated with the corresponding digest-indexed key.

For each recursive level $k$ and each local block $j$, define two successive hybrids.

*Hybrid* $\mathcal{H}_{k,j,1}^R$. Replace the real $\mathsf{LacE_{hi}}$ ciphertext for block $(k, j)$ and its $\overline{\chi}_{\mathsf{LacE}}$-masked evaluation-error leakage by the output of $\mathsf{LacE_{hi}.Sim}$ on the same block queries. By Lemma 15, this changes the receiver's view only negligibly.

*Hybrid* $\mathcal{H}_{k,j,2}^R$. Replace the corresponding $\mathsf{LHE_{hi}}$ bridge ciphertext, correction ciphertext, and key for block $(k, j)$ by $\mathsf{LHE_{hi}.Sim}$, and program the designated random-oracle entry with the simulated correction ciphertext. The residual given to $\mathsf{LHE_{hi}.Sim}$ is exactly the value that the receiver should hold after the local LacE term is subtracted. By Lemma 14, this replacement is computationally indistinguishable.

Iterating these hybrids over all polynomially many blocks and levels, starting from $\mathcal{H}_2^R$, replaces every truncated LacE and LHE object in the receiver's view by simulated objects. Equations (17) and (21) guarantee that each level denoises back to the same clean recursive interface used by the simulator, so the residual plaintexts given to the LHE simulator are consistent across levels.

The final hybrid is therefore distributed exactly as the ideal execution with receiver input $\mathbf{x}$ and output $\mathbf{m}$, except for the negligible hybrid gaps and the negligible $\mathsf{Bad}_{\mathsf{use}}$ event. Thus $\mathcal{D}_R^R \approx_c \mathcal{D}_I^R$.

Combining the corrupt-sender and corrupt-receiver cases completes the proof. □

# E Proofs for Malicious Security

## E.1 Roadmap and the malicious interface

The malicious proof has three jobs. First, for a corrupted receiver, the simulator must identify the effective input $x$, simulate the honest-sender objects that the receiver sees, and program the random-oracle-derived correction so that the receiver's final output is the ideal output. The receiver input can be recovered in two ways. The random-linear-check wrapper of Section 10.1.6 gives an explicit extraction mechanism, proved inline in the corrupted-receiver hybrids. Alternatively, Section E.4 gives a no-message knowledge extraction route for the public digest.

Second, for a corrupted sender, the simulator must extract the sender's effective correlation data, in particular the $\Delta$-side setup encoded by the reusable ciphertext package, define the fresh online mask from the sender's response, and then generate a receiver-facing transcript consistent with the ideal functionality. This is why the sender side is organized around an extractable setup proof; the malicious protocol does not use golden-seed grinding and has no separate sender-side online proof.

Third, the single-identifier statements must lift to executions with many identifiers. The shared setup is handled once, while the online messages, random-oracle labels, and ideal-functionality table entries are indexed by $\mathsf{id} = (\mathsf{sid}, \mathsf{mode})$. The final theorem therefore combines receiver extraction, sender-side certification, random-oracle programming, and a hybrid over identifiers.

## E.2 Boundary Transcripts

For the malicious proofs, a corrupted party's view is the boundary object from which that party can be rerun: its external input, coins, incoming messages, and random-oracle query/answer transcript. Outgoing messages are not listed in the sender's or receiver's own view; they are determined when the party is rerun from its view, and appear instead as incoming messages in the other party's view. We also do not enumerate values that are merely computable from the view.

Let $L$ denote the set of ordinary recursive shrink/expand setup levels. The definitions below are parameterized by the receiver-extraction choice. In the digest-knowledge variant, the check messages are empty. In the random-linear-check variant, the final check adds, for each $t \in [\tau_{\mathsf{chk}}]$, the receiver-to-sender message $u'_t$, the sender-to-receiver message $h_t$, the check challenge $\rho_{\mathsf{chk}}$, and the check-hash oracle queries described in Section 10.1.6. In the non-interactive mode, $\rho_{\mathsf{chk}}$ is obtained from the online random oracle. The scalar check correlations are the sacrificed VOLE output coordinates, not separate auxiliary inputs.

**Corrupted receiver.** Its offline incoming message is

$$\mathsf{Msg}_R^{\mathsf{off}} = \big((\mathsf{ct}_{\mathsf{lac}}{}^{(h)}, \mathsf{ct}_{\mathsf{r}}{}^{(h)}, \pi_{\mathsf{setup}}^{(h)})_{h\in L}, \mathsf{ct}_{\mathsf{lac,rt}}, \mathsf{ct}_{\mathsf{r,rt}}, B^*, \pi_{\mathsf{setup}}^{\mathsf{rt}}\big).$$

Its online sender-to-receiver message is

$$\mathsf{Msg}_R^{\mathsf{on}} = \begin{cases} (\mathsf{sk_{rt}}', \rho_{\mathsf{chk}}) & \text{in interactive random-linear-check mode,} \\ \mathsf{sk_{rt}}' & \text{in interactive digest-knowledge mode,} \\ \emptyset & \text{in non-interactive mode.} \end{cases}$$

In non-interactive mode, the corresponding online tuple is obtained through the online random oracle instead of through an incoming message: it is $(\mathsf{sk_{rt}}', \varsigma, \rho_{\mathsf{chk}})$ in the random-linear-check variant and $(\mathsf{sk_{rt}}', \varsigma)$ in the digest-knowledge variant. Its check incoming message is

$$\mathsf{Msg}_R^{\mathsf{chk}} = \begin{cases} (h_t)_{t\in[\tau_{\mathsf{chk}}]} & \text{in the random-linear-check variant,} \\ \emptyset & \text{in the digest-knowledge variant.} \end{cases}$$

The corrupted-receiver view is

$$\mathsf{View}_R = (\mathsf{inp}_R, \omega_R, \mathsf{Msg}_R^{\mathsf{off}}, \mathsf{Msg}_R^{\mathsf{on}}, \mathsf{Msg}_R^{\mathsf{chk}}, \mathsf{RO}_R),$$

where $\mathsf{inp}_R$ is the receiver's external input, $\omega_R$ is its coins, and $\mathsf{RO}_R$ is the ordered sequence of random-oracle query/answer pairs it observes. In particular, $\mathsf{RO}_R$ contains the online random-oracle query in non-interactive mode, the correction-ciphertext queries made during root and ordinary expansion, and, in the random-linear-check variant, the check-hash queries. The external input $\mathsf{inp}_R$ is not assumed to be the effective VOLE input extracted by the proof; the effective input may be hard-coded in the receiver, derived from its coins, or encoded in its interaction behavior.

**Corrupted sender.** The corrupted sender has no incoming offline protocol message from the receiver in this layer. Its online incoming message is the receiver's randomized root digest

$$\mathsf{Msg}_S^{\mathsf{on}} := d'_{\mathsf{rt}}.$$

Its check incoming message is

$$\mathsf{Msg}_S^{\mathsf{chk}} = \begin{cases} (u'_t)_{t\in[\tau_{\mathsf{chk}}]} & \text{in the random-linear-check variant,} \\ \emptyset & \text{in the digest-knowledge variant.} \end{cases}$$

The corrupted-sender view is

$$\mathsf{View}_S = (\mathsf{inp}_S, \omega_S, \mathsf{Msg}_S^{\mathsf{on}}, \mathsf{Msg}_S^{\mathsf{chk}}, \mathsf{RO}_S),$$

where $\mathsf{RO}_S$ is the ordered sequence of random-oracle query/answer pairs observed by the sender. In non-interactive mode, this includes the online random-oracle query used to derive either $(\mathsf{sk_{rt}}', \varsigma)$ or $(\mathsf{sk_{rt}}', \varsigma, \rho_{\mathsf{chk}})$; in the random-linear-check variant it also includes the check-hash queries made by the sender. The sender's setup package, setup proofs, online response, and check hashes are outgoing messages determined by the sender when this boundary view is replayed.

## E.3 Boundary objects for the simulators

The malicious protocol and its boundary transcripts are specified in Sections E.2 and 10.1. For the simulator descriptions below, we name the concrete honest-generated objects that are read or programmed in one execution of $\mathsf{LogVOLE}_{\mathcal{R}}$. This is not a second definition of a corrupted party's view; the view objects are exactly $\mathsf{View}_R$ and $\mathsf{View}_S$ from Section E.2. Let $w = \tau\rho\alpha^{\ell}$, and let

$L = \{0, \ldots, \ell - 1\}$ index the ordinary recursive shrink/expand setup levels. At ordinary level $h \in L$, the sender's current secret is expanded into a level vector

$$\mathbf{s}^{(h)} \in \mathcal{R}_q^{\alpha\tau\rho}.$$

The call to $\mathsf{ShrinkExpand}_{\mathcal{R}}.\mathsf{Offline}$ is the shrink/expand offline call

$$(\mathbb{S} : (r^{(h)}, \mathsf{sk_r}^{(h)}), \mathbb{R} : (\mathsf{ct_{lac}}^{(h)}, \mathsf{ct_r}^{(h)})) \leftarrow \text{ShrinkExpand.Offline}(\mathbb{S} : \mathbf{s}^{(h)}),$$

where

$$(r^{(h)}, \mathsf{ct_{lac}}^{(h)}) \leftarrow \mathsf{LacE.Enc}(\mathbf{s}^{(h)}), \qquad (\mathsf{ct_r}^{(h)}, \mathsf{sk_r}^{(h)}) \leftarrow \mathsf{LHE.Enc}_1(r^{(h)}).$$

Thus $\mathsf{ct_{lac}}^{(h)}$ is a LacE ciphertext for $\mathbf{s}^{(h)}$, and $\mathsf{ct_r}^{(h)}$ is the corresponding LHE first ciphertext for $r^{(h)}$. As specified in Section 10.1.3, this ordinary setup package is accompanied by a NIZK proof

$$\pi_{\mathsf{setup}}^{(h)}$$

for the setup statement

$$\mathsf{st}_{\mathsf{setup}}^{(h)} := (\mathsf{pp}^{(h)}, \mathsf{ct_{lac}}^{(h)}, \mathsf{ct_r}^{(h)}),$$

using the setup relation $R_{\mathsf{setup}}$. The root wrapper is one widened shrink/expand instance, denoted RootSE. It has width $\tau\rho + m$, with left input $\hat{\mathsf{sk}}_{\mathsf{r}} \in \mathcal{R}_q^{\tau\rho}$ and right public randomizer matrix $B^* \in \mathcal{R}_q^{1\times m}$. The offline root-wrapper call is

$$(\mathbb{S} : (r_{\mathsf{rt}}^{(1)}, \mathsf{sk_{r,rt}}), \mathbb{R} : (\mathsf{ct_{lac,rt}}, \mathsf{ct_{r,rt}}, B^*)) \leftarrow \text{RootSE.Offline}(\mathbb{S} : (\hat{\mathsf{sk}}_{\mathsf{r}}, B')),$$

where $B' = (\mathbf{b}_1^\top \; B^*)$. Accordingly, writing

$$\mathsf{ct_{lac,rt}} = (\mathbf{C}_{\mathsf{rt}}^{\mathsf{top}}, \mathsf{ct_{lacrt},}_L),$$

the root payload

$$(\mathbf{C}_{\mathsf{rt}}^{\mathsf{top}}, \mathsf{ct_{lacrt},}_L, \mathsf{ct_{r,rt}}, B^*)$$

has arity 4: two LacE-derived components, one LHE first ciphertext, and one uniform public randomizer. The widened root setup package is likewise accompanied by a NIZK proof

$$\pi_{\mathsf{setup}}^{\mathsf{rt}}$$

for the root setup statement

$$\mathsf{st}_{\mathsf{setup}}^{\mathsf{rt}} := (\mathsf{pp}^{\mathsf{rt}}, \mathsf{ct_{lac,rt}}, \mathsf{ct_{r,rt}}, B^*).$$

This is the arity-4 instance of the same setup-certification interface.

**Receiver-facing objects.** The receiver-facing offline message component from $\mathsf{View}_R$ is

$$\mathsf{Msg}_R^{\mathsf{off}} = ((\mathsf{ct_{lac}}^{(h)}, \mathsf{ct_r}^{(h)}, \pi_{\mathsf{setup}}^{(h)})_{h\in L}, \mathsf{ct_{lac,rt}}, \mathsf{ct_{r,rt}}, B^*, \pi_{\mathsf{setup}}^{\mathsf{rt}}).$$

After its own computation, the corrupted receiver sends some value $d'_{\mathsf{rt}}$ to the sender. This is an outgoing adversarial message and is not part of the incoming-message list. The malicious proof uses the no-golden variant of Section 10.1.1: the interactive sender does not run $\mathsf{FindGoldenSeed}$ and sends no separate online proof. The labels used as inputs to $H_{\mathsf{ct}}$ are session separated. In interactive mode we derive the label seed

$$\varsigma := H_{\mathsf{seed}}(\mathsf{id}, \mathsf{sk_{rt}}') \in \{0,1\}^\lambda.$$

This seed is only label material; the interactive protocol has no separate seed message. In non-interactive mode, the seed is the component derived from the online random oracle and labels are also bound to $d'_{\mathsf{rt}}$. In interactive mode, the sender's only online message to the receiver before the recursive expansion is

$$\mathbb{S} \to \mathbb{R} : \quad \mathsf{sk_{rt}}' \in \mathcal{R}_q$$

in the digest-knowledge variant, and

$$\mathbb{S} \to \mathbb{R} : \quad (\mathsf{sk_{rt}}', \rho_{\mathsf{chk}}) \in \mathcal{R}_q \times \{0,1\}^\lambda$$

in the random-linear-check variant, where $k'_{\mathsf{rt}} \leftarrow \mathcal{R}_q$ and

$$\mathsf{sk_{rt}}' := \mathsf{sk}_{\mathsf{r,rt}}^\top \mathbf{g}^{-1}(d'_{\mathsf{rt}}) + k'_{\mathsf{rt}}.$$

The check variant additionally samples $\rho_{\mathsf{chk}} \leftarrow \{0,1\}^\lambda$. In non-interactive mode, there is no sender message at this point. The receiver obtains the corresponding tuple as a random-oracle answer,

$$(\mathsf{sk_{rt}}', \varsigma) := H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}}) \in \mathcal{R}_q \times \{0,1\}^\lambda$$

in the digest-knowledge variant, and

$$(\mathsf{sk_{rt}}', \varsigma, \rho_{\mathsf{chk}}) := H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}}) \in \mathcal{R}_q \times \{0,1\}^\lambda \times \{0,1\}^\lambda$$

in the random-linear-check variant. Thus the incoming online message component is

$$\mathsf{Msg}_R^{\mathsf{on}} = \begin{cases} (\mathsf{sk_{rt}}', \rho_{\mathsf{chk}}) & \text{in interactive random-linear-check mode,} \\ \mathsf{sk_{rt}}' & \text{in interactive digest-knowledge mode,} \\ \emptyset & \text{in non-interactive mode.} \end{cases}$$

Let $\mathsf{RO}_R$ be the complete ordered sequence of random-oracle query/answer pairs observed by the receiver. In non-interactive mode, this includes the query $d'_{\mathsf{rt}}$ and answer $(\mathsf{sk_{rt}}', \varsigma)$ or $(\mathsf{sk_{rt}}', \varsigma, \rho_{\mathsf{chk}})$, depending on the receiver-extraction variant. The receiver's random oracle view includes the root correction-ciphertext answer

$$\mathsf{ct}_{\mathsf{k,rt}} := \begin{cases} H_{\mathsf{ct}}(\mathsf{id}, \varsigma, \mathsf{rt}, 0) & \text{if } \mathsf{mode} = \mathsf{int}, \\ H_{\mathsf{ct}}(\mathsf{id}, d'_{\mathsf{rt}}, \varsigma, \mathsf{rt}, 0) & \text{if } \mathsf{mode} = \mathsf{ni}. \end{cases}$$

For each ordinary online expand call at recursion depth $j$ and block index $i$, it includes the correction-ciphertext answer

$$\mathsf{ct}_{\mathsf{k}j,i} := \begin{cases} H_{\mathsf{ct}}(\mathsf{id}, \varsigma, j, i) & \text{if } \mathsf{mode} = \mathsf{int}, \\ H_{\mathsf{ct}}(\mathsf{id}, d'_{\mathsf{rt}}, \varsigma, j, i) & \text{if } \mathsf{mode} = \mathsf{ni}. \end{cases}$$

The optional check components are those listed in $\mathsf{View}_R$ in Section E.2.

**Sender-side honest state.** The sender's offline state consists of

$$\mathsf{OffState}_S = ((\mathbf{s}^{(h)}, r^{(h)}, \mathsf{sk_r}^{(h)})_{h \in L}, \hat{\mathsf{sk}}_{\mathsf{r}}, B', r_{\mathsf{rt}}^{(1)}, \mathsf{sk}_{\mathsf{r,rt}}).$$

The sender's offline messages to the receiver are exactly

$$\mathsf{OffMsg}_S = ((\mathsf{ct}_{\mathsf{lac}}{}^{(h)}, \mathsf{ct_r}^{(h)})_{h \in L}, \mathsf{ct}_{\mathsf{lac,rt}}, \mathsf{ct}_{\mathsf{r,rt}}, B^*, (\pi_{\mathsf{setup}}^{(h)})_{h \in L}, \pi_{\mathsf{setup}}^{\mathsf{rt}}).$$

Online, the sender receives $d'_{\mathsf{rt}}$. In the no-golden variant used for malicious security, the sender does not grind for a golden seed. In interactive mode it samples $k'_{\mathsf{rt}} \leftarrow \mathcal{R}_q$, computes

$$\mathsf{sk_{rt}}' = \mathsf{sk}_{\mathsf{r},\mathsf{rt}}^{\top}\mathbf{g}^{-1}(d'_{\mathsf{rt}}) + k'_{\mathsf{rt}},$$

and sends $\mathsf{sk_{rt}}'$, additionally sampling and sending $\rho_{\mathsf{chk}}$ in the random-linear-check variant. For correction labels in this mode, derive $\varsigma := H_{\mathsf{seed}}(\mathsf{id}, \mathsf{sk_{rt}}')$. In non-interactive mode, it computes

$$(\mathsf{sk_{rt}}', \varsigma) := H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}}),$$
$$\text{or} \qquad (\mathsf{sk_{rt}}', \varsigma, \rho_{\mathsf{chk}}) := H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}}),$$

depending on the receiver-extraction variant, and then sets

$$k'_{\mathsf{rt}} := -\mathsf{sk}_{\mathsf{r},\mathsf{rt}}^{\top}\mathbf{g}^{-1}(d'_{\mathsf{rt}}) + \mathsf{sk_{rt}}'.$$

The sender then runs the same widened root expand call and computes

$$k_{\mathsf{rt}} := \mathsf{Agg}(\mathsf{crt}((q_t)_{t\in[\rho]}, \hat{\mathbf{k}}_L), \tau \to 1) \in \mathcal{R}_q.$$

For every ordinary expand call $(j, i)$, the sender queries

$$\mathsf{ct}_{\mathsf{k}j,i} := \begin{cases} H_{\mathsf{ct}}(\mathsf{id}, \varsigma, j, i) & \text{if } \mathsf{mode} = \mathsf{int}, \\ H_{\mathsf{ct}}(\mathsf{id}, d'_{\mathsf{rt}}, \varsigma, j, i) & \text{if } \mathsf{mode} = \mathsf{ni}. \end{cases}$$

and computes its block noisy mask

$$\tilde{\mathbf{k}}_{j,i} := \mathsf{LHE.Dec}(\mathbf{k}'_{j,i}, \mathsf{ct}_{\mathsf{k}j,i}),$$

followed by the same deterministic denoising, CRT recombination, aggregation, and trimming steps as in the online algorithm. The sender's online state is therefore

$$\mathsf{OnState}_S = (d'_{\mathsf{rt}}, \mathsf{sk_{rt}}', \varsigma, k'_{\mathsf{rt}}, \rho_{\mathsf{chk}}, \mathsf{ct}_{\mathsf{krt}}, k_{\mathsf{rt}}, (\mathsf{ct}_{\mathsf{k}j,i}, \mathbf{k}'_{j,i}, \tilde{\mathbf{k}}_{j,i})_{j,i}),$$

and its online message in interactive mode is $\mathsf{sk_{rt}}'$ or $(\mathsf{sk_{rt}}', \rho_{\mathsf{chk}})$, depending on the receiver-extraction variant; in non-interactive mode it sends no online message after receiving $d'_{\mathsf{rt}}$.

### E.4 Knowledge Extraction of $x$

This section states the optional knowledge-extraction route for recovering the receiver's effective input from the public digest. The malicious construction does not rely on this route exclusively: the random-linear-check adaptor of Section 10.1.6 gives an explicit alternative with the same asymptotic efficiency. The explicit adaptor adds a final check round in the interactive protocol; in the non-interactive protocol, the check challenge is derived from the same online random-oracle response, so the round structure is unchanged.

We nevertheless include the knowledge route because it captures the apparent native security property of the digest layer. The malicious receiver does not only output a digest value; it subsequently uses that digest in the translation algebra. For this use to remain meaningful under the noise analysis, the relevant openings below the digest must be bounded. The assumption below formalizes the corresponding knowledge statement: whenever a digest is used successfully in this admissible bounded sense, an extractor can recover the bounded opening data that underlies that use. This is analogous in spirit to the random-oracle convention that a party using a hash value must have queried the oracle, but it is strictly narrower. We do not model $\mathsf{Digest}$ as a random oracle, and we do not assume that arbitrary digest outputs are invertible; the extraction target is only the bounded witness data that a successful receiver uses, explicitly or implicitly, inside this protocol.

### E.4.1 Digest-knowledge extraction from the public root

On the receiver side, the only object that enters the later translation algebra is the public digest $d_x$. Any simulation argument that wants to recover the leaves of $x$, or the branch data used by the translate/expand identities from Section 5.3, can therefore try to extract that information from the digest layer itself.

A generic proof that $d_x$ is a correct digest of a large vector $x$ is not the right tool here. The receiver input is large, and the whole point of the digest layer is to compress it. For the receiver side, we instead isolate the narrower knowledge-style property actually consumed by the simulator: whenever a malicious receiver produces a digest that admits an admissible use in the translation algebra, the corresponding branch data can be extracted from the receiver's computation.

The digest section has already done the computational work needed for this route. Corollary 1 identifies the bounded regime in which an opening can still affect the translation algebra, and Theorems 5 and 6 say that within that regime the digest is computationally tied to at most one opening tree of the type used by the protocol. What remains here is the extraction step. The point is not that arbitrary outputs of a hash-like function should be generically extractable. Rather, the claim is that in the present protocol, the only way for a malicious receiver to produce a digest that is actually useful in the later algebra is to have implicitly determined the corresponding input data below that digest. The extractor is a non-black-box device for recovering that implicit information from the adversary's computation.

This is best viewed as a knowledge-style heuristic. Related work of Canetti and Dakdouk on extractable perfectly one-way functions [CD08], and of Bitansky, Canetti, Chiesa, and Tromer on extractable collision-resistant hashes [BCCT11], gives useful language and precedent for this type of statement. In particular, the latter work explicitly studies extractability in settings built from Merkle-tree hashing. At the same time, those works do not directly prove the exact statement needed here, and we therefore state the required property directly as an assumption adapted to our protocol.

The important point is that the assumption is about recovering the admissible opening below one digest node, not about privileging any one particular implementation path by which the adversary arrived at that digest. An adversary might start from explicit opening data and compute a digest, modify internal data and compute a related digest, or arrive at the same output by some more complicated computation. The pair need not ever appear as an explicit variable in the adversary's state. The extractor is allowed to be non-black-box and to recover the admissible witness from the adversary's internal computation rather than from a literal stored opening.

To make the target explicit, let $T$ denote the accepted translate/expand transcript around one digest node $d$. Write

$$\mathsf{Use}_{\mathsf{dig}}(T, d; u_L, u_R) = 1$$

if $(u_L, u_R)$ is an admissible opening of $d$ in the sense of Definition 9, and the row evaluations in $T$ that use this node are explained by the lifted witness $w(u_L, u_R)$ from Section A. This relation is a proof-level abstraction of the algebra in Section 5.3: it means that the adversary has produced messages that are useful precisely because they behave as if evaluated on a small witness. An out-of-bound formal preimage is not a successful use in this sense, since Corollary 1 says that its row evaluations are already smudged. Thus "$d$ is later used successfully" will mean that there exists an ordered pair $(u_L, u_R)$ with $\mathsf{Use}_{\mathsf{dig}}(T, d; u_L, u_R) = 1$. By Theorem 5, such an admissible opening is computationally unique.

This is the protocol-specific reason the assumption is plausible: the extractor is not being asked to invert an arbitrary stand-alone hash output, but to recover the concrete small witness that the adversary is already using, either explicitly or implicitly, to drive the later linear translation algebra.

Accordingly, the extraction target used by the malicious proof is concrete. Fix one block of width $\mu$, and let $\ell := \log_2 \mu$ be the depth of the corresponding binary digest tree. The extractor recursively descends below the public root digest $d_x$ and attempts to recover the leaf values at depth $\ell$. If a branch stops early, the extractor records a hole there. Thus the final extracted object is a partial leaf vector

$$\hat{x} \in (R_q \cup \{\perp\})^\mu.$$

Here $\hat{x}_j \in R_q$ means that the extractor recovered the $j$-th leaf value together with a full root-to-leaf branch consistent with the concrete digest relation, while $\hat{x}_j = \perp$ means that the corresponding branch could not be fully resolved down to depth $\ell$.

It is convenient to package this as a partial tree $\mathcal{T}_x$. The root of $\mathcal{T}_x$ is labeled by $d_x$. Each extracted internal node is labeled by its ordered pair of children. A branch may terminate early, in which case no children are claimed below that node. Write

$$R_{\mathsf{dig}}(d_x; \mathcal{T}_x) = 1$$

for this partial-tree consistency condition, and let

$$\mathsf{Leaf}_\ell(\mathcal{T}_x) \in (R_q \cup \{\perp\})^\mu$$

denote the induced depth-$\ell$ partial leaf vector.

**Assumption 1** (Single-node digest extraction)**.** *There exists a probabilistic polynomial-time non-black-box extractor* $\mathsf{Ext}_{\mathsf{dig}}$*, together with functions* $t_{\mathsf{dig}}(\lambda)$ *and* $\epsilon_{\mathsf{dig}}(\lambda)$*, such that the following holds.*

*Let* $\mathbb{R}^\star$ *be any PPT receiver, and fix its auxiliary input* $\alpha$ *and random tape* $\rho$*. Let* $T$ *be an accepting transcript produced by running* $\mathbb{R}^\star(\alpha; \rho)$*. Let* $d \in \mathcal{R}_q$ *be either the root digest sent in* $T$*, or an internal digest value obtained from an earlier successful extraction while descending below that root. If* $d$ *is used admissibly in* $T$*, meaning that for some ordered pair* $(u_L, u_R)$*,*

$$\mathsf{Use}_{\mathsf{dig}}(T, d; u_L, u_R) = 1.$$

*then* $\mathsf{Ext}_{\mathsf{dig}}^{\mathbb{R}^\star}(\alpha, \rho, T, d)$ *runs in time at most* $t_{\mathsf{dig}}(\lambda)$ *and outputs an ordered pair* $(\hat{u}_L, \hat{u}_R)$ *satisfying*

$$d = \mathsf{NodeDigest}_B(\hat{u}_L, \hat{u}_R) \qquad \text{and} \qquad \|w(\hat{u}_L, \hat{u}_R)\|_2 \le \beta,$$

*except with probability at most* $\epsilon_{\mathsf{dig}}(\lambda)$*. Moreover, the output is the admissible opening used by* $T$*, up to the negligible ambiguity ruled out by Theorem 5.*

**Remark.** This assumption is stated for one binary digest node. At internal levels, the extracted pair $(u_L, u_R)$ is interpreted as the two child digests below that node. At the bottom level, the extracted pair is interpreted as the two child leaf values.

We extend the single-node extractor to a tree by starting at the public root $d_x$, recursing on every extracted child digest, and stopping a branch whenever extraction fails. At depth $\ell$, the resulting partial tree $\mathcal{T}_x$ induces a partial leaf vector

$$\hat{x} := \mathsf{Leaf}_\ell(\mathcal{T}_x) \in (R_q \cup \{\perp\})^\mu.$$

**Lemma 16** (Recursive extraction of a partial digest tree)**.** *Assume Assumption 1. Let* $d_x$ *be the public root digest of a block of width* $\mu$*, and let* $\ell := \log_2 \mu$*. Then there is a recursive non-black-box extractor* $\mathsf{Ext}_{\mathsf{tree}}$ *which, given code access to the malicious receiver that produced* $d_x$*, runs in time at most*

$$(2^\ell - 1)\, t_{\mathsf{dig}}(\lambda)$$

*and outputs a partial tree $\mathcal{T}_x$ such that, except with probability at most*

$$(2^\ell - 1)\,\epsilon_{\mathsf{dig}}(\lambda),$$

*every extracted node is consistent with the concrete digest relation. Hence $\mathcal{T}_x$ induces a partial leaf vector*

$$\hat{x} = \mathsf{Leaf}_\ell(\mathcal{T}_x) \in (R_q \cup \{\perp\})^\mu.$$

*If $\hat{x}_j \neq \perp$, the extractor recovered a full consistent root-to-leaf branch for coordinate $j$; if $\hat{x}_j = \perp$, the branch terminated early.*

*Proof.* A depth-$\ell$ binary tree has at most $2^\ell - 1$ internal nodes, so the recursive procedure invokes $\mathsf{Ext}_{\mathsf{dig}}$ at most that many times. The running-time bound follows immediately. By Assumption 1, each node extraction succeeds and returns the admissible opening except with probability $\epsilon_{\mathsf{dig}}(\lambda)$; a union bound over the calls gives the stated failure probability. Conditioned on this event, the assembled partial tree is consistent at every extracted node, and unresolved branches are exactly the branches on which the recursion stopped. □

**Role in the malicious proof.** Lemma 16 is the knowledge-extraction route for recovering the receiver input without adding the random-linear check. It does not treat $\mathsf{Digest}$ as a true random oracle. Rather, it formalizes the narrower knowledge-style property that the protocol needs: if the receiver uses a public digest $d_x$ in the later translation algebra, then the simulator may recursively extract a partial tree below $d_x$, together with explicit bounds on the extractor's running time and failure probability, and hence obtain the leaf data for every fully recovered branch.

To convert the partial leaf vector into an effective receiver input, the simulator fills unresolved leaves with a fixed canonical value, say 0:

$$x_j^{\mathsf{eff}} := \begin{cases} \hat{x}_j & \text{if } \hat{x}_j \neq \perp, \\ 0 & \text{if } \hat{x}_j = \perp. \end{cases}$$

Outside the extraction-failure event, every leaf that is fixed by an admissible use of the digest has been recovered. The remaining coordinates are not fixed by the digest-knowledge route, so the simulator may submit any valid value for them to $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$.

### E.5 Malicious security against a corrupted receiver

We now prove the first malicious theorem for the protocol. The proof first replaces the receiver-visible setup packages in the full receiver view, removing the receiver's useful information about the sender's hidden $\Delta$-side state. It then follows one of two extraction branches and finally performs a common programming step. For the random-linear-check wrapper, extraction is an active rewinding procedure that varies the check challenge while keeping the online root response fixed. For the digest-knowledge route, extraction invokes the optional knowledge extractor for the public digest. In both cases, once $x^{\mathsf{eff}}$ is known, the simulator performs one final execution and programs the random-oracle value defining $\mathsf{ct}_{\mathsf{k}}$ so that the corrupted receiver outputs the ideal value.

**Ideal functionality.** The ideal functionality is the staged CI-VOLE functionality $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ from Figure 7, used with maliciously corrupted parties. For the ring-level protocol we instantiate its parameter $\mathcal{M}$ as the relevant ring/message space, here $\mathcal{R}_q$, and use the instance width $w$. Write the protocol identifier as $\mathsf{id} = (\mathsf{sid}, \mathsf{mode})$, where $\mathsf{mode} \in \{\mathsf{ni}, \mathsf{int}\}$; the mode selects the corresponding

setter and release commands of *Figure* 7. In the corrupt-receiver case, the simulator must extract an effective receiver input $x^{\mathsf{eff}} \in \mathcal{R}_q^w$, submit it using $\mathsf{setx\text{-}ni}$ or $\mathsf{setx\text{-}int}$, and obtain the ideal receiver output $m^{\mathsf{id}} \in \mathcal{R}_q^w$ from the matching mode-specific release point. The simulator does not receive the honest sender's mask $k$, nor the sender's correlation input $\Delta$. The simulator then completes the protocol transcript so that the wrapped receiver outputs exactly $m^{\mathsf{id}}$.

**Imported correctness invariant.** The malicious hybrids use one consequence of the semi-honest correctness analysis rather than repeating that analysis. By Lemma 9 and the induction in Section B.3, once the setup objects, root value, and internal correction ciphertexts along a path are fixed, the remaining leaf expand step has the following form. The leaf correction ciphertext enters the raw receiver or sender value through an affine bijection on the correction ciphertext space, and the subsequent denoising, CRT/gadget recombination, aggregation, and trimming are the same deterministic clean-output map used in the semi-honest proof. On the complement of the no-golden correctness bad event for the malicious parameters, this clean-output map is a surjective quotient onto the retained block message space, with a kernel $K_{\mathsf{ct}}$ that depends only on the fixed transcript prefix and the block shape. Thus each output block has a fiber that is an affine coset of $K_{\mathsf{ct}}$, and a uniform correction ciphertext conditioned on a given clean block output is uniform on that coset. The same statement applies to the sender-side mask computation after fixing the accepted correction path. This imported invariant is the only place where the malicious proof uses the recursive correctness analysis.

The receiver-visible setup packages are instances of one setup-package interface. Such a package is a public payload $\mathsf{P} = (\mathsf{U}_1, \ldots, \mathsf{U}_t)$, for an allowed arity $t \in \{2, 4\}$, together with a setup proof $\pi_{\mathsf{setup}}$ for the corresponding setup relation. In this protocol the packages are

$$\mathsf{Pkg}_h := (\mathsf{ct}_{\mathsf{lac}}{}^{(h)}, \mathsf{ct}_{\mathsf{r}}{}^{(h)}, \pi_{\mathsf{setup}}^{(h)}) \quad (h \in L),$$

and

$$\mathsf{Pkg}_{\mathsf{rt}} := (\mathbf{C}_{\mathsf{rt}}^{\mathsf{top}}, \mathsf{ct}_{\mathsf{lacrt},L}, \mathsf{ct}_{\mathsf{r,rt}}, B^*, \pi_{\mathsf{setup}}^{\mathsf{rt}}).$$

These are the only sender-generated setup objects in the receiver view that carry the honest sender's hidden $\Delta$-side state. The interactive online message $\mathsf{sk_{rt}}'$, when present, is sampled in the no-golden mode of Section 10.1.1; in the random-linear-check variant it is accompanied by the check challenge $\rho_{\mathsf{chk}}$. The fresh mask $k'_{\mathsf{rt}} \leftarrow \mathcal{R}_q$ makes $\mathsf{sk_{rt}}'$ independent of the setup witness. In non-interactive mode the corresponding pair is a random-oracle answer. Thus the online root response does not require a separate $H_1$ replacement.

**Theorem 11** (Malicious security against a corrupted receiver)**.** *Fix one protocol identifier* $\mathsf{id} = (\mathsf{sid}, \mathsf{mode})$*, and let* $R^\star$ *be a PPT malicious receiver. Assume:*

1. *the adaptive setup-package simulation used in* $H_1$ *below for every allowed package arity* $t \in \{2, 4\}$*, instantiated from Sections 3.6.2 and 3.6.4 and the simulated-proof property of Section 3.8;*
2. *either the random-linear-check extraction route of Section 10.1.6 or the digest-knowledge extraction route of Section E.4.1;*
3. *the no-golden correctness condition under which the imported semi-honest correctness invariant above holds for arbitrary hash-derived correction ciphertexts;*
4. *programmable random-oracle security for the domain-separated* $H_{\mathsf{seed}}$*,* $H_{\mathsf{on}}$*, and* $H_{\mathsf{ct}}$ *query points used below.*

*Then there exists a PPT simulator* $\mathsf{Sim}_R$ *such that the real protocol execution is indistinguishable from the ideal execution with* $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$*, up to the stated ciphertext-simulation, extraction, setup-proof simulation, and random-oracle programming losses. In the ideal execution, the wrapped receiver's final output is exactly the ideal receiver output returned by* $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$*.*

**Lemma 17** (Regular cleanup map)**.** *Fix the CRT moduli $(q_t)_{t\in[\rho]}$, the gadget basis $(1, g, \ldots, g^{\tau-1})$, and the retained output coordinates. Let $\mathcal{C}$ be the raw correction-vector space before cleanup and let $\mathcal{M}$ be the retained clean-output space. The map*

$$G = \mathsf{Trim} \circ \mathsf{Agg} \circ \mathsf{CRT} \circ \mathsf{Denoise} : \mathcal{C} \to \mathcal{M}$$

*is a regular surjection.*

*Proof.* Each scale-and-round map in $\mathsf{Denoise}$ is regular: for $\delta_t = q/q_t$, the chosen rounding rule partitions the centered representatives modulo $q$ into $q_t$ cells of size $\delta_t$. Products of these maps are regular. $\mathsf{CRT}$ is a bijection. $\mathsf{Agg}$ is a surjective group homomorphism because the gadget basis contains 1; therefore all its fibers are cosets of its kernel. $\mathsf{Trim}$ is a coordinate projection, hence also a regular surjection. A composition of regular surjections is regular. □

**Lemma 18** (Regular correction programming)**.** *Let $G : \mathcal{C} \to \mathcal{M}$ be the regular cleanup map from Lemma 17. For any fixed $\alpha \in \mathcal{C}$, define*

$$F_\alpha(C) = G(\alpha + C).$$

*If $M \leftarrow \mathcal{M}$ is uniform and*

$$C^\star \leftarrow F_\alpha^{-1}(M)$$

*is sampled uniformly from the fiber above $M$, then $C^\star$ is uniform over $\mathcal{C}$.*

*Proof.* Translation by $\alpha$ is a bijection on $\mathcal{C}$, so $F_\alpha$ is a regular surjection with the same fiber size as $G$. For any $c \in \mathcal{C}$,

$$\Pr[C^\star = c] = \sum_{m\in\mathcal{M}} \Pr[M = m] \Pr[C^\star = c \mid M = m].$$

Only $m = F_\alpha(c)$ contributes. Since $M$ is uniform and every fiber has size $|\mathcal{C}|/|\mathcal{M}|$, this probability is

$$\frac{1}{|\mathcal{M}|} \cdot \frac{|\mathcal{M}|}{|\mathcal{C}|} = \frac{1}{|\mathcal{C}|}.$$

□

*Proof of Theorem 11.* The hybrids separate the receiver-only extraction argument from the final joint-output distribution. The target simulator extracts an effective receiver input $x^{\mathsf{eff}}$: from the digest itself in the digest-knowledge branch, and from a rewinding experiment in the random-linear-check branch. It sends $x^{\mathsf{eff}}$ to $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ and receives the ideal receiver output $m^{\mathsf{id}}$. It then runs the honest receiver continuation internally to determine which leaf $H_{\mathsf{ct}}$ queries affect the final output and what fixed offset they induce. For each such query, it samples the oracle answer from the appropriate fiber $F_{\mathsf{pref}}^{-1}(m^{\mathsf{id}})$, so that the honest receiver computation outputs $m^{\mathsf{id}}$.

**$H_0$: real execution.** Hybrid $H_0$ is the real malicious execution with corrupted receiver $R^\star$ and honest sender. The setup packages and setup proofs are real, all random-oracle answers are sampled lazily as uniform strings in their output spaces, and the honest sender output and receiver output are the real protocol outputs.

**$H_1$: simulated setup, extraction, and real-output programming.** Hybrid $H_1$ replaces the receiver-visible setup packages with simulated $\mathsf{LacE}/\mathsf{LHE}$ packages and simulated setup proofs. For each package instance, the replacement uses the adaptive simulators from Sections 3.6.2 and 3.6.4: $\mathsf{LacE.Sim}_1$ generates the simulated $\mathsf{LacE}$ ciphertext state and $\mathsf{LacE.Sim}_2$ answers the adaptive evaluation leakages, while $\mathsf{LHE.Sim}_1, \mathsf{LHE.Sim}_2, \mathsf{LHE.Sim}_3$ generate the simulated initial ciphertext, adaptive second ciphertexts, and adaptive functional keys. Public non-ciphertext components, such as $B^*$, keep their real distribution, and the setup proof is replaced by the simulated proof for the resulting public statement. This is an all-at-once adaptive replacement for each package instance; the finite tuple of package instances is handled by the standard product/hybrid loss over those independent instances.

After the setup replacement, $H_1$ extracts $x^{\mathsf{eff}}$ using one of the two branches below. It then completes a sender-facing execution with the real external sender while acting as an honest receiver: it generates a fresh digest $d_x^\star$ consistent with $x^{\mathsf{eff}}$, using independent digest randomness, and sends $d_x^\star$ to the sender through the correct mode-specific interface. This execution fixes a sender-facing label seed $\varsigma^\star$, and the simulator computes the receiver value $m^{\mathsf{real}}$ from that honest-receiver execution. The malicious receiver and distinguisher are not given $d_x^\star$, $\varsigma^\star$, or the sender-facing correction-oracle labels. Instead of answering the designated leaf $H_{\mathsf{ct}}$ queries with independent random strings, $H_1$ programs them from the regular fibers that force the deterministic honest receiver computation to output $m^{\mathsf{real}}$. Thus the auxiliary extraction experiment may omit the honest sender's output, but the completed $H_1$ experiment preserves the joint output distribution seen by a distinguisher.

**Digest extraction branch.** The digest-knowledge variant performs no rewinding. Once the receiver's public root digest $d_x$ is fixed, the simulator applies Lemma 16. The extractor returns a partial digest tree and its induced partial leaf vector $\hat{x} \in (\mathcal{R}_q \cup \{\perp\})^w$. The simulator defines

$$x_j^{\mathsf{eff}} := \begin{cases} \hat{x}_j & \text{if } \hat{x}_j \neq \perp, \\ 0 & \text{if } \hat{x}_j = \perp. \end{cases}$$

Bounded-opening uniqueness says that any accepting digest opening agrees with this extracted partial tree except with the digest-extraction loss from Section E.4. This branch does not use the random-linear rewinding experiment below.

**Random-linear-check extraction experiment.** This paragraph applies only to the random-linear-check variant. The simulator uses the following auxiliary experiment. Fix the coins of $R^\star$, so that it is a deterministic receiver.

Let $P_0$ be the receiver transcript sampled by this experiment, and let $\mathsf{out}_R(P_0)$ be the receiver output determined by $P_0$. Here the receiver transcript means the receiver's formal inputs together with every protocol message and random-oracle answer delivered to $R^\star$: setup packages, setup proofs, online messages, correction-oracle answers, and check challenges or check-hash answers. It does not include messages generated by $R^\star$, since those are determined by its coins and incoming transcript. The distinguisher for this auxiliary experiment receives $(P_0, \mathsf{out}_R(P_0))$, but not the honest sender's output. Thus the experiment tests whether the receiver's behavior changes; it does not test whether the receiver output is jointly consistent with the sender output.

Define $P_1$ from $P_0$ by replacing the receiver-visible setup packages with the corresponding adaptive ciphertext-simulator outputs and simulated setup proofs. The random-oracle answers, including correction answers, remain ordinary oracle answers in this internal experiment.

**Lemma 19** (Receiver-transcript setup replacement)**.** *In the random-linear-check auxiliary experiment above,*

$$(P_0, \mathsf{out}_R(P_0)) \approx_c (P_1, \mathsf{out}_R(P_1)).$$

*Proof.* With the coins of $R^\star$ fixed, $\mathsf{out}_R(P_b)$ is a deterministic function of $P_b$, so it suffices to compare the receiver transcripts. After setup, the receiver-visible values are random-oracle answers or fresh sender-sampled masks with uniform marginals. The only non-oracle point is the interactive root response: for fixed public values,

$$\mathsf{sk_{rt}}' = \mathsf{sk}_{\mathsf{r,rt}}^{\top}\mathbf{g}^{-1}(d'_{\mathsf{rt}}) + k'_{\mathsf{rt}}$$

with $k'_{\mathsf{rt}} \leftarrow \mathcal{R}_q$. Translation by $\mathsf{sk}_{\mathsf{r,rt}}^{\top}\mathbf{g}^{-1}(d'_{\mathsf{rt}})$ is a bijection on $\mathcal{R}_q$, so $\mathsf{sk_{rt}}'$ is uniform and independent of the hidden setup state. In the random-oracle online mode the value returned by $H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}})$ is uniform by definition.

Thus, in this auxiliary experiment, the setup packages are the only receiver-visible values that can reveal the sender's hidden setup state. Since the honest sender's output is not given to the distinguisher, joint consistency with that output is outside the experiment. For each package instance, the adaptive LacE/LHE simulators replace the whole receiver-visible package interface: ciphertext state, adaptive evaluations, and adaptive keys. The finite tuple of package instances is handled by a standard product/hybrid over the package instances, and the accompanying setup proofs are replaced by their simulated proofs. Deterministic evaluation of $R^\star$ on the resulting transcript preserves indistinguishability. □

The lemma also gives that $P_1$ is independent of the honest sender's $\Delta$, except for the accumulated setup-replacement losses.

**Random-linear-check extraction.** The simulator extracts from the challenge-free prefix inside $P_1$. Let $T$ be this prefix. In interactive mode, $T$ contains the receiver's incoming transcript up to and including the online root response $\mathsf{sk_{rt}}'$, but excludes $\rho_{\mathsf{chk}}$. We parse the sender's response as the challenge-independent part $\mathsf{sk_{rt}}'$ followed by the check challenge, even if they are delivered in one packet. The label seed $\varsigma = H_{\mathsf{seed}}(\mathsf{id}, \mathsf{sk_{rt}}')$ is therefore fixed by $T$. In random-oracle online mode, the simulator stops before answering $H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}})$, fixes the first two components $(\mathsf{sk_{rt}}', \varsigma)$, and treats them as part of $T$. Each probe answers the same query as

$$H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}}) = (\mathsf{sk_{rt}}', \varsigma, \rho^{(r)}_{\mathsf{chk}})$$

with a fresh independent challenge field.

The simulator stores $R^\star$'s deterministic state at the end of $T$. For probe $r = 1, 2, \ldots$, it resumes from this state using the same setup messages, the same online values fixed by $T$, and scratch random-oracle tables for the probe suffix. It samples a fresh challenge $\rho^{(r)}_{\mathsf{chk}}$, derives the check coefficients

$$\chi^{(r,t)} \in \mathcal{R}_q^w, \qquad t \in [\tau_{\mathsf{chk}}],$$

and runs the receiver to the end of the check. If the probe accepts, the receiver messages $(u'^{(r)}_t)_{t\in[\tau_{\mathsf{chk}}]}$ give the linear equations

$$u'^{(r)}_t = \sum_{i\in[w]} \chi_i^{(r,t)} x_i + x_{w+t} \qquad t \in [\tau_{\mathsf{chk}}].$$

These equations are interpreted over the check domain: over $\mathcal{R}_q$ for the ring-form wrapper, or after the chosen field/CRT projection for a field-form wrapper. Thus the unknown vector is

$$X = (x_1, \ldots, x_w, x_{w+1}, \ldots, x_{w+\tau_{\mathsf{chk}}})$$

in that domain.

The simulator keeps accepting probes until it has $N_{\mathsf{ext}}$ of them, or until the rewinding budget is exhausted. Choose $N_{\mathsf{ext}}$ so that the induced linear system in $X$ has full rank except with probability $\epsilon_{\mathsf{rank}}$. For example, over a field $\mathbb{F}$ with fully random coefficients, it is enough to take

$$\tau_{\mathsf{chk}}(N_{\mathsf{ext}} - 1) \geq w + \lambda,$$

which gives $\epsilon_{\mathsf{rank}} \leq |\mathbb{F}|^{-\lambda}$ by the usual kernel-counting bound for the block matrix $(\chi^{(r,t)} \mid e_t)$. For the ring-form check, the same statement uses the rank guarantee of the chosen ring-linear hash family or of the CRT projections used by the check.

Let $B_\Delta(T)$ be the event that this fixed challenge-free prefix admits two accepting completions whose final receiver-side VOLE outputs are valid for two different application inputs under the same hidden sender state. Since $T$ is taken from $P_1$, it is $\Delta$-independent except for the setup-replacement losses, and two such completions would subtract to a nonzero relation involving the hidden $\Delta$. Therefore

$$\Pr[B_\Delta(T)] \leq \epsilon_\Delta.$$

Conditioned on $\neg B_\Delta(T)$, all accepting probes are explained by one effective input. Once the probe equations have full rank, solving them recovers this input, denoted $x^{\mathsf{eff}}$.

The extraction loss is the union of four events: $B_\Delta(T)$, failure to obtain $N_{\mathsf{ext}}$ accepting probes, rank failure of the probe matrix, and check/hash soundness failure. Thus it is bounded by

$$\epsilon_\Delta + \epsilon_{\mathsf{rew}} + \epsilon_{\mathsf{rank}} + \epsilon_{\mathsf{chk}}.$$

The probe suffixes are internal to extraction; they are not part of the final receiver transcript.

**$H_1$ final transcript.** After extraction, $H_1$ runs one final receiver execution, using simulated setup packages and simulated setup proofs. In the random-linear-check branch the incoming receiver transcript before the programmed leaf answers is distributed as in $P_1$; in the digest-knowledge branch it is produced by the same setup simulators without running the rewinding experiment. The simulator chooses the online root response or programs $H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}})$ with a fresh mode-specific label seed. If a designated correction label has already been queried before this seed is fixed in the receiver's view, the simulator enters the prequery bad event.

The designated labels are exactly the leaf-level correction labels. Root and internal correction values are not programmed in this step: by the time a leaf answer is chosen, all root and internal values on the path have already been fixed and are part of the affine offset for that leaf computation. Conditioned on no prequery, all designated $H_{\mathsf{ct}}$ labels are fresh. If $N_{\mathsf{leaf}}$ is the number of leaf blocks and $Q_{\mathsf{ct}}$ is the number of $H_{\mathsf{ct}}$ queries made before the mode-specific label seed is fixed in the receiver's view, then for this invocation

$$\Pr[\mathsf{Bad}_{\mathsf{preq}}] \leq \frac{Q_{\mathsf{ct}} N_{\mathsf{leaf}}}{2^\lambda},$$

plus the negligible probability of reusing a label seed across invocations.

The value $m^{\mathsf{real}}$ comes from the simulator's sender-facing execution with the real external sender. In that execution the simulator uses the fresh $x^{\mathsf{eff}}$-consistent digest $d_x^\star$, obtains the corresponding

sender-facing label seed $\varsigma^\star$, and completes the protocol as an honest receiver. Neither $d_x^\star$ nor $\varsigma^\star$ is delivered in the malicious receiver transcript. Let $\mathsf{Bad}_{\mathsf{seed}^\star}$ be the event that $R^\star$, or the distinguisher through its oracle interface, queries a leaf $H_{\mathsf{ct}}$ label containing $\varsigma^\star$. If $Q_{\mathsf{ct}}$ also bounds these queries, then

$$\Pr[\mathsf{Bad}_{\mathsf{seed}^\star}] \le \frac{Q_{\mathsf{ct}} N_{\mathsf{leaf}}}{2^\lambda},$$

again up to the negligible probability of a repeated label seed. Conditioned on no $\mathsf{Bad}_{\mathsf{seed}^\star}$, the oracle entries used inside the sender-facing completion with digest $d_x^\star$ are not reachable from the receiver-visible oracle table.

Let $\alpha_{\mathsf{pref}}$ be the fixed affine offset contributed by the already fixed values for the designated leaf computations, and set

$$F_{\mathsf{pref}} = F_{\alpha_{\mathsf{pref}}}$$

as in Lemma 18. The simulator samples

$$C^\star \leftarrow F_{\mathsf{pref}}^{-1}(m^{\mathsf{real}})$$

uniformly from the fiber above the real-protocol output and programs the designated $H_{\mathsf{ct}}$ labels to the components of $C^\star$.

**Lemma 20** (Real-output programming preserves oracle distribution)**.** *Conditioned on no* $\mathsf{Bad}_{\mathsf{preq}}$ *and no* $\mathsf{Bad}_{\mathsf{seed}^\star}$*, replacing the ordinary leaf* $H_{\mathsf{ct}}$ *answers by the programmed vector* $C^\star$ *does not change the receiver/distinguisher view.*

*Proof.* Fix the receiver transcript before the designated leaf answers and fix $\alpha_{\mathsf{pref}}$. In the ordinary execution, the designated leaf answers form a uniform vector $C$ over the correction-ciphertext space. Then $F_{\mathsf{pref}}(C)$ is uniform over the receiver-output space because $F_{\mathsf{pref}}$ is a regular surjection by Lemma 18. Conditioned on no $\mathsf{Bad}_{\mathsf{seed}^\star}$, the oracle answers used to sample $m^{\mathsf{real}}$ are independent of the receiver transcript and of the receiver-visible oracle table. Once the sender-side non-oracle state is fixed, the honest-receiver completion with the real sender applies the same fixed-offset cleanup form to fresh correction answers under $\varsigma^\star$. The fixed offset does not affect uniformity, and regularity of the cleanup map gives a uniform receiver-output value. Therefore $m^{\mathsf{real}}$ is a uniform target in the receiver/distinguisher view. Sampling $C^\star$ uniformly from $F_{\mathsf{pref}}^{-1}(m^{\mathsf{real}})$ gives a uniform correction vector by the fiber statement in Lemma 18. Thus the programmed answers have exactly the same distribution as ordinary fresh random-oracle answers on all labels visible to the receiver/distinguisher. □

Substituting these correction ciphertexts into the deterministic receiver computation makes the wrapped receiver output exactly $m^{\mathsf{real}}$. Therefore a distinguisher that receives both parties' outputs sees the same joint output distribution in $H_0$ and $H_1$, except for the setup-replacement, extraction, prequery, and hidden-sender-seed query losses.

For the random-linear-check wrapper, the simulator handles the final check using the extracted input. When the final receiver message $u'_t$ agrees with

$$u'_t = \sum_{i\in[w]} \chi_i^{(t)} x_i^{\mathsf{eff}} + x_{w+t}^{\mathsf{eff}},$$

the simulator computes the receiver-side checked value from $m^{\mathsf{real}}$ and returns the corresponding $H_{\mathsf{chk}}$-hash. If the equation is violated, the binding and check-soundness bounds for the extraction procedure imply that the receiver accepts only with the stated check/hash error.

$H_2$**: ideal-output programming.** Hybrid $H_2$ is the target simulation. It replaces the real sender-side sampling of $m^{\mathsf{real}}$ in $H_1$ with the ideal call: the simulator submits $x^{\mathsf{eff}}$ to $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ using the mode-specific receiver command

$$\begin{cases} \mathcal{F}_{\mathsf{CI\text{-}VOLE}}.\mathsf{setx\text{-}ni}(\mathbb{R} : \mathsf{sid}, x^{\mathsf{eff}}) & \text{if } \mathsf{mode} = \mathsf{ni}, \\ \mathcal{F}_{\mathsf{CI\text{-}VOLE}}.\mathsf{setx\text{-}int}(\mathbb{R} : \mathsf{sid}, x^{\mathsf{eff}}) & \text{if } \mathsf{mode} = \mathsf{int}. \end{cases}$$

In non-interactive mode, the call returns the corrupted receiver's ideal output $m^{\mathsf{id}}$ immediately. In interactive mode, the honest sender has fixed its mask through $\mathsf{setk\text{-}int}$, and the simulator obtains $m^{\mathsf{id}}$ at the matching release point. For fixed $x^{\mathsf{eff}}$, the honest sender in the real protocol and $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ sample the same VOLE output distribution; in particular the receiver component is uniform over the receiver-output space, including the padded check coordinates in the random-linear-check variant. The simulator therefore repeats the $H_1$ programming step with

$$C^\star \leftarrow F_{\mathsf{pref}}^{-1}(m^{\mathsf{id}}).$$

Lemma 18 again gives that $C^\star$ is distributed as an ordinary fresh random-oracle answer vector, and the deterministic receiver computation outputs $m^{\mathsf{id}}$.

**Hybrid conclusion.** The transition $H_0 \to H_1$ is justified by setup simulation, the inline receiver-transcript setup-replacement lemma for the random-linear-check auxiliary experiment, the digest extractor in the digest-knowledge branch, the random-linear extraction loss in the random-linear-check branch, and regular correction programming with $m^{\mathsf{real}}$. This transition is secure for distinguishers that receive the joint sender/receiver output because the sender-side oracle labels used to sample $m^{\mathsf{real}}$ are hidden from the receiver view, and $H_1$ programs the receiver output to the real-protocol value paired with the honest sender output. The transition $H_1 \to H_2$ replaces that real-protocol output sampling with the identical ideal VOLE output distribution and uses the same regular-correction programming. Therefore the real and ideal receiver views are indistinguishable up to the setup-simulation, setup-proof simulation, extraction, prequery, hidden-sender-seed query, regular-correction programming, and no-golden correctness losses. □

## E.6 Malicious security against a corrupted sender

For a corrupted sender, the setup proofs from Section 10.1.3 certify that the public recursive ciphertext package is explained by one bounded sender state. By the NIZK argument-of-knowledge interface of Section 3.8, the simulator extracts that state. In particular, it extracts the global correlation $\Delta^{\mathsf{eff}}$, the certified recursive setup state, and the root state $(\mathbf{r}_{\mathsf{rt}}^{(1)}, \mathsf{sk}_{\mathsf{r,rt}})$.

**Theorem 12** (Malicious sender security)**.** *Assume the setup proofs of Section 10.1.3 satisfy the argument-of-knowledge and zero-knowledge properties of Section 3.8, assume the randomized-root hiding lemma of Lemma 7 with root min-entropy $\lambda_{\mathsf{rt}}$ before the receiver releases $d'_{\mathsf{rt}}$, and assume the no-golden correctness condition under which the imported semi-honest correctness invariant above holds for arbitrary hash-derived correction ciphertexts. Then the malicious protocol securely realizes $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ against a malicious sender.*

*Proof.* Let $S^\star$ be a PPT malicious sender. The simulator runs $S^\star$ and mediates its interaction with the honest receiver.

**$H_0$: real execution.** Hybrid $H_0$ is the real malicious execution with $S^\star$ and an honest receiver, using the no-golden malicious protocol of Section 10.1. The sender supplies the ordinary setup packages, the widened root package, and the accompanying setup proofs. If any proof fails verification, the honest receiver aborts.

**$H_1$: extract the certified setup.** In $H_1$, whenever all setup proofs verify, the simulator invokes the NIZK extractor from Section 3.8 on every accepting setup proof. Except with setup-extraction error $\epsilon_{\mathsf{zk}}$, it obtains witnesses satisfying $R_{\mathsf{setup}}$ for all setup packages. This only augments the experiment with extracted witnesses; the adversary sees the same transcript as in $H_0$.

The extracted witnesses determine one effective sender setup. We write

$$(\Delta^{\mathsf{eff}}, w, \mathsf{st}_{\mathbb{S}}^{\mathsf{eff}}) := \mathsf{ParseSetup}(W_{\mathsf{setup}}),$$

where $\mathsf{ParseSetup}$ is the deterministic parsing map induced by the wrapper and the setup relation. Concretely, at the top level it inverts the public label-to-ring embedding used in Figure 4; the remaining entries of $\mathsf{st}_{\mathbb{S}}^{\mathsf{eff}}$ are the certified recursive masks, bridge states, and root state extracted from the setup witnesses. The relation $R_{\mathsf{setup}}$ guarantees that these objects are exactly the objects used by the public ciphertext packages. The simulator now submits

$$\mathcal{F}_{\mathsf{CI\text{-}VOLE}}.\mathsf{set\text{-}delta}(\mathbb{S} : \Delta^{\mathsf{eff}}, w)$$

to the ideal functionality.

**$H_2$: name the sender-induced root mask.** Hybrid $H_2$ is distributed identically to $H_1$. Its only purpose is to name the values that will be used in the ideal switch. Let $d'_{\mathsf{rt}}$ denote the public digest sent by the honest receiver. The proof uses only this public ring element; it is not computed from, opened to, or used to recover the receiver's private input.

In interactive mode, an accepted sender response $\mathsf{sk_{rt}}'$ defines the root-side sender mask

$$k'_{\mathsf{rt}} := \mathsf{sk_{rt}}' - \mathsf{sk}_{\mathsf{r},\mathsf{rt}}^{\top}\mathbf{g}^{-1}(d'_{\mathsf{rt}}).$$

In non-interactive mode, the online random oracle first fixes $(\mathsf{sk_{rt}}', \varsigma)$ at $H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}})$, and the same equation defines $k'_{\mathsf{rt}}$. This changes no messages and no distributions.

**$H_3^{\mathsf{int}}$: extract the interactive sender mask.** This hybrid applies when $\mathsf{mode} = \mathsf{int}$. The simulator maintains the table for $H_{\mathsf{ct}}$. If $S^\star$ queries

$$H_{\mathsf{ct}}(\mathsf{id}, s, j, i)$$

for some $s \in \{0,1\}^\lambda$ before sending $\mathsf{sk_{rt}}'$ as its online response, the simulator returns a fresh value of the correct ciphertext type and records it at that exact label. After $S^\star$ sends the online response $\mathsf{sk_{rt}}'$, derive $\varsigma := H_{\mathsf{seed}}(\mathsf{id}, \mathsf{sk_{rt}}')$. The sender-side output is computed using the recorded values at the accepted labels

$$(\mathsf{id}, \varsigma, j, i).$$

For any accepted label that was not queried, the simulator samples the corresponding $H_{\mathsf{ct}}$ value at that time. Queries with other second components are ordinary random-oracle queries and do not affect the mask that the simulator extracts from the accepted $\mathsf{sk_{rt}}'$ response. The only structural property needed of the internal correction labels is uniqueness: after $(\mathsf{id}, \varsigma)$ is fixed, every root, internal, and leaf correction point has a single accepted oracle label, and domain separation prevents

collisions with other identifiers or modes. If the recursive algorithm reaches the same accepted label twice, both visits use the same recorded oracle answer.

After setup extraction and the definition of $k'_{\mathsf{rt}}$, the sender-side output on each leaf block is a deterministic function of the accepted correction ciphertexts. Concretely, the lower-level expand algorithm computes a raw LHE decryption

$$\tilde{\mathbf{k}}_i(\mathsf{ct}_{\mathsf{k}\ell,i}) = \mathsf{LHE.Dec}(k_i^{\mathsf{loc}}, \mathsf{ct}_{\mathsf{k}\ell,i}),$$

where $k_i^{\mathsf{loc}}$ is the carried sender key determined by the extracted setup state, the accepted root mask, and the accepted internal correction ciphertexts on the path to leaf $i$. It then applies the same denoising and output post-processing used by the honest algorithm. Let

$$\mathsf{Post}_{\mathsf{ct}}^S : \mathcal{C}_{\mathsf{ct}} \to \mathcal{M}^\mu$$

be this sender-side post-processing map. By the imported correctness invariant, this is the sender-side instance of the same quotient map after the accepted correction path has been fixed. For each leaf block $i$, the $H_{\mathsf{ct}}$ value at the accepted label therefore determines one mask block

$$\mathbf{k}_i^{\mathsf{eff}} := \mathsf{Post}_{\mathsf{ct}}^S(\tilde{\mathbf{k}}_i(\mathsf{ct}_{\mathsf{k}\ell,i})).$$

The concatenation of these blocks is the extracted mask $\mathbf{k}^{\mathsf{eff}}$. No oracle programming is needed in this branch. In the ideal hybrid, the simulator submits this mask to the interactive functionality as the corrupted sender's chosen mask.

**$H_3^{\mathsf{ni}}$: couple the non-interactive sender mask.** This hybrid applies when $\mathsf{mode} = \mathsf{ni}$. The non-interactive functionality samples the sender mask itself, so the simulator does not extract a sender-chosen mask in this branch. Instead, it uses the hidden release digest to couple the sender's later view to the ideal mask.

The honest receiver's ideal setx-ni call is made by the honest receiver, not by the simulator. The simulator does not learn the receiver input or the receiver output. The value it must match is the sender mask $\mathbf{k}$ sampled by the ideal functionality and later released to the corrupted sender. The relevant random-oracle inputs are controlled by the receiver's release message $d'_{\mathsf{rt}}$. Before that message is delivered to $S^\star$, the randomized-root hiding lemma and its leftover-hash instantiation make $(B^*, d'_{\mathsf{rt}})$ statistically close to a source in which $d'_{\mathsf{rt}}$ has conditional min-entropy at least $\lambda_{\mathsf{rt}}$ given the sender's prefix view. Thus a prior query to $H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}})$, or to a derived label

$$H_{\mathsf{ct}}(\mathsf{id}, d'_{\mathsf{rt}}, \varsigma, j, i),$$

must guess the receiver's release digest. If $Q_{\mathsf{on}}$ and $Q_{\mathsf{ct}}$ are the sender's pre-release query counts, and $N_{\mathsf{ct}}$ is the number of correction labels used by the invocation, then

$$\Pr[\mathsf{Bad}_{\mathsf{preq}}^S] \le (Q_{\mathsf{on}} + Q_{\mathsf{ct}} N_{\mathsf{ct}}) 2^{-\lambda_{\mathsf{rt}}} + \epsilon_{\mathsf{root\text{-}hide}}.$$

The simulator aborts into this bad event if such a query occurred. Otherwise, after the ideal setx-ni call has sampled the pending mask and before $d'_{\mathsf{rt}}$ is delivered to $S^\star$, the labels that will be used to expose this mask are fresh in the final random-oracle table. This is the same freshness invariant as above, now for the sender's VOLE component: if a label tied to the released mask was already present, it was a pre-release guess of $d'_{\mathsf{rt}}$ and the execution has already entered $\mathsf{Bad}_{\mathsf{preq}}^S$. The simulator fixes the $H_{\mathsf{on}}$ answer as

$$H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}}) = (\mathsf{sk_{rt}}', \varsigma) \quad \text{or} \quad H_{\mathsf{on}}(\mathsf{id}, d'_{\mathsf{rt}}) = (\mathsf{sk_{rt}}', \varsigma, \rho_{\mathsf{chk}})$$

depending on the receiver-extraction variant. It then sets the $H_{\mathsf{ct}}$ values at the labels $(\mathsf{id}, d'_{\mathsf{rt}}, \varsigma, j, i)$ so that the sender-side expand computation yields the same $\mathbf{k}$. The visible $H_{\mathsf{ct}}$ answers are sampled using the same affine-fiber sampler as in the corrupted-receiver proof. After the setup state, root mask, and accepted internal correction ciphertexts are fixed, the leaf correction ciphertext enters the sender's raw value through an affine bijection, and the imported invariant gives the same fixed-kernel quotient for the clean mask block. Sampling uniformly from the corresponding coset gives exactly the distribution of a uniform random-oracle answer conditioned on the released sender mask. Since the ideal functionality samples $\mathbf{k}$ uniformly, the conditioned oracle answers have the same joint distribution as the real answers, except for the prequery and correctness bad events already named. The induced value $k'_{\mathsf{rt}} = \mathsf{sk}_{\mathsf{rt}}{}' - \mathsf{sk}_{\mathsf{r},\mathsf{rt}}^{\top}\mathbf{g}^{-1}(d'_{\mathsf{rt}})$ is the effective non-interactive online mask for this execution.

**$H_4$: switch the receiver to the ideal functionality.** The honest receiver now interacts with $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ using its own private input. The simulator does not submit this input and does not learn it.

In interactive mode, the simulator submits the extracted mask $\mathbf{k}^{\mathsf{eff}}$ using

$$\mathcal{F}_{\mathsf{CI\text{-}VOLE}}.\mathsf{setk\text{-}int}(\mathbb{S} : \mathsf{sid}, \mathbf{k}^{\mathsf{eff}}).$$

The honest receiver submits its input through $\mathsf{setx\text{-}int}$, and the simulator invokes

$$\mathcal{F}_{\mathsf{CI\text{-}VOLE}}.\mathsf{release\text{-}int}(\mathbb{S} : \mathsf{sid})$$

to deliver the ideal output to the receiver. In non-interactive mode, the honest receiver's $\mathsf{setx\text{-}ni}$ call has already fixed both its ideal output and the sender mask; when the protocol reaches the release point, the simulator delivers the released mask to $S^{\star}$.

By the setup relation, the extracted $\Delta^{\mathsf{eff}}$ and recursive state are exactly the state certified by the public setup package. By the online definition of $k'_{\mathsf{rt}}$, the root relation is satisfied. By the affine leaf equations above and the correctness theorem for the recursive expand computation, the real receiver computation would have produced exactly the same value that $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ returns for the receiver's private input and the extracted sender state. Therefore replacing the real receiver continuation by the ideal receiver continuation is perfect outside denoising failures, the no-golden correctness bad event, and $\mathsf{Bad}^{S}_{\mathsf{preq}}$.

**$H_5$: replace the root digest in the sender view.** At this point the honest receiver's output is supplied by the ideal functionality, so the simulator no longer needs a root digest tied to the receiver's private input. It samples a dummy value

$$\widehat{d}_{\mathsf{rt}} \leftarrow \mathcal{R}_q$$

and uses $\widehat{d}_{\mathsf{rt}}$ as the root digest delivered to $S^{\star}$, with the $H_{\mathsf{on}}$ and $H_{\mathsf{ct}}$ labels formed consistently using that value. By the randomized-root hiding lemma of Lemma 7, instantiated through the ring leftover-hash bound of Theorem 4, the real pair $(B^*, d'_{\mathsf{rt}})$ is statistically close to $(B^*, \widehat{d}_{\mathsf{rt}})$. This replacement affects only the corrupted sender's view.

The final hybrid is exactly the ideal execution with a corrupted sender. Summing the setup-extraction, root-hiding, non-interactive prequery, random-oracle programming, and correctness losses completes the proof. □

## E.7 Putting the Corruption Cases Together

The preceding theorems are stated for one fixed protocol identifier $\mathsf{id} = (\mathsf{sid}, \mathsf{mode})$. With the two corruption cases isolated, we first combine them into a malicious security statement for that fixed identifier. We then lift the statement to executions with many identifiers by a hybrid over identifiers.

**Theorem 13** (Malicious security for one fixed identifier)**.** *Assuming Theorems 11 and 12, the protocol realizes $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ with malicious security for one fixed $\mathsf{id} = (\mathsf{sid}, \mathsf{mode})$.*

*Proof.* The two theorems cover the two possible static corruptions for the fixed identifier. Both simulators use the same ideal functionality syntax from Figure 7: first set-delta fixes the effective global correlation and width. In non-interactive mode the receiver uses setx-ni and later release-ni; in interactive mode the sender uses setk-int, the receiver uses setx-int, and the sender completes the call with release-int.

If the receiver is corrupted, Theorem 11 extracts the effective receiver input and programs the honest-sender correction values so that the receiver obtains the ideal output. If the sender is corrupted, Theorem 12 extracts the certified sender setup, derives $\Delta^{\mathsf{eff}}$, and aligns the sender mask and honest-receiver output with the ideal functionality. The bad events are disjointly accounted for by the corresponding proof: receiver-input extraction and $\Delta$-hiding losses occur only in the corrupted-receiver case, while setup-proof extraction, no-golden correctness, and sender-side oracle programming losses occur only in the corrupted-sender case. A union bound over the relevant bad events gives the fixed-identifier malicious security statement. □

**Theorem 14** (Full malicious CI-VOLE theorem)**.** *Assume the fixed-identifier theorem from Theorem 13. Then the full protocol securely realizes $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ with malicious security for polynomially many identifiers.*

*Proof.* A full execution may contain many identifiers. The setup state and global correlation are fixed once by set-delta. Thereafter, online messages, random-oracle labels, ideal-functionality entries, and release commands are keyed by id. The multi-identifier simulator therefore performs the setup step only once: in the corrupted-receiver case it uses the single simulated setup view for all identifiers, and in the corrupted-sender case it extracts one certified reusable setup witness and submits the corresponding $\Delta^{\mathsf{eff}}$ through the one-time setup command

$$\mathcal{F}_{\mathsf{CI\text{-}VOLE}}.\mathsf{set\text{-}delta}(\mathbb{S} : \Delta^{\mathsf{eff}}, w).$$

It also maintains one global random-oracle table.

The hybrids then replace identifiers one at a time while keeping that shared setup and shared oracle table fixed. For the current identifier, the fixed-id theorem supplies the online transcript replacement and the matching ideal-functionality commands. Oracle entries whose labels contain the current id are handled by that replacement; entries for other identifiers remain in the global table and are answered exactly as before. Since every online random-oracle label and every functionality-table entry contains id, cross-identifier correlations appear only through the shared setup already handled at the beginning or through collisions in the domain-separated oracle labels. Those collision events are included in the same prequery/collision bad events as above. A union bound over the polynomially many identifiers and their bad events gives the full multi-identifier statement. Finally, the deterministic $\mathsf{LogVOLE}_{\mathbb{Z}}$ wrapper applies the label-to-ring and ring-to-label maps from Figure 4, yielding the stated $\mathcal{F}_{\mathsf{CI\text{-}VOLE}}$ functionality over the top-level message space. □

- PARAMETERS: $\mathsf{LogVOLE}$ over $\mathbb{Z}_p$, circuit $C_{\mathsf{st}}$, binding commitment $\mathsf{Com}$, and $\tau_{\mathsf{chk}}$ linear-check repetitions.
- INPUT: $\mathbb{P}$ : $\mathbf{w}$. $\mathbb{V}$ : public statement.
- PROTOCOL:

1. $\mathbb{V}$: sample $\Delta \xleftarrow{\$} \mathbb{Z}_p$.
2. $\mathbb{V}$: compute $(T_{\mathsf{set}}, \omega_{\mathbb{V}}) \leftarrow \mathsf{FreshSetup}_{\mathsf{LogVOLE}}(\Delta)$.
3. $\mathbb{V} \to \mathbb{P} : T_{\mathsf{set}}$.
4. $\mathbb{P}$: compute $\mathbf{x} = (x_1, \ldots, x_w) \leftarrow C_{\mathsf{st}}(\mathbf{w})$.
5. $\mathbb{P}$: sample $u_1, \ldots, u_{\tau_{\mathsf{chk}}}, V_1^* \xleftarrow{\$} \mathbb{Z}_p$.
6. $\mathbb{P}$: set $\mathbf{x}' := (x_1, \ldots, x_w, u_1, \ldots, u_{\tau_{\mathsf{chk}}}, V_1^*)$, $j_t := w + t$, and $r := w + \tau_{\mathsf{chk}} + 1$.
7. $\mathbb{V}$: sample $\mathbf{k}_{\mathbf{x}'} = (k_1, \ldots, k_r) \xleftarrow{\$} \mathbb{Z}_p^r$.
8. Run the fresh $\mathsf{LogVOLE}$ online phase on $\mathbf{x}'$, with $\mathbb{V}$ as sender and $\mathbb{P}$ as receiver. $\mathbb{P}$ receives $\mathbf{m}_{\mathbf{x}'} = (m_1, \ldots, m_r)$, where $m_i = k_i + x'_i\Delta$.
9. For $t \in [\tau_{\mathsf{chk}}]$, $\mathbb{V}$: sample $\chi^{(t)} = (\chi_1^{(t)}, \ldots, \chi_w^{(t)}) \xleftarrow{\$} \mathbb{Z}_p^w$.
10. $\mathbb{V} \to \mathbb{P} : (\chi^{(t)})_t$.
11. $\mathbb{P}$: set $u'_t := u_t + \sum_{i\in[w]} \chi_i^{(t)} x_i$ for all $t \in [\tau_{\mathsf{chk}}]$.
12. $\mathbb{P}$: sample $\eta_{\mathsf{rlc}} \xleftarrow{\$} \mathsf{Coins}$ and set $C_{\mathsf{rlc}} := \mathsf{Com}((u'_t)_t; \eta_{\mathsf{rlc}})$.
13. $\mathbb{P} \to \mathbb{V} : C_{\mathsf{rlc}}$.
14. Define $\psi : [t_M] \to \{g \in C_{\mathsf{st}} : g = (a, b, c, \mathsf{Mult})\}$ and $\phi : [t_A] \to \{g \in C_{\mathsf{st}} : g = (a, b, c, \mathsf{Add})\}$.
15. For $g = \psi(i) = (a, b, c, \mathsf{Mult})$, $\mathbb{P}$: set $V_{1,g} := m_c - x_a m_b - x_b m_a$ and $V_{2,g} := m_a m_b$. $\mathbb{V}$: set $U_g := k_a k_b + k_c\Delta$.
16. For $g = \phi(i) = (a, b, c, \mathsf{Add})$, $\mathbb{P}$: set $V_g := m_c - m_a - m_b$. $\mathbb{V}$: set $U_g := k_c - k_a - k_b$.
17. For output wire $o$, $\mathbb{P}$: set $V_o := m_o$. $\mathbb{V}$: set $U_o := k_o$.
18. Derive batching coefficients $(\rho_{M,i})_{i\in[t_M]}, (\rho_{A,i})_{i\in[t_A]}, \rho_o \leftarrow H(T_{\mathsf{set}}, C_{\mathsf{rlc}}, \text{transcript})$.
19. $\mathbb{P}$: set $\widehat{V}_1 := V_1^* + \sum_{i\in[t_M]} \rho_{M,i} V_{1,\psi(i)}$.
20. $\mathbb{P}$: set $\widehat{V}'_2 := -m_r + \sum_{i\in[t_M]} \rho_{M,i} V_{2,\psi(i)}$ and $\widehat{V}_2 := \widehat{V}'_2 + \sum_{i\in[t_A]} \rho_{A,i} V_{\phi(i)} + \rho_o V_o$.
21. $\mathbb{P}$: sample $\eta_{\mathsf{itmac}} \xleftarrow{\$} \mathsf{Coins}$ and set $C_{\mathsf{itmac}} := \mathsf{Com}((\widehat{V}_1, \widehat{V}_2); \eta_{\mathsf{itmac}})$.
22. $\mathbb{P} \to \mathbb{V} : C_{\mathsf{itmac}}$.
23. $\mathbb{V}$: set $\widehat{U}' := -k_r + \sum_{i\in[t_M]} \rho_{M,i} U_{\psi(i)}$ and $\widehat{U} := \widehat{U}' + \sum_{i\in[t_A]} \rho_{A,i} U_{\phi(i)} + \rho_o U_o$.
24. $\mathbb{V} \to \mathbb{P} : (\Delta, \omega_{\mathbb{V}}, \mathbf{k}_{\mathbf{x}'})$.
25. $\mathbb{P}$: reject unless $T_{\mathsf{set}}$ recomputes from $(\Delta, \omega_{\mathbb{V}})$.
26. $\mathbb{P}$: reject unless $m_i = k_i + x'_i\Delta$ for all $i \in [r]$.
27. $\mathbb{P}$: open $C_{\mathsf{rlc}}$ and $C_{\mathsf{itmac}}$.
28. For $t \in [\tau_{\mathsf{chk}}]$, $\mathbb{V}$: check $k_{j_t} + u'_t\Delta + \sum_{i\in[w]} \chi_i^{(t)} k_i = m_{j_t} + \sum_{i\in[w]} \chi_i^{(t)} m_i$.
29. $\mathbb{V}$: accept iff all checks pass and $\widehat{U} = \widehat{V}_2 + \widehat{V}_1\Delta$.

Figure 17: Opened-setup IT-MAC-based ZK protocol.

# F Opened ZK Protocol and Proof Sketch

This appendix describes the opened ZK protocol from Section 11. The verifier uses a fresh semi-honest $\mathsf{LogVOLE}$ setup for one proof session, and opens that setup after the prover has committed to the values that must be fixed before $\Delta$ is known. This removes the sender-side NIZKs needed by the long-term public-key route, while retaining the golden-seed optimization.

## F.1 Why the Opening Is Sound

The verifier opening serves one purpose: it turns setup well-formedness into a local check by the prover. In the public-key route, the verifier cannot reveal $\Delta$, so the setup must be accompanied by malicious-VOLE certification proofs. In the opened route, the setup is fresh and single use. Once the prover has committed to the random-linear-check answers and the final IT-MAC-check values, the verifier can reveal $\Delta$ and all setup coins. The prover recomputes the prover-visible setup messages and rejects if they do not match.

The timing is the essential invariant. The prover sees $\Delta$ only after its accepting proof values are committed. Therefore revealing $\Delta$ lets the prover audit the verifier setup, but it does not let the prover adapt the linear-check answers or the final IT-MAC equation.

## F.2 Proof Sketch

**Completeness.** For an honest verifier, the opened setup recomputes to the transcript that the prover already used, so the prover's setup audit accepts. The random-linear-check identities hold by the final $\mathsf{LogVOLE}$ output relation $m_i = k_i + x_i\Delta$. The IT-MAC batching equation is the gate identity from Section 11; for a valid trace all gate errors are zero, so the verifier accepts.

**Soundness.** Fix a cheating prover and condition on a valid verifier opening. If the opening is invalid, the prover rejects and there is no accepting transcript. Before the verifier opens $\Delta$, the $\mathsf{LogVOLE}$ transcript is fixed and the prover has committed to the random-linear-check answers and the final values $(\widehat{V}_1, \widehat{V}_2)$. By the semi-honest security of the fresh $\mathsf{LogVOLE}$ setup, this pre-opening view hides $\Delta$ except with negligible probability.

The random-linear-check adaptor now gives one effective trace. Rewind the prover to the point after its challenge-free transcript is fixed and answer with fresh independent check challenges. Each accepting rewind gives linear equations in the same output vector, with the sacrificed coordinates included as unknowns. Except for the $1/p$ random-hit event per independent field check, enough accepting rewinds determine a unique vector $\mathbf{x}^{\mathsf{eff}}$. This is exactly the extraction role of Section 10.1.6, specialized to the final field-level output used by the ZK application.

It remains to check the circuit. The prover's committed $(\widehat{V}_1, \widehat{V}_2)$ must satisfy

$$\widehat{U} = \widehat{V}_2 + \widehat{V}_1\Delta.$$

For the extracted trace, expanding the affine and multiplication checks gives

$$\widehat{U} - \widehat{V}_2 - \widehat{V}_1\Delta = E_M\Delta^2 + E_A\Delta + E_o\Delta,$$

where $E_M$, $E_A$, and $E_o$ are the randomly batched multiplication, addition, and output errors of $\mathbf{x}^{\mathsf{eff}}$. If the extracted trace does not satisfy the verification circuit, at least one of these batched error coefficients is nonzero except with probability $O(1/p)$ over the batching challenge. Since $\Delta$ was hidden when the prover committed to the final values, the remaining equation can hold only with the stated negligible plus statistical error.

**Zero knowledge.** The simulator samples the verifier's fresh setup using the honest verifier algorithm and simulates the semi-honest $\mathsf{LogVOLE}$ transcript. It commits to random-looking prover values before the opening, receives the verifier opening, and checks it exactly as the honest prover would. Given $\Delta$, the simulator samples $\widehat{V}_1 \leftarrow \mathbb{Z}_p$ uniformly and sets

$$\widehat{V}_2 := \widehat{U} - \widehat{V}_1\Delta.$$

This has the same distribution as the honest proof because the protocol includes the fresh random masking coordinate used in the IT-MAC check. The random-linear-check messages are simulated using their sacrificed coordinates, which mask the checked linear views. Thus the verifier's view is distributed as in a real execution, up to the semi-honest $\mathsf{LogVOLE}$ simulation and the hiding of the commitments.

**Reusable public keys.** This proof deliberately uses the verifier opening and is therefore not a proof for a long-lived public key. For that setting, the verifier keeps $\Delta$ hidden and the ZK application should instantiate its label-generation layer with the malicious $\mathsf{CI\text{-}VOLE}$ protocol of Section 10, including the sender-side setup certification proofs.

# G Omitted Details about [AMR25]

Here, we provide a detailed comparison with [AMR25]. We expect that readers have already read our overview (Section 4).

## G.1 Cost Analysis of [AMR25]

[AMR25] reports its communication complexity vaguely as $\mathsf{poly}(\lambda) \cdot \log |\mathbf{x}|$ in its abstract and does not provide a clear analytic cost for computation. They use different notations for their cost accounting, making a straightforward comparison difficult. Here, we provide a detailed analysis of their analytic cost using our notation.

As background, [AMR25] considers a slightly more general problem where two parties want to evaluate the outer product $\mathbf{x} \cdot \mathbf{y}$ for their respective inputs $\mathbf{x}$ and $\mathbf{y}$. The mapping between our notation and [AMR25] is summarized as follows:

- **Roles**: The $\mathbb{S}$ corresponds to the *encoder*; the $\mathbb{R}$ corresponds to the *hasher*.
- **Artifacts**: The *encoding* corresponds to the one-time setup (committing to $\mathbf{y}$); the *digest* corresponds to the per-instance communication (evaluating the product).
- **Parameters**:
  - $\ell = |\mathbf{y}|$: Dimension of the vector $\mathbf{y}$.
  - $m = |\mathbf{x}|$: Dimension of the vector $\mathbf{x}$.
  - $r = O(\log |\mathbf{x}|)$: Number of homomorphic hashing levels.
  - $t > 1$: a constant
  - $n = k \cdot \log q$: Dimension of the public LWE matrix. We also have $n = \tilde{\Omega}(\lambda^2)$
  - $k = \Theta(\lambda)$.
  - $q$: Modulus for the LWE samples. We also have $\log q = \tilde{\Omega}(\lambda)$

We first focus on the **(one-time setup) encoding cost**. The protocol is listed as $\mathsf{Encode}$ in Construction 3.6 of [AMR25, p. 20]. Its communication, shown in [AMR25, p. 23], involves sending a series of matrices over $\mathbb{Z}_q$, whose total number of entries is $\ell \cdot r \cdot t^2 \cdot n^2$, totaling $\ell \cdot r \cdot t^2 \cdot n^2 \cdot \log q$ bits of communication. Here, $\ell = |\mathbf{y}|$, $r = O(\log |\mathbf{x}|)$ is the number of bootstrap levels, and $t > 1$ is a constant. The most significant costs come from $n$ and $q$, which require careful analysis. $n$ is the number of rows in the public LWE matrix ($\mathbf{A}$) and $n = k \cdot \log q$ [AMR25, p. 20]. $k(\lambda)$ is set as $\Theta(\lambda)$ for security [AMR25, p. 18]. The magnitude of $q$ is critical, and $q = 2^{\tilde{\Omega}(\lambda)}$ for two reasons. First, their construction generally only guarantees approximate correctness, *i.e.*, the results are noisy. If the parties want only negligible errors in the results, they need to lift their messages in $\mathbb{Z}_p$ by $q/p \in \lambda^{\omega(\lambda)}$ [AMR25, p. 17]. In other words, $\log q = \tilde{\Omega}(\lambda)$. Secondly, for our applications of interest such as ZK and GC, their labels must be from a sufficiently large domain for security. Specifically,

we need the message modulus to be $p = 2^{\Omega(\lambda)}$. As $p < q$, we also have $\log q = \tilde{\Omega}(\lambda)$. This gives us $n = O(k \log q) = \tilde{O}(\lambda^2)$. Plugging in these parameters to the encoding size and expressing it in terms of $\log |\mathbf{x}|$ and $\lambda$, we find that the encoding size is $O(\log |\mathbf{x}| \cdot k^2 \log^3 q) = O(\log |\mathbf{x}| \cdot \lambda^5)$.

For the **(per-instance message) digest**, the size corresponds to a length-$n$ vector of integers, each having a bit-length of $r \log(tn) + \log ||x||_\infty$ [AMR25, Eq. 6]. Using the parameter conversions above, we have the digest size as $\tilde{O}(\lambda^2(\log |\mathbf{x}| + \lambda))$.

Their **online computation cost** is more implicit as their main focus was showing succinct communication. For each instance, the hasher and the encoder evaluate $\mathsf{HashEval}$ and $\mathsf{EncEval}$, respectively, to obtain their shares. Instead of inspecting every operation in these protocols, we observe that both protocols need to multiply vectors by a matrix $\Pi_{i=1}^{r-1} \mathbf{P}_i$, where $\mathbf{P}_i = (\mathbf{I}_{t^{r-i}} \cdot \mathbf{P})$, $\mathbf{I}_\mu$ is the identity matrix of dimension $\mu \times \mu$, and $\mathbf{P} \in \mathbb{Z}^{n^2 \times (t \cdot n^2)}$ is a 0/1 matrix in which each row has at most $n^2$ non-zero entries [AMR25, p. 21]. Hence, $\Pi_{i=1}^{r-1} \mathbf{P}_i$ is a $|\mathbf{x}| \times (t \cdot n^2)$ matrix whose entries have norms bounded by $n^{2r}$. Plugging in the parameter $n = \tilde{O}(\lambda^2)$, we know multiplying a vector by this matrix would cost $\tilde{O}(|\mathbf{x}| \cdot \lambda^4)$ (potentially large) integer arithmetic operations, and this is the lower bound on computation per instance.

## G.2 Detailed Comparison about Constructions

Readers familiar with [AMR25] might find that both their work and ours exhibit a similar structure: at each level of bootstrap (or recursion), the protocols derive a share of the tensor product (or broadcast multiplication) from the shares of a smaller set of products at the level above. Also, [AMR25] relies on the LWE hardness assumption, while our work assumes hardness of RLWE. Nevertheless, the underlying techniques are vastly different. [AMR25] relies on a theorem in [BLMR13] stating that LWE w.r.t. random matrices, *i.e.*, using $\mathbf{A}^T \cdot \mathbf{B}$ as the LWE matrix where $\mathbf{A}$ and $\mathbf{B}$ are random binary matrices, is as hard as standard LWE. This is the key theorem enabling the hasher to reduce the problem size via multiplying its inputs by a fat matrix $\mathbf{A}$, which allows the parties to later expand their shares by multiplying them by a tall matrix $\mathbf{B}$. It is unclear if there is any equivalent structure/theorem in RLWE that enables such a mechanism without introducing new or less-tested hardness assumptions. In contrast, our construction relies on a RLWE-based laconic encryption scheme [DKL$^+$23a]. Leveraging this scheme requires overcoming a different set of challenges, as highlighted in Section 6.1 and Section 6.2. For example, our construction must remove the noise at every recursion level to maintain correctness, while [AMR25] allows the noise to accumulate across levels.