Onion Ring ORAM: Efficient Constant Bandwidth Oblivious RAM from (Leveled) TFHE

Hao Chen[1], Ilaria Chillotti[2], and Ling Ren[3]

[1]Microsoft Research
[2]KU Leuven
[3]VMware Research

June 20, 2019

Abstract

Oblivious RAM (ORAM) is a cryptographic primitive that allows a client to hide access pattern to its data encrypted and stored at a remote server. Traditionally, ORAM algorithms assume the server acts purely as a storage device. Under this assumption, ORAM has at least $\log(N)$ bandwidth blowup for N data entries. After three decades of improvements, ORAM algorithms have reached the optimal logarithmic bandwidth blowup. Nonetheless, in many practical use-cases a constant bandwidth overhead is desirable. To this purpose, Devadas et al. (TCC 2016) formalized the server computation model for ORAM and proposed *Onion ORAM* which relies on homomorphic computation to achieve constant worst-case bandwidth blowup. This line of work is generally believed to be purely theoretical, due to the large overheads of homomorphic computation. In this paper, we present Onion Ring ORAM, the first efficient constant bandwidth ORAM scheme in the single server model, based on the Onion ORAM construction and the leveled version of the TFHE scheme by Chillotti et al.. We propose a series of improvements, most notably including a more efficient homomorphic permutation protocol. We implement Onion Ring ORAM and show that it can outperform state-of-the-art logarithmic-bandwidth ORAM like Path ORAMs and Ring ORAM when the network throughput is limited. Under one setting, our construction improves the end-to-end latency by 1.5× over Ring ORAM and 7.3× over Path ORAM.

1 Introduction

Oblivious RAM (ORAM), first introduced by Goldreich and Ostrovsky [28, 29] allows a client to outsource its private data to an untrusted server and perform read/write requests to its data without leaking any information through the addresses of its requests.

In the past few years, ORAM has started to draw more attention as recent works have demonstrated sensitive information leakage through access patterns across various application domains, including secure enclaves [66, 63], storage outsourcing [35, 36], and searchable encryptions [34, 12, 67]. In light of this, researchers from multiple communities have applied ORAM to protect against these access pattern attacks. The two most prominent use cases of ORAM are storage outsourcing [59, 9, 13, 19] and secure hardware [23, 53, 43, 40, 50, 46, 55].

ORAM provides cryptographic-grade protection to access patterns; however, its strong security also comes with a high price. The primary source of overhead, which is also ORAM's main metric, is its *bandwidth blowup*. At a high level, an ORAM algorithm turns each logical memory access into many random-looking accesses. Bandwidth blowup, sometimes referred to as bandwidth for short, is the amount of communication needed between the client and the server to serve one logical access. Numerous works steadily improved ORAM bandwidth blowup [28, 29, 65, 30, 37, 60, 56, 61, 64, 52, 48, 6]. Despite these efforts and progress, the most efficient ORAMs [60, 57, 52, 64, 6] to date still incur $\Theta(\log N)$ bandwidth blowup where N is the total number of logical memory blocks.

Moreover, it is also well known that any ORAM in the classic model (discussed further below) must suffer from $\Omega(\log N)$ bandwidth blowup [29, 39], so ORAMs have reached asymptotic optimality. Combined with the fact that there exist ORAM algorithms with very small hidden constants [60, 52], it seems necessary to explore new models if we are to find further algorithmic improvements. The classic ORAM model assumes the server is a plain storage device, that is, it only supports read operations and write operations. A natural extension to the classic model is to assume that the server can perform computation. This extended model is called the "server-computation model". It is important and promising for two important reasons. First, the server computation model, in fact, better matches reality for the cloud storage scenario, in which the server (e.g., Amazon AWS or Microsoft Azure) has ample computational power. [1] Second, the server computation model allows for ORAM schemes with $O(1)$ bandwidth blowup [22], and thus holds potential for significant improvements.

The state-of-the-art single-server constant-bandwidth ORAM is Onion ORAM by Devadas et al. [22]. Its two variants rely on Damgård-Jurik [20] additively homomorphic encryption (AHE) and BGV [10] somewhat homomorphic encryption (SWHE)[2], respectively. But Onion ORAM was proposed, and has been regarded, as a purely theoretical advance primarily because Damgård-Jurik AHE and BGV-style SWHE both demand heavy computation. For example, it was estimated in [32] that the end-to-end delay AHE variant of Onion ORAM would be take more than 10,000 seconds for retrieving a 128KB block from a 40GB database, compared to less than 30 seconds for Path ORAM and Ring ORAM.

1.1 Our Contributions

We present Onion Ring ORAM, a constant-bandwidth ORAM based on the leveled mode of the TFHE homomorphic encryption scheme. We estimate that our construction improves the performance of Onion ORAM by at least three orders of magnitude, through a series of algorithmic innovations. We show that our construction can outperform state-of-the-art logarithmic-bandwidth ORAMs (e.g., Path ORAM and Ring ORAM) in the oblivious cloud storage setting when network throughput is limited.

- We designed an efficient homomorphic permutation algorithm, which permutes an array of ciphertexts according to an encrypted permutation, and applied it to improve the eviction phase of Onion ORAM. Our construction homomorphically evaluates a Waksman permutation network, taking advantage of TFHE's multiplexer operations (in leveled mode).
- We adapted the homomorphic expansion algorithm in [3], combining it with TFHE external product, to reduce the communication cost of sending the encrypted "swap bits" in a Waksman network. As a result, we are able to pack multiple swap bits in a single ciphertext and use the homomorphic expansion algorithm to unpack them for use in homomorphic permutation.
- We adapted the permuted buckets technique and the XOR trick from [52] to our construction, making necessary modifications to Onion ORAM to gain efficiency. For example, we replaced the XOR trick with homomorphic addition. We also avoided costly PIR operations in Onion ORAM.

We implemented Onion Ring ORAM based on the experimental version of the TFHE library. With the concrete parameterization in our implementation, Onion Ring ORAM has a bandwidth blowup of $12.8\times$ and spends (amortized) 7.6 seconds on computation to access a 768KB block for $N = 2^{22}$ total logical blocks. Under similar parameters, Path ORAM has a bandwidth blowup of $160\times$ and Ring ORAM has a bandwidth blowup of $34\times$. When the client-server network throughput is low, Onion Ring ORAM offers better performance. Concretely, for an ORAM with 2^{22} blocks of size 1.5MB, assuming a 10 Mbps network,

[1]For secure hardware, server computation would mean in-memory or in-disk computation, which is not yet widely available and is under active research [7].

[2]SWHE is similar to FHE but supports only circuits of a given depth. It avoids the costly bootstrapping step in FHE.

Onion ORAM can finish an access in 26.5 seconds, compared to 40.7 seconds for Ring ORAM and 192 seconds for Path ORAM.

While we motivate homomorphic permutation and expansion in the context of ORAM, they are basic building blocks that can be of independent interest. For example, Gentry et al. [26] introduced a blinded permutation protocol based on AHE. Its communication cost is at least linear in the size of the entire array being permuted and it requires sending three messages. In contrast, the communication cost of our homomorphic permutation depends only on the number of elements in the array, and requires only one message from client to server.

1.2 Related works

Many ORAMs leveraged server computation without realizing the power of this new model [25, 44, 21, 52]. Thus, these works still aim for $\Theta(\log N)$ (or higher) bandwidth.

Research on constant-bandwidth ORAM to date has been entirely theoretical. Apon et al. [5] is the first to observe that $O(1)$ bandwidth can be achieved in the server computation model. Their work focused on feasibility rather than efficiency: they observe that any ORAM algorithm can be made $O(1)$ bandwidth if executed under fully homomorphic encryption (FHE).

Devadas et al. [22] presented Onion ORAM which achieves $O(1)$ bandwidth without resorting to full-blown FHE. Onion ORAM focused on asymptotic complexity. As mentioned, both variants of Onion ORAM require heavy computation. Nonetheless, it remained to this day the most efficient constant-bandwidth ORAM. [3]

Another line of ORAM work leverages multiple non-colluding servers to reduce bandwidth without resorting to costly homomorphic operations [47, 41, 1, 38]. The state of art in this direction has sub-logarithmic, e.g., $O(\log N/\log\log N)$ bandwidth blowup [1, 38]. Two other works, Stefanov-Shi [58] and Hoang et al. [32] achieve $O(1)$ client-server bandwidth blowup and $O(\log N)$ inter-server bandwidth blowup. They achieve good concrete efficiency based on the observation that inter-server network throughput can be much higher than that between client and server. Our construction is in the single-server with computation model.

2 Background

2.1 ORAM with Server Computation

An ORAM algorithm involves two parties, a trusted client and an untrusted server. The ORAM algorithm provides the client an `Access` function that takes as input a tuple $(a, \mathsf{op}, \mathsf{data})$ where a represents the logical address of the data block the client requests, $\mathsf{op} \in \{\mathsf{Read}, \mathsf{Write}\}$, and data is the data being written ($\mathsf{data} = \perp$ if $\mathsf{op} = \mathsf{Read}$).

Let $\mathbf{y} = ((a_1, \mathsf{op}_1, \mathsf{data}_1), \ldots, (a_m, \mathsf{op}_m, \mathsf{data}_m))$ be the client's sequence of logical accesses. Let ORAM($\mathbf{y}$) represent the client's sequence of interactions with the server to fulfill $\mathbf{y}$. Here, interactions include not only physical read/write requests but also any other sub-protocols between the client and the server. In particular, in our construction, the client-server interaction includes multiple invocations of homomorphic permutation defined in the next subsection.

We say an ORAM algorithm is *correct* if, except for negligible (in some security parameter) probability, the output of each $\mathtt{Access}(a_i, \mathsf{op}_i, \mathsf{data}_i)$ operation to the client is consistent with $\mathbf{y}$, that is, the last data that was written to address a_i prior to the i-th access. We say an ORAM algorithm is *secure* if for any two logical client request sequences $\mathbf{y}$ and $\mathbf{z}$ with the same length $|\mathbf{y}| = |\mathbf{z}|$, the resulting client-server interactions ORAM($\mathbf{y}$) and ORAM($\mathbf{z}$) are computationally indistinguishable.

Threat model. In this work we assume the server is semi-honest. That is, it will faithfully follow the prescribed protocol, but it may run additional experiments in an attempt to infer the client's secret (access pattern in our case). We discuss the options and costs to defend against a malicious server in Section 6.

[3]C-ORAM [45] was an attempt to improve Onion ORAM. But it was later found to be broken [1] (together with the follow-up CHf-ORAM).

2.2 Onion ORAM

Onion ORAM is an ORAM algorithm in the server computation model with $O(1)$ bandwidth blowup [22]. We briefly explain Onion ORAM in this subsection. We will, however, leave out certain optimizations and techniques from Onion ORAM that are superseded by our techniques.

Onion ORAM follows the binary tree ORAM paradigm of Shi et al. [56]. The server storage is structured as a binary tree with L levels. Each node in the tree is a bucket that contains Z slots. Each slot can hold a logical block or a dummy block. Each block has a size of B bits. Later, we will see that Z decides the correctness failure probability. Once we select Z, L should be set such that the total number of blocks $Z(2^L - 1)$ is between $2N$ to $4N$.

Onion ORAM maintains the **key invariant** of binary tree ORAMs: each logical block is mapped to a random *path* in the tree, and must reside somewhere on that path. The client locally tracks all the block-to-path mappings as well as slot-to-block mappings. That is, for each logical address $a \in [N]$, the client tracks $\mathsf{pos}[a]$, the path block a is mapped to; for each slot (i, j) where $i \in [2^L - 1], j \in [Z]$, the client also tracks $\mathsf{id}[i][j]$, the logical address of the block that resides in the j-th slot of the i-th bucket in the tree.

Having the client store the above two metadata arrays locally is a common design choice in oblivious cloud storage [59, 21, 19]. Both arrays have $\Theta(N)$ entries of $\Theta(\log N)$ bits each. It is also typically assumed that the size of each logical block D is at least several Kilo Bytes. Thus, the client still enjoys significant storage savings despite having to store the two metadata arrays. In contrast, a theoretical ORAM algorithm (Onion ORAM included) is typically required to use only $O(B)$ bits of client storage, where B is the size of each block; hence, the id array is stored encrypted on the server and the pos array is stored in recursive ORAMs [56], adding extra overheads.

Now, accessing a block involves the following steps. Suppose the client requests the block with logical address a.

1. The client locally looks up $x = \mathsf{pos}[a]$, the path block a is mapped to.
2. The client locally looks up $\mathsf{id}[i][j]$ for each slot (i, j) on path x. Due to the invariant, one of these slots stores block a.
3. The client uses private information retrieval (PIR) based on either AHE or SWHE to download block a from the path. The client can now read or update block a.
4. The client remaps block a to a fresh random path x' and updates $\mathsf{pos}[a]$.
5. The client puts block a to the next available slot in the root bucket. (Note that the root bucket is on every path, so the invariant is honored.)
6. After $A < Z$ accesses, the client and server jointly perform an eviction procedure, which will empty the root so that the client can make another A accesses.

Barring the eviction step, correctness and security of Onion ORAM are relatively straightforward. Correctness is guaranteed by the key invariant. Security holds because the client-server interaction on each access is a PIR operation on a path. The path was generated at random on the last access to block a and it has never been revealed to the server until this access, and the PIR operation reveals no additional information by definition.

We omit Onion ORAM's eviction procedure because it needs to be modified to account for the permuted bucket technique, which we will describe in Section 3.2.

2.3 Leveled TFHE

TFHE is a fully homomorphic encryption scheme presented in 2016 by Chillotti et al. [14] and improved in 2017 by the same authors [15]. TFHE uses a ring variant of the GSW construction [27] together with two other encryption schemes. The security of TFHE is based on the hardness of the Learning With Errors (LWE) problem and its ring variant (RLWE) [51, 62, 42]. The scheme is available in open source on GitHub [17, 16].

TFHE can be used in both fully homomorphic mode with bootstrapping after every single gate, or in leveled mode where the number of operations allowed is limited by the parameter choice. For efficiency reasons, our construction avoids the use of bootstrapping. Instead, we make non-black-box use of various building blocks in the leveled mode of TFHE to achieve our efficiency goals.

The TFHE scheme works over the real torus $\mathbb{T} = \mathbb{R} \mod 1$ and over the torus polynomials $\mathbb{T}_n(X) = \mathbb{R}[X]/(X^n + 1) \mod 1$. Here n is the ring dimension parameter of the scheme, and is a power of two integer. We denote as $R = \mathbb{Z}[X]/(X^n + 1)$: we observe that $\mathbb{T}$ is a $\mathbb{Z}$-module, while $\mathbb{T}_n(X)$ is a R-module.

Ring LWE. A RLWE secret s is a polynomial sampled form a key distribution χ in R (generally uniformly distributed binary polynomials). RLWE ciphertexts encrypt torus polynomial messages $\mu \in \mathcal{M}$, where $\mathcal{M}$ is a discrete subset of $\mathbb{T}_n(X)$. In our case, we suppose $\mathcal{M}$ is the set of torus polynomials with coefficients in $\{0, \frac{1}{t}, \ldots, \frac{t-1}{t}\}$, for a fixed positive integer t. Then, a RLWE ciphertext is a pair of torus polynomials $(a, b) \in \mathbb{T}_n(X)^2$ where $b - a \cdot s = e + \mu$, where the "noise term" e is small. To decrypt, we use the secret key s to compute $\varphi = b - a \cdot s$ and we approximate it to the nearest message in $\mathcal{M}$. In our setting, decryption is correct given that the norm of the error $||e||_\infty$ is smaller than $\frac{1}{2t}$.

A RLWE public key is equal to a fresh RLWE encryption of 0: $pk = (a_{pk}, b_{pk} = a_{pk} \cdot s + e_{pk}) \in \mathbb{T}_n(X)^2$. In order to encrypt a message $\mu \in \mathbb{T}_n(X)$, we sample a polynomial u from the secret key distribution χ and two small polynomials e_0, e_1 (from a Gaussian distribution centered in 0 with standard deviation α). A public key RLWE encryption of μ is then the pair of torus polynomials $(a_{pk} \cdot u + e_0, b_{pk} \cdot u + e_1 + \mu) \in \mathbb{T}_n(X)^2$. RLWE ciphertexts support additive homomorphism.

Ring GSW. Given a base B and a length parameter ℓ, we define the RGSW gadget as $\mathbf{g} = (1/B, \ldots, 1/B^\ell)$. The RGSW gadget matrix is defined as follows

$$\mathbf{G} = \mathbf{I}_2 \otimes \mathbf{g} = \begin{bmatrix} \mathbf{g} & 0 \\ 0 & \mathbf{g} \end{bmatrix} \in \mathbb{T}_n(X)^{2\ell \times 2}.$$

We define the gadget decomposition of a vector $\mathbf{u} \in \mathbb{T}_n(X)^2$ as the 2ℓ-dimensional vector $\mathbf{G}^{-1}(\mathbf{u}) \in R^{2\ell}$ with coefficients in $\mathbb{Z} \cap (-B/2, B/2]$, such that the decomposition error $||\mathbf{G}^{-1}(\mathbf{u}) \cdot \mathbf{G} - \mathbf{u}||_\infty$ is upper bounded by $1/(2B^{\ell+1})$.

Then, an RGSW ciphertext encrypting an integer polynomial $\mu \in R$ with respect to a secret RLWE key s is defined as $\mathbf{C} = \mathbf{Z} + \mu \cdot \mathbf{G}$, where $\mathbf{Z} \in \mathbb{T}_n(X)^{2\ell \times 2}$ satisfies the property that $||Z \cdot (-s, 1)^T||_\infty$ is small.

External RLWE product. The external RLWE product takes as input a RLWE ciphertext $\mathbf{d}$, encrypting $\mu_\mathbf{d} \in \mathbb{T}_n(X)$, and a RGSW ciphertext $\mathbf{C}$, encrypting $\mu_\mathbf{C} \in R$, with respect to the same secret s. The output is a RLWE ciphertext encrypting the product $\mu_\mathbf{d} \cdot \mu_\mathbf{C} \in \mathbb{T}_n(X)$. The external product is denoted by $\boxdot$ and defined as $\mathbf{C} \boxdot \mathbf{d} = \mathbf{G}^{-1}(\mathbf{d}) \cdot \mathbf{C}$. For more details on the external product, we refer to [15, 18].

CMux. Building on the external product, Chillotti et al. [14, 15, 18] defined a ternary operation called `CMux`, the homomorphic multiplexer. `CMux` takes as input two RLWE ciphertexts $\mathbf{d_0}$ and $\mathbf{d_1}$, encrypting respectively μ_0 and μ_1 in $\mathbb{T}_n(X)$, and a RGSW ciphertext $\mathbf{C}$, encrypting a bit $b \in \{0, 1\}$, all with respect to the same secret key, and it outputs a RLWE encryption of μ_b. In other words, it implements the multiplexer gate. The homomorphic operation `CMux` is defined as $\mathbf{c}' = \mathtt{CMux}(\mathbf{C}, \mathbf{d_1}, \mathbf{d_0}) = \mathbf{C} \boxdot (\mathbf{d_1} - \mathbf{d_0}) + \mathbf{d_0}$. We let $\mathsf{Var}(c)$ denote the maximum variance of the noise in cipehrtext c. From the noise analysis done in [15], the noise after `CMux` satisfies:

$$\mathsf{Var}(\mathbf{c}') \leq \max(\mathsf{Var}(\mathbf{d_0}), \mathsf{Var}(\mathbf{d_1})) + V_\mathbf{C} \tag{1}$$

where $V_\mathbf{C}$ only depends on the noise in the RGSW ciphertext and the TFHE parameters. We stress that the important property of `CMux` is that the noise growth is *additive*. If we were to perform this operation solely over RLWE ciphertexts, using a somewhat homomorphic encryption scheme such as BGV, then the noise growth will be multiplicative, i.e., $\mathsf{Var}(\mathbf{c}') \geq C \cdot \max(\mathsf{Var}(\mathbf{d_0}), \mathsf{Var}(\mathbf{d_1}))$ for some constant $C > 1$ depending on $\mathsf{Var}(\mathbf{C})$. Hence if we perform L `CMux` operations sequentially, the final noise will be a factor $\Omega(2^L)$ larger than the input noise, whereas the TFHE construction of `CMux` has final noise only a linear $O(L)$ factor larger than input noise.

2.4 Homomorphic Permutation

A homomorphic permutation protocol involves two parties, a trusted client and an untrusted server. We assume that the client has generated a pair of keys (sk, pk) for a homomorphic encryption scheme. It consists of two functions run by the two parties, respectively.

- $\hat{\pi} \leftarrow \texttt{genPerm}(pk, \pi)$ is run by the client. It takes as input a m-by-m permutation and outputs an encryption of some encoding of $\hat{\pi}$.
- $\{\mathbf{c}'_i\}_{i=1}^m = \texttt{evalPerm}(pk, \hat{\pi}, \{\mathbf{c}_i\}_{i=1}^m)$ is executed by the server. It takes as input the above $\hat{\pi}$ and m ciphertexts and permutes the m ciphertexts homomorphically.

We say the above protocol is correct if except for negligible probability, $\texttt{Dec}(sk, \mathbf{c}'_i) = \texttt{Dec}(sk, \mathbf{c}_{\pi(i)})$ where $\texttt{Dec}$ is the decryption function of the underlying homomorphic encryption scheme. The homomorphic permutation is secure if, for any π_1 and π_2, the generated encrypted permutations $\hat{\pi}_1$ and $\hat{\pi}_2$ are computationally indistinguishable.

3 Onion Ring ORAM

In this section, we present the Onion Ring ORAM algorithm assuming a black-box homomorphic permutation algorithm. We present our modifications to Onion ORAM and fill in some details omitted by the original construction. A major modification is to incorporate the permuted bucket technique introduced by Ren et al. [52]. The Onion ORAM briefly mentioned the permuted bucket technique but did not provide details on how to integrate it with their algorithms.

3.1 Initialization

First, the client generate keys for the three encryption schemes utilized in Onion Ring ORAM: AES, RLWE and RGSW (RGSW and RLWE use the same keys). The client then sends the RLWE public key to the server. The server initializes all slots of the ORAM tree to dummy blocks, i.e., RLWE encryptions of zero, using the public key. Next, the client initializes the block-to-path mappings pos and the slot-to-block mappings id. Each entry in pos is initialized to a random integer between 0 and $2^L - 1$, representing a random path in the tree. Each entry in id is initialized to dummy, indicating that every slot in the ORAM tree contains a dummy block in the beginning.

3.2 Access

Algorithm 1 present the access algorithm of Onion Ring ORAM. The biggest advantage of Onion Ring ORAM is that it avoids PIR operations by combining perumuted buckets with RLWE.

The permuted bucket technique adds S additional slots reserved for dummy blocks to each bucket. A real block b is stored as $\mathsf{RLWE}(\mathsf{AES}(b))$, whereas a dummy block has the form $\mathsf{RLWE}(0)$. When a bucket is involved in eviction (Section 3.3), its $Z + S$ slots are randomly permuted.

We will set $S = Z$ so that each bucket has $2Z$ slots and can be at most half full. Setting $S = Z$ gives the best option for Onion Ring ORAM since it matches two types of failure probabilities (cf. Section 3.4). We remark that Ring ORAM [52], on the other hand, can use a small number slots for both real and dummy blocks and perform extra on-demand reshuffles (called early reshuffles) to blocks that run out of dummy blocks before their next scheduled eviction.

In the access algorithm, the client can now specify one block from each bucket on the path: the requested block from the bucket containing it and a random untouched dummy block from every other bucket. The server homomorphically "add up" these blocks and return it to the client, which yields $\mathsf{RLWE}(\mathsf{AES}(b) + 0 + ... + 0)$ where b is the requested block. After this step, the client locally marks all slots involved as "touched" (in the id metadata array). These slots will not be touched again until they are re-permuted during evictions. (Recall that the requested block will be relocated to other buckets after this access.)

The permuted bucket technique requires only homomorphic add operations, which are cheaper than PIR which requires homomorphic multiplications. Therefore, it reduces *online latency*, i.e., the latency for the client to receive the requested block.

However, with permuted bucket it is trickier to put the requested block back to the root (Step 5 in Section 2.2). Without permuted bucket, the client simply uploads the requested block to the next available slot in the root bucket. Now, it has to keep the root bucket randomly permuted. Onion ORAM suggests using a "PIR write" operation to insert the requested block to a random slot in the root bucket, and they do this immediately after each access to keep the client storage a constant number of blocks. Since we do not target constant client storage, we simply let the client buffer the requested blocks locally until the next eviction. The buffer consumes $A < Z$ blocks of client storage where A is the eviction period (i.e., we perform one eviction per A accesses).

3.3 Eviction

Algorithm 2 present the eviction algorithm of Onion Ring ORAM. The purpose of the eviction procedure is to move blocks towards the leafs of the tree to free up space in and near the root bucket. Like in Onion ORAM, eviction happens every A accesses.

As the first step of Onion Ring ORAM eviction, the client AES-encrypts the blocks in the locally stored root bucket and uploads them to the server. The server encrypts them using the RLWE public key and supplies $(2Z - A)$ RLWE-encryptions of zero as dummy blocks. The client and server then randomly permute the root bucket using the homomorphic permutation protocol in Section 4. Compare to the PIR write approach, our design saves both bandwidth and computation at the expense of a reasonable amount (a few hundred blocks) of extra client storage.

Each eviction operation goes down a path. The eviction paths are selected according to a reverse lexical order [25], which interprets increasing integers in their binary representation flipped. As an example, for a tree with eight paths numbered 0 to 7 from left to right, the reverse lexical order is $[000, 001, 010, 011, 100, 101, 110, 111]$ $\rightarrow [000, 100, 010, 110, 001, 101, 011, 111] = [0, 4, 2, 6, 1, 5, 3, 7]$.
Intuitively, performing evictions according to reverse lexical order is efficient because it minimizes the amount of redundant work by consecutive evictions, since it minimizes the number of overlapping buckets between consecutive paths.

Once an eviction path is selected, the Onion ORAM eviction algorithm takes each non-leaf bucket on the path and moves all real blocks in it to its two children. This step is called a *triplet eviction* in Onion ORAM. Each eviction consists of $L - 1$ triplet evictions along the selected path.

Next we explain what happens during a triplet eviction. To ease notation, we call the child on the eviction path the *destination bucket* and the other child the *sibling* bucket. The parent bucket is also called the *source bucket*. The client and server engage in the homomorphic permutation protocol to move blocks from the source bucket to the two child buckets. The client splits the source bucket into two halves by specifying Z slots that include all real blocks mapped to the sibling bucket (and the right amount of dummy blocks to to make the total Z). The server takes these Z blocks, supplies Z encrypted dummy blocks, puts them into the sibling bucket (which is empty before this step [22]), and homomorphically permutes them according to an encrypted random permutation chosen by the client. The other Z slots in the source bucket then include all real blocks mapped to the destination bucket. The client further specifies Z blocks from the destination bucket, which include the remaining untouched real blocks (and the right number of untouched dummy blocks to make the total Z). The server takes these $2Z$ blocks, puts them into the destination bucket, and homomorphically permutes them according to an encrypted random permutation chosen by the client.

Lastly, the destination bucket in the last triplet, which is a leaf bucket, needs to be decrypted and re-encrypted to keep the noise level low in the ciphertexts. Concretely, the client download Z blocks (which includes all real blocks) from random slots, decrypts these Z blocks, re-encrypts them under AES, and sends them back to the server. The server further encrypts the AES ciphertext under RLWE, supplies Z encrypted dummy blocks of zero, and carries out a homomorphic permutation according to an encrypted random permutation chosen by the client. Note that the eviction algorithm makes extensive use of a homomorphic permutation protocol, which will be explained in detail in Section 4.

Algorithm 1: Onion Ring ORAM Access Algorithm.

Input: $(a, \mathsf{op}, \mathsf{data}')$ where a is the logical address of the requested block, op is either Read or Write, and data' is the data written to address a if $\mathsf{op} = \mathsf{Write}$

Data Structure:

- G, number of evictions that happened so far
- g, number of accesses since the last eviction
- pos, block-to-path mappings stored locally at client
- id, slot-to-block mappings stored locally at client
- R, root bucket stored locally at client.

Notation :

- N, number of logical blocks
- L, number of levels in ORAM tree
- Z, maximum number of real blocks per bucket (also the number of reserved dummy blocks per bucket)
- $\mathcal{P}(x, l)$ denotes the l-th bucket on path x
- shaded part is executed by server

```
1  Client locally looks up $x = \mathsf{pos}[a]$
2  Initialize a vector $S$ of length $L$
3  for $l = 0 : L - 1$ do
4      $i = \mathcal{P}(x, l)$                      ▷ the $l$-th bucket on path $x$
5      if $\exists j$ s.t. $\mathsf{id}[i][j] == a$ then
6          $S[l] = (i, j)$                           ▷ requested block resides in this bucket
7      end
8      else
9          $S[l] = (i, j')$ where $\mathsf{id}[i][j] == \mathsf{dummy}$
10     end
11     $\mathsf{id}[i][j] = \perp$                   ▷ mark slot $(i, j)$ as touched
12 end
13 Client sends $S$ to server

14 $Y = \mathsf{RLWE}(0)$
15 for $l = 1 : L - 1$ do
16     $(i, j) = S[l]$
17     Server fetches data $y$ from slot $(i, j)$
18     $Y = Y \oplus y$                              ▷ $\oplus$ is homomorphic add
19 end
20 Server sends $Y$ to client

21 $(\perp, j) = S[0]$
22 $Y = Y \oplus \mathsf{R}[j]$                     ▷ add block from local root
23 Client decrypts $Y$ to obtain $\mathsf{data}$ of block $a$
24 $\mathsf{data} = \mathsf{data}'$ if $\mathsf{op}$ is write
25 $\mathsf{R}[g] = \mathsf{data}$                  ▷ add to locally stored root
26 $\mathsf{id}[0][g] = a$
27 $\mathsf{pos}[a] = \mathtt{rand}() \mod 2^L$
28 $g = g + 1$
29 if $g == A$ then
30     Evict()
31     $g = 0$
32 end
```

Algorithm 2: Onion Ring ORAM Eviction Algorithm $\mathsf{Evict}()$.

```
   Data Structure and Notation as defined in Algorithm 1
1  Eviction path $x = \mathsf{ReverseLexicalOrder}(G)$
2  Client AES-encrypts $\mathsf{R}$ to get $\{r_j\}_{j=1}^{A} = \{\mathsf{AES}(\mathsf{R}[j])\}_{j=1}^{A}$
3  Client generates a random permutation $\pi$, computes $\hat{\pi} = \texttt{genPerm}(pk, \pi)$, and sends $\hat{\pi}$ to server
4  Server RLWE-encrypts $\{r_j\}$ to get $\{\mathbf{c}_j\}_{j=1}^{A} = \{\mathsf{RLWE}(r_j)\}_{j=1}^{A}$
5  Server supplies $(2Z - A)$ RLWE encryptions of zero $\mathsf{RLWE}(0)$ as $\{\mathbf{c}_j\}_{j=A+1}^{2Z}$
6  Server computes $\{\mathbf{c}'_j\}_{j=1}^{2Z} = \texttt{evalPerm}(pk, \hat{\pi}, \{\mathbf{c}_j\}_{j=1}^{2Z})$ and puts $\{\mathbf{c}'_j\}_{j=1}^{2Z}$ to the root
7  for $l = 0 : L - 2$ do
8      $\mathsf{src} = \mathcal{P}(x, l)$                         ▷ the $l$-th source bucket
9      $\mathsf{dst} = \mathcal{P}(x, l + 1)$                     ▷ destination bucket
10     $\mathsf{sib}$ = sibling of $\mathsf{dst}$                 ▷ sibling bucket
11     Client generates a random permutation $\pi$, computes $\hat{\pi} = \texttt{genPerm}(pk, \pi)$, and sends $\hat{\pi}$ to server
12     Client specifies $Z$ slots $\{s_j\}_{j=1}^{Z}$ in $\mathsf{src}$ which include all real blocks in $\mathsf{src}$ mapped to $\mathsf{sib}$ (pad to $Z$
         with dummy)
13     Server fetches $\{\mathbf{c}_j\}_{j=1}^{Z} = \{\mathsf{src}[s_j]\}_{j=1}^{Z}$ Server supplies $Z$ $\mathsf{RLWE}(0)$ as $\{\mathbf{c}_j\}_{j=Z+1}^{2Z}$
14     Server computes $\{\mathbf{c}'_j\}_{j=1}^{2Z} = \texttt{evalPerm}(pk, \hat{\pi}, \{\mathbf{c}_j\}_{j=1}^{2Z})$ and puts $\{\mathbf{c}'_j\}_{j=1}^{2Z}$ to $\mathsf{sib}$
15     Client generates a random permutation $\pi$, computes $\hat{\pi} = \texttt{genPerm}(pk, \pi)$, and sends $\hat{\pi}$ to server
16     Client specifies $Z$ slots $\{s_j\}_{j=1}^{Z}$ in $\mathsf{dst}$ which include all remaining untouched real blocks (pad to $Z$
         with dummy)
17     Server fetches $\{\mathbf{c}_j\}_{j=1}^{Z} = \{\mathsf{dst}[s_j]\}_{j=1}^{Z}$
18     Server fetches the other $Z$ blocks in $\mathsf{src}$ as $\{\mathbf{c}_j\}_{j=Z+1}^{2Z}$
19     Server computes $\{\mathbf{c}'_j\}_{j=1}^{2Z} = \texttt{evalPerm}(pk, \hat{\pi}, \{\mathbf{c}_j\}_{j=1}^{2Z})$ and puts $\{\mathbf{c}'_j\}_{j=1}^{2Z}$ to $\mathsf{dst}$
20     Server fills $\mathsf{src}$ with $2Z$ $\mathsf{RLWE}(0)$                ▷ $\mathsf{src}$ is now empty
21     Client marks all slots in $\mathsf{src}$ and $\mathsf{sib}$ as untouched and updates $\mathsf{id}$ to track the blocks currently
         residing in them
22 end
23 Client downloads $Z$ slots $\{\mathbf{c}_j\}_{j=1}^{Z}$ from $\mathsf{leaf} = P(x, L - 1)$ which include all real blocks (pad to $Z$ with
     dummy)
24 Client decrypts $\{\mathbf{c}_j\}_{j=1}^{Z}$ (both RLWE and AES) to get $\{b_j\}_{j=1}^{Z}$
25 Client re-encrypts using AES with fresh randomness to get $\{b'_j\}_{j=1}^{Z}$ and sends $\{b'_j\}_{j=1}^{Z}$ to server
26 Client generates a random permutation $\pi$, computes $\hat{\pi} = \texttt{genPerm}(pk, \pi)$, and sends $\hat{\pi}$ to server
27 Server RLWE-encrypts $\{b'_j\}$ to get $\{\mathbf{c}_j\}_{j=1}^{Z} = \{\mathsf{RLWE}(b'_j)\}_{j=1}^{Z}$
28 Server supplies $Z$ $\mathsf{RLWE}$ encryptions of zero as $\{\mathbf{c}_j\}_{j=Z+1}^{2Z}$
29 Server computes $\{\mathbf{c}'_j\}_{j=1}^{2Z} = \texttt{evalPerm}(pk, \hat{\pi}, \{\mathbf{c}_j\}_{j=1}^{2Z})$ and puts $\{\mathbf{c}'_j\}_{j=1}^{2Z}$ to $\mathsf{leaf}$
30 Client marks all slots in $\mathsf{leaf}$ as untouched and updates $\mathsf{id}$ to track the blocks currently residing in it
```

3.4 Correctness and Security

The access algorithm returns the data lastly written to the requested address unless some bucket overflows during eviction. A sibling or destination bucket overflows if more than Z blocks from the source bucket need to move into it. We will bound this probability at the end of this subsection.

To prove security, we argue that every piece of value the client reveals to the server contains no secret information and can hence be simulated.

In the access algorithm, the client first reveals a path $x = \mathsf{pos}[a]$. x is random following the standard tree-based ORAM security: x was selected the previous time block a was accessed and has not been revealed since then. Then, for each bucket on the path, the client specifies a slot, which contains either the untouched requested block a or an untouched dummy block. Any untouched block resides in a random slot of that bucket and once the slot is touched, it will not be touched again until the it is re-permuted in an eviction. Hence, the access algorithm is secure unless some bucket runs out of untouched dummy blocks before its next schedule eviction. At the end of this subsection, we will show that, except for negligible probability, a bucket is untouched less than Z times before it is involved in eviction, ensuring that it will not run out of untouched dummy blocks.

The eviction algorithm consists of $L - 1$ triplet evictions. In each triplet eviction, the client invokes the homomorphic permutation subroutine twice. The permutations and blocks are both encrypted under RLWE. The security of homomorphic permutation thus reduces to the IND-CPA security of the underlying RLWE encryption scheme, which can be reduced to the hardness of the decision RLWE problem. We remark that Peikert et al. [49] showed a reduction from worst-case ideal lattice problems to decision RLWE problems. In addition, the client specifies Z slots in the source bucket (line 12) and Z slots in the destination bucket (line 16). The locations of these slots look random as we explain below. The Z slots in the source bucket will look random because the source bucket has just been randomly permuted — the root bucket is immediately randomly permuted after being uploaded and each subsequent source bucket was the destination bucket of the previous triplet eviction. This argument also applies to the leaf bucket (line 23) as it was randomly permuted as a destination bucket just before that. The Z slots in the destination bucket will look random because they are untouched since the bucket was lastly permuted during eviction — again, unless the bucket is touched more than Z times since its last eviction, which happens with negligible probability.

It remains to choose suitable parameters Z and A to ensure that, except for negligible probability, the child buckets do not overflow during evictions and that buckets do not run out of dummy blocks. For dummy slots, a bucket at level $l > 0$ is permuted every $(2^{l-1}A)$ accesses (alternating between sibling and destination) and each access independently has a 2^{-l} probability of touching the bucket. Thus, the number of dummy slots touched before a bucket is re-permuted is captured by a binomial distribution. The probability of running out of dummy slots is hence the upper tail bound of the this distribution, $\Pr[X > Z]$ where X follows the above binomial distribution with mean $A/2$. The events for buckets to overflow conveniently follow the same binomial distribution, according to a more involved argument in Devadas et al. [22] (which is the reason we set the number of reserved dummy slots S to Z) A standard Chernoff bound shows that with $Z = A$, this probability is $\exp(-\Omega(Z))$, which suffices for a theoretical security proof. But the more precise binomial distribution will help us choose tighter parameters. In our implementation, we will set $Z = 254$ and $A = 249$ for a security level of 2^{-80} failure probability.

3.5 Bandwidth Analysis

We use pt to denote RLWE plaintext size and ct to denote RLWE ciphertext size. We let $F = ct/pt$ denote the ciphertext expansion factor of RLWE.

On each access, the client downloads one RLWE ciphertext. Thus, Onion Ring ORAM has online bandwidth blowup exactly F. On each eviction, the client (i) uploads A AES ciphertexts to the root bucket, and (ii) downloads Z RLWE ciphertexts and uploads Z AES ciphertexts to refresh the leaf.

The client also sends encrypted permutations during eviction. These bandwidth overheads can be absorbed by having multiple *chunks* per block. Each chunk is a RLWE ciphertext. The encrypted permutations are reused for every chunk in a block, so they account for very little bandwidth when the blocks have many chunks (cf. confirmed by experiments in Section 5.3).

Ignoring the bandwidth of encrypted permutation, each eviction consumes bandwidth $A \cdot pt + Z(pt + ct)$. The eviction bandwidth is amortized across A accesses. Hence, our construction has amortized bandwidth blowup

$$F + \frac{A \cdot pt + Z \cdot (pt + ct)}{A \cdot pt} = (F + 1)(1 + Z/A).$$

Since $A < Z$ and $F \geq 1$. Hence 4 is a lower bound on the bandwidth blowup of Onion Ring ORAM. Note that a smaller F also reduces Onion Ring ORAM's computation cost and server storage blowup.

4 Homomorphic Permutation from (Leveled) TFHE

In this section, we discuss one major technical contribution in this work – an efficient algorithm for performing a permutation on homomorphically encrypted data.

First, we briefly discuss the inefficiencies of the basic homomorphic permutation protocol in Onion ORAM. A straightforward way to implement homomorphic permutation is to use m 1-out-of-m selections. In this method, the client needs to send $O(m^2)$ ciphertexts and the server needs to perform $O(m^2)$ homomorphic multiplications. A better method is to use permutation networks (a.k.a sorting networks), which reduces both metrics above to $O(m \log m)$. This is indeed adopted in the AHE variant of Onion ORAM [22]. However, if we use permutation networks in the SWHE variant of Onion ORAM, its BGV-style ciphertext size needs to be increased by a factor of the multiplicative depth, which is $O(\log m)$ of permutation networks. The larger ciphertexts result in an $O(\log m)$ factor blowup in server storage and computation cost.

We base our algorithm on the `CMux` opertaion in TFHE and construct a homomorphic expansion algorithm allowing the server to generate the encryptions of swap bits from a small number of ciphertexts each encrypting many such bits, thus reducing the client to server communication overhead. We will see that using the `CMux` operation in TFHE to evaluate permutation networks on encrypted data avoids the above issues, and one reason is that for SWHE schemes like BGV, the noise growth is multiplicative in the depth $O(\log m)$, whereas in `CMux` the noise growth is *additive* (see Equation 1). This fact allows us to evaluate many permutation networks in sequence on large number of inputs, without much negative impact on either computational cost or ciphertext expansion.

4.1 Overview of Our Construction

One way to achieve homomorphic computation is to evaluate a Waksman permutation network [8] over homomorpically encrypted data. A Waksman permutation network is a circuit with m input and output wires consisting of $O(m \log m)$ controlled swap gates each specified by a swap bit. The depth of the network is $O(\log m)$. We encode the permutation π as the swap bits in a Waksman permutation network, which can be viewed as a circuit composed of controlled swap gates. A controlled swap takes two input messages x_0, x_1 and a swap bit b:

$$\texttt{CSwap}(b, x_0, x_1) = \begin{cases} (x_0, x_1) & \text{if } b = 0 \\ (x_1, x_0) & \text{if } b = 1. \end{cases}$$

A Waksman network on m inputs requires a total number of
$\omega(m) := \sum_{i=1}^{m} \lceil \log(i) \rceil \leq m \log m$ controlled swaps. At a high level, the client first configures a Waksman permutation network for π by locally computing the swap bits $b_1, \ldots, b_{\omega(m)}$. Then, the client encrypts all b_i's and sends them to the server. The server uses these encrypted swap bits to evaluate the permutation network.

Note that an encrypted version of `CSwap` can be built directly from TFHE's `CMux` operations. Therefore, we make use of the two encryption modes of TFHE to match the interface of `CMux`. The input ciphertexts are encrypted using RLWE mode. The swap bits are encrypted using RGSW mode. We can then evaluate the Waksman permutation network on these ciphertexts. From Equation 1, we can deduce that the noise of each output ciphertext satisfies

$$\mathsf{Var}(\mathbf{c}_{out}) \le \mathsf{Var}(\mathbf{c}_{in}) + O(\log m) \cdot V_{\text{GSW}}, \tag{2}$$

where the V_{GSW} term is independent of the noise of $\mathbf{c}_{in}$: it only depends on the noise of the RGSW-encrypted swap bits and the TFHE parameters. This observation accounts for a large performance improvement, because the small noise growth allows us to keep a constant ciphertext expansion ratio (hence ORAM bandwidth blowup) for sufficiently large blocks. Meanwhile, the homomorphic permutation takes $O(m \log m)$ operations, but importantly, the complexity of each operation only scales double-logarithmically with the length of the permutation m. We refer the readers to Section 4.2 for a detailed analysis.

Then, we turn our attention to reducing the size of the encrypted permutation. The client has several options to send the encrypted swap bits to the server. The naive way is to send each bit encrypted under RGSW, ready to be used in the permutation, which incurs a lot of communication in practice. The other end of extreme is to use an RLWE-to-LWE extraction followed by the *circuit bootstrapping* algorithm in [15, 16], which take LWE ciphertexts as input and outputs RGSW ciphertexts: this approach minimizes communication but incurs huge computational cost.

We strike a balance between communication and computation cost by adapting the homomorphic expansion algorithm in [3] (see Section 4.3) which makes use of RLWE key switching together with the TFHE external product. Roughly speaking, we pack multiple bits inside a small number of RLWE ciphertext. Then we let the server extract each encrypted bit and homomorphically expand it into a RGSW ciphertext to be used in the permutation. Our solution reduces the communication costs over the naive method by about three orders of magnitude, with the cost of some additional server computation which is independent of the block size. Hence for large blocks, the amount of additional server computation is small compared to the actual encrypted permutation network evaluation.

4.2 Efficiency Analysis of Homomorphic Permutation

Instead of using a single cryptosystem, our approach to homomorphic permutation makes use of two encryption schemes supported in TFHE. Namely, we encrypt the swapbits in RGSW and the input in RLWE. Then, the network evaluation is performed using the `CMux` operation (recalled in Section 2.3). We noted that the `CMux` operation works particularly nicely with our approach due to Equation 2. Moreover, in our construction inherited from [22], ORAM eviction requires us to perform L permutation networks in sequential manner. In this case, we can bound the output noise of all nodes after one eviction by

$$\mathsf{Var}(c_{out}) \le O(L \cdot \log m) \cdot V_{\text{GSW}}. \tag{3}$$

Below, we present an informal analysis to estimate the server's computation and storage overhead of our construction.

To prepare the analysis, we give some background on the relationship between the noise growth and computation complexity in TFHE. For the correctness of the RLWE decryption, we require the final noise to be smaller than $\frac{1}{2t}$. Also, the TFHE implementation chooses a cut-off modulus q, so that every torus element is represented by its most significant $\log q$ bits. Let α denote the standard deviation of the noise in the freshly generated ciphertexts, we have $V_{\text{GSW}} = O(\alpha^2)$. The correctness of the scheme requires $q = \Omega(\frac{1}{\alpha})$ (since otherwise the noise will be dominated by a larger rounding error). Now, taking square roots on both sides of Equation 3, we found that the standard deviation of the output noise is $\sigma \le O(\sqrt{L \log m})\alpha$. Hence it suffices to select $q = 2t \cdot O(\sqrt{L \log m})$, i.e., $\log q = \log t + O(\log \log m + \log L)$ to ensure decryption succeeds with an overwhelming probability. Recall that Each RLWE ciphertext is of $2n \log q$ bits and the underlying plaintext has $n \log t$ bits. Hence the server's storage blowup of our construction is

$$2\frac{\log q}{\log t} = 2 + \frac{O(\log \log m + \log L)}{\log t}.$$

Hence, we conclude that for any fixed m and L, with sufficiently $\log t$, the server's storage blowup in our construction approaches the limiting optimal value of 2, though we do not achieve this limiting value in our

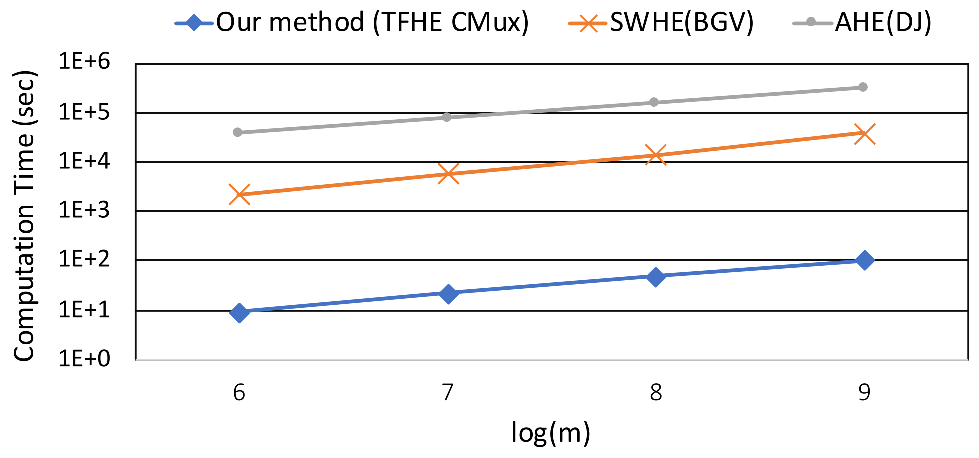


Figure 1: Single-core server time for computing a permutation network on m homomorphically encrypted inputs

current implementation: in our experiments, the ciphertext expansion factor is $2 \times 64/12 \approx 10.66$. This is due to the fact that the current TFHE library does not support increasing the parameters q. We envision that it will be possible with an arbitrary-precision implementation of TFHE.

On the other hand, the computation complexity of the `CMux` operation scales linearly with the complexity of multiplications of integers modulo q, which we assume to be $O(\log(q)^{1+\epsilon})$ for some ϵ between 0 and 1. Since $\log(q)$ scales linearly with $\log\log m + \log L$, the permutation input length m and the depth of the ORAM tree L have a mild impact on the complexity.

In Figure 1, we compare the concrete timings of our methods to evaluate a permutation network on m encrypted 128KB-blocks with the two methods proposed by [22]: the AHE approach using Dåmgard-Jurik cryptosystem and the SWHE approach using BGV. For AHE, we estimated the timing using the results in [4] with 1024-bit RSA modulus. For BGV, we used the HElib library [31]. We selected the parameters in HElib such that it provides 128 bits of security and it supports the depths of the network, namely $2\log(m) - 1$.

From Figure 1, we see that our method outperforms the SWHE and AHE approaches by roughly 2.5 and 4 orders of magnitudes, respectively. As for server's storage blowup, our method and AHE can both achieve $O(1)$ for sufficiently large blocks, whereas the SWHE method requires $O(\log m)$ blowup.

4.3 Reducing Client to Server Communication via Homomorphic Expansion

One problem remains of the above approach: during the ORAM eviction process, the client needs to send RGSW encryptions of $O(m \log m)$ swap bits to the server for each permutation of m encrypted blocks. However, the RGSW ciphertexts are large: each RGSW ciphertext is 32KB in TFHE and even larger for our parameter choices, whereas every ciphertext only encrypts a single bit. Thus, this naïve solution introduces a high communcication overhead when used in practice, which contradicts our initial goal of minimizing the concrete communication overhead of ORAM.

In order to reduce the client to server communication during ORAM eviction, we adapted a homomorphic expansion algorithm in Angel et al. [3], which uses repeated RLWE key switching to expand one RLWE ciphertext $\mathsf{RLWE}(X^j)$ into n ciphertexts $\mathsf{RLWE}(e_{ij})$ with $e_{ii} = 1$ and $e_{ij} = 0$ if $i \neq j$. Our algorithm shares some similarity with the one in [3] but they are different in two major aspects.

First, note that rather than expanding an integer index into a bit-vector as in [3], our task is to expand a ciphertext encrypting many bits into many encryptions of individual bits, For this task, we observed that the same idea can be applied to turn $\mathsf{RLWE}(\sum b_i X^i)$ into individual encryptions $\mathsf{RLWE}(b_i)$. To see how this is done, recall that the RLWE key switching operation could be used to perform a "substitution" operation

Algorithm 3: RLWE Expansion SubRoutine `expandRlwe`

```
Input: $\mathbf{c} = \mathsf{RLWE}(\sum_{i=0}^{n-1} b_i X^i), b_i \in \{0,1\}$
Output: $\mathbf{c}_i = \mathsf{RLWE}(n \cdot b_i), 0 \le i < n$
1  $\mathbf{c_0} := \mathbf{c}$
2  for $i = 1 : \log n$ do
3      $k = n/2^{i-1} + 1$
4      for $b = 0 : 2^i - 1$ do
5          $\mathbf{c}_{2b} = \mathbf{c}_b + \texttt{Subs}(\mathbf{c}_b, k)$
6          $\mathbf{c}_{2b+1} = \mathbf{c}_b - \texttt{Subs}(\mathbf{c}_b, k)) \cdot X^{-k}$
7      end
8  end
9  return $\{\mathbf{c}_j\}_{j=0}^{n-1}$
```

Algorithm 4: Homomorphic Expansion `homExpand`

```
Input: $\mathbf{c}_k = \mathsf{RLWE}(\sum_{i=0}^{n-1} \frac{b_i}{nB^{k+1}} X^i), b_i \in \{0,1\}$, $k = 0, 1, \dots, \ell - 1$
Input: $\mathbf{A} = \mathsf{RGSW}(-s)$
Output: $\mathbf{C}_i = \mathsf{RGSW}(b_i), 0 \le i < n$
1  for $k = 0 : \ell - 1$ do
2      $\mathbf{c}'_k := \texttt{expandRlwe}(\mathbf{c}_k)$
3  end
4  for $i = 0 : n - 1$ do
5      Initialize $\mathbf{C}_i$ as an empty $2\ell \times 2$ matrix of polynomials
6      for $k = 0 : \ell - 1$ do
7          $\mathbf{C}_i[k] = \mathbf{A} \boxdot \mathbf{c}'_k[i]$
8          $\mathbf{C}_i[k + \ell] = \mathbf{c}'_k[i]$
9      end
10 end
11 return $\{\mathbf{C}_i\}_{j=0}^{n-1}$
```

$\texttt{Subs}(\cdot, k)$ that transforms $\mathsf{RLWE}(\sum b_i X^i)$ to $\mathsf{RLWE}(\sum b_i (X^i)^k)$ for any odd integer k. For $k = n + 1$, we have $X^{ik} = (-1)^i X^i$. Now for a ciphertext $\mathbf{c} = \mathsf{RLWE}(\sum b_i X^i)$, we see that $\mathbf{c} + \texttt{Subs}(\mathbf{c}, n + 1)$ will encrypt $\sum_i 2b_{2i} X^{2i}$ and $X^{-1}(\mathbf{c} - \texttt{Subs}(\mathbf{c}, n + 1))$ will encrypt $\sum_i 2b_{2i+1} X^{2i}$. So in effect we extracted the even and odd parts of the message polynomial. Then, doing this recursively using substitutions with $k = n/2^s + 1$ for $s = 1, 2, \dots, \log(n) - 1$ results in encryptions of b_i. Algorithm 3 gives a description of this algorithm, which is used as a subroutine of our expansion algorithm. For completeness, we present the key switching and substitution algorithms in the appendix.

Also, we need to generate RGSW encryptions for the server to use in the homomorphic permutation, whereas in [3] it is sufficient to generate RLWE encryptions. To achieve this, we observe that the first ℓ rows of an RGSW encryption of b can be viewed as $\mathsf{RLWE}(b \cdot (-s)/B^k)$ for $1 \le k \le \ell$, where s is the secret key polynomial, and the second ℓ rows are $\mathsf{RLWE}(b/B^k)$. In order to generate the second half of the rows, we could repeat the subroutine in the previous paragraph ℓ times on $\mathsf{RLWE}(\sum b_i X^i / B^k)$. Then, we could perform the external product between $\mathsf{RLWE}(b_i/B^k)$ and $\mathsf{RGSW}(-s)$ to generate $\mathsf{RLWE}(b_i(-s)/B^k)$ for $k = 1, 2, \dots, \ell$. These ciphertexts together constitute an RGSW ciphertext encrypting b_i. See Algorithm 4 for a formal description of our homomorphic expansion algorithm.

Theorem 4.1 (Correctness of Algorithm 3)**.** *Suppose the RLWE key switching keys are instantiated with decomposition base B_{ks}, decomposition length ℓ_{ks} and noise variance θ_{ks}. Then, for an input ciphertext*

$\mathbf{c} = \mathsf{RLWE}(\sum_{i=0}^{n-1} b_i X^i)$, *Algorithm 1 outputs* $\mathbf{c}_j = \mathsf{RLWE}(n \cdot b_j)$ *with noise variance*

$$\mathsf{Var}(\mathbf{c}_j) \leq n^2 \mathsf{Var}(\mathbf{c}) + \frac{n^2 - 1}{3} V_{ks},$$

where $V_{ks} := \frac{n}{4B_{ks}^{2\ell_{ks}}} + \ell_{ks} \cdot n \cdot (\frac{B_{ks}}{2})^2 \theta_{ks}$.

Proof. See appendix. □

Theorem 4.2 (Correctness of Algorithm 4)**.** *Suppose* $\mathbf{A}$ *is an RGSW encryption of* $(-s)$ *with noise variance* $\theta_{\mathbf{A}}$*. If the input ciphertexts* $\mathbf{c}_k = \mathsf{RLWE}(\sum_{i=0}^{n-1} \frac{b_i X^i}{nB^{k+1}})$ *have noise variance bounded by* θ*, then Algorithm 4 outputs* $\mathbf{C}_i = \mathsf{RGSW}(b_i)$ *with noise variance*

$$\mathsf{Var}(\mathbf{C}_i) \leq n^3 \theta + \frac{n^3 - n}{3} V_{ks} + V_{ext},$$

where V_{ks} *is as in Theorem 4.1 and*

$$V_{ext} = 2\ell n \left(\frac{B}{2}\right)^2 \theta_{\mathbf{A}} + (1+n)n \left(\frac{1}{2B^\ell}\right)^2$$

Proof. See appendix. □

Computation Complexity of Algorithm 4 In addition to the ℓ calls to Algorithm 3 (a.k.a. `expandRlwe`), it requires $n \times \ell$ external products to generate n RGSW ciphertexts. Each `expandRlwe` requires n key switching operations, Hence the amortized cost for generating one RGSW ciphertext is ℓ external products plus ℓ key switchings, resulting in roughly ℓ^2 ring operations if we take $\ell_{ks} \approx \ell$.

Communication Complexity. With our homomorphic expansion algorithm, for n swap bits, the client sends ℓ ciphertexts to the server. The server then executes Algorithm 4 to generate n RGSW encryptions. The amortized communication is $\lceil \frac{\ell}{n} \rceil$ RLWE ciphertexts per swap bit. Note that the naïve method requires sending one RGSW ciphertext per swap bit, and the size of one RGSW ciphertext is 2ℓ times of an RLWE ciphertext. Hence our method saves communication by a factor of n, the ring dimension in TFHE. Since $n > 10^3$ in practice (our experiments used $n = 2^{11}$), our homomorphic expansion algorithm reduces the communication cost by *3 orders of magnitude* in our settings. We illustrate this improvement on communication in Figure 2.

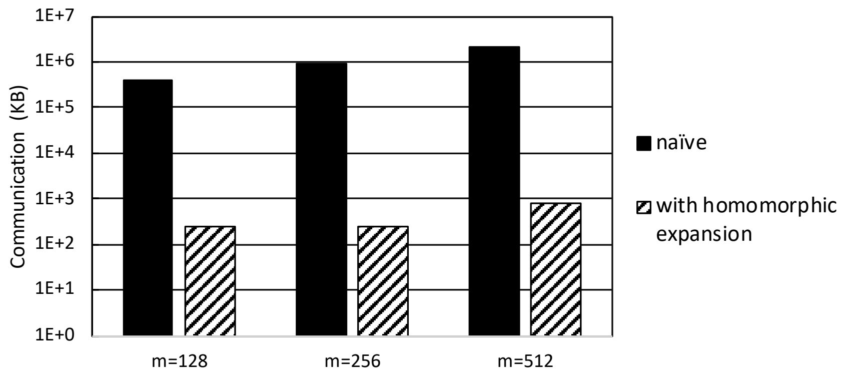


Figure 2: Client to server communication cost to send an encrypted permutation on $m \in \{128, 256, 512\}$ input blocks.

4.4 Putting it together

We separate the pseudocode for our homomorphic permutation into client part and server part, and include them in Algorithm 5 and Algorithm 6, respecitively. For line 1 in Algorithm 5, we note that there are standard ways to configure a permutation network to realize a particular permutation, and refer the readers to [33] for details of a configuration algorithm.

Algorithm 5: Client part of homomorphic permutation `genPerm`

```
  Input: Inputs a permutation $\pi$ on $m$ inputs
  Output: $\hat{\pi}$, encryptions of the swap bits
1 Configure a permutation network that realizes $\pi$ by computing the swap bits $\{b_i\} = \texttt{swapbits}(\pi)$ for
    $0 \leq i < \omega(m)$
2 $\mathbf{c}[j][k] = \mathsf{RLWE}(\sum_{i=0}^{n-1} \frac{b_{i+nj}}{nB^k} X^i)$ for $1 \leq k \leq \ell$, $1 \leq j \leq \lceil \omega(m)/n \rceil$.
3 return $\mathbf{c}$
```

Algorithm 6: Server part of homomorphic permutation `evalPerm`

```
  Input: $\hat{\pi}$, homomorphically encrypted swap bits in a Waksman network realizing the permutation $\pi$
  Input: $\mathbf{c}_i = \mathsf{RLWE}(\mu_i), 0 \leq i < m$
  Output: $\mathbf{c}'_i = \mathsf{RLWE}(\mu_{\pi(i)}), 0 \leq i < m$
1 $\hat{\Pi}$ = empty list of $omega(m)$ RGSW ciphertexts.
2 $\hat{\Pi} := \texttt{homExpand}(\mathbf{c}[1]) \| \cdots \| \texttt{homExpand}(\mathbf{c}[[k/n]])$
3 $\mathbf{c}'_i = \mathbf{c}_i$
4 evalWaksman($\hat{\pi}, \mathbf{c}'_i$)                         ▷ defined in appendix
5 return $\mathbf{c}'_i$
```

Correctness. The correctness of our homomorphic permutation protocol (Algorithms 5 and 6) follows from the correctness of the subroutines, which we prove in the appendix.

5 Implementation

5.1 Further optimizations

Reduce fresh ciphertext size via PRG. Observe that the first element (generally noted a) of every fresh RLWE ciphertext is a polynomial sampled uniformly at random. In practice, these polynomials can be generated from a pseudo-random generator initialized by a seed. In order to reduce the communication from the ORAM client to the ORAM server, the client could send only the second element of the RLWE ciphertexts together with the seed used to generate the a polynomials and ask the ORAM server to generate the a polynomials itself. This technique, already proposed in [14], could halve the size of ciphertexts from client to server.

Modulus switching. First proposed in [11], the modulus switching technique allows for noise control and ciphertext size reduction in homomorphic encryption. We use it to reduce the communication cost from ORAM server to ORAM client, which occurs both at online block retrieval and offline leaf refreshing phases. Put simply, modulus switching works as follows: suppose the server wishes to send a ciphertext $\mathbf{c} = (a, b)$ to the client where each polynomial has $\log q$ bits of coefficients. Instead of sending $\mathbf{c}$ itself, it only sends the most significant $\log q'$ bits of each coefficient. This reduces the ciphertext size by $\log q / \log q'$, with the side effect of adding some extra noise. The variance of the added noise is upper bounded by $(\frac{q'}{2q})^2(1 + ||s||_1) \leq (\frac{q'}{2q})^2(1 + n)$. For our parameters with $n = 2048$ and $\log(q) = 64$, we found that using

$\log(q') = 32$ does not affect the correctness of decryption. Thus, we can reduce the size of ciphertexts from server back to client by a half.

Combining the two techniques above, we reduced the effective ciphertext expansion factor F by roughly a half, from 10.66 to 5.33.

5.2 Experimental Setup

We run our experiments on an Azure virtual machine with the following configurations. The machine has an 8-core Intel Xeon E5-2673 CPU running at 2.30GHz and 32GB memory. We use all eight cores for the server, but only one core for the client.

ORAM parameters. In this paper, we set $Z = 254$. Thus, each homomorphic permutation takes $2Z = 508$ ciphertexts. The reason for this choice is that the packed swap bits for a 508-by-508 permutation fit in 16 RLWE ciphertexts. We then set $A = 249$ to obtain a security level of 2^{-80} failure probability. We use an ORAM tree with 15 levels and $N = 2^{22}$ blocks in total. We test three block sizes: 384KB, 768KB, and 1536KB. Each RLWE ciphertext encrypts 3KB. So the above sizes correspond to blocks broken up into 128, 256, and 512 RLWE ciphertexts, respectively.

TFHE parameters. In our implementation, we used the experimental version of TFHE, which is available in open source [16]. We use ring dimension $n = 2048$ and modulus $q = 2^{64}$. We fix the standard deviation of our error distribution to be $\alpha = 2^{-55}$, which yield about 120 bits of security according to the LWE estimator [2]. For the plaintext, we choose $t = 2^{12}$, so each ciphertext can encrypt up to $\log(t) \cdot N = 3$ KB of data. This gives $F = 64/12 = 5.33$. For the RGSW parameters, we use $B = 2^3$ and $\ell = 8$, except for the RGSW encryption of $-s$ we use $B = 2^7$ and $\ell = 7$. For RLWE key switching, we use $B_{ks} = 2^5$ and $\ell_{ks} = 10$.

5.3 Bandwidth and Computation Cost

In this subsection, we evaluate the bandwidth blowup and computation cost of Onion Ring ORAM and compare them to existing ORAM protocols.

Following the literature, we distinguish *online* and *amortized* cost. Online cost refers to the cost incurred between when the client makes an access and when it gets the returned data. Amortized cost refers to the total cost per access. Another related notion is *worst-case* cost. For Onion Ring ORAM, they are simply amortized costs multiplied by the eviction frequency A, so we omit reporting them. We remark that Onion Ring ORAM as presented has high worst-case cost, but de-amortization techniques of tree-based ORAMs have also been proposed in the literature [54, 19] and they apply to Onion Ring ORAM.

Bandwidth cost. Table 1 lists the bandwidth cost and blowup of Path ORAM, Ring ORAM, and Onion Ring ORAM. Path ORAM's bandwidth blowup under similar conditions (ORAM size, client-side position map) is $8\log(N/4) = 160\times$, half of which is incurred online. Ring ORAM under similar conditions should use a higher eviction frequency $A = 450$ and reserved number of dummies of $S = 500$ according to the analytic model in [52], giving an amortized bandwidth blowup of about $2.26L = 34.3\times$ (L is the same as Onion Ring ORAM since we use the same Z). Its online bandwidth is optimal due to the XOR technique [21]. Onion Ring ORAM has an online bandwidth blowup of $5.33\times$ and an amortized bandwidth blowup of $12.9\times$. Note that the theoretical formula in Section 3.5 gives $12.8\times$, confirming that the encrypted swap bits account for very small bandwidth after packing. Importantly, the cost of sending the encrypted swap bits is independent of the block size. Compared to Ring ORAM, Onion Ring ORAM reduces overall bandwidth blowup while increasing online bandwidth blowup.

Note that the homomorphic expansion technique plays a crucial role in our construction to achieve a low bandwidth blowup. Without using this technique, the encrypted swap bits become $2048\times$ larger and add 246MB of communication per access. In other words, the block size would have to be at least comparable to 246MB to achieve constant bandwidth blowup.

Computation cost. Table 2 shows the computation cost breakdown of Onion Ring ORAM.

Scheme	online bandwidth		amortized bandwidth	
	raw	blowup	raw	blowup
Path ORAM	30 MB	80	60 MB	160
Ring ORAM	384 KB	1	12.9 MB	34.3
Onion Ring ORAM	2 MB	5.33	4.8 MB	**12.9**

Table 1: Bandwidth cost and blowup under 384 KB block size

		Time (second)		
Block Size (KB)		384	768	1536
Online	Server add	0.005	0.01	0.02
	Client decrypt	0.11	0.19	0.38
	Total	0.11	0.2	0.4
Offline homomorphic permutation	Client genPerm	0.015	0.015	0.015
	Server expansion	31.7	31.7	31.7
	Server Waksman	15.1	30.1	60.2
	Total (of $2L$)	1404	1854	2757
	Amortized (over A)	5.6	7.4	11.1
Total		**5.7**	**7.6**	**11.5**

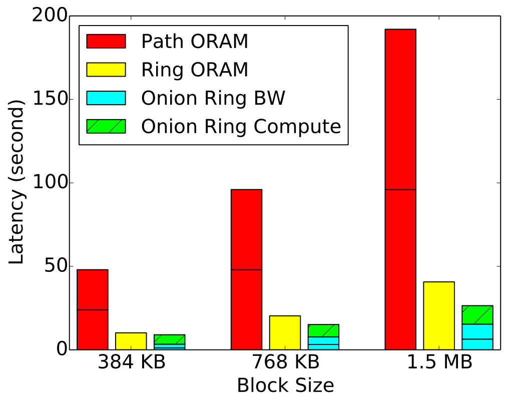
Table 2: Computation cost of each step in Onion Ring ORAM. The server uses 8 threads while the client uses 1 thread.

The computational component of online latency consists of the time for homomorphic addition on the server and the time for decryption on the client. The permuted bucket technique is effective in reducing online latency. As is shown in the table, homomorphic addition on the server takes very little time. Most of the online latency is incurred by the client to decrypt the block.

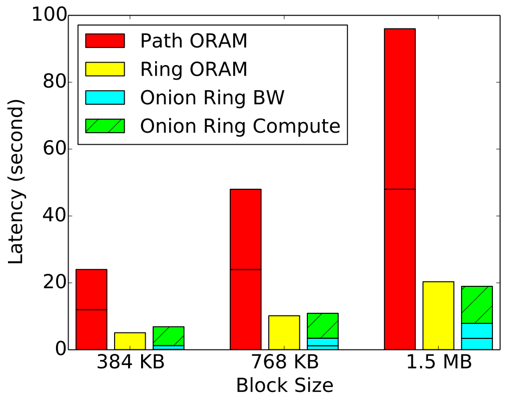
For each homomorphic permutation in eviction, the time client takes to generate encrypted swap bits is less than 0.02 second. The server takes 31.7 seconds to expand them into RGSW ciphertexts. These two components are the computational overheads of the homomorphic expansion techniques. Importantly, these two parts do not depend on the number of chunks in a block and hence do not depend on block size. Hence, for larger blocks, the overhead of homomorphic expansion on the total timing decreases. The time spent on homomorphically evaluating the Waksman permutation scales linearly with the number of ciphertext chunks. For 768KB block size (256 chunks), this step takes 30.1 seconds, which is comparable to the expansion step. One eviction uses 30 permutations, two per each non-leaf level, one for root and one for leaf. This cost is amortized among $A = 249$ accesses per eviction.

To summarize, again using 768KB as an example, the computation cost per accessing a 768KB clock is about 7.6 seconds, of which 7.4 seconds are spent offline (mostly on the server using 8 cores) and 0.2 second is spent online (mostly by the single-core client).

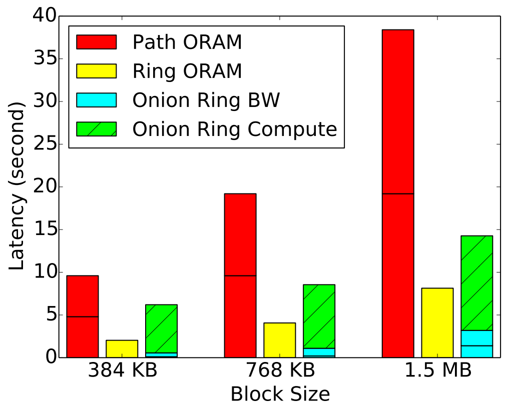
We remark that Path ORAM and Ring ORAM use minimum computation whose effects to performance can be ignored. Path ORAM uses no server computation and Ring ORAM uses only XOR operations. Clients in both schemes only use symmetric encryption. Onion ORAM with BGV or AHE, on the other hand, will incur prohibitive computational cost, 2.5 and 4 orders of magnitude, respectively, as shown in Figure 1.



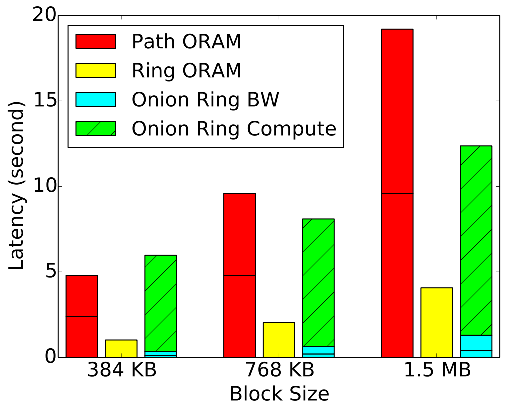
(a) 10 Mbps



(b) 20 Mbps



(c) 50 Mbps



(d) 100 Mbps

Figure 3: Online latency and overall (amortized) access time of Path ORAM, Ring ORAM and Onion ORAM under different block size and network throughput. The horizontal line in each bar divides online latency and offline time cost

5.4 Latency and Access Time

To compare the end-to-end latency between different ORAM protocols, we need to assume a network throughput between the client and server. We did not implement the network protocol between client and server. Instead, we simulate the online latency and overall amortized timing for Path ORAM, Ring ORAM, and Onion Ring ORAM under different network throughput. The simulations do not add network round-trip delays. Since all three ORAMs have a single round trip (recall that the position map is stored locally), we expect round-trip delays to affect all three ORAMs similarly.

In Figure 3, a horizontal line in each bar divides the time cost into two parts, online latency at the bottom and offline time cost on top. For Path ORAM, about half of the time cost is incurred online. Ring ORAM incurs almost its entire cost offline since its online bandwidth blowup is 1. Onion Ring ORAM also has larger online bandwidth than Ring ORAM but still give good online latency.

From Figure 3, we observe that the comparison between end-to-end latency heavily depends on network throughput. When network throughput is low, communication is the bottleneck of access time and Onion Ring ORAM outperforms Path ORAM and Ring ORAM due to its smaller bandwidth blowup. Taking the configuration of 1.5 MB block size and 10 Mbps network throughput as an example, the amortized access time Onion Ring ORAM is about 26.5 seconds, compared to 40.7 seconds for Ring ORAM and 192 seconds for Path ORAM. As the network throughput increases, the amortized access time of Path ORAM and Ring

ORAM scales inversely proportionally. The communication component of Onion Ring ORAM's access time also scales inversely proportionally, but its overall time cost converges to the aforementioned computation time. When the network throughput is at 100Mbps, our construction still outperforms Path ORAM.

In summary, Onion Ring ORAM is preferred in a cloud storage setting where the client-server communication is expensive or limited. Under our experimental settings, the cross-over point with Path ORAM is at about 100 Mbps and the cross-over point with Ring ORAM is at about 20 Mbps.

Lastly, we make a remark on S3ORAM [32]. It was left out in our performance comparisons, since it is under a different security model. By assuming three non-colluding servers, S3ORAM offers a better performance for slow client-server network settings, given that the inter-server network throughput is high. For example, when the client-server network is 9Mbps and inter-server network is 250Mbps, S3ORAM takes about 3 seconds to access a 512KB block.

6 Conclusions and Future Directions

We presented the first efficient single-server ORAM with $O(1)$ bandwidth blowup which we call Onion Ring ORAM. Our construction is based on the Onion ORAM and the TFHE homomorphic encryption scheme. To achieve our efficiency goal, we built an efficient homomorphic permutation protocol, and incorporated the permuted buckets and the XOR trick from Ring ORAM. We implemented our ORAM scheme and performed experiments in various network bandwidth settings. We observed that in low-throughput networks, our construction can have better end-to-end latency than state-of-the-art ORAM constructions such as Path ORAM and Ring ORAM. We believe our work can serve as the first step to exploring practical impacts of cutting-edge homomorphic encryption techniques to improve the efficiency of ORAM.

We list some directions for future work:

Reducing bandwidth blowup. In our implementation, we have the effective ciphertext expansion factor $F = 5.33$ and $Z/A = 254/249 = 1.02$, which results in the $12.9\times$ bandwidth blowup we reported (the encrypted swap bits contributed little to this blowup). Note that Z/A is already close to their theoretical limit of 1. Nonetheless, from the heuristic analysis in Section 4.2, we see that there is still a lot of room for improving the ciphertext expansion factor F. We did not implement these improvements in this work, since a further reduction of F requires a larger precision $\log q$, and the TFHE library only supports $\log q$ up to 64, due to the use of double precision complex FFT. Hence, we leave the task for reducing F to future work.

Malicious security. Our implementation assumes an honest-but-curious server. If security against a malicious server is desired, there are two options. The first one is the position-map-based message authentication code integrity verification for ORAM [24]. In fact, since we store the position map locally, this method degenerates to per-block count-based message authentication codes. It adds very small overhead: one HMAC operation on the client per access. With this method, the client can reliably detect any incorrect return data.

However, the above method may allow a subtle type of leakage through detection. For example, say the server overwrites a slot with a dummy block. If the slot contained a dummy block to begin with, then this action will not result in any observable outcome to the client; but if the overwritten block was a real block, the client will detect incorrect return data. If the client stops interacting with the server upon detection (which seems to be a natural thing to do), the server learns whether the overwritten block was real or dummy. Nevertheless, this leakage seems reasonable: it only leaks the time of detection, which is akin to the total number of accesses leakage allowed in the ORAM definition. Thus, we believe this method still provides a reasonable solution to malicious security. Onion ORAM gave a method based on erasure coding and the "cut-and-choose" trick to defend against a malicious server without any extra leakage [22]. Their method works for any server computation ORAM but it is largely theoretical and incurs large overhead. It is interesting future work to design efficient defenses against malicious servers for our construction.

References

[1] Ittai Abraham, Christopher Fletcher, Kartik Nayak, Benny Pinkas, and Ling Ren. Asymptotically tight bounds for composing ORAM with PIR. In *IACR International Workshop on Public Key Cryptography*, pages 91–120. Springer, 2017.

[2] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Mathematical Cryptology*, 9(3):169–203, 2015.

[3] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 962–979. IEEE, 2018.

[4] Anastasov Anton. Implementing onion oram: A constant bandwidth oram using ahe. 2016.

[5] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In *International Workshop on Public Key Cryptography*, pages 131–148. Springer, 2014.

[6] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, and Elaine Shi. OptORAMa: Optimal oblivious RAM. Technical report, Cryptology ePrint Archive, Report 2018/892, 2018.

[7] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 34(4):36–42, 2014.

[8] Bruno Beauquier and E Darrot. On arbitrary size waksman networks and their vulnerability. *Parallel Processing Letters*, 12(03n04):287–296, 2002.

[9] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 837–849. ACM, 2015.

[10] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 309–325, 2012.

[11] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on Computing*, 43(2):831–871, 2014.

[12] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 668–679. ACM, 2015.

[13] Zhao Chang, Dong Xie, and Feifei Li. Oblivious RAM: a dissection and experimental evaluation. *Proceedings of the VLDB Endowment*, 9(12):1113–1124, 2016.

[14] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, pages 3–33, 2016.

[15] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, pages 377–408, 2017.

[16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: experimental-tfhe repository. https://github.com/tfhe/experimental-tfhe, 2017.

[17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption library. https://tfhe.github.io/tfhe/, August 2016.

[18] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. Cryptology ePrint Archive, Report 2018/421, 2018. `https://eprint.iacr.org/2018/421`.

[19] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 727–743, 2018.

[20] Ivan Damgård and Mads Jurik. A generalisation, a simpli. cation and some applications of paillier's probabilistic public-key system. In *International Workshop on Public Key Cryptography*, pages 119–136. Springer, 2001.

[21] Jonathan L Dautrich Jr, Emil Stefanov, and Elaine Shi. Burst ORAM: Minimizing ORAM response times for bursty access patterns. In *USENIX Security Symposium*, pages 749–764. USENIX Association, 2014.

[22] Srinivas Devadas, Marten van Dijk, Christopher Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography Conference*, pages 145–174. Springer, 2016.

[23] Christopher Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 3–8. ACM, 2012.

[24] Christopher Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive oram:[nearly] free recursion and integrity verification for position-based oblivious ram. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 103–116. ACM, 2015.

[25] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 1–18. Springer, 2013.

[26] Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. Private database access with HE-over-ORAM architecture. In *International Conference on Applied Cryptography and Network Security*, pages 172–191. Springer, 2015.

[27] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. *IACR Cryptology ePrint Archive*, 2013:340, 2013.

[28] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194. ACM, 1987.

[29] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[30] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 95–100. ACM, 2011.

[31] Shai Halevi and Victor Shoup. HElib-An Implementation of homomorphic encryption. https://github.com/homenc/HElib, May 2019.

[32] Thang Hoang, Ceyhun D Ozkaptan, Attila A Yavuz, Jorge Guajardo, and Tam Nguyen. S3ORAM: A computation-efficient and constant client bandwidth blowup ORAM with Shamir secret sharing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 491–505. ACM, 2017.

[33] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.

[34] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security*, volume 20, page 12, 2012.

[35] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Inference attack against encrypted range queries on outsourced databases. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 235–246. ACM, 2014.

[36] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1340. ACM, 2016.

[37] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)-security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the 23rd annual ACM-SIAM symposium on Discrete Algorithms*, pages 143–156. Society for Industrial and Applied Mathematics, 2012.

[38] Eyal Kushilevitz and Tamer Mour. Sub-logarithmic distributed oblivious ram with small block size. *arXiv preprint arXiv:1802.05145*, 2018.

[39] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In *Annual International Cryptology Conference*, pages 523–542. Springer, 2018.

[40] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGPLAN Notices*, 50(4):87–101, 2015.

[41] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *Theory of Cryptography*, pages 377–396. Springer, 2013.

[42] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.

[43] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 311–324. ACM, 2013.

[44] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Efficient private file retrieval by combining ORAM and PIR. In *Network and Distributed System Security*, 2014.

[45] Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. Constant communication oram with small blocksize. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 862–873. ACM, 2015.

[46] Kartik Nayak, Christopher Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal. HOP: Hardware makes obfuscation practical. In *Network and Distributed System Security*, 2017.

[47] Rafail Ostrovsky and Victor Shoup. Private information storage. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 294–303. ACM, 1997.

[48] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 871–882. IEEE, 2018.

[49] Chris Peikert, Oded Regev, and Noah Stephens-Davidowitz. Pseudorandomness of ring-lwe for any ring and modulus. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 461–473. ACM, 2017.

[50] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security Symposium*, pages 431–446, 2015.

[51] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93, 2005.

[52] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX Security Symposium*, pages 415–430, 2015.

[53] Ling Ren, Christopher Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Design and implementation of the ascend secure processor. *IEEE Transactions on Dependable and Secure Computing*, 2018.

[54] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *IEEE Symposium on Security and Privacy*, pages 198–217. IEEE, 2016.

[55] Sajin Sasy, Sergey Gorbunov, and Christopher Fletcher. Zerotrace: Oblivious memory primitives from intel SGX. In *Network and Distributed System Security*, 2018.

[56] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with $O(\log^3 N)$ worst-case cost. In *International Conference on The Theory and Application of Cryptology and Information Security*, pages 197–214. Springer, 2011.

[57] Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious ram protocol. *Journal of the ACM*, 65(4):18:1–18:26, 2018.

[58] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 247–258. ACM, 2013.

[59] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy*, pages 253–267. IEEE, 2013.

[60] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. In *Network and Distributed System Security*, 2012.

[61] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

[62] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, pages 617–635, 2009.

[63] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434. ACM, 2017.

[64] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861. ACM, 2015.

[65] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 139–148. ACM, 2008.

[66] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[67] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium*, pages 707–720, 2016.

A RLWE Key Switching and Substitution

The algorithms below are known and we include them for the sake of self-containment. The RLWE key switching procedure transforms an encryption of message μ under secret s' to an encryption of same message under secret s. It is described in Algorithm 7 below.

Algorithm 7: RLWE Key Switching Algorithm

```
  Input: $\mathbf{c} = \mathsf{RLWE}_{s'}(\mu)$, $B_{ks}$ and $\ell_{ks}$
  Input: $\mathsf{KS}_{s'\to s}[j] = \mathsf{RLWE}_s(s'/B_{ks}^j)$ for $j = 1 : \ell_{ks}$
  Output: $\mathbf{c}' = \mathsf{RLWE}_s(\mu)$
1 $\mathbf{c} = (c_0, c_1)$
2 Decompose $c_0$ as $\sum_{j=1}^{\ell_{ks}} c_0[j]/B_{ks}^j + \epsilon_{dec}$ with $||c_0[j]||_\infty \le B_{ks}/2$ and $||\epsilon_{dec}||_\infty \le \frac{1}{2B_{ks}^\ell}$
3 return $\mathbf{c}' = (0, c_1) - \sum_{j=1}^{\ell_{ks}} c_0[j] \cdot \mathsf{KS}_{s'\to s}[j]$
```

Lemma A.1. *Suppose the input to Algorithm 7 is an RLWE ciphertext with noise variance θ and assume the key switching keys have noise variance θ_{ks}. Algorithm 7 outputs $c' = \mathsf{RLWE}_s(\mu)$ with*

$$\mathsf{Var}(\mathbf{c}') \le \theta + \frac{n}{4B_{ks}^{2\ell_{ks}}} + \ell_{ks} \cdot n \cdot \left(\frac{B_{ks}}{2}\right)^2 \theta_{ks}.$$

Proof. Define $\varphi_s(a, b) = b - as$. We have

$$\begin{aligned}
\varphi_s(\mathbf{c}') &= c_1 - \sum_j c_0[j]\varphi_s(\mathsf{KS}[j]) \\
&= c_1 - \sum_j c_0[j](s'/B_{ks}^j + e_{ks}^j) \\
&= \varphi_{s'}(\mathbf{c}) - (\epsilon_{dec}s' + \sum_j c_0[j]e_{ks}^j).
\end{aligned}$$

The result follows by noting that the terms each term e_{ks}^j has variance θ_{ks} and $||s'||_1 \leq n$. □

Since this key switching algorithm is not available in the TFHE library, we implemented it ourselves. Note that the only difference between this variant and the usual key switching for BGV scheme is that we allow small errors for the decomposition.

It is easy to construct the substitution algorithm $\texttt{Subs}(\mathbf{c}, k)$ from key switching. We include it for the reader's convenience in Algorithm 8 below.

Algorithm 8: RLWE Substitution Algorithm $\texttt{Subs}(\mathbf{c}, k)$

```
  Input: $\mathbf{c} = \mathsf{RLWE}_s(\mu(X))$; odd integer $k$
  Input: $\mathsf{KS}_{s(X^k)\to s}[j]$ for $j = 1 : \ell_{ks}$
  Output: $\mathbf{c}' = \mathsf{RLWE}_s(\mu(X^k))$
1 $\mathbf{c} = (c_0, c_1)$
2 Let $\mathbf{c}' = (c_0', c_1') = (c_0(X^k), c_1(X^k))$
3 Decompose $c_0'$ as $\sum_{j=1}^{\ell_{ks}} c_0'[j]/B_{ks}^j + \epsilon_{dec}$ with $||c_0'[j]||_\infty \leq B_{ks}/2$ and $||\epsilon_{dec}||_\infty \leq \frac{1}{2B_{ks}^{\ell}}$
4 return $\mathbf{c}' = (0, c_1) - \sum_{j=1}^{\ell_{ks}} c_0[j] \cdot \mathsf{KS}_{s(X^k)\to s}[j]$
```

To see the correctness of Algorithm 8, note that $\mathbf{c}' = (c_0(X^k), c_1(X^k))$ is an encryption of $\mu(X^k)$ under $s(X^k)$. This step only replaces the noise polynomial $e(X)$ with $e(X^k)$, which does not affect the maximal variance on its coefficients. Then, key switching from $s(X^k)$ to $s(X)$ will output an encryption of $\mu(X^k)$ under the original secret key s. The noise growth of $\texttt{Subs}$ is exactly the same as that of the underlying key switching.

B Proofs

Proof of Theorem 4.1. For convenience, we denote $V = \frac{n}{4B_{ks}^{2\ell_{ks}}} + \ell_{ks} \cdot n \cdot (\frac{B_{ks}}{2})^2 \theta_{ks}$. So we have $\mathsf{Var}(\texttt{Subs}(\mathbf{c}, k)) \leq \mathsf{Var}(\mathbf{c}) + V$. Now Algorithm 3 has $\log(n)$ iterations, and let $\mathbf{c}^i$ denote any single ciphertext obtained after the i-th iteration (with $\mathbf{c}^0 = \mathbf{c}$ the input ciphertext), and let e^i denote the error term in $\mathbf{c}^i$. Then we have the relation

$$e^i = e^{i-1} \pm e^{i-1}(X^k) + \epsilon_{ks}$$

where we use ϵ_{ks} to denote the key switching noise term whose variance is bounded above by V. Thus, we have

$$\mathsf{Var}(\mathbf{c}^i) \leq 4\mathsf{Var}(\mathbf{c}^{i-1}) + V.$$

Applying this relation recursively, we have

$$\begin{aligned}\mathsf{Var}(\mathbf{c}^i) &\leq 4^i\mathsf{Var}(\mathbf{c}) + V \cdot (1 + 4 + \ldots + 4^{i-1}).\\ &= 4^i\mathsf{Var}(\mathbf{c}) + \frac{4^i - 1}{3}V.\end{aligned}$$

The claim in Theorem 4.1 follows by taking $i = \log(n)$. □

Proof of Theorem 4.2. Since the first ℓ rows of the output of Algorithm 4 are obtained as an external product involving the second ℓ rows, it suffices to look at the first row of each ouptut. Let c be any output ciphertext of $\texttt{expandRlwe}$ with variance $\theta_{\mathbf{c}}$, and let $\theta_{\mathbf{A}}$ denote the variance of the RGSW encryption of $-s$. Then, we have

$$\begin{aligned}
\mathsf{Var}(\mathbf{A} \boxdot \mathbf{c}) &\leq \theta_{\mathbf{c}} ||(-s)||_2^2 + 2\ell n \left(\frac{B}{2}\right)^2 \theta_{\mathbf{A}} \\
&\quad + ||(-s)||_2^2 (1+n) \left(\frac{1}{2B^\ell}\right)^2 \\
&\leq \theta_{\mathbf{c}} n + 2\ell n \left(\frac{B}{2}\right)^2 \theta_{\mathbf{A}} + (1+n)n \left(\frac{1}{2B^\ell}\right)^2 \\
&= \theta_{\mathbf{c}} n + V_{ext}.
\end{aligned}$$

The last inequality follows from the fact s has binary coefficients. Now from Theorem 4.1, we have

$$\theta_{\mathbf{c}} \leq n^2\theta + \frac{n^2-1}{3} V_{ks},$$

where θ is the noise variance of the input ciphertexts. Substituting this into the previous inequality finishes the proof. □

C The `evalWaksman` Algorithm

We present the algorithm for `evalWaksman` in Algorithm 9. Note that it is mostly a rewriting the of the algorithm to evaluate a cleartext Waksman permutation network in [8], with the only difference that the TFHE `CMux` operation replaces the swapping of two values.

Algorithm 9: `evalWaksman`

```
Input: A vector $\hat{\Pi}$ of $\omega(m)$ RGSW encyrptions of the bits in swapbits($\pi$)
Input: a vector $\mathbf{c}[i] = \mathsf{RLWE}(\mu_i), 0 \leq i < m$
Output: $\mathbf{c}$ is modified in-place such that $\mathbf{c}[i] = \mathsf{RLWE}(\mu_{\pi(i)}), 0 \leq i < m$
if $m \leq 1$ then
    return
end
$ins = \lfloor m/2 \rfloor$
Initialize two arrays $\mathbf{c}^{up}, \mathbf{c}^{down}$
for $i = 0 : ins - 1$ do
    $temp = \texttt{CMux}(\hat{\Pi}[i], \mathbf{c}[2i], \mathbf{c}[2i+1])$
    $\mathbf{c}[2i+1] = \mathbf{c}[2i] + \mathbf{c}[2i+1] - temp$
    $\mathbf{c}[2i] = temp$                    ▷ using CMux to swap $\mathbf{c}[2i]$ and $\mathbf{c}[2i+1]$
end
for $i = 0 : ins - 1$ do
    $\mathbf{c}^{up}[i] = \mathbf{c}[2i]$
    $\mathbf{c}^{down}[i] = \mathbf{c}[2i+1]$
end
if $m$ is odd then
    $\mathbf{c}^{down}[ins] = \mathbf{c}[m-1]$
end
$j_1 = ins + \omega(ins)$
$j_2 = j_1 + \omega(ins + (m\%2))$
evalWaksman($\mathbf{c}^{up}, \hat{\Pi}[ins : j_1]$)
evalWaksman($\mathbf{c}^{down}, \hat{\Pi}[j_1, j_2]$)
for $i = 0 : ins - 1$ do
    $\mathbf{c}[2i] = \mathbf{c}^{up}[i]$
    $\mathbf{c}[2i+1] = \mathbf{c}^{down}[i]$
end
if $m$ is odd then
    $\mathbf{c}[m-1] = \mathbf{c}^{down}[ins]$
end
$outs = \lfloor m/2 \rfloor - 1$
if $m$ odd then
    $outs = outs + 1$
end

for $i = 0 : outs - 1$ do
    $temp = \texttt{CMux}(\hat{\Pi}[j_2 + i], \mathbf{c}[2i], \mathbf{c}[2i+1])$
    $\mathbf{c}[2i+1] = \mathbf{c}[2i] + \mathbf{c}[2i+1] - temp$
    $\mathbf{c}[2i] = temp$
end
return
```