Panacea: Non-interactive and Stateless Oblivious RAM

Kelong Cong, Debajyoti Das, Georgio Nicolas, and Jeongeun Park

imec-COSIC, KU Leuven, Leuven, Belgium.
{kelong.cong,debajyoti.das,georgio.nicolas,jeongeun.park}@esat.kuleuven.be

Abstract. Oblivious RAM (ORAM) allows a client to outsource storage to a remote server while hiding the data access pattern from the server. Many ORAM designs have been proposed to reduce the computational overhead and bandwidth blowup for the client. A recent work, Onion Ring ORAM (CCS'19), is able to achieve $O(1)$ bandwidth blowup in the online phase using fully homomorphic encryption (FHE) techniques. However, it requires a computationally expensive client-side offline phase to do so. Furthermore, it is a stateful construction, which means that the client has to maintain a state of the database locally. We present Panacea: a novel design of ORAM based on FHE techniques, that is non-interactive and stateless, achieves $O(1)$ bandwidth blowup, and does not require an expensive offline phase for the client to perform; in that sense, our design is the first of its kind among other ORAM designs. To provide the client with such performance benefits, our design pushes all of expensive computations to the resourceful server. We additionally show how to boost the server performance significantly using *probabilistic batch codes* at the cost of only 1.5x in additional bandwidth blowup and 3x expansion in server storage. Our experimental results show that our design, with the employment of this batching technique, is practical in terms of server computation overhead as well. Specifically, for a database size of 2^{19}, it takes only 1.16 seconds of amortized computation time for a server to respond to a query. As a result of our client's statelessness, low computational overhead and practical computational overhead with the server, our design is ideal to be deployed as a cloud-based privacy-preserving storage outsourcing solution.

1 Introduction

Oblivious RAM (ORAM) is a cryptographic primitive which allows a client to store its private data on an untrusted server without leaking any information to the server or any observer about the data or the data access patterns (sequence of read/write operations and the addresses of those operations).

For efficient *storage outsourcing*, where a client with low availability of local memory can utilize a cloud server to store its private data, an ideal ORAM design should incur low computational overhead and *bandwidth blowup*[1] for the client. However, the ORAM designs from the classical model [17,19,28,29], where the server acts as a plain storage device and supports only read or write operations, must incur $\Omega(\log n)$ bandwidth blowup for a database size of n elements. To overcome such high overhead on the client, many designs have been proposed in the *server computation model*, utilizing the computational power of the server to reduce the communication overhead [23,26,25,2,9,14,27]. The server computation model is also more realistic since a cloud server can have much more computational and memory resources than the client. Most of these designs still require $O(\log n)$ bandwidth blowup.

Only until recently were designs based on fully homomorphic encryption (FHE) techniques such as Onion ORAM [10] and Onion Ring ORAM [3] able to achieve a bandwidth blowup of $O(1)$. Despite their low asymptotic guarantees, both of these designs still incur significant overhead

[1] *Bandwidth blowup* is defined as the ratio between the communication cost of ORAM and the non-private case where the access pattern is not hidden.

on the client (in terms of both computational cost and bandwidth blowup). The construction of Onion ORAM is of purely theoretical interest and really expensive to implement in practice [18]. On the other hand, Onion Ring ORAM requires an expensive client-side offline phase to achieve its performance guarantees.

Can we achieve an ORAM with $O(1)$ bandwidth blowup and low computation overhead for clients without introducing an expensive offline phase? Is it possible to achieve so with a non-interactive and stateless design? We answer affirmatively to those questions by proposing a novel ORAM design: Panacea. Based on fully homomorphic encryption (FHE) techniques, Panacea achieves $O(1)$ bandwidth blowup and low client-side computational overhead by pushing almost all the computation to the server: The client only needs to encrypt the queries it sends to the server and decrypt responses upon reception; and thus does not need to take part in any expensive offline phase.

Both Onion ORAM and Onion Ring ORAM are inspired from the binary tree-based ORAM framework proposed by Shi et al. [23], and they use FHE techniques to reduce bandwidth blowup. Our design idea emanates from the following observation: such tree-like structures mainly help in minimizing the number of client-server interactions and are not required when FHE techniques are used. Instead, our design employs a technique called *homomorphic de-multiplexing* (adapted from [8]) which allows the server operate on all elements in a database while accessing the client-indexed element and keeping the server oblivious to which specific element was accessed. In that sense, Panacea introduces a completely new technique for ORAM protocol design.

We present two variations of Panacea: the first version takes the above idea directly, optimizing the number of bits required to represent an address and the total amount of computation required by the server. Since, it does not rely on any specific database structure, it does not require the client to maintain the state of the database locally (unlike Onion Ring ORAM or Path ORAM [24]). Therefore, this design allows the client to be completely stateless[2], and does not incur any memory overhead. With such a stateless design, the client can query the server, go offline, and then retrieve the results whenever it comes back online. Another notable advantage is that the client can use its key to access the database from multiple devices without the need for state synchronization, similar to the advantages of stateless digital signatures when compared to stateful digital signatures. Our evaluations based on a prototype implementation show that the computation overhead for the server remains reasonably practical while achieving a stateless, non-interactive client which is not required to perform any expensive offline phase after decrypting a response.

Our second version leverages existing batching techniques from the literature to amortize the server's computational cost for a batch of k queries at the expense of adding some bandwidth blowup. More specifically, we use probabilistic batch codes (PBC) which incur 1.5x of additional bandwidth and increase the database size by 3x in our settings. PBCs also introduce a very small failure probability of one-in-a-trillion. In this context, a failure translates to only some (but not all) of k read/write queries being executed. Nonetheless, such a failure does not leak any information to the server. PBCs are traditionally used for load-balancing applications and PIR schemes, and cannot be directly applied to an encrypted non-static database because mutation to individual elements causes consistency issues in the PBC encoding. We solve this problem using our *consistency correction* technique. To the best of our knowledge, this is the first time that PBCs are applied to an encrypted non-static database. While the asymptotic complexity of Panacea is higher compared

[2] The client needs only to maintain static states such as private keys, but it does not need to maintain a private dynamic state that depends on the state of the database

to that of the stateful design, we give concrete performance results on databases with up to 2^{19} data elements and show that our construction remains highly practical to deploy with around 1.16 seconds of amortized computation time for the server to respond to a query, and 8x of bandwidth blowup when communicating with the client.

To summarize our contributions:

1. we present a novel ORAM design, where a client can be stateless, based on the homomorphic de-multiplexer technique (Sections 4 and 5);
2. we use batching techniques from literature in combination with our consistency correction algorithm to make the server-side computational cost practical (Section 6);
3. we implement and evaluate the performance of our ORAM design along with the batching technique and describe a cloud-based deployment strategy (Section 7).

2 Background and Related Work

2.1 ORAM Syntax

We consider a single-server-single-client ORAM setup where a client wants to outsource the storage of its private data to an untrusted server. For a single query, the ORAM algorithm provides an Access function to the client which takes as input a 3-tuple $(\mathsf{a}, \mathsf{op}, \mathsf{data})$, where a represents the logical address of the block requested by the client, $\mathsf{op} \in \{\mathsf{Read}, \mathsf{Write}\}$, and data is the data being written to the block if $\mathsf{op} = \mathsf{Write}$ (if $\mathsf{op} = \mathsf{Read}$ the block is not updated and data is ignored). The goal of the ORAM algorithm is to completely hide the data access pattern (which blocks were read/written) from the server.

We consider that the client fetches/stores data on the server in *blocks*, where each block is of B bits. A typical size of a block for cloud storage could be a few to several hundred KB. We consider that the database has n distinct data blocks stored on the server. All relevant notations are summarized in Table 2.

One main metric to measure the ORAM algorithm's overhead is the *bandwidth blowup* which is defined as the ratio between the communication cost of ORAM and the non-private case where the access pattern is not hidden.

Threat Model. In our implementation, we consider an honest-but-curious server, i.e., the server follows the prescribed protocol correctly. However, the server can store all the ciphertexts and other data sent by the client and, afterwards, act as a probabilistic polynomial time adversary to perform additional computations in order to infer the private data or the access pattern of the client. We adopt a standard security definition similar to [26,21].

Definition 1 (Security Definition) *Let,*

$$\mathbf{y} := \{(\mathsf{a}_1, \mathsf{op}_1, \mathsf{data}_1), \ldots, (\mathsf{a}_s, \mathsf{op}_s, \mathsf{data}_s)\}$$

denote a sequence of data accesses by the client, where a_i denotes the address of the block being read from or written to on the i-th access, op_i denotes the whether the operation on a_i is a read or write, data_i denotes the data if the i-the operation is a write operation. Let $A(\mathbf{y})$ denote the sequence of communication between the client and the server under an ORAM protocol for the data access sequence $\mathbf{y}$. We say that the ORAM protocol is secure, if

1. *for any chosen pair of data access sequences* $\mathbf{y}$ *and* $\mathbf{y}'$ *of same length, the server cannot (computationally) distinguish between* $A(\mathbf{y})$ *and* $A(\mathbf{y}')$*,*
2. *for a sequence of input* $\mathbf{y}$ *the ORAM protocol returns output to the client consistent with* $\mathbf{y}$ *with probability* $> 1 - \mu$*, where* μ *is a very small quantity.*

Similar to other works, we do not consider leakage through side channels such as when or how frequently the client queries the ORAM. We discuss achieving integrity against a potentially malicious server in Section 8, however, we do not focus on integrity in our main presentation.

2.2 Existing ORAM Designs

Oblivious RAM or ORAM was first introduced by Goldreich and Ostrovsky [15,16] in their seminal work [16]. They prove that ORAM designs in classical model, where the server acts as a plain storage device to support only read and write operations, have a fundamental requirement of $\Omega(\log n)$ bandwidth blowup. Path ORAM [24] is a prominent example in this line of work which achieves a bandwidth blowup of $O(\log n)$ by utilizing a binary tree-based database structure. However, due to the inherent regular mutations to the structure of binary trees, the client in Path ORAM is required to locally maintain the state of the database's structure to be able to keep track of the location of data elements; which implies a stateful client with memory overhead.

Table 1: Comparison between our design and previous ORAM approaches in terms of storage requirement (in bits) and computational cost of both server and client for a batch of k queries. We denote L as the number of levels of ORAM tree of Onion Ring ORAM [3] and Path ORAM [24], Z as the number of slots in each node, n the number of data elements, and N and q as ciphertext parameters of underlying FHE scheme. w is the number of hash functions used for our batching technique. In our work we set $w = 3$.

	Ours(Method 1)	Ours(Method 2)	Onion Ring ORAM [3]	Path ORAM [24]
Memory (Server)	$O(n \cdot N \cdot \log q)$	$O(w \cdot n \cdot N \cdot \log q)$	$O(Z \cdot (2^L - 1) \cdot N \cdot \log q)$	$Z \cdot n$
Memory (Client)	$O(1)$	$O(w \cdot n \cdot \log n)$	$O(n \cdot \log n + Z \cdot N \cdot \log q)$	$O(logn) \cdot \omega(1)$
Stateful	×	×	✓	✓
Computation (Client-Query)	$O(k \cdot \log n)$	$O(k \cdot \log n)$	$O(k \cdot \log n)$	$O(k \cdot \log n)$
Computation (Client-Eviction)	×	×	$O(Z \cdot \log Z \cdot \log n)$	$O(k \cdot \log n)$

To circumvent the lower bound set by Goldreich and Ostrovsky, many ORAM designs have been proposed in the server computation model [23,26,25,2,9,14,27]. In the server computation model, the server does not just act as a plain storage device. Instead, it helps in offsetting the bandwidth blowup by performing additional expensive computations. However, most of these designs cannot achieve a bandwidth blowup that is better than $O(\log n)$. Only until recently were the designs of Onion ORAM and Onion Ring ORAM actually able to circumvent the $O(\log n)$ bandwidth blowup. Using FHE techniques, both designs provide a non-interactive online phase with a bandwidth blowup of $O(1)$. However, given they both employ binary tree-based database structures, similar to Path ORAM, the client in their case needs to maintain a state of the database's structure locally. Additionally, Onion ORAM and Onion Ring ORAM introduce expensive eviction (offline) phases to achieve their guarantees.[3] While Onion Ring ORAM significantly improves over Onion ORAM, it still nevertheless incurs computation and memory overheads on the stateful client: the client is required to perform expensive operations and interact with the server during the eviction phase.

We observe that such binary tree-based structures are not necessary when using FHE techniques: the server can operate on all elements while reading from/writing to only element, or a batch of them. This motivated us to take a step away from this design paradigm, and propose a novel design strategy which achieves a non-interactive and stateless client ORAM scheme while guaranteeing a bandwidth blowup of $O(1)$ and eliminating the need for offline eviction phases.

In our design (both with and without PBCs), the client prepares a query, sends it to the server, and then can go offline. After querying, it does not need to be involved in any computation, and can retrieve the result whenever it comes back online. In our core design, the client does not have any memory overhead. However, with our optimization with PBCs, the client needs to generate an $\mathsf{IndexMap}$ (c.f. Section 6). But, once the $\mathsf{IndexMap}$ is generated, it does not need to be updated once more: hence preserving the stateless property of the client. Furthermore, $\mathsf{IndexMap}$ can be assumed as public, unlike the requirement of secure client-side storage of non-key material by Path ORAM and Onion Ring ORAM. We compare our design with that of Onion Ring ORAM in Table 1 based on the trade-offs offered by each.

3 Preliminaries

3.1 Preliminaries of Underlying FHE Scheme

Notations. We define vectors and matrices in lowercase bold and uppercase bold, respectively. A dot product of two vectors $\mathbf{v}, \mathbf{w}$ is denoted by $\langle \mathbf{v}, \mathbf{w} \rangle$. For a vector $\mathbf{x}$, both $\mathbf{x}_i$ and $\mathbf{x}[i]$ denote either the i-th component scalar or the i-th element of an ordered finite set. $\log(\cdot)$ is the logarithm function with base 2. Let $\mathcal{R}$ and $\mathcal{R}_q$ denote $\mathbb{Z}[X]/(X^N+1)$ and $\mathbb{Z}[X]/(X^N+1) \mod q$, respectively, for positive integers q and N. We summarize the notations in Table 2.

Ciphertexts. We define the ciphertext modulus as q and the plaintext modulus as t, where $t \ll q$, and denote $\mathit{\Delta} = \lfloor q/t \rceil$.

- An RLWE ciphertext is defined as $c := (a, b) \in \mathcal{R}_q^2$, where $b = a \cdot s + \mathit{\Delta} \cdot m + e$ for a message polynomial $m \in \mathcal{R}_t$ and a secret key $s \in \mathcal{R}$. c is denoted by $\mathsf{RLWE}_s(m)$.
- Given a base B_g and $\ell = O(\log q)$, we define a gadget vector $\mathbf{g} = (1, g, \ldots, g^{\ell-1})^t$. An RGSW ciphertext is a form of $\mathbf{C} := (\mathbf{a}, \mathbf{b}) \in \mathcal{R}_q^{2\ell \times 2}$, where $\mathbf{b} = \mathbf{H} + m \cdot \mathbf{G}$, where each row of $\mathbf{H}$ is an $\mathsf{RLWE}_s(0)$ and $\mathbf{G}$ is a gadget matrix which is defined by $\mathbf{G} = \mathbf{I}_2 \otimes \mathbf{g}$. The ciphertext is denoted as $\mathsf{RGSW}_s(m)$. s can be omitted if it is not specified.

Homomorphic Operations and Basic Algorithms. We introduce some homomorphic operations and basic algorithms used in this paper.

Homomorphic Addition. Let $c_1 := (a_1, b_1)$ and $c_2 := (a_2, b_2)$ be two RLWE ciphertexts. The addition between two ciphertexts is defined as $c_+ := c_1 + c_2 = (a_1 + a_2, b_1 + b_2)$.

[3] Even for Path ORAM, all the write operations can be done as part of an eviction phase, however, the cost is similar to that of a read operation. On the other hand, the eviction phases in Onion ORAM and Onion Ring ORAM are much more expensive compared to Path ORAM.

Table 2: Protocol and system parameters for ORAM

DB	The database
n	Number of data blocks (elements) in the database
w	Number of hash functions for PBC
k	Number of queries in a single batch query
b	Number of buckets in DB to support PBC
p	Failure probability of PBC
op	The access type, $\mathsf{op} \in \{\mathsf{Read}, \mathsf{Write}\}$
$\langle \mathbf{v}, \mathbf{w} \rangle$	Dot product of two vectors $\mathbf{v}, \mathbf{w}$
$\mathbf{x}_i, \mathbf{x}[i]$	The i-th component of vector $\mathbf{x}$
$\log(\cdot)$	Logarithm function with base 2
N	The maximum degree of a polynomial
t	Plaintext modulus
q	Ciphertext modulus
ℓ	Decomposition parameter
$\mathcal{R}$	$\mathbb{Z}[X]/(X^N+1)$ for a positive integer N
$\mathcal{R}_q$	$\mathbb{Z}[X]/(X^N+1) \mod q$ for positive integers q, N

External Product. We denote a homomorphic multiplication between an RLWE ciphertext and an RGSW ciphertext as $\boxdot$, which is called external product in [5], and define it as follows: $a \boxdot \mathbf{B} = \langle \mathbf{G}^{-1}(a), \mathbf{B} \rangle$, where $\mathbf{B}$ is an $\mathsf{RGSW}(m_B)$ where $B \in \{0, 1\}$ and a is an $\mathsf{RLWE}(m_a)$ where $m_a \in \mathcal{R}_t$ and $\mathbf{G}^{-1}(\cdot)$ is the gadget decomposition function which satisfies $\langle \mathbf{G}^{-1}(a), \mathbf{G} \rangle = a$ for $a \in \mathcal{R}_q$.

CMUX gate. With the given external product, we can build a small circuit called CMUX [6] which takes three inputs a, b, (RLWE ciphertexts) encrypting m_a, m_b respectively, and $\mathbf{C}$ (an RGSW ciphertext encrypting a bit), and outputs either of a or b depending on the value which $\mathbf{C}$ encrypts. The CMUX gate with the three inputs are defined as follows: $\mathsf{CMUX}(a, b, \mathbf{C}) = (b - a) \boxdot \mathbf{C} + a =: c$, where c encrypts m_b if $\mathbf{C}$ is an encryption of 1, otherwise m_a.

Conversion. We refer to an efficient conversion algorithm (Algorithm 4 of [3]) to switch a format of ciphertext from RLWE form to RGSW, while preserving the message m. In our case, the algorithm accepts ℓ RLWE ciphertexts $\{\mathsf{RLWE}_s(m \cdot g^j)\}_{j \in [\ell]}$ as input and outputs $\mathsf{RGSW}_s(m)$. To run this algorithm, an additional public evaluation key denoted by ksk us needed. This key is precomputed and sent by the client beforehand. We call the algorithm $\mathsf{RLWEtoRGSW}$ throughout the paper.

3.2 Homomorphic De-Multiplexer

The goal of the homomorphic de-multiplexer in our ORAM is to output a unit vector which contains 1 on a desired position, obliviously. A similar functionality called homomorphic traversal is discussed in [8]. Its purpose is, given a binary tree of depth L, to move a value from the root to a desired leaf. In ORAM, the client needs to indicate the desired item to be accessed without revealing its index to server. So we leverage this technique to distinguish the desired item from other database elements. Assuming that a server stores n data elements, we prepare a complete binary tree with level $L = \log n$. If the client wants to touch the i-th element from the database, it can homomorphically index the i-th leaf with an encryption of 1, and the rest with encryptions of 0 via homomorphic traversal. Given these n leaves, the server can run the desired operation exclusively at the desired item because an operation on the i-th position can be activated with encryption of 1 only. We remind that the server cannot distinguish between encryptions of 0 or 1.

Unlike the approach of [8], the client only prepares $\lceil \log n \rceil$ bits which represent a bit-decompostion of the index i, encrypts them all in RGSW format and transmits the ciphertexts to server. The server runs the algorithm with the encrypted $\log n$ bits as controller bits, each in their corresponding level. In our case, all nodes on the same level of a tree use the same controller bit, whereas [8] requires different input for each decision node. At the end of the algorithm, we obtain an encryption of 1 on the i-th leaf from the left.

We call the algorithm HomDemux and present it in Algorithm 1. In our case, HomDemux takes as input an RLWE ciphertext (the root value of the tree) and $\log n$ RGSW ciphertexts (the encrypted controller bits) and outputs n RLWE ciphertexts (the leaf nodes).

Algorithm 1 Homomorphic de-multiplexer HomDemux

```
Input: $\log n$ controller elements $\{c_0, \ldots, c_{L-1}\}$.
Output: $n$ values of leaves: $z_0, \ldots, z_{n-1}$
$b_0 \leftarrow \mathsf{Enc}(1)$
for $i \leftarrow 0, \ldots, L-1$ do
    for $j \leftarrow 0, \ldots, 2^i - 1$ do
        $b_{2^{i+1}+2j-1} := b_{2^i-1+j} - b_{2^i-1+j} \cdot c_i$        ▷ $\cdot$ denotes homomorphic multiplication
        $b_{2^{i+1}+2j} := b_{2^i-1+j} \cdot c_i$
    end for
end for
for $i \leftarrow 0, \ldots, n-1$ do
    $z_i := b_{2^{L+1}-1+i}$
end for
return $z_0, \ldots, z_{n-1}$
```

4 Core Design of Panacea

In this section, we present the design for single query ORAM with $O(1)$ bandwidth blowup and no client-side memory overhead. In Sections 5 and 6 we extend our design to improve server performance by allowing multiple queries to be sent in batches.

4.1 Design Description

We begin with a high-level description of how our algorithm works in the plaintext setting, and then show how it is converted to the FHE setting.

1. The client sends $(\mathsf{a}, \mathsf{op}, \mathsf{data})$, where a is the logical address of a requested block[4], op represents either a read or write operation, and data is the new data.
2. The server de-multiplexes a into a unit vector $(l_0, \ldots, l_{n-1})$ where $l_{\mathsf{a}} = 1$, and the remaining values are 0, then returns the result of the dot-product $\langle (l_0, \ldots, l_{n-1}), (d_0, \ldots, d_{n-1}) \rangle$ to the client.
3. To update, the server runs a simple circuit on every element d_i of the database:
   - Let $\mathsf{temp}_i \leftarrow$ **if** $(\mathsf{op} = \mathsf{Read})$ d_i **else** data.
   - Update the database entry $d_i \leftarrow \mathsf{temp}_i$.

Now we describe the same construction in the FHE setting:

[4] A data block refers a database element in our paper. We use these terms interchangeably.

Client: Query Generation. A query consists of a 3-tuple: $(\mathsf{a}, \mathsf{op}, \mathsf{data})$, where a is the logical address of requested data element, op is an operation which is chosen by client (either read or write), and data is the value which would be written.

The client chooses an index $i \in [0, \ldots, n-1]$, and computes its binary bit-decomposition to obtain $\log n$ bits, denoted by $\epsilon_0, \ldots, \epsilon_{\log n - 1}$. The client then encrypts these $\log n$ bits to generate a. Based on the desired operation op (either Read or Write), the client and generates an encryption of 0 and 1 and places them in an ordered tuple as follows: $\mathsf{Read} = (\mathsf{Enc}(0), \mathsf{Enc}(1))$ and $\mathsf{Write} = (\mathsf{Enc}(1), \mathsf{Enc}(0))$. The final element in the query is data is which is an encryption of zero in the case of a $Read$ query or the actual data to be written in the case of a $Write$ query.

Server: Main computation. Once the server receives a query from the client, it executes the following two phases:

1. $\mathsf{Response\ phase}$: The server runs the homomorphic de-multiplexer ($\mathsf{HomDemux}(\mathsf{a})$) on the input to generate an encrypted unit vector of dimension n indicating the desired position that the client wants to access. Next, the server computes a dot product between the unit vector and the data array to obtain the value at the requested position. Finally, the server sends the result to the client. Regardless of the operation, the client retrieves the original value at the desired location. If the op is Read, the client can access what it intended to retrieve (the i-th item). In case of a write operation, the client will ignore the received value. The actual database update (write) phase occurs after the response.

2. $\mathsf{Update\ Phase}$: Server creates a temporary array of n elements which contains either original data elements or the updated value data, depending on op by running the function called RW that achieves the following: if op is Read, the temporary array copies the original data elements; and if op is Write, the client input data is copied to every n elements.

Then the server runs the CMUX gate with the n leaves (as a controller bit) which are the output of $\mathsf{HomDemux}$ and the corresponding temporary element and the original data denoted as d_j for each $j \in [0, n-1]$. In more detail, the n leaves contain only one encryption of 1 at the desired position (the i-th leaf) and the rest are encryptions of 0. Therefore, the i-th element will be updated with data if $\mathsf{op} = \mathsf{Write}$, otherwise, it is updated to the original value d_i, which means it remains same eventually.

We remind that every computation is done over encrypted elements, so that the server is oblivious to what operation is given and what block is updated or returned to the client.

4.2 Instantiation based on Homomorphic Encryption

We describe our protocol using FHE ciphertexts as defined in Section 3. Let i be binarized as $i = (\epsilon_0, \epsilon_1, \ldots, \epsilon_{s-1})_2$, where $2^s = n$. We show how data elements and the query elements are encoded in the following:

- d_j is a RLWE sample under client's key K, $\forall j \in [0, \ldots, n-1]$
- $\mathsf{a} := (\mathsf{RGSW}_K(\epsilon_0), \ldots, \mathsf{RGSW}_K(\epsilon_{s-1}))$
- $\mathsf{op} := (\mathsf{RGSW}_K(b_0), \mathsf{RGSW}_K(b_1))$, if $b_0 = 0$ and $b_1 = 1$, it encodes Read. If $b_0 = 1$, and $b_1 = 0$, it encodes Write.
- $\mathsf{data} := \mathsf{RLWE}_K(\alpha)$, where α is an update value if $\mathsf{op} = \mathsf{Write}$, otherwise, α is any random value.

As soon as the server receives a query from a client, it runs $\mathsf{RW}(\mathsf{op}, \mathsf{data}, d_j)$ for each data element d_j. The algorithm is defined as follows:

- $\mathsf{RW}((\mathbf{C}_0, \mathbf{C}_1), a, b) \to c$:
  - Given an input of two RGSW ciphertexts $\mathbf{C}_0, \mathbf{C}_1$ and two RLWE ciphertexts a, b encrypting m_0, m_1 respectively, it outputs a RLWE ciphertext which encrypts either m_0 or m_1.
  - Computes $c := a \boxdot \mathbf{C}_0 + b \boxdot \mathbf{C}_1$

Hence, if $\mathbf{C}_0 = \mathsf{RGSW}_K(0)$ and $\mathbf{C}_1 = \mathsf{RGSW}_K(1)$ (which is $\mathsf{op} = \mathsf{Read}$), the output c is an RLWE ciphertext encrypting m_1 (the original data). After each component of the output of HomDemux is converted to an RGSW ciphertext via RLWEtoRGSW [5], the RGSW ciphertexts are used as controller bits of a CMUX gate to select either temp_j or d_j. The final dot product step consists of performing n component-wise external products between n RGSW ciphertexts $\{\mathbf{L}_j\}_{j\in[0,\ldots,n-1]}$ and n RLWE ciphertexts $\{d_j\}_{j\in[0,\ldots,n-1]}$. With the above definitions of encoding and functions, we describe the instantiation of our ORAM protocol in Algorithm 2.

Algorithm 2 ORAM protocol on server's side

```
Input: ($\mathsf{a}, \mathsf{op}, \mathsf{data}$), where $\mathsf{a}$ is the logical address of the requested block, $\mathsf{op}$ is either Read or Write, and data is the value written to the block at address $\mathsf{a}$ if $\mathsf{op} = \mathsf{Write}$. We note that everything in the query is encrypted under an FHE scheme.
Notation:
    – $n :=$ the number of data element.
    – $d_i :=$ the $i$-th data element stored in a server.
    – $\mathcal{L} := \{l_0, \ldots, l_{n-1}\}$; the set of value of leaves.
    – $\mathsf{temp} :=$ temporary data element.
Server sets $y := 0$
for $i \leftarrow 0, \ldots, \ell - 1$ do
    Server runs $\mathsf{HomDemux}(\mathsf{a}, g^i) \to \mathcal{L}_i$
end for
for $j \leftarrow 0, \ldots, n - 1$ do
    Server runs $\mathsf{RLWEtoRGSW}(\{l_{i,j}\}_{i\in[0,\ldots\ell-1]}, \mathsf{ksk}) \to \mathbf{L}_j$
    Server computes $y := y + d_j \boxdot \mathbf{L}_j$
end for
Server sends $y$ to the client
for $j \leftarrow 0, \ldots, n - 1$ do
    Server runs $\mathsf{RW}(\mathsf{op}, \mathsf{data}, d_j) \to \mathsf{temp}$
    Server computes $d_j := \mathsf{CMUX}(\mathbf{L}_j, \mathsf{temp}, d_j)$
end for
```

4.3 Computation and Communication Complexity

The communication complexity of the query is $O(\log n)$ since the client sends only $\log n$ encrypted bits. The total number of bits also involves the ciphertext expansion factor, however, that only depends on the underlying FHE scheme, and not on the size of the database n. For every query, the server sends exactly one encrypted block to the client as response. Therefore, we also have an overall $O(1)$ bandwidth blow-up as existing FHE-based ORAM designs [10,3]. Due to our stateless strategy, the server has to touch all database elements to guarantee security, hence the computation complexity is $O(n)$. In more detail, we run $\ell \cdot n/2, 2n, \ell \cdot n, n$ and n external products for

[5] The conversion algorithm is adapted from Algorithm 4 of [3].

$\mathsf{HomDemux}, \mathsf{RW}, \mathsf{RLWEtoRGSW}$, computing y, and updating data, respectively. Since ℓ is a parameter of the underlying FHE scheme, it is independent of n.

Since we separate the protocol into two phases: Response and Update, the server's response latency from the client's perspective would be equivalent to the running time of the response phase, which consumes $(\ell/2 + \ell + 1) \cdot n$ external products. After responding, the server performs $3n$ external products in the update phase. The server's total time for each query would be the sum of the duration of both phases.

4.4 Security

The security of our ORAM scheme stems directly from the semantic security (IND-CPA) of the underlying homomorphic encryption scheme. Our queries consist of FHE ciphertexts, encoding the queried database index, desired operation, and potential update value. Semantic security ensures that the server cannot learn any information about the underlying plaintexts of the query elements. The security of our design can be expressed through the following security theorem.

Theorem 1 (Security of $\mathsf{P}anacea_{core}$). *Assuming IND-CPA security and correctness of the underlying FHE scheme, Panacea$_{core}$ provides security as defined in Definition 1 with μ negligible in n.*

We refer to Appendix A for the proof of the above theorem. The theorem relies on the correctness and security of the underlying FHE scheme. However, when instantiating, we require an additional assumption of circular security which is always assumed in any FHE based protocol due to the use of key material (the key switching key in our case for $\mathsf{RLWEtoRGSW}$) when encrypting key-dependent information. Since there is no known attack on circular security, we believe that it would be safe to rely on such an assumption without harm on our concrete security, as is the case in other similar works.

5 Multi-query ORAM with Amortized Computation Cost: Method 1

The design of ORAM presented in Section 4 handles one query at a time and incurs a server-side computational overhead of $O(n)$ for each query. In this section, we present a method to reduce the (amortized) server-side computation cost for a batch of k queries on distinct elements over τ processor threads. For simplicity, we assume the number of threads to be equal to the number of queries ($\tau = k$), however, all the available processor threads can be fully utilized as long as $\ell \cdot k \geq \tau$. Our protocol is described in Algorithm 3 and Algorithm 4.

Algorithm 3 parallelizes Algorithm 2 except for Lines 9 14. Note that, only one entry with a distinct position is not an encryption of zero in each temp_i. Therefore, in Algorithm 4, adding all temp_i values does not destroy the value of the entry due to $k-1$ zeros. As a result, $\bar{\mathsf{temp}}$ has k encryptions which encrypt non-zero values at the desired positions, with the rest being encryptions of zero. Moreover, the elements of $\bar{\mathbf{L}}$ indicate the positions of the non-zero elements.

In essence, the first step (Algorithm 3) results in $O(n)$ multiplications parallelized over each available thread, and the second step (Algorithm 4) merges the results by applying $2n$ multiplications and k additions. Thus, the latency to process and respond to k queries is same as that of a single query, as $O(n)$ operations are performed before responding to the client. Similarly, the response is sent to the client before performing the database update.

Algorithm 3 Multi-query ORAM: Multi-Threaded Step

```
Input: $\{(\mathsf{a}_i, \mathsf{op}_i, \mathsf{data}_i)\}, i \in [k]$, where $k$ is the number of queries and the rest are the same as Algorithm 2.
Notation: we use the same notation as Algorithm 2.
for every thread $i \in [\tau]$ do
    $\mathsf{HomDemux}(\mathbf{a}_i)$ followed by $\mathsf{RLWEtoRGSW}$ to obtain $\mathbf{L}_{i,j}$.
    $y_i := \mathsf{Enc}(0)$
    for $j \in [0, n-1]$ do
        $y_i := y_i + d_j \boxdot \mathbf{L}_{i,j}$
    end for
    The server sends $\{y_i\}_{i \in [k]}$ to the client.
    for $j \in [0, n-1]$ do
        $\mathsf{temp}_{i,j} := \mathsf{RW}(\mathsf{op}_i, d_j, \mathsf{data}_i) \boxdot \mathbf{L}_{i,j}$
    end for
    $(\mathbf{L}_{i,j}, \mathsf{temp}_{i,j})$ is kept for use in Algorithm 4.
end for
```

Algorithm 4 Multi-query ORAM: Merge and Update Step

```
let $\mathsf{temp}_i := \{\mathsf{temp}_{i,0}, \ldots, \mathsf{temp}_{i,n-1}\}$, $\mathbf{L}_i := \{\mathbf{L}_{i,0}, \ldots, \mathbf{L}_{i,n-1}\}$
$\bar{\mathsf{temp}} := \sum_{i=1}^{k} \mathsf{temp}_i, \bar{\mathbf{L}} := \sum_{i=1}^{k} \mathbf{L}_i$
$\mathsf{CMUX}(\bar{\mathbf{L}}_j, \bar{\mathsf{temp}}_j, \mathsf{Enc}(d_j)), \forall j \in [0, \ldots, n-1]$   ▷ Effectively, if $\bar{\mathbf{L}}_j = \mathsf{Enc}(1)$ it outputs $\bar{\mathsf{temp}}_j$, $\mathsf{Enc}(d_j)$ otherwise.
```

We make two observations where this method achieves performance gains when compared to running (a multithreaded version of) Algorithm 2 sequentially for k queries. First, during the demultiplexing step of Algorithm 2, if the number of threads available is greater than the number of decomposition levels ℓ, then the additional threads would remain unused. In the method described in this section, all threads available would be utilized as long as $\ell \cdot k \geq \tau$. Second, instead of performing $k \cdot n$ applications of the CMUX gate at the end, we sum the ciphertexts and only perform n applications instead.

6 Multi-query ORAM with Amortized Computation Cost: Method 2

In this method, we use an existing batching technique from the literature to achieve a sublinear server-side amortized computational cost when considering a batch of k queries. However, the proposed batching techniques come with their own trade-offs: the networking overhead when batching is much higher than the baseline. Additionally, batch codes require a redundancy of elements in the database, which significantly increases the server's storage overhead. We provide an overview of batch codes and introduce our construction below.

Batch Codes. A batch code, parameterized by the 4-tuple (n, m, k, b), takes as input a database with n elements, and outputs a set of m (possibly distinct) elements distributed among b buckets, such that any k elements from the original database can be retrieved by fetching at most one element from each of the b buckets. We want $m < kn$ to have any performance improvement for the server.

Probabilistic Batch Codes (PBC). A probabilistic batch code, parameterized the 5-tuple (n, m, k, b, p), is a relaxed version of batch codes which allows a failure probability of p to retrieve all k elements when the batch elements are chosen uniformly at random (without replacement). This introduces a probability p of encountering a batch of k elements such that there is no way to retrieve all the elements in that batch by fetching at most one element from each bucket. Ideally, we target minimizing the probability of failure as much as possible, in order to maximize that of successfully executing our ORAM protocol.

6.1 Improving Performance Using PBC

We use a PBC based on reverse cuckoo hashing. Our technique draws inspiration from the batching technique used in the PIR protocol: SealPIR [1]. One key difference between ORAM and PIR is that PIR protocols only provide read operations on the database, whereas ORAM schemes provide both read and write operations. Additionally, all data elements would be encrypted in the context of ORAM.

Using PBC, the database would be divided into b buckets using w pre-specified (non-cryptographic) hash functions. First, the server allocates a total of $m = w \cdot n$ elements into b buckets, such that there would exist w instances of each element spread across W buckets. Then, for each batch of $k < b$ queries, the client accesses exactly one element from each bucket. We describe the details of this design and its underlying components below.

Database Allocation by Server. Let $h_1, \cdots, h_w$ be w distinct hash functions. Since the client and the server need to agree on some hash functions, the hash functions are publicly specified as part of the protocol. Given a database of n elements, each element is hashed based on its index in the original database. The actual element is encrypted and remains encrypted throughout the process. Suppose an element at index ι produces the following hash values: $i_1 = h_1(\iota), \cdots, i_w = h_w(\iota)$. The server adds the elements from the buckets $i_1, \cdots, i_w$. The process is then repeated for all distinct elements in the database. Upon completion, each element will have w instances spread over b buckets (possibly non-distinct). Depending on the hash functions, it is possible that there are multiple instances of the same element in a bucket, but only with small probability. However, the presence of two elements in one bucket does not impact the security of ORAM and only influences the failure probability of PBC. We further discuss more this failure probability and its implication in Section 6.4.

Batch-query by Client. For every batch of k (read/write) queries, the client is expected to perform b accesses, one from each of the b buckets. For each access per bucket, the client can run the core protocol described in Section 4. The client selects the elements for those b accesses in a way that satisfies all the k desired read/write operations. To that end, the client uses the *reverse cuckoo hashing* technique with the w hash functions as described below.

The problem can be looked at as a balls-and-bins problem. The client has to place k balls (the intended read/write operations) into b bins (the buckets in the database), where each bin can have at most one ball. For a ball (the index of an element) ι the client computes w candidate buckets by applying the hash functions. Then places the ball in one of the empty candidate bins. If none of the candidate bins are empty, the client picks one uniformly at random, removes the existing ball ι_{old} from the chosen bin, and places the new ball ι there. For the removed ball ι_{old}, the client runs the insertion procedure again, and if that causes another ball to be removed, this procedure goes on recursively for a maximum number of iterations.

If the client can successfully place all the k balls into b bins, that becomes the assignment for elements in each bucket the client should access from the server. There is also a small chance of failure to place all the k balls into b bins, we evaluate the failure probability in Section 6.4. Since the client and the server are using the same hash functions, if assignment puts an element ι in bin i for the client, that element will be available in bucket i on the server database. Now, the client can run one instance of the core ORAM protocol described in Section 4 for each bucket. Note that, for $(b-k)$ buckets, which were empty after placing k balls into b bins, the client generates dummy read requests. Since the client accesses one element from each bucket, and no bucket is touched more than once in one batch, the server does not know which buckets correspond to which of the k accesses requested by the client. However, to access a specific element in a bucket, the client needs to know the index j of the element in that bucket for the index ι in the original database. Moreover, since the elements in the database are encrypted and each element has exactly w redundant copies, the other $(w-1)$ copies need to be updated as well for each write access. Otherwise, the database would lose consistency, and any future queries might not return correct (up-to-date) results. We describe below how we solve the above two problems.

Algorithm 5 ORAM protocol using PBC

```
$\{D_i\}_{i\in[b]}$ := the database divided into $b$ buckets, initially all buckets are empty.
$d_\iota$ := the element at index $\iota$ in the original database $DB$.
$\{h_t\}_{t\in[w]}$ := the hash functions used for PBC.
$I$ := $\mathsf{IndexMap}$: the mapping between an original database index $\iota$ and the pairs $(i, j)$ denoting bucket index and
    the location inside the bucket.
function SERVER:SETUP(Database $DB$)
    for $i \leftarrow 0, \dots, b-1$ do
        $\mathsf{counter}_i := 0$
    end for
    for each index $\iota$ in $DB$ do
        for $t \leftarrow 0, \dots, w$ do
            $i := h_t(\iota)$; Append to $D_i$ the element $d_\iota$
            Add the 3-tuple $(\iota, i, \mathsf{counter}_i)$ to $I$; $\mathsf{counter}_i := \mathsf{counter}_i + 1$
        end for
    end for
end function
function CLIENT:SENDBATCHQUERY($\{(\mathsf{a}_t, \mathsf{op}_t, \mathsf{data}_t)\}_{t\in[k]}$)
    $\{(i_t, j_t, \mathsf{op}'_t)\}_{t\in[b]} \leftarrow \mathsf{BatchEncode}(I, \{(\mathsf{a}_t, \mathsf{op}_t)\}_{t\in[k]})$
    for $t \leftarrow 0, \dots, b-1$ do
        Send request on $D_{i_t}$ for each input $(\mathsf{j_t}, \mathsf{op}'_t, \mathsf{data}'_t)$
    end for
end function
function SERVER:BATCHRESPONSE($\{(\mathsf{j_i}, \mathsf{op}_i, \mathsf{data}_i)\}_{i\in[b]}$)        ▷ $(\mathsf{j_i}, \mathsf{op}_i, \mathsf{data}_i)$ are encrypted for Server.
    for $i \leftarrow 0, \dots, b-1$ do
        Run Algorithm 2 on $D_i$ on input $\{(\mathsf{j_i}, \mathsf{op}_i, \mathsf{data}_i)\}$        ▷ The response $y_i$ is sent back during the process.
    end for
    ConsistencyCorrection()        ▷ See Algorithm 6.
end function
function CLIENT:BATCHEXTRACT($\{(\mathsf{a}_t, \mathsf{op}_t)\}_{t\in[k]}, \{y_1, \dots, y_b\}$)
    $x_1, \dots, x_b$ := Decrypt $\{y_1, \dots, y_b\}$
    return $\mathsf{BatchDecode}(I, \{(\mathsf{a}_t, \mathsf{op}_t)\}_{t\in[k]}, \{x_1, \dots, x_b\})$
end function
```

6.2 Mapping Indices

In order to access an element from a bucket the client needs to know the index j of the element in the bucket for its original index ι in the database. We solve that problem using an $\mathsf{IndexMap}$ which stores this mapping. The client can store the $\mathsf{IndexMap}$ throughout the time it uses the database which that adds a small memory overhead, in order to save query generation time. For a database with one million entries, the total size of $\mathsf{IndexMap}$ is less than 9 MB for our chosen parameters.

It is to be noted that the indices of the elements are used as inputs to the hash functions during the process of allocating them to the buckets. Therefore, once the buckets are allocated by the server, $\mathsf{IndexMap}$ is never modified. Other ORAM designs that rely on client storage require the client to maintain some metadata representing the state of the database, and that needs to be updated after every read/write access and requires secure storage to protect privacy. On the contrary, $\mathsf{IndexMap}$ in our case can be public and does not require any updates. Moreover, if the server uses a predefined source of randomness (which does not break the security of the ORAM protocol), $\mathsf{IndexMap}$ becomes deterministic and can be provided to the client as part of the setup. In that sense, our client remains stateless: even if the client goes offline, it can easily retrieve/reconstruct the $\mathsf{IndexMap}$ from setup information.

We denote the method of producing b pairs of bucket indices and the corresponding indices in the buckets from k indices from the original database as $\mathsf{BatchEncode}()$. This operation accepts as input the k tuples $\{(\iota_1, \mathsf{op}_1), \ldots, (\iota_k, \mathsf{op}_k)\}$ and outputs the 3-tuples $\{(i_1, j_1, \mathsf{op}_1), \ldots, (i_b, j_b, \mathsf{op}_1)\}$. When the client receives the server's response, it just needs to discard the $(b-k)$ responses corresponding to the dummy requests mentioned above. We denote this method as $\mathsf{BatchDecode}()$. We describe the ORAM algorithm using PBC in Algorithm 5.

6.3 Consistency Correction

Since our batching technique requires redundancy, all w copies of data elements need to be updated for each write access. With the above method alone[6], only one of w copies would be updated, leaving the remaining copies as encryptions of the old data. Therefore, we introduce a new technique which also updates the remaining redundant $w-1$ blocks to avoid data inconsistency.

We explain our technique to correctly update all the copies with $w = 3$ for simplicity (we also choose $w = 3$ in our setting which we explain in Section 6.4). This method can be easily extended to any value of w.

We note that this correction is run after the merge and update phase (Algorithm 4) is executed on each bucket. Suppose that the three hash functions ($w = 3$) are applied to index ι and the resulting indices are given by (i_1, j_1), (i_2, j_2) and (i_3, j_3), where i_* and j_* represent the bucket index and the position of an element inside the bucket respectively. After processing a batch, the server obtains the following for each pair (i, j) as a result of applying the homomorphic de-multiplexer (Section 3.2) on each bucket:

$L_{i,j}$: an RGSW ciphertext that encrypts 1 if the j-th element in the i-th bucket is accessed, or 0 otherwise.

B_i: an RGSW ciphertext that encrypts 1 if the operation in the i-th bucket is a write operation, or 0 otherwise. This is the same as the first element of the op tuple from the client's query (Section 4.2).

[6] PBC without consistency correction

Note that only one $L_{i,j}$ corresponding to the same element will contain 1, since no element can be accessed twice in the same batch. Similarly, only one $L_{i,j}$ will contain 1 for a given bucket i. The server derives B_i from the op parameter in the client's query, then applies the following consistency correction formula to update the database:

$$\begin{aligned} V_\iota^{new} &= V_{i_1,j_1} \boxdot (1 - L_{i_1,j_1} \boxdot B_{i_1} - L_{i_2,j_2} \boxdot B_{i_2} - L_{i_3,j_3} \boxdot B_{i_3}) \\ &+ (V_{i_1,j_1} \boxdot L_{i_1,j_1} \boxdot B_{i_1} + V_{i_2,j_2} \boxdot L_{i_2,j_2} \boxdot B_{i_2} + V_{i_3,j_3} \boxdot L_{i_3,j_3} \boxdot B_{i_3}), \end{aligned} \tag{1}$$

where $V_{i,j}$ denotes the ciphertext at (i, j). V_ι^{new} is the updated value which is copied to locations $(i_1, j_1), (i_2, j_2)$ and (i_3, j_3), replacing V_{i_1,j_1}, V_{i_2,j_2} and V_{i_3,j_3} respectively.

If we consider one of the $L_{i,j}$ to be an encryption of 1, then intuitively, the top line of Equation (1) evaluates to an encryption of 0 if it is a write operation, otherwise it is an encryption of V_{i_1,j_1}. The bottom line of Equation (1) evaluates to an encryption of 0 if it is a read operation, otherwise it is the new value that is written to the database. If none of $L_{i,j}$ is an encryption of 1, then Equation (1) evaluates to V_{i_1,j_1}. As a result, if the client performs a write operation to one of the indices i_1, j_1, i_2, j_2 and i_3, j_3, then subsequently the new value is returned, otherwise the old value is returned. We assume that the client is honest and does not attempt to submit queries which can make its own database inconsistent, i.e., more or equal to two of L_{i_1,j_1}, L_{i_2,j_2} and L_{i_3,j_3} is an encryption of 1. We present the method in Algorithm 6.

Algorithm 6 ConsistencyCorrection by server after every batch

```
Notation:
    – $V_{i,j}$ := the $j$-th data element stored in the $i$-th bucket.
    – $\{h_t\}_{t\in\{1,2,3\}}$ := the hash functions used for PBC. // We write the algorithm considering 3 hash functions.
    – $I$ := IndexMap: stores the mapping between an original database index $\iota$ and the pairs $(i, j)$ denoting bucket
      index and the location in the bucket.
    – $\{L_{i,j}\}$; the set of leaf values after running HomDemux on bucket $i$.
for $\iota \leftarrow 0, \ldots, n-1$ do
    $(i_1, j_1), (i_2, j_2), (i_3, j_3)$ := query $I$ for $\iota$.
    for $t \leftarrow 1, 2, 3$ do
        $B_{i_t} \leftarrow \mathsf{op}_{i_t}[0]$
    end for
    $v := V_{i_1,j_1} \boxdot (1 - L_{i_1,j_1} \boxdot B_{i_1} - L_{i_2,j_2} \boxdot B_{i_2} - L_{i_3,j_3} \boxdot B_{i_3}) + (V_{i_1,j_1} \boxdot L_{i_1,j_1} \boxdot B_{i_1} + V_{i_2,j_2} \boxdot L_{i_2,j_2} \boxdot B_{i_2} +$
$V_{i_3,j_3} \boxdot L_{i_3,j_3} \boxdot B_{i_3})$
    $D_{i_1}[j_1] := D_{i_2}[j_2] := D_{i_2}[j_2] := v$
end for
```

6.4 Failure Probability and Parameter Choices

There is a very small probability that the assignment of k balls in b bins will fail. However, that will only result in a failure to retrieve some of the k elements for a batch. And, the client will know that the batch will fail, before even sending the batch query to the server.

Analyzing the exact probability of failure, and determining the constant factors of cuckoo hashing still remains an open problem. However, there are empirical studies [12,4,20] to estimate this probability for different parameter choices. Chen et al. [4] estimates that the failure probability is $\approx 2^{-40}$ when $w = 3$ and $b = 1.5k$, for $k > 200$. Following their analysis, we choose $k = 256$, $w = 3$, and $b = 384$. With that, the database has an additional 3x storage overhead, and the client has an additional 1.5x bandwidth overhead. However, since the access in each bucket is independent of other buckets for a single batch, then the server can run multiple accesses targeting different buckets in parallel.

6.5 Tradeoffs And Comparison With Method 1

With our batching technique, we reduce the server-side computational cost of by several factors (as demonstrated in our experiments in Section 7). However, this improvement comes at a bandwidth cost of 1.5x when compared to our Method 1 or the core protocol and a storage cost as the database takes 3x more space. Additionally, there is a slight increase in client-side memory overhead compared to Method 1 due to the storage of the hash table (see Table 1). Since each component of $\mathsf{IndexMap}$ consists of a few bits (bit length of an index), the actual overhead does not have a big impact on the client's performance. On the other hand, if running cuckoo hashing is fast enough, then the client does not need to store $\mathsf{IndexMap}$, instead, he generates the hash every time he queries. Then the memory complexity on the client side becomes $O(1)$, but the query generation complexity will increase to $O(n)$ (running hash functions over n elements), which is a fair trade-off.

In Method 1, we mainly optimize the number of read/write operations to the database: the overall computation cost for the server is still $O(n)$. If disk-IO per block is expensive (e.g., very large block size) or the cost of computation is not a concern (availability of a powerful cluster), Method 1 could be beneficial to optimize the bandwidth. However, given that server-side computational overhead is much higher than disk access, Method 2 would be more useful to optimize the server computation time. Especially, with the high speed ($200 - 500$ MB/s) read/write capabilities of solid-state-devices and typical block size (several KB), we consider that disk-IO is not a bottleneck, and the cost of 1.5x additional bandwidth blowup and 3x expansion in the size of database are reasonable trade-offs. Therefore, we mainly focus on evaluating Method 2 in Section 7.

7 Implementation and Evaluation

Our protocol leverages the TFHE scheme [6] which supports every homomorphic operation we use. We provide an implementation in the Rust Programming Language[7] using the Concrete Core library[8] [7] for homomorphic operations. The source code is available on GitHub.[9] We include the dependency manifest (`Cargo.lock`) and the parameter sets used in our experiments (`params.json`) to ensure reproducible results.

7.1 Experimental Setup

To simulate a Server implementing our scheme, we benchmarked our implementation on a dedicated cloud instance running Red Hat Linux. The instance was equipped with an Intel® Xeon™ Platinum 8360Y CPU (36-core@2.4ghz) and 2TB of RAM.

To simulate a client, we compiled our code, targeting a Linux PC equipped with an Intel® Core™ i5-6600 CPU (4-core@3.3ghz) and 8GB of RAM, as well as a 2020 Apple MacBook Pro with

[7] Toolchain: Rust version 1.68.0 nightly-x86_64-unknown-linux-gnu

[8] ”concrete-core” version: 1.0.2

[9] `https://github.com/KULeuven-COSIC/Panacea`

an M1 chip and 8GB of RAM. In contrast with our server setup, benchmarks evaluated on both client setups were limited to a single-thread for ease of comparison.

We assume that the client and the server agree on the database and FHE parameters prior to execution. Additionally, during setup, the client generates their keys and sends the evaluation keys to the server. The server sets up a dummy database (or a database consisting of trivial RLWE encryptions of zeros). We benchmark successive simulations of our protocol for a variable database size progressing from $n = 2^{12}$ to $n = 2^{19}$ using the same hardware. Our instantiations use parameters in Table 3, yielding 128 bits of security.

Table 3: TFHE Parameters

Parameter	**Value**
standard deviation	2^{-55}
polynomial degree N	2048
decomposition parameters for key switching (g, ℓ)	$(2^5, 11)$
decomposition parameters for query (g, ℓ)	$(2^5, 9)$
plaintext modulus t	2^8
ciphertext modulus q	2^{64}

7.2 Bandwidth

Table 4: Bandwidth blowup and the actual cost for 384KB data element size when $n = 2^{19}$. We used method 2 (PBC) for our amortized scenraio.

ORAM type		**Bandwidth blowup**		**amortized blowup**	
Stateful	Path ORAM	28.5 MB	76	57 MB	152
	Onion Ring ORAM	2 MB	5.33	4.8 MB	12.9
Stateless	Panacea	14.8 MB	39.46	20.71 MB	55.22
	Panacea (with query packing)	3 MB	8	3 MB	8

We compare the bandwidth of our design with that of existing stateful designs (specifically, PathORAM and Onion Ring ORAM) under similar conditions. Unlike previous stateful designs, our query cannot contain any plaintext to guarantee the security.

Therefore, the actual communication cost for queries would be worse than that of many stateful designs due to size of FHE ciphertexts. With the concrete parameters used in our implementation, one RGSW ciphertext costs 576KB and one RLWE ciphertext (encrypting 2KB of message) as one response costs 16KB (we use the same technique of OnionRing ORAM [3] to reduce the size of response by reducing modulus from 64 bits to 32 bits). We require $\log n$ RGSW ciphertexts to encode the chosen index a and two for the operation op.

When $n = 2^{19}$, the query size is $21 \cdot 576$KB $= 11.8$MB. Then the server's answer consists of 192 RLWE ciphertexts each of size 384KB, resulting in 3MB in total. Due to the choice of parameters in Onion Ring ORAM [3] $t = 2^{12}$, it was possible that more data could be loaded (up to 3KB) in one ciphertext, resulting in a reduced server response size of 2MB. It is possible for us to set the same t in theory, but for large $n = 2^{19}$, we had to reduce the parameter to guarantee the correctness in our *actual implementation.*

For the amortized scenario (with Method 2), the query size is 1.5x larger than that of the single query case. In any case, the query size is the dominant factor in the total bandwidth overhead. Indeed, Onion Ring ORAM incurs less bandwidth overhead than us. However, we can achieve similar low bandwidth overhead by using their query packing method to pack all the query bits in ℓ RLWE ciphertexts, instead of sending $\log n{+}2$ RGSW ciphertexts as we currently do. Even without this optimization, we still have much less bandwidth overhead than PathORAM due to our constant bandwidth blowup.

Query Packing. If we use the query packing method of Onion Ring ORAM to reduce the query size, the server would have to unpack queries upon reception and convert them into the right format to be processed. We call this conversion operation query unpacking. Then, there would be an additional server-side computational overhead that has $\log n$ complexity. Query unpacking can be trivially parallelized, therefore it does not cause a significant impact to the total computation time. When we pack everything in the client's query (i.e. indices of the item, operation encoding) into ℓ RLWE ciphertexts, the bandwidth consumption would be dominated by the server's response: 3MB in this case. The result is given in the last row of Table 4.

We have also implemented our design with query packing by using a tweaked version of Algorithm 4 of [3] for $n = 2^{19}$ without modifying our parameters. We note that their unpacking method (Algorithm 4 of [3]) causes the presence of a lot of additional noise, given the same set of parameters. Therefore, we use a tweaked version of their algorithm (discussed in Appendix B) to guarantee correctness. In our implementation, the unpacking algorithm consumed 69 ms per RLWE ciphertext, hence the additional cost is around 13 seconds to unpack 21 RGSW ciphertexts for $n = 2^{19}$.

We note that our bandwidth blowup is only affected by the ciphertext expansion factor (ratio between ciphertext and plaintext) of the underlying FHE scheme: Panacea maintains a constant bandwidth blowup independent of the block size. On the other hand, Onion Ring ORAM [3] has another factor Z (the number of real blocks in each node/bucket of their tree structure) which depends on n. In order to keep the bandwidth blowup constant in practice, Onion Ring ORAM requires large block size in combination with query packing.

7.3 Computation Overhead

To demonstrate the practicality of our design we compare the computational overhead of our design with that of Onion Ring ORAM, since it is the only other practical design based on homomorphic encryption techniques, and achieves constant bandwidth blowup. However, we want to highlight that their amortization technique is fundamentally different from ours, and we achieve new properties like statelessness, non-interactiveness.

Single query and Method 1. For our single query performance, with $n = 2^{12}$, the response time is 1.79s and the update duration is 1.39s. We used $k = 36$ threads when implementing the single query protocol to benchmark the performance. Afterwards, we used Method 1 to handle k queries in one go using the same setup. As expected, our Method 1 for k queries is faster than running our single query (Algorithm 2) protocol k times on the same number of cores. This is due to the inability to parallelize all parts of the computation in the single query case as we observed in Section 5. Specifically, to process k queries, Section 5 consumes 25% less time on average when compared to Algorithm 2. For larger database sizes we refer to the batched version of our protocol with PBC shown next.

Batching with PBC. When the protocol uses PBC, the client batches 256 queries and submits the whole batch to the server which processes it at once. In our experiment we use a total of 36 threads (one thread per core) to parallelize server-side computations. Numbers in Table 5 show the averages taken over 5 independent runs of the experiment using the same parameters. The sum of the response duration and update duration represents the total time spent by the server to treat a query. The response duration specifies the elapsed duration between the server receiving the client's query and transmitting the response. The update duration is equivalent to the time spent by the server to update the database after responding to the client. Figure 1 displays the linear growth of the amortized duration of server computation as the number of database elements increases.

We find a clear performance advantage when comparing with the single query mode. Using a database size of 2^{12}, the server is over 180 times faster in terms of access duration.

Table 5: Computation time required by the server for database size n, with n from 2^{12} to 2^{19}, for the size of the batch $k = 256$ with PBC. Numbers in brackets are amortized cost. All times are in seconds.

n	**Response Duration**	**Update Duration**	**Total Time**
2^{12}	2.47 (0.0096)	1.01 (0.0004)	3.48 (0.014)
2^{14}	9.53 (0.037)	2.89 (0.011)	12.42 (0.049)
2^{16}	38.08 (0.15)	11.04 (0.043)	49.13 (0.19)
2^{18}	147.92 (0.58)	48.02 (0.19)	195.94 (0.77)
2^{19}	296.43 (1.16)	94.83 (0.37)	391.26 (1.53)

We came across some difficulties when comparing our numbers directly with those of Onion Ring ORAM, [3] since their implementation is not publicly available. Moreover, we conjecture that the numbers presented in [3] are theoretical projections which do not account for bottlenecks from data transfer speeds between CPU and RAM which we noticed during our experiments when using large database sizes. Memory bandwidth is a well-known bottleneck for FHE computation [22,13]. Therefore, we provide an analytical comparison below.

Explanation and Analytical Comparison with Onion Ring ORAM. The server's main computation is performed over all n data elements to guarantee the security, either sequentially or in parallel depending on the server's resources and deployment scenario. Therefore, $O(n)$ is an inevitable complexity. In more detail, the response phase which mainly influences the latency consists of $(\ell/2+\ell+1)\cdot n$ external product in practice, where ℓ is a parameter of the underlying FHE scheme. In our experiments, we set $\ell = 9$. Hence, at least $14.5 \cdot n$ external products are performed. If we use Method 2, this number is multiplied by $w = 3$ to handle $k \geq 200$ queries at a time.

When comparing with Onion Ring ORAM, we note that their construction has a default complexity of $O(\log n)$ for both online and offline phases. During the online phase, the only computation performed is an addition of $\log n$ ciphertexts, which is extremely cheap when compared to other homomorphic operations. However, for every A accesses, they require an interactive offline phase

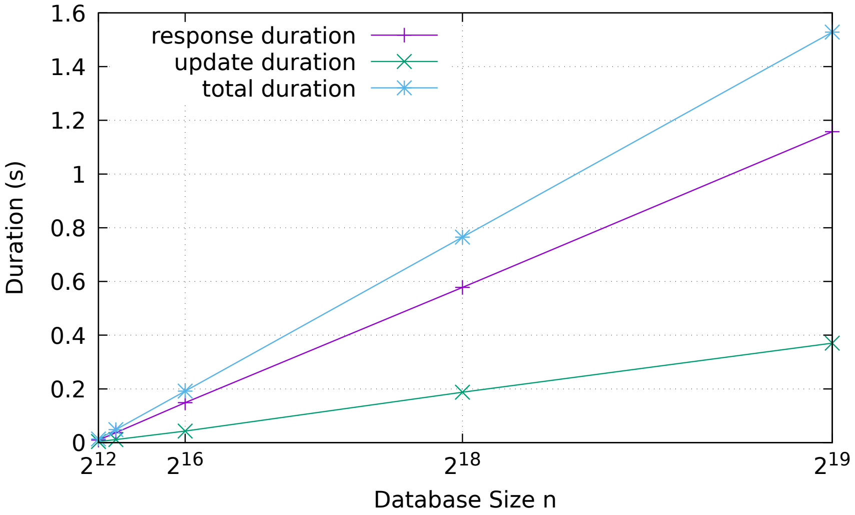

Fig. 1: Amortized duration of server computation with database size n from 2^{12} to 2^{19} and batch size $k = 256$. The x-axis denotes the database size n, and the y-axis denotes the duration (in seconds).

called eviction, during which the server runs an expensive algorithm $\log n$ times interactively with the client. This algorithm is instantiated with FHE and some concrete parameters in [3]. However, we note that there exists a hidden factor of $Z \log Z$ in their complexity analysis of eviction. This factor is a result of reshuffling Z blocks in each node (bucket) on a selected path during eviction. In detail, the dominant complexity during the eviction comes from performing $Z \cdot \log Z \cdot \log n \approx 2^{17}$ external products, where $Z \approx 2^9$ according to their choice of parameters, when $n = 2^{22}$. Therefore, the dominant factor is not limited by $\log n$ anymore, and the number of slots (Z) in every bucket is the key factor which impacts real performance. As a result, Z should also be chosen carefully according to n.

If we set $k = A$, we can compare two different designs in the amortized scenario with k queries. Therefore, our experimental numbers show that our stateless design can achieve comparable performance to that of stateful Onion Ring ORAM. The actual latency of their eviction phase also depends on how fast the client is at answering since it requires multiple client-server interactions. Additionally, our protocol parallelizes much better given additional server resources, whereas their eviction should be run sequentially and interactively, which also contributes positively to our practical performance. These factors should also be taken into account when applying such protocols in the real world.

Client Performance: Method 2. One of the benefits of our stateless design is that the client has much less of a computational burden compared to stateful constructions. The client-side computation only consists of generating queries (encrypting indices) and decrypting responses from the server. In Table 6, we demonstrate query generation time on two different consumer-grade laptops. The computation cost is small enough for the client to run efficiently even when using a normal computer.

On the other hand, Onion Ring ORAM necessarily requests the client to join the offline (eviction) phase after every A accesses. In the eviction phase, a client has to generate a permutation to reshuffle blocks in $\log n$ buckets on a chosen path, resulting in having to perform $O(\log n \cdot Z \cdot \log Z)$ encryptions. Moreover, the client also has to download Z data elements (which are encrypted) multiple times during the eviction phase.

Table 6: Duration of performing one RGSW Encryption on client setups

Intel i5-6600	**Apple M1**
0.17s	0.065s

7.4 Memory Usage

For the server's memory overhead, our approach does not rely on the structure of the database; it always has a constant overhead, apart from Method 2 which relies on the number of hash functions that we use. On the other hand, Onion Ring ORAM and other stateful designs rely heavily on the tree depth (L) and the number of slots (Z) in each node to guarantee the security. For Onion Ring ORAM, both of these parameters are highly dependent on n. However, with the concrete parameters used for their implementation, they chose Z and L carefully for $Z \cdot 2^L$ to be bounded by $4n$ for optimal performance results. Therefore, with our choice of $w = 3$ in Method 2, we store $3n$ elements with the server, and our storage overhead is similar to Onion Ring ORAM in practice.

Regarding real-time memory consumption on the server's side, recall from Section 6 that we need to store $w \cdot n$ RGSW ciphertexts $L_{i,j}$ in memory temporarily to perform the consistency correction algorithm, which introduces considerable memory usage since one RGSW ciphertext consists of 2ℓ RLWE ciphertexts. But it is possible to reduce the memory usage by a half if the consistency correction is performed as soon as possible, i.e., by performing consistency correction on the three entries in the encoded database that are mapped from the same original index right after they are available, instead of waiting for $\mathsf{P}anacea_{core}$ to be completed on every bucket. We leave this optimization for future work.

7.5 Monetary Cost

Using the same parameter set from Table 3 and $k = 256$, we present the concrete cost of operating our server using Method 2 without query packing. Appendix C provides a reference price list retrieved from the website[10] of Google Cloud Platform (GCP).

Data Transfer and Storage. Given that internal storage traffic (i.e. transfers between a cloud instance and a standard storage bucket) is free of charge on GCP, we can neglect the monetary cost of transferring the database from client to server. The same applies for Ingress traffic from the client to the cloud, which means that the client can transmit their queries with no server-side cost.

Our fixed parameters grant us a fixed server response size of 12.5MB, regardless of the database size. This translates to \$0.0028 of billable egress traffic per response. Referring to Table 9, we present the of the database to be stored size in GB with n from 2^{12} to 2^{19} and the estimated cost in US Dollars per month.

[10] `https://cloud.google.com/compute/all-pricing`

Table 7: Growth of database in GB with n from 2^{12} to 2^{19} and estimated cost of storage per month in US Dollars

n	**Database Size (GB)**	**Cost per Month**
2^{12}	0.403	\$0.010478
2^{14}	1.611	\$0.041886
2^{16}	6.445	\$0.16757
2^{18}	25.782	\$0.670332
2^{19}	51.564	\$1.340664

Table 8: Server time to load the database from storage, perform the ORAM computation, store the updated database in storage, and their sum (total server time) in seconds with the estimated monetary cost in US Dollars.

n	**Load**	**ORAM**	**Update**	**Total**	**Cost**
2^{12}	0.1	3.48	0.1	3.68	\$0.02576
2^{14}	0.402	12.42	0.402	13.2	\$0.092568
2^{16}	1.611	49.13	1.611	52.35	\$0.366464
2^{18}	6.445	195.94	6.445	208.83	\$1.46181
2^{19}	12.891	391.26	12.891	417.042	\$2.919294

Server-Side Protocol Execution. We assume that our server runs on demand. More details on our proposed server setup can be found in Section 7.6. For simplicity, we assume that regardless of the value of n, a server instance would be equipped with 36 cores and 2TB of RAM to replicate our experimental setup. Thus referring to Table 9 we can assume that running our server would cost \$25.2/h or \$0.007/s. Table 8 provides an estimated cost in US Dollars per batch of 256 queries with n from 2^{12} to 2^{19}.

7.6 On-Demand Server

During the initial protocol setup, no server interaction is required with the client. The parameters and encrypted database are transferred from the client to the storage solution, which in our case is a GCP Standard Storage Bucket. Whenever a query is submitted by a client, the query is ingested by a cloud scheduler, operating at negligible cost, which boots up an instance of our server from a snapshot. This instance would exist only for the duration of performing one computation and database update as per Table 5, to minimize billing costs. The server instance, provided with the client's query by the scheduler, would then fetch the database and parameters from the storage bucket and store them directly to on a ramdisk partition. We assume that the speed at which the ramdisk partition on the server can communicate bi-directionally with the storage solution at a rate of 4GB/s[11].

Having received all data it requires, the server then computes the response, transmits it back to the client, updates the database and stores it on the storage solution then gets shut down automatically. The total billable time during which our server operates can be calculated by summing the time to load the encrypted database, the total computation running time and the time to store the updated database.

[11] https://cloud.google.com/compute/docs/disks/performance

8 Discussion

8.1 Noise Growth

Due to the nature of FHE schemes, after performing a certain number of operations, ciphertexts accumulate noise over time. When the noise becomes too high, it would no longer be possible to decrypt ciphertexts successfully. This applies to our ORAM protocol: after a certain number of (sequential) accesses, the result returned by the server may respond with ciphertexts that cannot be decrypted by the client anymore.

We show that our protocol can support over 100,000 (amortized) queries (Method 2) before bootstrapping is needed to refresh the noise budget. It is because the noise contained in the data is additively accumulated per access due to the homomorphic operation that we use called external product in [6].

The multiplicative depth of the protocol grows linearly in the number of sequential accesses. In every access, the new noise term to be added is generated via a series of external products ($\mathsf{HomDemux}$ and $\mathsf{RLWEtoRGSW}$) among ciphertexts which have low noise. Let's denote this term by v_q which does not depend on the noise contained in data, hence it has always same variance bound for a fixed set of parameters. The noise contained in a data element e_{data} is updated as $e_{data} := c_1 \cdot e_{data} + c_2 \cdot v_q$ after one access, where c_1, c_2 are small constants less than 5. Therefore, the noise (in term of number of bits) grows with $O(\log Q)$ complexity, where Q is the number of accesses. That is why we can keep small noise growth. Specifically, for $n = 2^{19}, t = 2^8$ running Method 2 (Section 6) with batch size $k = 256$, and fixing our other parameters as described, we were able to process 425 successive batched queries, averaged after 10 samples, before we reached a decryption failure.

8.2 Defense Against Malicious Servers

The design presented in this paper mainly focuses on an honest but curious server (adversary). Our design can be extended with minor modifications to support the assumptions required when considering a malicious server. We use a technique similar to [11], where the client stores message authentication codes along with the elements. More specifically, for every element d_i the client stores $\mathsf{Enc}(d_i, \mathsf{MAC}_K(i, d_i))$ at slot i in the database. This allows the client to immediately detect if the server does not follow the protocol and returns a bad ciphertext or a ciphertext from any location other than the queried index for a Read access. However, the server might still be able to replay old elements from the same location. One way to mitigate against replay attacks, and guarantee the freshness of the blocks, would be to maintain a separate counter for each element, and use that counter as part of the input of MAC. After every Write access the corresponding counter needs to be increased. However, this requires that client maintains a state of all the counters locally, which adds state to our stateless design.

So we suggest a probabilistic defense mechanism: For every t Write accesses, the client chooses one of them at random for verification purposes. The index corresponding to that verification access would be Read immediately after a Write operation at the same location. If the server's response does not match the previously updated value on the position, then the client can immediately detect the server's malicious behavior. Note that our privacy guarantees imply that the server cannot learn anything about the type of intended access, and therefore, the server has no way to infer access patterns to identify the for verification accesses. For every malicious action, the client will detect the server with a probability of at least $\frac{1}{t}$.

When the client detects such a malicious action by the server, if the client aborts immediately, that provides a unique side-channel to the server. The server knows that after (maliciously) modifying a certain block the client has aborted the protocol. We leave it for future work to handle such side-channel attacks. However, the server will try to avoid detection for financial purposes, e.g., if it is a cloud server running storage as a service. If the clients can detect such malicious behavior from the server, they will eventually take their businesses elsewhere.

9 Conclusion

This work introduces a novel ORAM paradigm which supports a stateless client and has a comparatively low bandwidth blowup; a major departure from the tree-based ORAM constructions in the existing literature. We propose our core protocol supporting single-access queries along with two variations for multi-access: The first variation uses parallelism to reduce the server's computation time by 25% when compared to the core protocol; the second variation (Section 6) uses probabilistic batch codes to significantly improve the performance in the amortized setting (by over 180 times when compared to the core protocol for $n = 2^{12}$), achieving only 1.53 seconds of access time for $n = 2^{19}$. While the asymptotic computation overhead for the server in our protocol looks high compared to existing designs, our implementation's concrete performance is more than acceptable, especially for small to moderately sized databases.

Acknowledgement

The cloud resources and services used in this work were partly provided by the VSC (Flemish Supercomputer Center), funded by the Research Foundation - Flanders (FWO) and the Flemish Government. This work is partially supported by the Research Council KU Leuven under the grant C24/18/049, CyberSecurity Research Flanders with reference number VR20192203, and Defense Advanced Research Projects Agency (DARPA) under contract number FA8750-19-C-0502. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of the funders.

References

1. Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society Press, May 2018.
2. Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 131–148. Springer, Heidelberg, March 2014.
3. Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring ORAM: Efficient constant bandwidth oblivious RAM from (leveled) TFHE. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 345–360. ACM Press, November 2019.
4. Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1243–1255. ACM Press, October / November 2017.
5. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 3–33. Springer, Heidelberg, December 2016.
6. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020.

7. Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Concrete: Concrete operates on ciphertexts rapidly by extending tfhe. In *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, volume 15, 2020.
8. Kelong Cong, Debajyoti Das, Jeongeun Park, and Hilder V.L. Pereira. Sortinghat: Efficient private decision tree evaluation via homomorphic encryption and transciphering. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 563–577, New York, NY, USA, 2022. Association for Computing Machinery.
9. Jonathan L. Dautrich Jr., Emil Stefanov, and Elaine Shi. Burst ORAM: Minimizing ORAM response times for bursty access patterns. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 749–764. USENIX Association, August 2014.
10. Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 145–174. Springer, Heidelberg, January 2016.
11. Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram. *SIGPLAN Not.*, 50(4):103–116, mar 2015.
12. Alan M. Frieze and Tony Johansson. On the insertion time of random walk cuckoo hashing. In Philip N. Klein, editor, *28th SODA*, pages 1497–1502. ACM-SIAM, January 2017.
13. Robin Geelen, Michiel Van Beirendonck, Hilder V. L. Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios Dimou, Ingrid Verbauwhede, Frederik Vercauteren, and David W. Archer. BASALISC: Flexible asynchronous hardware accelerator for fully homomorphic encryption. Cryptology ePrint Archive, Report 2022/657, 2022. `https://eprint.iacr.org/2022/657`.
14. Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In Emiliano De Cristofaro and Matthew K. Wright, editors, *PETS 2013*, volume 7981 of *LNCS*, pages 1–18. Springer, Heidelberg, July 2013.
15. Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th ACM STOC*, pages 182–194. ACM Press, May 1987.
16. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, may 1996.
17. Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In Yuval Rabani, editor, *23rd SODA*, pages 157–167. ACM-SIAM, January 2012.
18. 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. Cryptology ePrint Archive, Report 2017/819, 2017. `https://eprint.iacr.org/2017/819`.
19. Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In Yuval Rabani, editor, *23rd SODA*, pages 143–156. ACM-SIAM, January 2012.
20. Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on OT extension. Cryptology ePrint Archive, Report 2016/930, 2016. `https://eprint.iacr.org/2016/930`.
21. Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 415–430. USENIX Association, August 2015.
22. Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 238–252, New York, NY, USA, 2021. Association for Computing Machinery.
23. Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 197–214. Springer, Heidelberg, December 2011.
24. 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. *J. ACM*, 65(4), apr 2018.
25. Emil Stefanov and Elaine Shi. ObliviStore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy*, pages 253–267. IEEE Computer Society Press, May 2013.
26. Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. In *NDSS 2012*. The Internet Society, February 2012.

27. Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 299–310. ACM Press, November 2013.
28. 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*, CCS '15, page 850–861, New York, NY, USA, 2015. Association for Computing Machinery.
29. Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, abhi shelat, and Elaine Shi. SCORAM: Oblivious RAM for secure computation. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 191–202. ACM Press, November 2014.

A Security

Here we revisit our security definition and theorems and provide the proofs.

A.1 Security Definition

Recall that we adopt a standard security definition similar to [26,21].

Definition 1 (Security Definition) *Let,*

$$\mathbf{y} := \{(\mathsf{a}_1, \mathsf{op}_1, \mathsf{data}_1), \ldots, (\mathsf{a}_s, \mathsf{op}_s, \mathsf{data}_s)\}$$

denote a sequence of data accesses by the client, where a_i denotes the address of the block being read or written on the i-th access, op_i denotes the whether the operation on a_i is a read or write, data_i denotes the data if the i-the operation is a write operation. Let $A(\mathbf{y})$ denote the sequence of communication between the client and the server under an ORAM protocol for the data access sequence $\mathbf{y}$. We say that the ORAM protocol is secure, if

1. *for any chosen pair of data access sequences $\mathbf{y}$ and $\mathbf{y}'$ of same length, the server cannot (computationally) distinguish between $A(\mathbf{y})$ and $A(\mathbf{y}')$,*
2. *for a sequence of input $\mathbf{y}$ the ORAM protocol returns output to the client consistent with $\mathbf{y}$ with probability $> 1 - \mu$, where μ is a very small quantity.*

Note that the parameter μ in our definition is not a negligible function in the asymptotic sense. Instead, we show concrete security for $\mu \approx 2^{-40}$, for the version with probabilistic batch codes (PBC). However, the version without PBC always returns the correct output. Recall that we have an honest-but-curious server which can store all ciphertexts and other data sent by the client and, afterwards, act as a probabilistic polynomial time adversary to perform additional computations in order to infer the client's private data or access patterns.

A.2 Security Proof for P$anacea_{core}$

Theorem 1 (Security of P$anacea_{core}$) *Assuming IND-CPA security and correctness of the underlying FHE scheme, Panacea$_{core}$ provides security as defined in Definition 1 with μ negligible in n.*

Proof (Proof-sketch). We perform the proof in two parts:

1. first, we show that it is difficult for the server to distinguish between two communication sequences corresponding to data access sequences $\mathbf{y}$ and $\mathbf{y}'$, where $|\mathbf{y}| = |\mathbf{y}'|$;

2. second, we show that the ORAM protocol always returns correct output and updates the accessed elements correctly.

Claim. Given two sequences of data accesses $\mathbf{y} = \{(\mathsf{a}_1, \mathsf{op}_1, \mathsf{data}_1), \dots, (\mathsf{a}_s, \mathsf{op}_s, \mathsf{data}_s)\}$ and $\mathbf{y}' = \{(\mathsf{a}'_1, \mathsf{op}'_1, \mathsf{data}'_1), \dots, (\mathsf{a}'_s, \mathsf{op}'_s, \mathsf{data}'_s)\}$ of same length $s \in poly(\lambda)$, the server cannot (computationally) distinguish between the communication sequences $A(\mathbf{y})$ and $A(\mathbf{y}')$ under the protocol $\mathsf{P}anacea_{core}$.

Proof: Suppose, for the sake of contradiction, that the server can run a distinguisher $\mathcal{D}$ to distinguish between $A(\mathbf{y})$ and $A(\mathbf{y}')$ with non-negligible probability p.

We want to show that using this distinguisher $\mathcal{D}$, we can construct an adversary $\mathcal{A}dv$ for the IND-CPA game of the underlying FHE scheme. For our proof we consider the IND-CPA game where the adversary is allowed to send multiple challenge queries.[12]

The adversary $\mathcal{A}dv$ acts as the challenger in the ORAM security game against the server. The server can choose two sequences of data access $\mathbf{y} = \{(\mathsf{a}_1, \mathsf{op}_1, \mathsf{data}_1), \dots, (\mathsf{a}_s, \mathsf{op}_s, \mathsf{data}_s)\}$ and $\mathbf{y}' = \{(\mathsf{a}'_1, \mathsf{op}'_1, \mathsf{data}'_1), \dots, (\mathsf{a}'_s, \mathsf{op}'_s, \mathsf{data}'_s)\}$ of same length s. The challenger ($\mathcal{A}dv$ in this case) picks one of $\mathbf{y}$ and $\mathbf{y}'$ based on the challenge bit b of the IND-CPA game and runs the ORAM protocol as described below:

1. $\mathcal{A}dv$ allows the client to directly query the oracle of the IND-CPA game to receive encryptions for all database elements. The client provides the ciphertexts to $\mathcal{A}dv$, and $\mathcal{A}dv$ provides them to the server to set up the database for the ORAM protocol.
2. After $\mathcal{A}dv$ receives y and y' from the server, for each $i \in [1, s]$:
   - if $\mathsf{a}_i \neq \mathsf{a}'_i$, $\mathcal{A}dv$ uses $\mathsf{a}_i, \mathsf{a}'_i$ as a pair of challenge plaintexts for the IND-CPA game and the response to construct the corresponding ORAM access request to the server;
   - if $\mathsf{op}_i \neq \mathsf{op}'_i$, $\mathcal{A}dv$ uses op_i and op'_i as a pair of challenge plaintexts for the IND-CPA game and the response to construct the corresponding request to the server;
   - if $\mathsf{data}_i \neq \mathsf{data}'_i$, $\mathcal{A}dv$ uses $\mathsf{data}_i, \mathsf{data}'_i$ as a pair of challenge plaintexts for the IND-CPA game and the response to construct the corresponding request to the server;
   - $\mathcal{A}dv$ receives the ciphertexts corresponding to all remaining elements of y (or y', since those elements are same) by querying the oracle of the IND-CPA game before sending the challenge pairs. Using them, $\mathcal{A}dv$ can construct all s access requests. $\mathcal{A}dv$ instructs the client to send the requests to the server.
3. Using the ciphertexts received from the IND-CPA game (as responses to the challenge pairs and oracle queries), the client runs the ORAM protocol with the server for s rounds.
4. If the distinguisher $\mathcal{D}$ for the ORAM protocol returns 1, $\mathcal{A}dv$ outputs 1 to the IND-CPA game.

Note that, whenever some parts of $\mathsf{y}[i]$ and $\mathsf{y}'[i]$ are different, $\mathcal{A}dv$ (on behalf of the client for the ORAM protocol) chooses the corresponding ciphertext based on the response of the IND-CPA game. Therefore, $\mathcal{A}dv$ effectively picks y or y' based on the challenge bit b of the IND-CPA game without actually knowing b. $\mathcal{A}dv$ does not observe any of the database elements (during setup or the protocol run), or the actual queries in plaintext. Therefore, the server does not have any information other than the distinguisher $\mathcal{D}$, and the observed ciphertexts.

According to the ORAM protocol, the client sends all query elements $(\mathsf{a}_i, \mathsf{op}_i, \mathsf{data}_i)$ as encrypted values. All database elements are also encrypted and all encrypted ciphertexts are re-randomized

[12] Security in a (polynomially bounded) multi-challenge IND-CPA game implies security in a single challenge IND-CPA game.

(as a result of operations under the FHE scheme) after every query. The response received by the client for each access is also encrypted.

For every access in ORAM, the server touches database elements: so that the access pattern is identical for every query.

This means that the distinguisher $\mathcal{D}$ only has the ciphertexts that the client sends to the server or the results of homomorphic computations at its disposal. Therefore, $\mathcal{A}dv$ wins the multi-query IND-CPA game with probability p. Which contradicts the IND-CPA security of the FHE scheme unless p is negligible. $\diamond$

Claim. Given an input 3-tuple $(\mathsf{a}, \mathsf{op}, \mathsf{data})$ to the ORAM access protocol, the client always receives the correct response and the accessed element is correctly updated in case of a write operation.

Proof: The correctness of the ORAM protocol is implied from: (1) correctness of the underlying FHE scheme, (2) the homomorphic traversal algorithm accessing the correct location, (3) the final step returning the correct value if the above conditions are satisfied, (4) and, if $\mathsf{op} = \mathsf{Write}$, the location is updated correctly.

By assumption, the FHE scheme is correct: all valid homomorphic operations will generate correct ciphertexts which can be correctly decrypted.

We can prove the correctness of homomorphic traversal by the method of induction on the number of database elements n. For simplicity, we assume $n = 2^\ell$ for some integer ℓ.

We first show, as the base case, that homomorphic traversal is correct for $\ell = 1$, i.e., $n = 2$. In that case, the traversal tree consists of only the root and two leaves directly connected to the root. Before the traversal, the root contains the value $b = \mathsf{Enc}(1)$, and each of the leaves contain $\mathsf{Enc}(0)$. The client sends only one bit of a (encrypted), which that acts as the controller bit for the root. Let us denote that encrypted controller bit with c. We update the right leaf node with $b \cdot c$, which is 1 if $c = 1$, and 0 otherwise (assuming the correctness of the FHE scheme). Similarly, the left node is updated with $(1 - c) \cdot b$, which is 1 if $c = 0$, and 0 otherwise.

Now, as an inductive hypothesis, let us assume that homomorphic traversal is correct up to a certain tree depth ℓ. This means that for an ℓ-bit a, the traversal produces $n = 2^\ell$ ciphertexts: Of which $n - 1$ are $\mathsf{Enc}(0)$, and one is $\mathsf{Enc}(1)$ at index a. Now we want to show that homomorphic traversal is correct for depth $\ell + 1$.

There are 2^ℓ nodes at the ℓ-th level, and $n = 2^{\ell+1}$ leaf nodes at level $\ell + 1$. Now, a has $\ell + 1$ bits. After, ℓ iterations of the homomorphic traversal, all nodes at level ell contain $\mathsf{Enc}(0)$, except for one at location denoted by $\mathbf{x} = \{\mathsf{a}[i]\}_{i \in [0, \ell-1]}$ which contains $\mathsf{Enc}(1)$ (by inductive hypothesis). Let us denote those nodes with $b_i, i \in [0, 2^\ell - 1]$; for all i, $b_i = \mathsf{Enc}(0)$ except $b_x = \mathsf{Enc}(1)$. Consequently, during the $(\ell + 1)$-th iteration (which is also the last iteration), all leaf nodes will get the value $\mathsf{Enc}(0)$ except for the child nodes of b_x. We know from the structure of binary tree, that the child nodes of b_x are at indices $x||0$ and $x||1$ among the leaf nodes. Similar to the argument of base case, if $\mathsf{a}[\ell + 1] = 0$, then the left child is $\mathsf{Enc}(1)$ and the right child is $\mathsf{Enc}(0)$, and vice-versa. Therefore, homomorphic traversal sets the correct element at depth $\ell + 1$.

Now the only thing remaining is to prove is that the final output is correct. Additionally, iff $\mathsf{op} = \mathsf{Write}$, then the database element is updated correctly with data. The first condition is implied by the correctness of the dot product between the output of the homomorphic traversal and the database element, which is in turn implied by the correctness of the FHE scheme. The second condition (correct update) is implied by the correctness of the RW and CMUX operations. RW ensures that the data element is updated iff $\mathsf{op} = \mathsf{Write}$, AND CMUX ensures that any update happens only at the location corresponding to the leaf with value $\mathsf{Enc}(1)$ after homomorphic traversal. The correctness of both of these operations are implied by the correctness of the FHE scheme. $\diamond$

From the above two claims we can claim that $\mathsf{P}anacea_{core}$ does not leak information about the client's access pattern $\mathbf{y}$, and always returns the correct result for all possible sequence $\mathbf{y}$. □

A.3 Security Proof for $\mathsf{P}\boldsymbol{anacea_{pbc}}$

Theorem 2 (Security of $\mathsf{P}anacea_{pbc}$) *Assuming IND-CPA security and correctness of the underlying FHE scheme, given a sequence of data accesses* $\mathbf{y} = \{(\mathsf{a}_1, \mathsf{op}_1, \mathsf{data}_1), \ldots, (\mathsf{a}_s, \mathsf{op}_s, \mathsf{data}_s)\}$ *of length s, with each batch of size k, and a failure probability p of the probabilistic batch code, $\mathsf{P}anacea_{pbc}$ provides security as defined in Definition 1 with* $\mu < p \cdot \frac{s}{k}$.

Proof (proof-sketch). Similar to the proof of Theorem 1, we also provide this proof in two parts:

1. we show that it is difficult for the server to distinguish between two communication sequences corresponding to data access sequences $\mathbf{y}$ and $\mathbf{y}'$, where $|\mathbf{y}| = |\mathbf{y}'|$;
2. we show that the ORAM protocol returns correct output with a probability very close to 1.

We will first prove our second claim, then prove the first one. For simplicity, since the client always sends queries in batches of size k, we assume that $s = \alpha k$ where α is an integer. A similar argument can be extended easily for other values of α assuming that the client will add dummy access requests to complete the size k of the last batch.

Claim. Given a sequence of inputs $\mathbf{y} = \{(\mathsf{a}_1, \mathsf{op}_1, \mathsf{data}_1), \ldots, (\mathsf{a}_s, \mathsf{op}_s, \mathsf{data}_s)\}$ to the ORAM access protocol, the access operations (read or write) are correctly executed for all the input tuples $(\mathsf{a}_i, \mathsf{op}_i, \mathsf{data}_i)$ with probability $1 - \mu$ where $\mu < p \cdot \frac{s}{k}$.

Proof: In the proof of Appendix A.2 we have argued that the homomorphic traversal yields the correct output. However, $\mathsf{P}anacea_{pbc}$ additionally applies batching, and individually runs $\mathsf{P}anacea_{core}$ over each of the b buckets. Therefore, we can argue that each individual run over all buckets preserves correctness with 0 failure probability, when an access request is made by the client.

However, the batching technique comes with a failure probability p: for a (randomly chosen) batch of size k, the client will fail to create a batch which can retrieve all k elements with probability p. In such a case, we consider that our ORAM protocol has failed to generate the correct output.

For a sequence of length s, there is a total of $\frac{s}{k} = \alpha$ batches. The probability that none of them fails is quantified by:

$$P = (1 - p)^{\alpha} > 1 - \alpha p$$

Therefore, the probability that at least one of them fails is quantified by:

$$\mu = 1 - P < \alpha p$$

If all batches are successful, then the client retrieves correct output for all access requests in the sequence $\mathbf{y}$ (implied from the success condition of the batching and correctness of the FHE scheme). The client fails to retrieve all the s elements if one of the batches fails; which happens with probability $\mu < \alpha p = p \cdot \frac{s}{k}$. $\diamond$

Now we move to proving the first claim which states that two communication sequences $A(\mathbf{y})$ and $A(\mathbf{y}')$ corresponding to $\mathbf{y}$ and $\mathbf{y}'$ are indistinguishable by the server. However, to ensure that no leakage emerges from the number of batches actually requested by the client, in case there is a failure with PBC, we assume that the client completes the batch with dummy requests and sends the batch even if this action would not help the client retrieve/update the k elements. With that additional assumption, we can claim the following:

Claim. Given two sequences of data accesses $\mathbf{y} = \{(\mathsf{a}_1, \mathsf{op}_1, \mathsf{data}_1), \ldots, (\mathsf{a}_s, \mathsf{op}_s, \mathsf{data}_s)\}$ and $\mathbf{y}' = \{(\mathsf{a}'_1, \mathsf{op}'_1, \mathsf{data}'_1), \ldots, (\mathsf{a}'_s, \mathsf{op}'_s, \mathsf{data}'_s)\}$ of same length s, the server cannot (computationally) distinguish between the communication sequences $A(\mathbf{y})$ and $A(\mathbf{y}')$ under the protocol $\mathsf{P}anacea_{pbc}$.

Proof: The proof of this claim is very similar to that of Appendix A.2. The only difference is that the queries are now sent in batches of k; but nevertheless, the server still touches all the elements in the database for each batch. Same as in the proof of Appendix A.2, the server can break the IND-CPA game for the underlying FHE scheme if the security of the ORAM scheme is broken.

The server can choose two sequences with one different element (which is at position s): $(\mathsf{a}_i, \mathsf{op}_i, \mathsf{data}_i) = (\mathsf{a}'_i, \mathsf{op}'_i, \mathsf{data}'_i) \forall 1 \leq i \leq (s-1)$, but $\mathsf{a}_s \neq \mathsf{a}'_s$ and $\mathsf{data}_s \neq \mathsf{data}'_s$; however, the server picks $\mathsf{op}_s = \mathsf{op}'_s$. The server picks either $\mathsf{op}_s = \mathsf{Read}$ or Write uniformly at random. Then the server tries to distinguish which one of $\mathbf{y}$ and $\mathbf{y}'$ is chosen by the client.

The server plays two IND-CPA games $\mathcal{G}_1$ and $\mathcal{G}_2$ as an adversary. In $\mathcal{G}_1$, the server chooses the pair $(\mathsf{a}_s, \mathsf{a}'_s)$ as challenge plaintexts for the IND-CPA game, and in $\mathcal{G}_2$ the pair data_s and data'_s as challenge plaintexts. If the distinguisher $\mathcal{D}$ for the ORAM protocol returns 1, the server outputs 1 to both games. We skip the detailed proof here. $\diamond$

From the above two claims we can claim that $\mathsf{P}anacea_{pbc}$ provides security as defined in Definition 1 with $\mu < p \cdot \frac{s}{k}$

In our batching technique with PBC, we choose parameters such that $p \approx 2^{-40}$. That makes $\mu \approx 2^{-40} \times \frac{s}{k}$. Even after a million batch queries (with each batch containing k queries), the probability that any of the batches would fail is almost one-in-a-million.

B Query Unpacking

In this section, we discuss how to pack a query for slower noise growth, while keeping a practical computation time. Algorithm 3 of [3] has the least computation complexity among the conversion methods from $\mathsf{RLWE}(M(X))$ to $\mathsf{RLWE}(M_i)$, where $i \in [N]$ with $\emptyset(logN)$ complexity. However, this approach results in a lot of noise at the end as discussed in their paper. Therefore, we slightly modify the algorithm by adopting a key switching method (Algorithm 7 of [3]).

Before discussing the algorithm, we define below a ciphertext form which we will employ:

- $\mathsf{RLWE}'_s(m) := \mathbf{Z} + (0, \mathbf{g} \cdot m \in R_q^{\ell \times 2}$, where $\mathbf{Z}$ is a matrix where each row is an $\mathsf{RLWE}_s(0)$, and $\mathbf{g} := (b^0, b^1, \ldots, b^{\ell-1})^t$ for a positive integer b. The secret key s is defined as $s_0 + s_1 X + \cdots s_{N-1} X^{N-1}$.

Our query consists of $C := \mathsf{RLWE}(M(X))$ packing all bits in coefficients of $M(X)$. Then we rearrange the coefficient of C to place the desired coefficient at the constant term. Next, we run the following key switching algorithm to output $C_i := \mathsf{RLWE}(M_i)$.

Then, we switch the call of Algorithm 3 by the above algorithm in Algorithm 4 of Onion Ring ORAM to generate the same output.

The algorithm consists of N scalar products between an integer and polynomial, which costs less than N external products, with additive noise growth. Consequently, the output noise only has N factor in the variance, whereas Algorithm 3 incurs a factor of N^2.

Algorithm 7 Key Switching from an RLWE to an RLWE

```
Input: $\mathbf{c} := \mathsf{RLWE}_{\mathbf{s}}(M(X)) = (a, b)$, where $b = -a \cdot s + \Delta M(X) + e$, and $\mathsf{ksk} := \{\mathsf{ksk}_i := \mathsf{RLWE}'_s(s_i)\}_{i \in [0,\ldots,N]}$
Output: $c := \mathsf{RLWE}_s(M_0) \in R_q^2$
Parse $a := a_0 + a_1 X + \cdots + a_{N-1} X^{N-1}$
for $i \leftarrow 1$ to $N$ do
    Compute $c_i := (\langle \mathbf{g}^{-1}(a_i), \mathsf{ksk}_i[1] \rangle, \langle \mathbf{g}^{-1}(a_i), \mathsf{ksk}_i[2] \rangle)$
end for
Compute $c := (\sum_{i=1}^{n} c_i[1], b + \sum_{i=1}^{n} c_i[2])$
return $c$
```

C Google Cloud Platform Reference Price List

The price list from Google Cloud Platform is added in Table 9 for reference.

Table 9: Cloud services required and reference unitary price list from Google Cloud Platform (GCP) in US Dollars for Europe region

Service	**Unit**	**Price per Unit**
Standard Storage	GB/month	\$0.026
Storage Traffic	GB	Free
Compute CPU	Core/hour	\$0.0831
Compute Memory	GB/hour	\$0.00563
Premium Ingress Traffic	GB	Free
Premium Egress Traffic	GB	\$0.23