Applications of Doubly Efficient PIR

Ariel Hamlin[1]

Khoury College of Computer Sciences, Northeastern University, Boston, MA, USA hamlin.a@northeastern.edu

Abstract. Outsourced data is ubiquitous raises the privacy model in which a client must interact and store data on a potentially untrusted server. Databases may be encrypted but usage in the form of *access patterns* still may leak information about both data and queries. This motivates us to examine techniques to conceal access patterns from an untrusted server. Doubly-Efficient Private Information Retrieval (DEPIR) is one technique that allows a single server to perform an read with overhead (both bandwidth and server computation) that is sub-linear in the database size.
In this thesis proposal, I will discuss three different applications of DEPIR for outsourced data use. Private Anonymous Data Access (PANDA) uses techniques from DEPIR to achieve a multi-client private and anonymous data access. Rewindable ORAM (R-ORAM) leverages DEPIR to provide a variant of ORAM used in our Fully Homomorphic Encryption for RAM construction. Finally, I will explore DEPIR's applications to Distributed ORAM (DORAM) in the secure computation model.

1 Introduction

We have entered an era where most data has migrated to an outsourced cloud. This creates a privacy model in which the person hosting the data is potentially untrusted and must be defended against. However, this model may go beyond simple storage and support the usage of the data by one or more clients. This could take the form of targeted queries to retrieve the medical records of a patient in a medical study, or a wider demographic computation to compute the average number of cars in a Massachusetts household. It is possible to encrypt the data to prevent the housing server to learn the contents, but simple encryption does not hide the *physical location* which the data is accessed on the server. We call that location data for any access an *access pattern* and there have been a litany of attacks [] that show query and data recovery are possible if access patterns are leaked.

This motivates us to examine techniques to conceal access patterns from an untrusted server. Two such techniques are Private Information Retrieval (PIR) and Oblivious RAM (ORAM), which at their simplest forms, allows a client to retrieve a record from a untrusted server without revealing their access patterns. There are key differences between the techniques: ORAM relies on computational assumptions and focuses on optimizing the *overhead* needed to read each block, PIR on the other hand, can either be computational or information theoretic and seeks to optimize the *bandwidth* for each block. As a result, PIR schemes often require *linear* work in database size on the behalf of the server for a single access. This issue motivated [] to introduce Doubly-Efficient Private Information Retrieval (DEPIR), which preserves the bandwidth efficiency of basic PIR schemes and allows a single server to perform an access with work that is *sub-linear in the database size*. This is accomplished by allowing some pre-processing on the contents of the database.

In this proposal, I will discuss my work that applies DEPIR (and DEPIR techniques) to a variety of different settings:

1. **Private Anonymous Data Access (PANDA)** [12] seeks to allow multiple clients (a portion of which may collude with the server) to perform accesses to a single server while preserving their privacy *and* anonymity. We construct three versions of PANDA, a bounded read-only, a unbounded read-only, and writable version using techniques drawn from DEPIR and using FHE to eliminate the usage of their new assumption.
2. **Rewindable Oblivious RAM (R-ORAM)** [11] achieves a new variant of ORAM which allows the server to 'rewind' to a previous state and allow executions to proceed creating a 'fork'. We introduce two versions of R-ORAM, initial-state (ISR-ORAM) and any-state (ASR-ORAM) which support rewinding

to different intermediary states. We use DEPIR directly in both constructions. This work was originally part of the construction of Fully Homomorphic Encryption for RAM (RAM-FHE), but has applications for multi-client ORAM.

3. **Distributed ORAM (DORAM)** [13] our construction builds on an existing multi-server ORAM variant in which the client is executed as a secure computation between the servers. Our construction is the first to achieve sublinear computation for the servers and constant rounds of communication. We also provide a variant with better asymptotics when the number of reads is much greater than the number of writes — we 'MPC'-ify DEPIR to achieve this result.

2 Preliminaries

In this section, we provide several definitions and constructions of existing cryptographic primitives that we leverage in this work. We begin with a brief summary of our notation.

We use the convention of 0-indexing, with $[N] = \{0, 1, \ldots, N-1\}$ as containing all whole numbers less than N. Additionally, $S \times S'$ denotes the Cartesian product of two sets. Bold letters $\boldsymbol{v}$ denote vectors, subscripts $\boldsymbol{v}_i$ indicate the i^{th} element of a vector, and $(w_i)_{i\in[N]}$ constructs a vector from an ordered list of items $w_0, w_1, \ldots, w_{N-1}$. The notation $\|$ denotes concatenation of bitstrings, sets, or vectors into a single object of longer length containing the (ordered) union of all elements.

The notation $x \leftarrow \mathcal{D}$ indicates taking a sample from a probability distribution $\mathcal{D}$. By abuse of notation, $x \leftarrow S$ indicates sampling from the uniform distribution over set S; we sometimes use $x \xleftarrow{\$} S$ for emphasis. We use $\approx$ to indicate computational indistinguishability of two distributions; that is, $\mathcal{D} \approx \mathcal{D}'$ if no probabilistic polynomial time adversary $\mathcal{A}$ has a noticeable difference in output when given a sample from $\mathcal{D}$ or $\mathcal{D}'$. We use $|S|$ to denote cardinality of set S.

2.1 Doubly Efficient Private Information Retrieval

First introduced by Canetti et al. and Boyle et al. [1, 4], Doubly Efficent Private Information Retrieval (DEPIR) is a variant of PIR achieving sub-linear server work by allowing pre-processing of the database. The major building block DEPIR is locally decodably codes (LDCs). Specifically, an application of Reed-Muller Codes, which allows for *smooth* LDCs.

Definition 1 (Smooth LDC). *A s-smooth, k-query* locally decodable code *with message length N, codeword size M, with alphabet Σ is denoted by $(s, k, N, M)_\Sigma$-*smooth *LDC and consists of a tuple of PPT algorithms* $(\mathsf{Enc}, \mathsf{Query}, \mathsf{Dec})$ *with the following syntax:*

- Enc *takes a message $m \in \Sigma^N$ and outputs a codeword $c \in \Sigma^M$*
- Query *takes a index $i \in [N]$ and outputs a vector $\boldsymbol{x} = (x_1, \ldots, x_k) \in [M]^N$*
- Dec *takes in vector codeword symbols $\boldsymbol{c} = (c_{x_1}, \ldots, c_{x_k}) \in \Sigma^N$ and outputs a symbol $y \in \Sigma$*

And has the following properties:

- ***Local Decodability:*** *For all messages $m \in \Sigma^L$ and every index $i \in [N]$:*

$$\Pr[\mathsf{Dec}(\mathsf{Enc}(m)_{\boldsymbol{x}}) = m_i \ : \ \boldsymbol{x} \leftarrow \mathsf{Query}(i)] = 1$$

- ***Smoothness:*** *For all indices $i \in [N]$, a LDC is s-smooth if when sampling $(x_1, \ldots, x_k) \leftarrow \mathsf{Query}(i)$, $(x_1, \ldots, x_k)$ is uniformly distributed on $[N]^s$ for every distinct subset of size s.*

We now formally introduce DEPIR, in particular the secret key variant, called SK-DEPIR.

Definition 2 (Doubly Efficient PIR). *A* Doubly Efficient PIR (DEPIR) *for alphabet Σ consists of a tuple of PPT algorithms* $(\mathsf{KeyGen}, \mathsf{Process}, \mathsf{Query}, \mathsf{Resp}, \mathsf{Dec})$ *with the following syntax:*

- KeyGen *takes the security parameter 1^λ and outputs the key k*

– $\mathsf{Process}$ *takes a key* k*, database* $\mathsf{DB} \in \Sigma^N$ *and outputs processed database* $\tilde{\mathsf{DB}}$
– Query *takes a key* k*, database index* $i \in [N]$ *and outputs a query* q *and temporary state* State
– Resp *takes a query* q*, processed database* $\tilde{\mathsf{DB}}$ *and outputs a server response* c
– Dec *takes a key* k*, server response* c*, temporary state* State *and outputs a database symbol* $y \in \Sigma$

And has the following properties:

– ***Correctness:*** *For all* $\mathsf{DB} \in \Sigma^N$ *and* $i \in [N]$*:*

$$\Pr\left[\mathsf{Dec}(k, \mathsf{State}, c) = \mathsf{DB}_i \;:\; \begin{array}{r} k \leftarrow \mathsf{KeyGen}\left(1^\lambda\right) \\ \tilde{\mathsf{DB}} \leftarrow \mathsf{Process}\left(k, \mathsf{DB}\right) \\ (q, \mathsf{State}) \leftarrow \mathsf{Query}\left(k, i\right) \\ c \leftarrow \mathsf{Resp}\left(\tilde{\mathsf{DB}}, q\right) \end{array}\right] = 1$$

– ***Double Efficiency:*** *The runtime of* KeyGen *is* $\mathrm{poly}\left(\lambda\right)$*, the runtime of* $\mathsf{Process}$ *is* $\mathrm{poly}\left(N, \lambda\right)$*, and the runtime of* $\mathsf{Query}, \mathsf{Dec}$ *is* $o\left(N\right) \cdot \mathrm{poly}\left(\lambda\right)$*, where* N *is the database size.*
– ***Security:*** *Any non-uniform PPT adversary* $\mathcal{A}$ *has only* $\mathrm{negl}\left(\lambda\right)$ *advantage in the following security game with a challenger* $\mathcal{C}$*:*
  1. $\mathcal{A}$ *sends to* $\mathcal{C}$ *a database* $\mathsf{DB} \in \Sigma^N$*.*
  2. $\mathcal{C}$ *picks a random bit* $b \leftarrow \{0,1\}$*, and runs* $k \leftarrow \mathsf{KeyGen}\left(1^\lambda\right)$ *to obtain a key* k*, and then runs* $\tilde{\mathsf{DB}} \leftarrow \mathsf{Process}\left(k, \mathsf{DB}\right)$ *to obtain a processed database* $\tilde{\mathsf{DB}}$*, which it sends to* $\mathcal{A}$*.*
  3. $\mathcal{A}$ *selects two addresses* $i^0, i^1 \in [N]$*, and sends* (i^0, i^1) *to* $\mathcal{C}$*.*
  4. $\mathcal{C}$ *samples* $(q, \mathsf{State}) \leftarrow \mathsf{Query}(k, i^b)$*, and sends* 1 *to* $\mathcal{A}$*.*
  5. *Steps 3 and 4 are repeated an arbitrary (polynomial) number of times.*
  6. $\mathcal{A}$ *outputs a bit* b'*, and his* `advantage` *in the game is defined to be* $\Pr[b = b'] - \frac{1}{2}$*.*

As shown in [1, 4] we can achieve SK-DEPIR with sublinear or poly-log parameters.

Lemma 1. *There exists SK-DEPIR schemes with the following parameters, where* N *is the database size and* λ *is the security parameter:*

– ***Sublinear SK-DEPIR:*** *For any* $\epsilon > 0$*, the running time of* $\mathsf{Process}$ *can be* $N^{1+\epsilon} \cdot \mathrm{poly}(\lambda)$*, and the running time of* Query *and* Dec *can be* $N^{\epsilon} \cdot \mathrm{poly}(\lambda)$*.*
– ***Polylog SK-DEPIR:*** *The running time of* $\mathsf{Process}$ *can be* $\mathrm{poly}(\lambda, N)$*, and the running time of* Query *and* Dec *can be* $\mathrm{poly}(\lambda, \log N)$*.*

3 Private Anonymous Data Access

In this section we introduce Private Anonymous Data Access (PANDA). Starting with the primitive definitions for Read-Only and Public-Writes PANDA from [12].

3.1 Read-Only PANDA.

In this section we describe our read-only PANDA scheme. We first formally define this notion. At a high level, a PANDA scheme is run between a server S and n clients $C_1, \cdots, C_n$, and allows clients to *securely* access a database D, even in the presence of a (semi-honestly) corrupted coalition consisting of the server S and a subset of at most t of the clients. In this section, we focus on the setting of a *read-only, public* database, in which the security guarantee is that read operations of honest clients remain entirely private and anonymous, meaning the corrupted coalition learns nothing about the identity of the client performed the operation, or which location was accessed.

Definition 3 (RO-PANDA). *A* Read-Only Private Anonymous Data Access (RO-PANDA) *scheme consists of procedures* $(\mathsf{Setup}, \mathit{read})$ *with the following syntax:*

- $\mathsf{Setup}(1^\lambda, 1^n, 1^t, \mathsf{D})$ *is a function that takes as input a security parameter* λ*, the number of clients* n*, a collusion bound* t*, and a database* $\mathsf{D} \in \{0,1\}^L$*, and outputs the initial server state* State_S*, and client keys* $\mathsf{sk}_1, \cdots, \mathsf{sk}_n$*. We require that the size of the client keys* $|\mathsf{sk}_j|$ *is bounded by some fixed polynomial in the security parameter* λ*, independent of* $n, t, |\mathsf{D}|$*.*
- *read is a protocol between the server* S *and a client* C_j*. The client holds as input an address* $\mathsf{addr} \in [L]$ *and the client key* sk_j*, and the server holds its current states* State_S*. The output of the protocol is a value* val *to the client, and an updated server state* State'_S*.*

We require the following correctness and security properties.

- ***Correctness:*** *In any execution of the* Setup *algorithm followed by a sequence of read protocols between various clients and the server, each client always outputs the correct database value* $\mathsf{val} = \mathsf{D}_{\mathsf{addr}}$ *at the end of each protocol.*
- ***Security:*** *Any PPT adversary* $\mathcal{A}$ *has only* $\mathrm{negl}(\lambda)$ *advantage in the following security game with a challenger* $\mathcal{C}$*:*
  - $\mathcal{A}$ *sends to* $\mathcal{C}$*:*
    - *The values* n, t *and the database* $\mathsf{D} \in \{0,1\}^L$*.*
    - *A subset* $T \subset [n]$ *of corrupted clients with* $|T| \leq t$*.*
    - *A pair of read sequences* $R^0 = \left(j_l^0, \mathsf{addr}_l^0\right)_{1 \leq l \leq q}, R^1 = \left(j_l^1, \mathsf{addr}_l^1\right)_{1 \leq l \leq q}$ *(for some* $q \in \mathbb{N}$*) , where* $\left(j_l^b, \mathsf{addr}_l^b\right)$ *denotes that client* $j_l^b \in [n]$ *reads address* $\mathsf{addr}_l^b \in [L]$*.*

    We require that $\left(j_l^0, \mathsf{addr}_l^0\right) = \left(j_l^1, \mathsf{addr}_l^1\right)$ *for every* $l \in [q]$ *such that* $j_l^0 \in T \vee j_l^1 \in T$*.*
  - $\mathcal{C}$ *performs the following:*
    - *Picks a random bit* $b \leftarrow \{0,1\}$*.*
    - *Initializes the scheme by computing* $\mathsf{Setup}\left(1^\lambda, 1^n, 1^t, \mathsf{D}\right)$*.*
    - *Sequentially executes the sequence* R^b *of read protocol executions between the honest server and clients. It sends to* $\mathcal{A}$ *the views of the server* S *and the corrupted clients* $\{C_j\}_{j \in T}$ *during these protocol executions, where the view of each party consists of its internal state, randomness, and all protocol messages received.*
  - $\mathcal{A}$ *outputs a bit* b'*.*

  The advantage $\mathsf{adv}_{\mathcal{A}}(\lambda)$ *of* $\mathcal{A}$ *in the security game is defined as:* $\mathsf{adv}_{\mathcal{A}}(\lambda) = |\Pr[b' = b] - \frac{1}{2}|$*.*

Efficiency Goals. Since a secure PANDA scheme can be trivially obtained by having the client store the entire database locally, or having the server send the entire database to the client in every read request, the *efficiency* of the scheme is our main concern. We focus on minimizing the client storage and the client/server run-time during each read protocol. At the very least, we require these to be $t \cdot o(|\mathsf{D}|)$.

In [12], we construct a *bounded-collusion* PANDA scheme, where we assume some upper bound t on the number of clients that collude with the server. The client and server efficiency scales linearly with t, but is otherwise poly-logarithmic in the data size and the total number of clients. In particular, our PANDA scheme allows for up to a poly-logarithmic collusion size t while maintaining poly-logarithmic efficiency for the server and the client. Our construction relies on the generic use of (leveled) *Fully Homomorphic Encryption (FHE)* [7,17] which is in turn implied by the *Learning With Errors (LWE)* assumption [16]. Our basic construction provides security against a semi-honest adversary.

We rely on an s-smooth, k-query LDC where $s = \lambda$ is set to be the security parameter. We think of the server S as consisting of $k' = k^2 t$ different "virtual servers", where t is the collusion bound. Each virtual server contains a permuted copy of the LDC codeword under a fresh PRP. Each client is assigned a random committee consisting of k out of k' of the virtual servers and gets the corresponding PRP keys. To retrieve an entry from the database, the client runs the LDC local decoding algorithm, which requests to see k codeword locations, and reads these locations using the virtual servers on its committee by applying the corresponding PRPs. It also reads uniformly random locations from the $k' - k$ virtual servers that are not on its committee.

In summary, we get the following theorem.

Theorem 1. *Assuming the existence of FHE, there exists a (read-only) PANDA scheme with n clients, t collusion bound, database size L and security parameter λ such that, for any constant $\epsilon > 0$, we get:*

- *The client/server run-time per* read *operation is $t \cdot \mathrm{poly}(\lambda, \log L)$.*
- *The server storage is $t \cdot L^{1+\epsilon} \cdot \mathrm{poly}(\lambda, \log L)$.*

3.2 PANDA with Writes.

We construct a PANDA scheme for public databases that supports write operations, but only guarantee privacy of read operations, a primitive we call *public-writes PANDA (PW-PANDA)*. Notice that this is the "best possible" security guarantee when there is (even) a (single) corrupted client. (Indeed, as the database is public, a corrupted coalition can always learn what values were written to which locations by simply reading the entire database after every operation.) We note that it suffices to consider this weaker security guarantee *when all clients are honest*, since any public-writes PANDA scheme can be generically transformed into a PANDA scheme which guarantees the privacy of write operations *when all clients are honest*. Indeed, one can implement a (standard) single-client ORAM scheme on top of the public-writes PANDA scheme, for which all clients know the private client key. (We note that the transformation might require FHE-encrypting the PANDA, to allow the server to perform operations on the PANDA which are caused by client operations on the ORAM.)

We now formally define the notion of a public-writes PANDA scheme.

Definition 4 (Public-Writes PANDA (PW-PANDA)). *A* public-writes PANDA (PW-PANDA) *scheme consists of procedures* $(\mathsf{Setup}, \mathit{read}, \mathit{write})$*, where* $\mathsf{Setup}, \mathit{read}$ *have the syntax of Definition 3, and* write *has the following syntax. It is a protocol between the server S and a client C_j. The client holds as input an address $\mathsf{addr} \in [L]$, a value v, and the client key sk_j, and the server holds its current states State_S. The output of the protocol is an updated server state State'_S.*

We require the following correctness and security properties.

- ***Correctness:*** *In any execution of the* Setup *algorithm followed by a sequence of* read *and* write *protocols between various clients and the server, where the* write *protocols were executed with a sequence Q of values, the output of each client in a* read *operation is the value it would have read from the database if (the prefix of) Q (performed before the corresponding* read *protocol) was performed directly on the database.*
- ***Security:*** *Any PPT adversary $\mathcal{A}$ has only* $\mathrm{negl}(\lambda)$ *advantage in the following security game with a challenger $\mathcal{C}$:*
  - *$\mathcal{A}$ sends to $\mathcal{C}$:*
    - *The values n, t, and the database $\mathsf{D} \in \{0,1\}^L$.*
    - *A subset $T \subset [n]$ of corrupted clients with $|T| \leq t$.*
    - *A pair of access sequences $Q^0 = \left(\mathsf{op}_l, \mathsf{val}_l^0, j_l^0, \mathsf{addr}_l^0\right)_{1 \leq l \leq q}, Q^1 = \left(\mathsf{op}_l, \mathsf{val}_l^1, j_l^1, \mathsf{addr}_l^1\right)_{1 \leq l \leq q}$, where $\left(\mathsf{op}_l, \mathsf{val}_l^b, j_l^b, \mathsf{addr}_l^b\right)$ denotes that client j_l^b performs operation op_l at address addr_l^b with value val_l^b (which, if $\mathsf{op}_l = \mathit{read}$, is $\perp$).*

    We require that $\left(\mathsf{op}_l, \mathsf{val}_l^0, j_l^0, \mathsf{addr}_l^0\right) = \left(\mathsf{op}_l, \mathsf{val}_l^1, j_l^1, \mathsf{addr}_l^1\right)$ for every $l \in [q]$ such that $j_l^0 \in T \vee j_l^1 \in T$; and $\left(\mathsf{val}_l^0, \mathsf{addr}_l^0\right) = \left(\mathsf{val}_l^1, \mathsf{addr}_l^1\right)$ for every $l \in [q]$ such that $\mathsf{op}_l = \mathit{write}$ (in particular, write operations differ only in the identity of the client performing the operation).
  - *$\mathcal{C}$ performs the following:*
    - *Picks a random bit $b \leftarrow \{0,1\}$.*
    - *Initializes the scheme by computing $\mathsf{Setup}\left(1^\lambda, 1^n, 1^t, \mathsf{D}\right)$.*
    - *Sequentially executes the sequence Q^b of* read *and* write *protocol executions between the honest server and clients. It sends to $\mathcal{A}$ the views of the server S and the corrupted clients $\{C_j\}_{j \in T}$ during these protocol executions, where the view of each party consists of its internal state, randomness, and all protocol messages received.*
  - *$\mathcal{A}$ outputs a bit b'.*

The advantage $\mathsf{adv}_{\mathcal{A}}(\lambda)$ *of* $\mathcal{A}$ *in the security game is defined as:* $\mathsf{adv}_{\mathcal{A}}(\lambda) = |\Pr[b' = b] - \frac{1}{2}|$.

We also consider extensions of PANDA to a setting that supports writes to the database. If the database is public and shared by all clients, then the location and content of write operations is inherently public as well. However, we still want to maintain privacy and anonymity for read operations, as well as anonymity for write operations. We call this a *public-writes* PANDA and it may, for example, be used to implement a public message board where clients can anonymously post and read messages, while hiding from the server which messages are being read. We also consider an alternate scenario where each client has her own individual private database which only she can access. In this case we want to maintain privacy and anonymity for *both the reads and writes* of each client, so that the server does not learn the content of the data, which clients are accessing their data, or what parts of their data they are accessing. We call this a *secret-writes* PANDA.

Public-writes PANDA scheme consists of $\log L$ levels of increasing size (growing from top to bottom), each containing size-λ "buckets" that hold several data blocks, and implemented with a B-bounded-access PE-PANDA scheme. To initialize our PANDA scheme, we generate PE-PANDA public- and secret-keys for every level. Initially, all levels are empty, except for the lowest level, which consists of a PE-PANDA for the database D. read operations will look for the data block in all levels (returning the top-most copy), whereas write operations will write to the top-most level, causing a reshuffle at predefined intervals to prevent levels from overflowing. We note that adding a *new copy* of the data block (instead of updating the existing data block wherever it is located) allows us to change *only* the content of the top level. This is crucial to obtaining a non-trivial scheme, since levels are implemented using a *read-only* PANDA, and so can only be updated by generating a *new* scheme for the *entire* content of the level, which might be expensive (and so must not be performed too often for lower levels).

Notice that since the levels are implemented using a PE-PANDA scheme (which, in particular, is only secure against a bounded number of accesses), security is guaranteed only as long as each level is accessed at most an a-priori bounded number of times. To guarantee security against *any* (polynomial) number of accesses, we "regenerate" each level when the number of times it has been accessed reaches the bound. This regeneration is performed by running the garble algorithm of the PE-PANDA scheme with a new label, consisting of the epoch number of the current level and the number of regeneration operations performed during the current epoch (this guarantees that every label is used at most once in each level). In summary, each level can be updated in one of two forms: (1) through a reshuffle operation that merges an upper level into it; or (2) through a regenerate operation, in which the PE-PANDA of the level is updated (but the actual data blocks stored in it do not change). We note that (unlike standard hierarchical ORAM) the reshuffling and regeneration need not be done obliviously, since the server knows the contents of all levels.

In [12], we show the following results.

Theorem 2. *Assuming the existence of FHE, there exists a* public-writes *PANDA with n clients, t collusion bound, database size L and security parameter λ such that, for any constant $\epsilon > 0$, we get:*

- *The client/server run-time per* read *operation is $t \cdot \mathrm{poly}(\lambda, \log L)$.*
- *The client run-time per* write *is $O(\log L)$, and the server run-time is $t \cdot L^{\epsilon} \cdot \mathrm{poly}(\lambda, \log L)$.*
- *The server storage is $t \cdot L^{1+\epsilon} \cdot \mathrm{poly}(\lambda, \log L)$.*

The same results as above hold for secret-writes *PANDA, except that the client run-time per* write *increases to $t \cdot \mathrm{poly}(\lambda, \log L)$, and L now denotes the sum of the initial database size and the total number of writes performed throughout the lifetime of the system.*

4 Rewindable Oblivious RAM

We define two ORAM variants which guarantee security against rewinding attacks. They are both used in the context of Fully Homomorphic Encryption for RAM [11]. We will focus on Rewindable ORAM as a stand-alone building block however. The two notions differ in the type of attacks they can handle. We first recall the notion of an access pattern, and the standard ORAM definition [8, 9, 15].

Access pattern A length-q *access pattern* Q consists of a list $(\mathsf{op}_l, \mathsf{val}_l, \mathsf{addr}_l)_{1 \le l \le q}$ of instructions, where instruction $(\mathsf{op}_l, \mathsf{val}_l, \mathsf{addr}_l)$ denotes that the client performs operation $\mathsf{op}_l \in \{\mathsf{read}, \mathsf{write}\}$ at address addr_l with value val_l (which, if $\mathsf{op}_l = \mathsf{read}$, is $\perp$).

Informally, an ORAM scheme allows a client to store his database, or "logical memory", remotely on a server, or "physical memory". Following a Setup procedure which generates client and server states, reads and writes to logical memory are performed through an interactive protocol Access between the client and server, where in each round the client generates a read request and an update request for the server. The access pattern to physical memory during the Access protocol completely hides from the server the database contents and access pattern to logical memory (see the full version for the formal definition).

4.1 Rewindable ORAM Security

We now describe a game that formalizes the security of our ORAM variants. The adversarial server in the game chooses a pair of initial databases, and (as in standard ORAM) two sequences of access patterns, with the goal of distinguishing between the executions of these sequences on the two databases. Unlike standard ORAM, the adversarial server in our security game can also *rewind* the execution to a previous state, and continue the execution from that state.

Definition 5 (Rewindable ORAM security game). *The ORAM security game is run between an adversary $\mathcal{A}$, and a challenger $\mathcal{C}$.*

1. *$\mathcal{A}$ sends to $\mathcal{C}$ two databases $\mathsf{D}^0, \mathsf{D}^1 \in \{0,1\}^N$.*
2. *$\mathcal{C}$ picks a random bit $b \leftarrow \{0,1\}$, and runs $\mathsf{Setup}\left(1^\lambda, \mathsf{D}^b\right)$ to obtain client and server states $\mathsf{sk}, \mathsf{State}$. $\mathcal{C}$ sends State to $\mathcal{A}$.*
3. *Let $\mathsf{State}_0 = \mathsf{State}$ and $\mathsf{sk}_0 = \mathsf{sk}$. Repeat the following $\mathrm{poly}(\lambda)$ times, where in the i'th iteration:*
   (a) *$\mathcal{A}$ sends to $\mathcal{C}$ an index $j_i \in \{0, 1, \ldots, i-1\}$, as well as two sequences of instructions $Q_i^0 = \left(\mathsf{op}_{i,l}, \mathsf{addr}_{i,l}^0, \mathsf{val}_{i,l}^0\right)_{l \in [q_i]}$, and $Q_i^1 = \left(\mathsf{op}_{i,l}, \mathsf{addr}_{i,l}^1, \mathsf{val}_{i,l}^1\right)_{l \in [q_i]}$, where $q_i \le \mathrm{poly}(\lambda)$, $\mathsf{op}_{i,l} \in \{\mathit{read}, \mathit{write}\}$, $\mathsf{addr}_{i,l}^0, \mathsf{addr}_{i,l}^1 \in [N]$, and $\mathsf{val}_{i,l}^0, \mathsf{val}_{i,l}^1 \in \{0,1\}$.*
   (b) *Starting from server state State_{j_i} and client state sk_{j_i}, $\mathcal{C}$ executes $\mathsf{Access}\left(\mathsf{op}_{i,l}, \mathsf{addr}_{i,l}^b, \mathsf{val}_{i,l}^b\right)$ for $1 \le l \le q_i$. Let $\mathsf{sk}_i, \mathsf{State}_i$ denote the updated client and server states (respectively) at the end of this sequence of executions. Let acc_i denote the access pattern to physical memory during this sequence of Access executions.*
   (c) *$\mathcal{C}$ sends acc_i to $\mathcal{A}$.*
4. *$\mathcal{A}$ outputs a bit b', and his advantage in the game is defined as $\Pr[b = b'] - \frac{1}{2}$.*

Discussion. The rewindable ORAM security game of Definition 5 captures several security variants, depending on the permissible choice of j_i. First, notice that the security game with $\mathrm{poly}(\lambda)$ iterations in the security game, when the adversary is restricted to choose $j_i = i-1$ in each iteration, and $\mathsf{D}^0 = \mathsf{D}^1$, yields the standard ORAM security definition without rewinds. Second, restricting the adversary to choose $j_i = \{0, i-1\}$ in every iteration i means the adversary can only rewind the execution to the initial state, but can adaptively decide to "extend" a previous execution. Restricting the adversary to choose $j_i = 0$ in every iteration corresponds to an adversary that can only rewind the execution to the initial state, where any rewind "finalizes" the current branch of the execution, and the adversary cannot later extend it. In the most general form, when j_i can take any value in $\{0, 1, \ldots, i-1\}$, we can assume without loss of generality that the adversary chooses a length-1 sequence in each iteration of the security game. This corresponds to an adversary that can rewind the ORAM to any intermediate state. The security game of Definition 5 can be used to capture various other security variants; we choose to focus on the latter two notions. Formally,

Definition 6 (Any-State Rewindable ORAM (ASR-ORAM)). *We say that an ORAM scheme is* Any-State Rewindable (ASR) *if any PPT adversary $\mathcal{A}$ has a $\mathrm{negl}(\lambda)$ advantage in the rewindable ORAM security game of Definition 5.*

Definition 7 (Initial-State Rewindable ORAM (ISR-ORAM)). *We say that an adversary $\mathcal{A}$ is* initial-state restricted *if in every iteration i of the rewindable ORAM security game of Definition 5, it chooses $j_i = 0$. We say that an ORAM scheme is* Initial-State Rewindable (ISR) *if any initial-state restricted PPT adversary $\mathcal{A}$ has a* $\mathrm{negl}(\lambda)$ *advantage in the rewindable ORAM security game of Definition 5.*

4.2 Rewindable ORAM Construction

Constructing rewindable (even ISR-) ORAM appears to be difficult, and none of the standard ORAM constructions suffice. Indeed, all standard ORAM constructions follow the "balls and bins" model in which each data block is represented as a "ball" and stored on the server in some "bin". Such structures cannot guarantee even ISR-ORAM security since, as noted above, if the client state is reset between accesses then the server can distinguish whether the client is accessing the same data block or not (when accessing the same block, the client will access the same "ball" on the server). Thus, we need fundamentally different techniques than prior ORAM constructions.

Our new approach to rewindable ORAM leverages SK-DEPIR [2, 5], which can be viewed as a *stateless read-only* ORAM. Informally, following a setup phase in which the client receives a secret key k and the server receives an encoding $\widetilde{D}$ of the database D, the client can privately read arbitrary locations i of D by reading a few positions in $\widetilde{D}$, without having to update the client/server state during the process. The server should learn nothing about the underlying locations i being read. In particular, we can think of SK-DEPIR as a very restricted form of ISR-ORAM for the class of RAM program $P_i(D)$ that read and output the i'th location of D.

ISR-ORAM from SK-DEPIR and standard ORAM. The ISR-ORAM scheme is conceptually simple. Recall that SK-DEPIR is read-only, while ISR-ORAM supports arbitrary RAM programs that can both read and write to the database. In both cases, we can rewind the state to its initial value after an execution while maintaining privacy of the underlying access pattern. The high-level idea is to use the SK-DEPIR to support reads, and use a *standard* ORAM scheme to support writes.

Specifically, the initial states in our ISR-ORAM are the client and server states $k, \widetilde{D}$ of the SK-DEPIR. To execute a RAM program P, the client initializes a fresh copy of a standard, non-rewindable ORAM O, which is initially empty. Writes are executed using the ORAM scheme O. To read some location i, the client reads i from both the ORAM O and the SK-DEPIR. If location i was found in O, the client uses that value, otherwise he uses the SK-DEPIR value. Thus, the client always gets the freshest copy of the value in any location. Note that rewinding the ISR-ORAM client/server to their initial states erases all information about O (which was initialized only in the first access), so we do not require rewindable security from O: the next access will instantiate a completely fresh ORAM scheme O for the execution. The scheme is described in [11].

Theorem 3 (ISR-ORAM). *Assume there exist OWFs and SK-DEPIR. Then there exists an ISR-ORAM scheme.*

Moreover, if the Query *and* Dec *algorithms of the SK-DEPIR scheme have* $\mathrm{poly}(\lambda)$ *complexity for databases of size N and security parameter λ, and the client (resp., server) state has size* $\mathrm{poly}(\lambda)$ *(resp.,* $\mathrm{poly}(\lambda, N)$*), then the* Access *complexity of the ISR-ORAM is* $\mathrm{poly}(\lambda)$*, and the client (resp., server) state has size* $\mathrm{poly}(\lambda)$ *(*$\mathrm{poly}(\lambda, N)$*).*

ASR-ORAM from SK-DEPIR via a hierarchical structure. The ASR-ORAM construction is more complex. ASR-ORAM should support repeated sequential execution of different programs, and remain secure when the adversary can rewind to any intermediate state from which it starts a new sequence of program executions. Unfortunately, this precludes our previous solution of storing intermediate values written during the execution in a standard, non-rewindable ORAM: rewinding to an intermediate point will rewind the ORAM.

We solve this problem by combining SK-DEPIR with techniques from hierarchical ORAM [9, 15]. In particular, our ASR-ORAM consists of a hierarchy of SK-DEPIR schemes of exponentially increasing size, where the top-most scheme has size 1 and the bottom-most scheme has size N. Initially, the data is entirely contained in the bottom-most scheme. To read a location i we try to read it using the SK-DEPIR schemes at

Fig. 1. Functionality $\mathcal{F}_{mem}$

1. On input of $(\mathsf{Init}, \tilde{\mathsf{DB}})$, set $\mathsf{DB} = \tilde{\mathsf{DB}}$, return random additive shares of DB^s to party s.
2. On input additive shares of $(\mathsf{op}, \mathsf{elem}, \mathsf{DB})$ from two parties do:
   (a) if $\mathsf{op} = \mathsf{read}$ then set $o = \mathsf{DB}[\mathsf{addr}]$
   (b) if $\mathsf{op} = \mathsf{write}$ then set $o = \mathsf{DB}[\mathsf{addr}]$ and $\mathsf{DB}[\mathsf{addr}] = \mathsf{val}$
   (c) Let o^1, o^2 be random, additive shares of o, and DB^s be random additive shares of DB. Return (o^s, DB^s) to party s.

all levels, and use the value found in the top-most scheme that contains i. To write a location i, we write it to the top level (which requires re-generating its SK-DEPIR scheme). As in Hierarchical ORAM this requires "reshuffles": every pre-determined number of writes, we need to merge sufficiently many of the top levels to ensure that their combined size is large enough to hold the database. Since levels are implemented using SK-DEPIR, this requires reading and re-writing the levels in their entirety. However, as levels get larger, they are "reshuffled" with decreasing frequency so the overall amortized complexity is low.

Notice that reshuffles reveal no information, even under arbitrary rewinding, because they occur at pre-determined times (independent of the access history), and reads are secure by the security of the (stateless) SK-DEPIR even under arbitrary rewinding.

Theorem 4 (ASR-ORAM). *Assume the existence of OWFs and SK-DEPIR, then there exists an ASR-ORAM scheme. Moreover, if for $\epsilon > 0$ the* Query *and* Dec *algorithms of the SK-DEPIR scheme have $N^\epsilon \cdot \mathrm{poly}(\lambda)$ complexity, and* Process *has $N^{1+\epsilon} \cdot \mathrm{poly}(\lambda)$ complexity for databases of size N and security parameter λ, then:*

- *The complexity of* Access *is $N^\epsilon \cdot \mathrm{poly}(\lambda)$.*
- *The client state has size $\mathrm{poly}(\lambda)$, and the server state has size $N^{1+\epsilon} \cdot \mathrm{poly}(\lambda)$.*

5 Distributed ORAM with sublinear computation and constant rounds

Distributed Memory First introduced by Bunn et al. [3], the ideal functionality $\mathcal{F}_{mem}$ in Figure 1 captures the behavior achieved by a DORAM executed through secure computation. The database is initialized on secret shares of the database, and subsequent accesses are also secret shared, as is their resulting output. Previous constructions have focused on achieving constant rounds or sublinear server computation. Our construction is the first to achieve both constant rounds *and* sublinear server computation.

5.1 DORAM Construction

In this section, we describe both of our DORAM constructions in more detail.

Sublinear DORAM We start with describing the original square-root ORAM (introduced by Goldreich and Ostrovsky [10]) that our construction is based on. There is a single read-only array of size N, which we call the *store*, and a writable *stash* of $\sqrt{N}$ size. Elements in the store are (address, value) pairs; at initialization, the elements are permuted with a permutation known only to the client, and all elements are encrypted. To perform a read at a particular address, the client checks the stash using a linear scan; if not present then it reads the permuted element from the read-only store, and if present then it is retrieved from the stash and a random 'dummy' element is made to the store instead. The newly-read element is placed in the stash, in order to maintain the invariant that each element is read only *once* from the store. In the case of a write, a dummy is read from the store and the element is written in the stash. After enough queries have been made

to fill the stash, a duration that which we call an *epoch*, the elements from the stash are *reshuffled* back into the main store, with only the newest write at each location being kept.

While the basic square-root ORAM construction achieves constant rounds with sublinear communication and server computation, it is non-trivial to convert it to a two-party DORAM. There are two major issues incurred by shifting this to the two party case: (1) representing the permutation over the elements of the store and (2) merging the elements from the stash back into the store.

We first discuss how to represent the permutation that maps addresses to physical locations in the store. In [18], which is also based on square root ORAM, they choose to represent the permutation as a shared array in recursive ORAMs. This improves computation complexity but leads to $O(\log N)$ rounds of communication. To maintain constant rounds, we must instead find a compact representation of the permutation. We look for inspiration from the original square-root scheme. There, they generate a random 'tag' for each element in the store using a random oracle and then sort the elements according to the tag. A lookup then involves only a random oracle evaluation and a binary search across the sorted elements. However, because it is a single server scheme, they must use an oblivious sorting network in order to break the correlation between items in different epochs, which does not run efficiently in constant rounds. We leverage the fact that we have a two servers to break up the oblivious sort into its two components, 'oblivious' + 'sort'. To prevent the server from mapping items between epochs, we use a simple constant round functionality to obliviously permute elements that allows each server to permute the elements in turn. As long as one server is honest, the data is permuted obliviously. This allows us to generate the tags using an oblivious pseudorandom function (OPRF), rather than a random oracle, on the newly obliviously permuted elements and then sort the tags locally. Lookup again is just an OPRF evaluation on the address shares and then a local binary search on the store.

The second challenge arises during the reshuffling phase of the protocol. In the original square-root ORAM, elements are simply moved back into their original locations (updated elements in the store, dummies back in the stash) by executing another oblivious shuffle. To solve this in constant rounds, we again exploit the ability to obliviously permute elements by using our two server architecture. In order to do that though, we must ensure that the elements that we are permuting do not contain any duplicates. For example, if a read was executed on index i, there would be two copies of element i, one in the stash and one in the store. To solve this issue, we note that the elements that have been read in the store is public knowledge to both servers. As long as we maintain the invariant *if an element has been read (or written to), it is in the stash, and each element only occurs in the stash once*, we can simply concatenate elements in the stash with the *unread* elements in the store at the end of an epoch. Once we have concatenated the elements we can obliviously permute them to get our new store. The stash can then just be filled with new dummy elements.

A more detailed discussion of our construction can be found in [13]

DORAM with Unlimited Reads. Thanks to the modularity of our base scheme, the components are easily extensible. We improve the performance of the read-only data store while keeping the rest of the construction (the stash, our periodic shuffling technique at the end of each epoch, etc) mostly intact.

The separation of our read-only store from a read-and-writable stash suggests an intriguing tradeoff: if we are willing to leak whether each operation is a read or a write operation, then it is beneficial to design an efficient read-only store that supports unlimited reads, and only pay for accessing the stash on (hopefully infrequent) write operations. This optimization allows us to increase the duration of each epoch, or in other words to amortize the cost of each shuffle over more reads. Concretely, in a scenario where the ratio of reads-to-writes is about N-to-1, then for any constant $\epsilon > 0$ we can construct a read-only store where whose amortized cost per query is just $O_\lambda(N^\epsilon)$. Here, the notation O_λ means that we suppress $\text{poly}(\lambda)$ terms in order to focus on the dependency on the database size. By reducing the size of the stash to $O_\lambda(N^\epsilon)$, we can support write operations with this performance as well.

Our strategy to construct a unlimited-reads store might seem counter-intuitive at first: we start from a doubly efficient PIR [1, 4] that supports unlimited reads and convert it into a two-server distributed data store. A *doubly efficient private information retrieval* (DEPIR) scheme is a client-server protocol for oblivious access to a public dataset that only requires sublinear computation for both the client and server operations

and constant rounds of communication between the two. At first glance, it may seem that a 1-server DEPIR is a strictly stronger primitive than a 2-server DORAM, so we might expect to construct the latter generically as a secure computation of the former. However, this intuition isn't true because there are three properties that we aim to satisfy with DORAM, but that (even a doubly efficient) PIR does not:

- Support for writes,
- Hiding the contents of the database, in addition to access patterns, and
- Ensuring that the secure computation is constant rounds when the two parties collectively emulate the (sublinear but not constant time) client, in addition to the client-server communication.

The main observation underlying this approach is that the SK-DEPIR protocol of Canetti et al. [4] is highly amenable toward secure computation since its operations mostly involve linear algebra in a finite field that can be done purely locally, plus bitstring and set operations that are easy to handle in constant rounds. The Canetti et al. SK-DEPIR construction is based on a locally decodable code (LDC) in the style of a Reed-Muller code, which encodes a dataset as a multivariate polynomial. As a result, the most challenging part of our multiparty computation protocol involves securely emulating the client's procedures to evaluate or interpolate a multivariate polynomial at $O(N)$ points. The naive methods for polynomial evaluation (via application of the Vandermonde matrix) or polynomial interpolation (via the Lagrange interpolation polynomial) involve multiplication of a public matrix by a secret-shared vector, which can be done non-interactively but requires $O(N^2)$ computation, which is too slow for our purposes.

Given a binary field $\mathbb{F} = \mathrm{GF}(2^\ell)$ and a subspace $H^m \subset \mathbb{F}^m$, we construct secure computation protocols for evaluating or interpolating an m-variate polynomial $p \in \mathbb{F}[x_1, \ldots, x_m]$ on all points in H^m in time that is quasilinear (rather than quadratic) in $|H^m|$. This protocol may be of independent interest, and in our protocol it is needed to achieve our goal of sublinear computation for the overall DORAM scheme. We construct this secure computation scheme in two stages: first we construct a secure computation protocol for the Additive Fast Fourier Transform protocol of Gao and Mateer [6] for univariate polynomials over a binary field, and then we bootstrap this protocol to handle multivariate polynomials by using recursion on the number of variables in the polynomial as previously shown by Kedlaya and Umans [14]. All operations in this protocol reduce to linear combinations of secret variables, so the entire computation can be done locally by each party on their own boolean secret shares without the need for any communication.

6 Future Work & Milestones

January In early January I am targeting to propose my thesis.

January - May There are two paths for potential work during the next several months:

1. Explore the applications of Rewindable ORAM to a multi-client ORAM setting. In particular, in a setting where clients cannot communicate with each other and must rely on the server to represent an accurate accounting of the state - is Rewindable ORAM required to achieve security in this setting without any additional timing assumptions?
2. Expand Rewindable ORAM to a 'rewindable' Garbled RAM setting. In existing Garbled RAM constructions, the list of programs the client wishes to execute either need to be specified up front, or in the adaptive setting, iteratively. A client cannot go back and 'branch off' of a previous state, which in a setting where there a many clients wanting to run different programs on the same initial copy of the database, can be prohibitive to garble dataset for them all independently.

June In June I hope to defend my thesis.

References

1. Boyle, E., Ishai, Y., Pass, R., Wootters, M.: Can we access a database both locally and privately? In: TCC (2). Lecture Notes in Computer Science, vol. 10678, pp. 662–693. Springer (2017)

2. Boyle, E., Ishai, Y., Pass, R., Wootters, M.: Can we access a database both locally and privately? In: TCC'17, Proceedings, Part II. pp. 662–693 (2017). https://doi.org/10.1007/978-3-319-70503-3_22, https://doi.org/10.1007/978-3-319-70503-3_22
3. Bunn, P., Katz, J., Kushilevitz, E., Ostrovsky, R.: Efficient 3-party distributed ORAM. In: SCN. Lecture Notes in Computer Science, vol. 12238, pp. 215–232. Springer (2020)
4. Canetti, R., Holmgren, J., Richelson, S.: Towards doubly efficient private information retrieval. In: TCC (2). Lecture Notes in Computer Science, vol. 10678, pp. 694–726. Springer (2017)
5. Canetti, R., Holmgren, J., Richelson, S.: Towards doubly efficient private information retrieval. In: TCC'17, Proceedings, Part II. pp. 694–726 (2017). https://doi.org/10.1007/978-3-319-70503-3_23, https://doi.org/10.1007/978-3-319-70503-3_23
6. Gao, S., Mateer, T.D.: Additive fast fourier transforms over finite fields. IEEE Trans. Inf. Theory **56**(12), 6265–6272 (2010)
7. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009. pp. 169–178 (2009). https://doi.org/10.1145/1536414.1536440, http://doi.acm.org/10.1145/1536414.1536440
8. Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: STOC'87. pp. 182–194 (1987)
9. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM **43**(3), 431–473 (1996)
10. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM **43**(3), 431–473 (1996)
11. Hamlin, A., Holmgren, J., Weiss, M., Wichs, D.: On the plausibility of fully homomorphic encryption for rams. In: CRYPTO (1). Lecture Notes in Computer Science, vol. 11692, pp. 589–619. Springer (2019)
12. Hamlin, A., Ostrovsky, R., Weiss, M., Wichs, D.: Private anonymous data access. In: EUROCRYPT (2). Lecture Notes in Computer Science, vol. 11477, pp. 244–273. Springer (2019)
13. Hamlin, A., Varia, M.: Two-server distributed ORAM with sublinear computation and constant rounds. IACR Cryptol. ePrint Arch. **2020**, 1547 (2020)
14. Kedlaya, K.S., Umans, C.: Fast modular composition in any characteristic. In: FOCS. pp. 146–155. IEEE Computer Society (2008)
15. Ostrovsky, R.: Efficient computation on oblivious RAMs. In: STOC'90. pp. 514–523 (1990)
16. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. J. ACM **56**(6), 34:1–34:40 (2009). https://doi.org/10.1145/1568318.1568324, http://doi.acm.org/10.1145/1568318.1568324
17. Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Foundations of secure computation **4**(11), 169–180 (1978)
18. Zahur, S., Wang, X., Raykova, M., Gascón, A., Doerner, J., Evans, D., Katz, J.: Revisiting square-root ORAM: efficient random access in multi-party computation. In: IEEE Symposium on Security and Privacy. pp. 218–234. IEEE Computer Society (2016)