Verifiable Oblivious Storage

Daniel Apon[*] Jonathan Katz[*] Elaine Shi[*] Aishwarya Thiruvengadam[*]

University of Maryland, College Park

Abstract

We formalize the notion of *Verifiable Oblivious Storage* (VOS), where a client outsources the storage of data to a server while ensuring data confidentiality, access pattern privacy, and integrity and freshness of data accesses. VOS generalizes the notion of Oblivious RAM (ORAM) in that it allows the server to perform computation, and also explicitly considers data integrity and freshness.

We show that allowing server-side computation enables us to construct asymptotically more efficient VOS schemes whose bandwidth overhead cannot be matched by any ORAM scheme, due to a known lower bound by Goldreich and Ostrovsky. Specifically, for large block sizes we can construct a VOS scheme with constant bandwidth per query; further, answering queries requires only poly-logarithmic server computation. We describe applications of VOS to Dynamic Proofs of Retrievability, and RAM-model secure multi-party computation.

1 Introduction

Oblivious RAM (ORAM) is a notion first proposed by Goldreich and Ostrovsky [21] in the context of protecting software from piracy. They consider an application in which a trusted CPU wishes to hide its memory-access patterns from an attacker who can view (and possibly modify) the entire contents of memory. Recently, as cloud computing has gained in popularity, ORAM has been recast as a means to securely outsource storage to an untrusted server, while hiding access patterns from the server.

In this paper, we propose Verifiable Oblivious Storage (VOS), which generalizes the notion of ORAM by allowing the storage medium to perform computation. In addition, it also explicitly incorporates notions of integrity and freshness. We will refer to integrity and freshness as verifiability in this paper.

Formally defining VOS. Our first contribution is to formally define VOS, and to differentiate the notion of VOS from ORAM. While we are the first to formalize the VOS notion, VOS has implicitly been used by other researchers earlier, often being referred to as ORAM. For example, Williams and Sion [36] recently proposed a scheme that improves round-complexity to $O(1)$ — since their scheme leverages server-side computation, it is implicitly a VOS scheme.

An important difference between VOS and ORAM schemes is that VOS schemes can be constructed to achieve asymptotically better bandwidth overhead than what can be achieved by any ORAM scheme. This is because all ORAM schemes are subject to a well-known lower bound result by Goldreich and Ostrovsky [21]. This result, however, does not apply to VOS.

[*] {dapon, jkatz, elaine, aish}@cs.umd.edu

Several applications where ORAM was previously employed can immediately achieve asymptotic bandwidth savings if we simply replace the ORAM with a VOS construction. For example, we know from prior work that RAM-model secure multi-party computation [25] and Dynamic Proofs of Retrievability [9] can be built using ORAM as a building block. In both these applications, the party storing data (or a share of the data) can perform computation. By replacing the underlying ORAM with a VOS in these constructions [9,25], we can immediately obtain asymptotic bandwidth savings as illustrated in Section 5.

Asymptotically efficient VOS construction. We show that, by allowing server-side computation, VOS schemes can be constructed that beat the known logarithmic lower bound on the bandwidth cost for any ORAM scheme [21]. Specifically, we show that there exists a VOS scheme with block size $\beta = \tilde{\Omega}(\lambda)$ (where λ is the security parameter) having $O(\beta)$ bandwidth cost for reading or writing a block; this scheme has $O(\beta)$ client-side storage, and uses only $O(1)$ roundtrips and requires only $O(\beta \cdot \text{poly} \log n) \cdot \text{poly}(\lambda)$ server-side computation per data access. This is asymptotically better than what any ORAM scheme can hope to achieve since, due to the lower bound by Goldreich and Ostrovsky, any ORAM scheme must have bandwidth cost $\Omega(\beta \log n)$ to read or write a block of β bits. *Note that this lower bound holds regardless of the block size β.*

1.1 Technical Highlight

Generic ORAM-to-VOS compiler in the semi-honest model. To construct efficient VOS schemes, we rely on fully homomorphic encryption (FHE) to encrypt and outsource the entire ORAM memory, as well as the ORAM client's secret state. The server can now perform computation on behalf of the client, without learning any secrets. The only occasion when the server needs to contact the client is to seek the client's help to decrypt the next physical address or sequence of physical addresses to read or write. The use of FHE or PIR to outsource the ORAM client's computation has been mentioned in earlier works [17, 27]. The main challenge, however, is *how to ensure security when the server is malicious, and may arbitrarily deviate from the prescribed computation.*

Generic ORAM-to-VOS compiler in the malicious model. Achieving security against a malicious server is much trickier in the VOS setting than in ORAM. While ORAM achieves integrity and freshness in a straightforward way by employing standard storage integrity techniques such as message authentication codes and Merkle hash trees, in VOS, we need to worry about a server that can arbitrarily deviate from the prescribed computation.

Naive applications of well-known techniques such as SNARKS [2, 3, 16] result in server computation that is linear in the size of the dataset. Instead, we leverage efficient Verifiable RAM computation (VC-RAM) to enforce honest server behavior. This allows us to achieve sublinear server computation. In VC-RAM, a client outsources a large memory array to a server in a preprocessing step. Afterwards in an online stage, the client specifies a sequence of inputs, and asks the server to compute a RAM program over the outsourced memory array and the inputs. Each query made by the client can result in updates to the server's memory array. VC-RAM allows a client to verify the result of these RAM computations, and meanwhile, the server's computation overhead is sublinear (in the data size) for sublinear-time queries.

Although VC-RAM has been informally mentioned in earlier works [1, 3, 7], we make the contribution of explicitly formalizing *stateful* VC-RAM (for repeated queries) and its security. We also present an efficient VC-RAM scheme with constant proof size and prover computation that is comparable to the run-time of the RAM program (as opposed to the dataset size).

Non-generic optimizations for specific schemes. We then apply these techniques to two existing ORAM schemes, the Path ORAM [34] and the Hierarchical ORAM by Goodrich and Mitzenmacher [22] that was later improved by Kushilevitz et al. [26]. The resulting VOS schemes are referred to as Path VOS and Hierarchical VOS respectively.

Applying Verifiable RAM Computation (VC-RAM) straight out-of-the-box is not sufficient to achieve the claimed asymptotic bounds for the Path VOS. We show how to tailor our VC-RAM techniques for the Path VOS to shave a $O(\log n)$ factor off the server computation. Similarly, for the hierarchical VOS, we propose rebalancing techniques that can shave $\text{poly}\log\log n$ to $\log n$ factors from the bandwidth cost, at increased (but still sublinear) server computation.

While the Path VOS is asymptotically better than the hierarchical VOS, the hierarchical VOS is necessary for our dynamic PoR application, since the Path VOS does not satisfy the next-read pattern hiding property [9].

1.2 Related Work

Oblivious RAM (ORAM) was first proposed by Goldreich and Ostrovsky [21], and later improved in a series of works [13,14,17,20,22–24,26,28–31,33–37]. Recently, ORAM has been used in outsourcing storage [22,35,37], and in secure two-party computation to achieve sublinear amortized cost [17,25].

ORAM with implicit server computation has appeared in several works [17,36], while still being referred to as ORAM. Williams and Sion rely on server-side computation to achieve a single-round ORAM scheme [36]. Their scheme ensures privacy against a malicious server but not integrity and freshness and has an asymptotic bandwidth cost of $\tilde{O}(\beta \log^2 n)$. In comparison, our VOS scheme is asymptotically more efficient, and ensures both privacy and integrity/freshness against a malicious server. Gentry et al. [17] proposed using homomorphic encryption to improve ORAM bandwidth cost. However, their scheme is only secure in the semi-honest model, and is also asymptotically more expensive in bandwidth than our construction. Mayberry et al. also proposed to leverage PIR techniques in combination with ORAM [27]. They too are implicitly using VOS; their scheme is not secure in the malicious model, and is asymptotically less efficient than our construction.

Private Information Retrieval (PIR) [6, 11, 12, 19] allows a client to access a dataset on the server obliviously. Single-server PIR techniques can achieve $O(\beta)$ bandwidth cost per query using FHE techniques [6] for large enough block sizes β. However, single-server PIR requires server computation that is linear in the size of the dataset. Also, PIR works for public datasets; in VOS, we consider a private dataset owned by the client, which is not exposed to the server.

Recently, Mayberry et al. propose a scheme combining a binary tree ORAM and a PIR scheme [27]. Their setting is effectively the VOS setting, since the server needs to perform computation. The Mayberry et al. scheme also does not achieve the standard notion of malicious security even though it is claimed in one paragraph in the paper. An obvious attack is that the server can leave out a block during the PIR computation. If the client does not abort, the server learns that this block is not the one intended.

2 Definitions of Verifiable Oblivious Storage

We use $((c\_out, c\_state), (s\_out, s\_state)) \leftarrow \mathsf{protocol}((c\_in,\ c\_state), (s\_in,\ s\_state))$ to denote a (stateful) protocol between a client and server, where *c_in* and *c_out* are the client's input and output; *s_in* and *s_out* are the server's input and output; and *c_state* and *s_state* are the client and server's states before and after the protocol.

We define the notion of *Verifiable Oblivious Storage* (VOS), in which a client outsources the storage of data to a server while ensuring privacy of the data and verifiability and obliviousness of access to that data.

Definition 1 (Verifiable Oblivious Storage)**.** *A Verifiable Oblivious Storage (VOS) scheme consists of the following interactive protocols between a client and a server.*

$((\perp, z), (\perp, Z)) \leftarrow \mathsf{Setup}(1^\lambda, (D, \perp), (\perp, \perp))$: An interactive protocol where the client's input is a memory array $D[1..n]$ where each memory *block* has bit-length β; and the server's input is $\perp$. At the end of the Setup protocol, the client has secret state z, and server's state is Z (which typically encodes the memory array D).

$((\mathsf{data}, z'), (\perp, Z')) \leftarrow \mathsf{Access}((\mathsf{op}, z), (\perp, Z))$: To access data, the client starts in state z, with an input op where $\mathsf{op} := (\mathsf{read}, \mathit{ind})$ or $\mathsf{op} := (\mathsf{write}, \mathit{ind}, \mathsf{data})$; the server starts in state Z, and has no input. In a correct execution of the protocol, the client's output data is the current value of the memory D at location ind (for writes, the output is the old value of $D[\mathit{ind}]$ before the write takes place). The client and server also update their states to z' and Z' respectively. The client outputs $\mathsf{data} := \perp$ if the protocol execution aborted.

We say that a VOS scheme is correct, if for any initial memory $D \in \{0,1\}^{\beta n}$, for any operation sequence $\mathsf{op}_1, \mathsf{op}_2, \ldots, \mathsf{op}_m$ where $m = \mathrm{poly}(\lambda)$, an $\mathsf{op} := (\mathsf{read}, \mathit{ind})$ operation would always return the last value written to the logical location ind (except with negligible probability).

2.1 Security Definition

We adopt a standard simulation-based definition of secure computation [8], requiring that a real-world execution "simulate" an ideal-world (reactive) functionality $\mathcal{F}$. At an intuitive level, our definition captures the privacy and verifiability requirements for an honest client, in the presence of a malicious server.

Ideal world. We define an ideal functionality $\mathcal{F}$ that maintains an up-to-date version of the data D on behalf of the client, and answers the client's access queries.

- *Setup.* An environment $\mathcal{Z}$ gives an initial database D to the client. The client sends D to an ideal functionality $\mathcal{F}$. $\mathcal{F}$ notifies the ideal-world adversary $\mathcal{S}$ (of the setup operation, but not of the data contents D). The ideal-world adversary $\mathcal{S}$ says ok or abort to $\mathcal{F}$. $\mathcal{F}$ then says ok or $\perp$ to the client accordingly.

- *Access.* In each time step, the environment $\mathcal{Z}$ specifies an operation $\mathsf{op} := (\mathsf{read}, \mathit{ind})$ or $\mathsf{op} := (\mathsf{write}, \mathit{ind}, \mathsf{data})$ as the client's input. The client sends op to $\mathcal{F}$. $\mathcal{F}$ notifies the ideal-world adversary $\mathcal{S}$ (without revealing to $\mathcal{S}$ the operation op). If $\mathcal{S}$ says ok to $\mathcal{F}$, $\mathcal{F}$ sends $D[\mathit{ind}]$ to the client, and updates $D[\mathit{ind}] := \mathsf{data}$ accordingly if this is a write operation. The client then forwards $D[\mathit{ind}]$ to the environment $\mathcal{Z}$. If $\mathcal{S}$ says abort to $\mathcal{F}$, $\mathcal{F}$ sends $\perp$ to the client.

Real world. In the real world, an environment $\mathcal{Z}$ gives an honest client a database D. The honest client runs the Setup protocol with the server $\mathcal{A}$. Then at each time step, $\mathcal{Z}$ specifies an input $\mathsf{op} := (\mathsf{read}, ind)$ or $\mathsf{op} := (\mathsf{write}, ind, \mathsf{data})$ to the client. The client then runs the Access protocol with the server. The environment $\mathcal{Z}$ gets the view of the adversary $\mathcal{A}$ after every operation. The client outputs to the environment the data fetched or $\perp$ (indicating abort).

Definition 2 (Simulation-based security: privacy + verifiability)**.** *We say that a protocol* $\Pi_{\mathcal{F}}$ *securely computes the ideal functionality* $\mathcal{F}$ *if for any probabilistic polynomial-time real-world adversary (i.e., server)* $\mathcal{A}$*, there exists an ideal-world adversary* $\mathcal{S}$*, such that for all non-uniform, polynomial-time environment* $\mathcal{Z}$*, there exists a negligible function* negl *such that*

$$\left|\Pr\left[\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda) = 1\right] - \Pr\left[\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda)$$

This definition is simulation-based [8] where the client is honest, and the server is corrupted. (The client is never malicious in our setting.) The definition also simultaneously captures *privacy* and *verifiability.* Intuitively, privacy ensures that the server cannot observe the data contents or the access pattern. Verifiability ensures that the client is guaranteed to read the correct data from the server — if the server happens to cheat, the client can detect it and abort the protocol.

3 ORAM to VOS: Generic Compilation Techniques

In this section, we describe how to generically transform any given ORAM scheme to an efficient VOS scheme. In Section 4, we give two specific VOS schemes - Path VOS and Hierarchical VOS. These are derived from the two classes of ORAM schemes, the hierarchical construction [21] and its variants [22, 24, 26, 30, 35–37], and the binary-tree scheme [31] and its variants [13, 17, 34].

3.1 Preliminary: Oblivious RAM

In this paper, we use a slightly different formalization of ORAM from that of Goldreich-Ostrovsky [21] to make notation simpler for our generic compiler.

An ORAM can be defined by a pair of algorithms $\mathsf{ORAM} := (\mathsf{Init}, \mathsf{Next})$:

- $(D_o, st) \leftarrow \mathsf{Init}(1^\lambda, D)$: Takes in storage array D containing n blocks each of bit length β, produces storage array D_o, and initial ORAM client state st.

- $(\mathsf{out}, \{\mathsf{raddr}\}, \{\mathsf{waddr}\}, \{\mathsf{data}\}, st) \leftarrow \mathsf{Next}(\mathsf{op}, st, \{\mathsf{fetched}\})$: Each ORAM operation $\mathsf{op} := (\mathsf{read}, ind)$ or $\mathsf{op} := (\mathsf{write}, ind, \mathsf{data})$ will proceed in *multiple rounds.* Each round will invoke the $\mathsf{ORAM.Next}$ algorithm with the following inputs: 1) current read/write operation op; 2) the (secret) ORAM client state st; and 3) a set of blocks $\{\mathsf{fetched}\}$ fetched from the last round. If this is the first round for an operation op, this fetched set is empty by convention. The $\mathsf{ORAM.Next}$ function in turn outputs a set of addresses to read in the next round denoted $\{\mathsf{raddr}\}$; a set of addresses $\{\mathsf{waddr}\}$ and data $\{\mathsf{data}\}$ to write in the next round; updates the client state st; and if this is the last round, $\mathsf{ORAM.Next}$ also outputs the block read out.

  The Next algorithm performs one round of the ORAM client computation.

Our notation is explained in the table below:

st	secret ORAM client state	$\{\mathsf{raddr}\}$	physical addr to read from
ind	logical index of a block	$\{\mathsf{waddr}\}$	physical addr to write to
$\mathsf{op} := (\mathsf{read}, ind)$ or $\mathsf{op} := (\mathsf{write}, ind, \mathsf{data})$	a read/write operation	$\{\mathsf{fetched}\}$	data blocks fetched from storage
out	the last logical block read	$\{\mathsf{data}\}$	data blocks to be written

Security is defined in terms of the inability of any PPT adversary to distinguish the access patterns generated by an honest execution of the ORAM client, from those output by a simulator that does not see the sequence of logical operations.

Definition 3 (ORAM security)**.** *We say that an ORAM scheme is secure, if there exists a stateful simulator* Sim*, such that for any PPT adversary* $\mathcal{A}$*,*

$$\left|\Pr\left[\mathcal{A}^{O[st](\cdot)}(1^\lambda) = 1\right] - \Pr\left[\mathcal{A}^{\mathsf{Sim}(1^\lambda, m)}(1^\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda) \tag{1}$$

where m is the number of oracle queries made by the adversary $\mathcal{A}$; and $O[st](\cdot)$ denotes a stateful oracle O, with secret state st. The oracle O takes in an operation op *and outputs a sequence of read and write physical addresses. Formally,*

Oracle O:

Initialization. On input D containing n blocks each of β bits, initialize a storage array D_o containing n_o blocks each of size β_o. Initialize the set $\{\mathsf{fetched}\}$ to be an empty set. Run $st := \mathsf{ORAM.Init}(1^\lambda, n, \beta)$.

Data access. On the j-th input op_j, $j \in \mathbb{N}$, perform the following:

- First, initialize the output array $\Gamma := \emptyset$.
- For $\mathsf{rnd} = 1$ to R_j where R_j is the total number of rounds for the j-th operation[a]:
    - Run $(\mathsf{out}, \{\mathsf{raddr}\}, \{\mathsf{waddr}\}, \{\mathsf{data}\}, st) \leftarrow \mathsf{ORAM.Next}(\mathsf{op}, st, \{\mathsf{fetched}\})$
    - Let $D_o[\{\mathsf{waddr}\}] := \{\mathsf{data}\}$, and let $\{\mathsf{fetched}\} := D_o[\{\mathsf{raddr}\}]$.
    - Append $\{\mathsf{raddr}\}$ and $\{\mathsf{waddr}\}$ to the output set Γ.
- Finally, output Γ.

[a] In all known ORAM constructions, due to the obliviousness requirement, R_j is a public value determined by the ORAM scheme description itself, and does not depend on the input sequence.

We use $D_o[\{\mathsf{waddr}\}] := \{\mathsf{data}\}$ and $\{\mathsf{fetched}\} := D_o[\{\mathsf{raddr}\}]$ to denote writing $\{\mathsf{data}\}$ to a set of write addresses $\{\mathsf{waddr}\}$, and reading from a set of read addresses $\{\mathsf{raddr}\}$ respectively. We assume that $\{\mathsf{waddr}\}$ and $\{\mathsf{data}\}$ are ordered sets, and we simply write each block data into each waddr in the specified order.

Deterministic vs. randomized ORAM. In general, the ORAM client algorithms Init and Next can be randomized. However, the Next algorithm can be made deterministic by choosing a PRF key k at random and including it in the client state st. Whenever Next requires random bits, this can be generated pseudorandomly from key k. If the randomized ORAM is secure, then the resulting ORAM with a deterministic Next algorithm is also secure due to the security of the PRF. Therefore, without loss of generality, in our generic ORAM-to-VOS compiler, we will assume an ORAM scheme with a deterministic Next algorithm.

- Setup: Client runs $(pk, sk) \leftarrow \mathsf{FHE.KeyGen}(1^\lambda)$. Client runs $(D_o, st) \leftarrow \mathsf{ORAM.Init}(1^\lambda, D)$.

  For $i = 1$ to $|D_o|$, the client computes $\overline{D_o}[i] := \mathsf{FHE.Enc}_{pk}(D_o[i])$. The client also computes $\overline{st} := \mathsf{FHE.Enc}_{pk}(st)$. Finally, the client sends $\big(pk, \{\overline{D_o}[i]\}_{i \in |D_o|}, \overline{st}\}\big)$ to the server.

- Access: For the j-th operation op, let R_j denote the number of ORAM rounds necessary for the j-th operation.

  First, the client encrypts $\overline{\mathsf{op}} := \mathsf{FHE}_{pk}(\mathsf{op})$ and sends it to the server.

  For $\mathsf{rnd} = 1$ to R_j:

  - If this is not the first round, i.e., if $\mathsf{rnd} \neq 1$, the server performs memory reads and writes: $\overline{D_o}[\{\mathsf{waddr}\}] := \{\overline{\mathsf{data}}\}$, and $\{\overline{\mathsf{fetched}}\} := \overline{D_o}[\{\mathsf{raddr}\}]$, where $\{\mathsf{raddr}\}$ and $\{\mathsf{waddr}\}$ are the read and write addresses returned by the client in the previous round, and data is the part of the FHE evaluation outcome in the previous round.
  - The server homomorphically evaluates the $\mathsf{ORAM.Next}$ circuit once[a]: $(\overline{\mathsf{out}}, \{\overline{\mathsf{raddr}}\}, \{\overline{\mathsf{waddr}}\}, \{\overline{\mathsf{data}}\}, \overline{st}) \leftarrow \mathsf{FHE.Eval}(\mathsf{ORAM.Next}(\overline{\mathsf{op}}, \overline{st}, \{\overline{\mathsf{fetched}}\}))$
  - Server sends client $\{\overline{\mathsf{raddr}}\}, \{\overline{\mathsf{waddr}}\}$. The client decrypts them using sk, and sends the clear-text $\{\mathsf{raddr}\}, \{\mathsf{waddr}\}$ to the server.

  Finally, server sends $\overline{\mathsf{out}}$ to the client, and the client decrypts it.

[a] The first round of the first operation does not depend on $\{\overline{\mathsf{fetched}}\}$. Therefore $\{\overline{\mathsf{fetched}}\}$ need not be provided as an input.

Figure 1: ORAM-to-VOS generic compiler: semi-honest model.

3.2 Compilation in the Semi-Honest Model

Intuition. The intuition is to have the client outsource the ORAM memory encrypted under an FHE scheme to the server. The client can then outsource all its computation to the server as well, since the server can homomorphically operate over the encrypted data. In this manner, the server only contacts the client whenever it is necessary for interaction during the computation.

ORAM-to-VOS compiler in the semi-honest model. Figure 1 describes how to transform an ORAM scheme to a VOS scheme that is secure under a semi-honest server.

Theorem 1. *Let* $\mathsf{FHE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ *be a semantically secure FHE scheme and let* $\mathsf{ORAM} = (\mathsf{Init}, \mathsf{Next})$ *be a secure ORAM scheme. Then, the generic compiler in Figure 1 gives a Verifiable Oblivious Storage (VOS) construction secure under a semi-honest server.*

The proof of Theorem 1 reduces to the security of the encryption scheme and the ORAM scheme in a straightforward manner. We refer the reader to Appendix C for the proof.

Optimization: Handling addresses independent of secret information. In the construction above, the server performs as much computation as possible and only seeks the client's help when it needs to decrypt the next set of physical addresses to read from or write to. In many ORAM schemes, there are read/write operations whose physical addresses do not depend on secret client state, memory contents, or the logical addresses accessed. Examples are the reshuffling operations of the hierarchical ORAM scheme [21] and its variants [22–24, 26, 35, 37] and the eviction operations of the binary-tree based ORAM [31] and its variants [34]. To achieve better efficiency, such reshuffling and eviction operations, can be performed by the server (on its own) homomorphically, without seeking the client's help to decrypt the physical addresses.

3.3 Handling Malicious Servers

One way to handle a malicious server is to rely on a Succinct Non-Interactive Argument of Knowledge (SNARK). However, if done naively, the circuit for the SNARK will have size that is at least linear in n, i .e., the size of the outsourced memory D. This requires the server to perform a linear amount of computation to produce a proof of correctness.

Instead, we rely on efficient Verifiable RAM Computation (VC-RAM) to enforce honest server behavior. Verifiable RAM computation has been informally introduced in the literature by Ben-Sasson et al. [1] and Bitansky et al. [3]. We, however, need a stateful version of verifiable RAM computation. Braun et al. also informally proposed and implemented verifiable RAM computation [7].

We define a *stateful* version of verifiable RAM computation, where each query can result in updates to the outsourced dataset. Below, we explicitly formalize this notion of stateful, multi-query VC-RAM. Relying on the same ideas as Ben-Sasson et al. [1] and Bitansky et al. [3], we show that verifying RAM computation can be done efficiently, resulting in server computation that is comparable to the run-time of the RAM program (as opposed to the size of the memory); succinct proofs of size $O(\lambda)$; and efficient client verification time that is not too much worse than simply reading the input and output.

3.3.1 Verifiable RAM Computation

Consider a scenario where a client outsources a memory array D to a server. Let f denote a RAM program agreed upon by the client and the server. At each time step t, the client supplies a small input x_i, and the server computes the RAM program f over x_i and the current state of memory D. The RAM program produces an answer which is sent to the client. It may also update the memory contents outsourced to the server – hence our notion of VC-RAM is stateful. Verifiability requires that the client be able to check that the RAM computation results returned by the server are correct.

Definition 4 (Verifiable RAM Computation)**.** *A (non-interactive) Verifiable RAM Computation (VC-RAM) scheme consists of the following algorithms:*

$(z, Z) \leftarrow \mathsf{Setup}(1^\lambda, D, f)$: Given an initial memory array $D[1..n]$ where each memory word has bit-length ℓ, a RAM program description f, output initial server state Z (which typically encodes D), and the initial client state z.

$(\overline{y}, Z') \leftarrow \mathsf{Compute}(x, Z)$: Given a small input x to the RAM program f, the server's current state Z, output an encoded answer $\overline{y}$, and updated server state Z'. [1]

[1] In the specific VC-RAM construction we describe, the encoded answer $\overline{y}$ includes the answer itself y, a proof vouching for its correctness, and an updated digest of the outsourced memory.

$(y, b, z') \leftarrow \mathsf{Verify}(x, \overline{y}, z)$: Given the input x, the client's current state z, an encoded answer $\overline{y}$, output a decoded answer y, a bit b indicating whether to accept this answer, and updated client state z'.

Correctness is defined as usual. We require that for any parameters n and ℓ, for any initial memory array $D \in \{0,1\}^{\ell n}$, for any polynomial-sized RAM program f which terminates in polynomial time, for any query sequence $x_1, x_2, \ldots, x_m$ where $m = \text{poly}(\lambda)$,

$$\Pr\left[\exists i : \begin{array}{l} (y_i \neq f(D, x_1, x_2, \ldots, x_i)) \\ \vee (b_i = 0) \end{array} \;\middle|\; \begin{array}{l} (z, Z_0) \leftarrow \mathsf{Setup}(1^\lambda, D, f) \\ \forall i \in \{1, 2, \ldots, m\} : \\ \qquad (\overline{y}_i, Z_i) \leftarrow \mathsf{Compute}(x_i, Z_{i-1}) \\ \qquad (y_i, b_i, z) \leftarrow \mathsf{Verify}(x_i, \overline{y}_i, z) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

In particular, we use the notation $y_i := f(D, x_1, x_2, \ldots, x_i)$ to denote the outcome of the i-th query, starting with an initial memory array of D, and after computing the RAM program f on queries $x_1, x_2, \ldots, x_i$. Note that each query is stateful, i.e., may result in updates to the memory array D.

Definition 5 (Verifiability of VC-RAM). *We say that a VC-RAM scheme is verifiable, if for any polynomial time (stateful) adversary $\mathcal{A}$ the following holds.*

$$\Pr\left[\begin{array}{l} \exists i : (b_i = 1) \wedge \\ (y_i \neq f(D, x_1, x_2, \ldots, x_i)) \end{array} \;\middle|\; \begin{array}{l} (D, f) \leftarrow \mathcal{A}(1^\lambda) \\ (z, Z) \leftarrow \mathsf{Setup}(1^\lambda, D, f) \\ (x_1, \overline{y}_1) \leftarrow \mathcal{A}(Z) \\ \forall i \in \{1, 2, \ldots, m\} : \\ \qquad (y_i, b_i, z) \leftarrow \mathsf{Verify}(x_i, \overline{y}_i, z) \\ \qquad (x_{i+1}, \overline{y}_{i+1}) \leftarrow \mathcal{A}(y_i, b_i) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Note again that the adversary $\mathcal{A}$ is stateful, and we do not write its state explicitly for simplicity.

Theorem 2. *There exists a non-interactive VC-RAM scheme such that for each query: the server runs in time $\tilde{O}(\tau \log n)\text{poly}(\lambda)$ where τ is the run-time of the RAM program in the unauthenticated setting; the verifier runs in time $O((|x| + |y|)\lambda)$; and the client-server bandwidth cost is $|x| + |y| + O(\lambda)$.*

Note that the client-server bandwidth cost has to be at least $|x| + |y|$, i.e., the number of bits necessary to transmit the query x and the answer y. Therefore, the only additional cost is $O(\lambda)$ for transmitting an updated digest of the outsourced memory and a proof vouching for the correctness of the result.

We explain the intuition for the VC-RAM construction. The full construction can be found in Appendix A. The high level idea is to build a Merkle tree over all outsourced memory, such that the client keeps the up-to-date root digest. To verify a RAM computation, we build a "verifier circuit" which takes in a trace of the computation, including 1) the CPU states before and after every computation step; 2) the memory contents fetched in every computation step; and 3) the Merkle-tree digest before and after each computation step. This verifier circuit checks the trace of the computation: 1) it checks that every memory read and write is correct using memory checking; and 2) it checks that every CPU computation step is correct. The server then constructs a SNARK for this "verifier circuit". Since this verifier circuit has size that is roughly the time of the RAM computation, we can achieve prover efficiency, i.e., the prover time is roughly the time of the RAM computation rather than the size of the entire dataset.

3.3.2 Relying on VC-RAM to Enforce Honest Server Behavior

In our semi-honest VOS construction described in Section 3.2, the client essentially outsources all of its ORAM memory (encrypted under FHE) to the server, as well as the ORAM's secret client state (also encrypted under FHE).

During each data access operation, in every round of interaction, the server performs some RAM computation on behalf of the client, and sends a message to the client to seek its help decrypting certain physical addresses. Using VC-RAM, the server can attach a succinct proof along with every message sent to the client, vouching for the correctness of the message. If the message sent to the client deviates from correct message, the client will surely detect it (except with negligible probability).

We state the theorem below, and give the formal presentation of the malicious-model ORAM-to-VOS compiler in Appendix B.

Theorem 3. *Assuming existence of SNARKs, collision resistant hash functions, and a semantically secure FHE scheme, the VOS construction in Appendix B is secure against a malicious server.*

Proof. (sketch.) Due to the proof of the semi-honest model compiler (Theorem 1), it suffices to show that a malicious server cannot deviate from the protocol without being detected — this is ensured by the security of the VC-RAM scheme. □

4 Optimizations for Specific ORAM Schemes

4.1 Background on Path ORAM

Stefanov et al. recently proposed the Path ORAM [34]. They formally prove that to achieve $n^{-\alpha(n)}$ failure probability, the (recursive) Path ORAM construction achieves $O(\alpha(n)\beta \log^2 n/\log\chi)$ client-side storage, and $O(\beta \log^2 n/\log\chi)$ bandwidth cost — χ is a term related to the block size where the block size $\beta = \chi \log n$ bits. Specifically, to make the failure probability negligible, we can use any $\alpha(n) := \omega(1)$.

Of particular interest is the case when the block size is $\Omega(\lambda)$ — in practical storage outsourcing applications, this is typically the case. Since $n = \text{poly}(\lambda)$, the number of recursions would be $O(1)$.

Lemma 1 (Path ORAM [34]). *For reasonably large block sizes $\beta = \Omega(\lambda)$, Path ORAM achieves bandwidth cost of $O(\beta \log n)$, a client-side storage of*
$O(\alpha(n)\beta \log n)$, and $O(1)$ rounds, with a failure probability of $n^{-\alpha(n)}$. Specifically, to achieve negligible failure probability, it suffices to use any $\alpha(n) := \omega(1)$.

We briefly introduce the Path ORAM algorithm below.

Server data layout. The blocks on the server are organized into a binary tree. of height roughly $\log n$. Each node in the tree is a bucket of $O(1)$ capacity. We use the notation $\mathcal{P}(x)$ to denote the path from the leaf node x to the root node, containing all buckets on the path. Additionally, $\mathcal{P}(x,\ell)$ denotes the bucket in $\mathcal{P}(x)$ at level ℓ in the tree.

Client data layout. The client holds a position map where $\mathsf{pos}[ind]$ records the up-to-date designated leaf node for block ind. A block ind's designated leaf node is x implies that the block resides somewhere along the path $\mathcal{P}(x)$.

```
Access(op):

Let $\mathsf{op} := (\mathsf{read}, ind)$ or $\mathsf{op} := (\mathsf{write}, ind, \mathsf{data})$ denote the current operation.

  1. Set $x := \mathsf{pos}[ind]$. Pick a fresh new leaf $x_n$. Store $\mathsf{pos}[ind] = x_n$.

  2. Request all blocks in the path $\mathcal{P}(x)$ from the server.

  3. Set $\mathsf{stash} := \mathsf{stash} \cup \mathcal{P}(x)$.

  4. Let $\mathsf{data}^*$ be the current block in $\mathsf{stash}$ with index $ind$.
     If $\mathsf{op}$ is a $\mathsf{write}$ operation, set $\mathsf{stash} := (\mathsf{stash} - \{(ind, x, \mathsf{data}^*)\}) \cup \{(ind, x_n, \mathsf{data})\}$. Else let
     $\mathsf{stash} := (\mathsf{stash} - \{(ind, x, \mathsf{data})\}) \cup \{(ind, x_n, \mathsf{data})\}$.

  5. For $\ell = L$ to $0$ (where $L$ is the leaf level, and $0$ is the root), do:
     Let $S$ be the set of all $\{(ind', x', \mathsf{data}')\} \in \mathsf{stash}$ such that $\mathcal{P}(x, \ell) = \mathcal{P}(x', \ell)$.
     $S :=$ Select $\min(|S|, \mathsf{bucketsize})$ blocks from $S$.
     Set $\mathsf{stash} := \mathsf{stash} - S$.
     If $|S| < \mathsf{bucketsize}$, pad $S$ with dummy blocks to $\mathsf{bucketsize}$.
     Client sends $S$ to the server to write in bucket $\mathcal{P}(x, \ell)$.

The output to the client is $\mathsf{data}^*$, plus the updated position map $\mathsf{pos}$.
```

Figure 2: Access protocol for Path ORAM (non-recursive).

The client also holds a small stash of size $O(\alpha(n) \log n)$ for overflowing blocks, where any $\alpha(n) := \omega(1)$ allows us to achieve negligible failure probability.

Data access. To perform any data access operation op, where $\mathsf{op} := (\mathsf{read}, ind)$ or $\mathsf{op} := (\mathsf{write}, ind, \mathsf{data})$, the client runs the Access protocol described in Figure 2. At Step 1, the block being read or written to is randomly remapped to a new leaf. At Step 2, the client requests a path of data blocks from the server. At Step 3, the local is merged with the data received from the server. At Step 4, the read/write operation is performed. At Step 5, the stash is written back into the tree, greedily pushing data blocks as close to the leaves as possible.

Recursive Path ORAM. The Path ORAM construction above requires the client to store a position map of $O(n \log n)$ bits. However, the client can store the position map on the server in a smaller ORAM. This is called the recursive Path ORAM. Particularly, if the block size $\beta := \Omega(\lambda)$, and $n = \mathrm{poly}(\lambda)$, then the depth of the recursion is constant.

4.2 Path VOS

We can use the generic compilation techniques described in Section 3 to compile Path ORAM to a VOS scheme — henceforth referred to as the Path VOS algorithm.

The semi-honest version of the Path VOS protocol is described in Figure 3. We can use the VC-RAM techniques described in Section 3.3.1 to compile the semi-honest protocol to one that is secure against a malicious server.

Recursive Path VOS. In the above (non-recursive) Path VOS protocol, the client needs to store

Path VOS (non-recursive, semi-honest model)

Setup. Given a memory array D, client lays out D into an initial ORAM-tree as in the Path ORAM algorithm, and creates an initial position map accordingly. The client encrypts the initial ORAM-tree under FHE, and an empty stash, and outsources both the FHE-encrypted ORAM-tree and stash to the server. The client keeps the position map locally.

Access. Let $\mathsf{op} := (\mathsf{read}, ind)$ or $\mathsf{op} := (\mathsf{write}, ind, \mathsf{data})$ denote the current operation.

- *Client*: Looks up its local position map $x := \mathsf{pos}[ind]$. Pick a fresh random new leaf x'. Compute $\overline{\mathsf{op}} := \mathsf{FHE.Enc}(\mathsf{op})$, $\overline{x}' := \mathsf{FHE.Enc}(x')$. Send $(\overline{\mathsf{op}}, x, \overline{x}')$ to the server.

- *Server*: Let $\mathsf{WritePath}(\mathcal{P}, \mathsf{stash}, \mathsf{op}, x')$ denote the circuit (Steps 3 to 5 in Figure 2) that on inputting a path $\mathcal{P}$, a stash stash, and the current operation op, returns the current value of the requested block ind, overwrites the block ind's designated leaf tag to x', overwrites the block ind with new data if this is a write operation, and write back blocks in $\mathcal{P} \cup \mathsf{stash}$ to the path $\mathcal{P}$, greedily packing them as close to the leaf as possible. The server homomorphically computes $(\overline{\mathsf{out}}, \overline{\mathcal{P}(x)}, \overline{\mathsf{stash}}) \leftarrow \mathsf{FHE.Eval}(\mathsf{WritePath}(\overline{\mathcal{P}(x)}, \overline{\mathsf{stash}}, \overline{\mathsf{op}}, \overline{x}'))$. The server sends to the client the FHE-encrypted result of the read $\overline{\mathsf{out}}$.

Figure 3: Path VOS (non-recursive, semi-honest model).

a position map of size $O(n \log n)$ bits. This client-side storage may be avoided by recursively outsourcing the position map to the server in a smaller VOS scheme. When the block size is $\beta = \Omega(\lambda)$, the depth of recursion is $O(1)$. The resulting recursive Path VOS scheme will therefore have $O(1)$ roundtrips for each data access.

Tailored VC-RAM techniques for Path ORAM. Based on the semi-honest protocol described above, and the VC-RAM techniques described in Section 3.3.1, we immediately obtain a Path VOS protocol with $\tilde{O}(\beta \log^2 n)\text{poly}(\lambda)$ server computation per data access[2], for block sizes $\beta = \tilde{\Omega}(\lambda)$. (The small increase in block size is due to FHE.)

We observe that by overlaying the Path ORAM tree structure on top of the Merkle tree, we can shave a logarithmic factor off the server computation. Recall that in our VC-RAM construction, the client maintains a Merkle-hash tree digest of the ORAM-memory outsourced to the server. To prove that any RAM computation is correct, the server computes a SNARK for a "verifier circuit" which verifies 1) that every memory access is correct (through the Merkle tree); and 2) every step of CPU computation is correct. In particular, the extra $\log n$ factor comes from the cost.

In the case of Path ORAM, since Path ORAM itself is a tree structure, we can overlay the Merkle tree on top of Path ORAM's tree structure. In this way, when Path ORAM accesses a path from the root to a leaf, the underlying memory checking scheme can vouch for the correctness of the entire path with $O(\log n)$ hashes. This can allow us to shave a $O(\log n)$ factor off the server computation for Path VOS.

Theorem 4 (Path VOS)**.** *Assume collision resistant hash functions, the ring LWE assumption with suitable parametrization [5, 18], and the q-PDH and q-PKE assumptions [16]. Let $\alpha(n)$ denote any function such that $\alpha(n) := \omega(1)$.*

[2] Throughout this paper, the notation $\tilde{O}(f(n))$ hides $\log(f(n))$ factors.

There exists a secure VOS scheme for reasonably large block size $\beta = \tilde{\Omega}(\lambda)$, with $O(\beta)$ bandwidth cost, $\tilde{O}(\alpha(n) \cdot \beta \log n)\mathrm{poly}(\lambda)$ server computation per data access, $O(\beta n)$ server-side storage, $O(\beta)$ client-side storage, and $O(\beta + \lambda^2)$ client computation per data access. Furthermore, the failure probability is $n^{-\alpha(n)}$, i.e., negligible in n for any $\alpha(n) := \omega(1)$.

Proof of security follows in a similar manner as the security proof for the generic compiler in the malicious model (Theorem 3).

4.3 The Hierarchical VOS

We propose a hierarchical VOS construction based on the Goodrich-Mitzenmacher ORAM (GM-ORAM) scheme [22] and its variants [26]. Although this hierarchical VOS construction achieves worse asymptotics than the Path VOS mentioned in the previous section, it is necessary later for our dynamic proofs of retrievability scheme — since the Path VOS scheme does not satisfy the next-read pattern hiding property (NRPH) proposed by Cash et al. [9]. (All of our VOS compilers are NRPH-preserving since they do not alter the sequence of accesses as dictated by the underlying ORAM.)

Although the basic idea is similar as before, to use FHE to outsource computation to the server, and use SNARK to enforce honest server behavior, we propose a "read/write (un)balancing" trick that allows us to reduce the bandwidth cost. The idea is that if we apply the generic ORAM-to-VOS compiler on the GM-ORAM scheme, reads will require more bandwidth than writes, since write is basically a homomorphic shuffling operation which the server can perform all on its own without interacting with the client. Therefore, we adjust the scheme to penalize writes while reducing the cost of reads. Note that in the traditional ORAM setting, writes cost more bandwidth, and that is why Kushilevitz et al. [26] propose a read/write balancing trick where they penalize reads to save on writes — our trick is the opposite of theirs since the read/write cost comparison is reverse in the VOS setting. We only give our main theorem for the Hierarchical VOS below. The detailed construction can be found in Appendix D.

Theorem 5. *Let $g(n)$ denote some function on n. Assume collision resistant hash functions, the ring LWE assumption with suitable parametrization [5, 18], and the q-PDH and q-PKE assumptions [16]. Then, there exists a VOS scheme for a reasonably large block size $\beta = \tilde{\Omega}(\lambda)$, with $O(\beta \log n / \log g(n))$ bandwidth cost, and $\tilde{O}(\beta g(n) \log^3 n / \log g(n))\mathrm{poly}(\lambda)$ server computation (per data access), where n is the total number of blocks and λ is the security parameter.*

The following table shows some interesting special cases of Theorem 8.

$g(n)$	server computation	bandwidth overhead
n^ϵ for constant $\epsilon < 1$	$\tilde{O}(\beta n^\epsilon \log^2 n)\mathrm{poly}(\lambda)$	$O(\beta)$
$\log n$	$\tilde{O}(\beta \log^4 n / \log \log n)\mathrm{poly}(\lambda)$	$O(\beta \log n / \log \log n)$
constant $c > 1$	$\tilde{O}(\beta \log^3 n)\mathrm{poly}(\lambda)$	$O(\beta \log n)$

5 Applications: Efficient Dynamic Proofs of Retrievability

For applications such as Dynamic Proofs of Retrievability, and RAM-model secure multi-party computation where the party storing the data (or a share of the data) can perform computation, often, just directly replacing the ORAM scheme with a VOS scheme can reduce the asymptotic communication overhead.

We show how VOS can be useful in Dynamic Proofs of Retrievability, based on the results of Cash et al. [9]. We note that two recent results have yielded more practical dynamic PoR schemes [10, 32]. Our dynamic PoR description helps demonstrate why distinguishing between VOS and ORAM can aid theoretical understanding. For a practical implementation, the recent schemes by Shi et al. [32] and Chandran et al. [10] are recommended.

Recently Cash et al. [9] show how to leverage a blackbox ORAM scheme to construct a dynamic proof of retrievability (PoR) scheme with $O(\beta\lambda \log^2 n)$ cost (both in terms of bandwidth and server computation) per data access. They require the underlying ORAM to have a special property which they call *"next-read pattern hiding"* (NRPH).

In the dynamic PoR scheme by Cash et al., they assume a passive server which does not perform any active computation. We observe that if we replaced the ORAM scheme in their construction with a VOS scheme (which also needs to satisfy the NRPH property), we would be able to obtain a dynamic PoR scheme (with server computation), which achieves smaller asymptotic bandwidth cost than Cash et al. [9].

The Path ORAM algorithm (and hence Path VOS too), however, does not satisfy the NRPH property, as pointed out by Cash et al. [9]. However, they showed that the GM-ORAM scheme and its variants indeed satisfy the NRPH property. Therefore, we rely on the hierarchical VOS described in Section 4.3 to build our dynamic PoR scheme.

Theorem 6. *Let $g(n)$ denote some function on n. Assume collision resistant hash functions, the ring LWE assumption with suitable parametrization [5, 18], and the q-PDH and q-PKE assumptions [16]. Then, there exists a dynamic proof of retrievability scheme for reasonably large block size $\beta = \tilde{\Omega}(\lambda)$, with $O(\beta \log n / \log g(n))$ bandwidth cost and $O(\beta g(n) \log^3 n / \log g(n))\mathrm{poly}(\lambda)$ server computation for each read operation; $O(\beta\lambda \log n / \log g(n))$ bandwidth cost and $O(\beta\lambda g(n) \log^3 n / \log g(n))\mathrm{poly}(\lambda)$ server computation for each write or audit operation; with $O(\beta)$ client-storage and $O(\beta n)$ server storage. In the above, n is the total number of blocks and λ is the security parameter.*

Below are some interesting special cases of the above theorem. "R:" stands for read cost, and "W/A:" stands for write/audit cost.

$g(n)$	server computation	bandwidth overhead
n^ϵ for constant $\epsilon < 1$	R: $\tilde{O}(\beta n^\epsilon \log^2 n)\mathrm{poly}(\lambda)$ W/A: $\tilde{O}(\beta\lambda n^\epsilon \log^2 n)\mathrm{poly}(\lambda)$	R: $O(\beta)$ W/A: $O(\beta\lambda)$
$\log n$	R: $\tilde{O}(\beta \log^4 n / \log\log n)\mathrm{poly}(\lambda)$ W/A: $\tilde{O}(\beta\lambda \log^4 n / \log\log n)\mathrm{poly}(\lambda)$	R: $O(\beta \log n / \log\log n)$ W/A: $O(\beta\lambda \log n / \log\log n)$
constant $c > 1$	R: $\tilde{O}(\beta \log^3 n)\mathrm{poly}(\lambda)$ W/A: $\tilde{O}(\beta\lambda \log^3 n)\mathrm{poly}(\lambda)$	R: $O(\beta \log n)$ W/A: $O(\beta\lambda \log n)$

In comparison to Cash et al. [9], using Verifiable Oblivious Storage (VOS), we can reduce the bandwidth cost to $O(\beta \log n / \mathrm{poly} \log\log n)$ for reads, and
$O(\beta\lambda \log n / \mathrm{poly} \log\log n)$ for writes, with poly-logarithmic server computation. Furthermore, we can reduce the bandwidth cost to $O(\beta)$ for reads, and $O(\beta\lambda)$ for writes, with $O(\beta n^\epsilon)\mathrm{poly}(\lambda)$ amount of server computation for a constant $\epsilon < 1$.

Other applications. Gordon et al. recently proposed to use ORAM to achieve amortized sublinear-time secure two-party computation [25]. In their setting, Alice's input is a large database, and Bob repeatedly makes queries over the database. Alice wishes to protect the privacy of her database, while Bob wishes to protect the privacy of his query. Using ORAM, Gordon et al. show that the cost of securely querying the database can be sublinear when amortizing the ORAM setup cost over all future queries. Since both parties (each storing a share of the data) perform computation in this setting, we can simply replace the ORAM with VOS, and asymptotically, this gives savings in terms of bandwidth overhead.

6 Conclusion and Open Problems

This paper separates VOS from ORAM, and shows that VOS need not be subject to ORAM's lower bounds, since it is a different model where server computation is allowed. The constructions proposed in this paper use general primitives such as FHE and SNARKs. An interesting open question is to see how to construct a practically efficient VOS scheme (potentially without FHE or SNARKs) that outperforms the best known ORAM in terms of bandwidth overhead. It would also be interesting to consider how to construct VOS schemes that are asymptotically more bandwidth efficient than ORAM from weaker assumptions, e.g., without SNARKs or non-falsifiable assumptions.

Acknowledgments

This research was funded by NSF under grant number CNS-1314857, by a Google Faculty Research Award, and by the US Army Research Laboratory and the UK Ministry of Defence under Agreement Number W911NF-06-3-0001. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, the U.S. Government, the UK Ministry of Defense, or the UK Government. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

We thank Hubert Chan, Charalampos Papamanthou, Emil Stefanov, and Hong-Sheng Zhou for helpful discussions, and the anonymous reviewers for their insightful comments.

References

[1] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. Fast reductions from rams to delegatable succinct constraint satisfaction problems: extended abstract. In *ITCS*, 2013.

[2] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, 2012.

[3] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *STOC*, 2013.

[4] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *Algorithmica*, 12: 225–244, 1994.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS*, 2012.

[6] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *FOCS*, 2011.

[7] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, 2013.

[8] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[9] D. Cash, A. Küpçü, and D. Wichs. Dynamic proofs of retrievability via oblivious ram. In *Eurocrypt*, 2013.

[10] N. Chandran, B. Kanukurthi, and R. Ostrovsky. Locally updatable and locally decodable codes. In *TCC*, 2014.

[11] B. Chor and N. Gilboa. Computationally private information retrieval (extended abstract). In *STOC*, 1997.

[12] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 41–50, 1995.

[13] K.-M. Chung and R. Pass. A simple oram. `https://eprint.iacr.org/2013/243.pdf`, 2013.

[14] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *IACR Theory of Cryptography Conference (TCC)*, pages 144–163, 2011.

[15] S. Garg, C. Gentry, S. Halevi, and M. Raykova. Two-round secure MPC from indistinguishability obfuscation. In *IACR Theory of Cryptography Conference (TCC)*, pages 74–94, 2014.

[16] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Eurocrypt*, 2013.

[17] C. Gentry, K. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs. Optimizing oram and using it efficiently for secure computation. In *PETS*, 2013.

[18] C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, pages 465–482, 2012.

[19] C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *ICALP*, 2005.

[20] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[21] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[22] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[23] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *CCSW*, 2011.

[24] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.

[25] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *ACM CCS*, 2012.

[26] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[27] T. Mayberry, E.-O. Blass, and A. Chan. Efficient private file retrieval by combining oram and pir. `http://eprint.iacr.org/2013/086`, 2013.

[28] R. Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, 1990.

[29] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, 1997.

[30] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.

[31] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[32] E. Shi, E. Stefanov, and C. Papamanthou. Practical dynamic proofs of retrievability. In *ACM CCS*, 2013.

[33] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[34] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *ACM CCS*, 2013.

[35] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.

[36] P. Williams and R. Sion. Single round access privacy on outsourced storage. In *CCS*, 2012.

[37] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM CCS*, 2008.

Appendices

A VC-RAM Construction

Earlier works by Ben-Sasson et al. [1] and by Bitansky et al. [3] implicitly give rise to an efficient VC-RAM construction. For the purpose of this paper, we formally recast the VC-RAM definition in Section 3.3. In this section, sketch an efficient VC-RAM construction based on the ideas proposed by Ben-Sasson et al. [1] and Bitansky et al. [3].

The VC-RAM construction described below is based on any publicly-verifiable memory-checking scheme and SNARK. The construction is publicly verifiable if the underlying SNARK is publicly verifiable; else it is secretly verifiable.

A.1 Preliminiaries

Memory checking. We recast memory checking [4] in a form better suited for our purposes, however it is clear that our definition is equivalent to that used in prior work. If D is a memory array, then the "instruction" $I = (\mathsf{data}, \mathsf{waddr}, \mathsf{raddr})$ sets $D[\mathsf{waddr}] = \mathsf{data}$ and returns $\mathsf{fetched} = D[\mathsf{raddr}]$. If a sequence of instructions $I_1 = (\mathsf{data}_1,\ \mathsf{waddr}_1,\ \mathsf{raddr}_1),\ \ldots,\ I_m = (\mathsf{data}_m,\ \mathsf{waddr}_m, \mathsf{raddr}_m)$ is executed, the correct answer to the final instruction (i.e., the final value $\mathsf{fetched}_m$) is defined in the obvious way: if $\mathsf{raddr}_m \notin \{\mathsf{waddr}_1, \ldots, \mathsf{waddr}_{m-1}\}$ then $\mathsf{fetched}_m = D[\mathsf{raddr}_m]$, i.e., the contents of the original memory at location raddr_m. Otherwise, let $t < m$ be maximal such that $\mathsf{raddr}_m = \mathsf{waddr}_t$; then $\mathsf{fetched}_m = \mathsf{data}_t$.

Definition 6. *A* memory-checking scheme *consists of algorithms* (Setup, Prove, Vrfy) *such that:*

- Setup *takes as input a security parameter* 1^λ *and an array* D*, and outputs a transformed array* $\tilde{D}$ *along with a digest* d*.*
- Prove *takes as input* $\tilde{D}$ *and an instruction* $I = (\mathsf{data}, \mathsf{waddr}, \mathsf{raddr})$*. It outputs an updated array* $\tilde{D}$*, a value* fetched*, and a proof* π*.*
- Vrfy *takes as input a digest* d*, an instruction* $I = (\mathsf{data}, \mathsf{waddr}, \mathsf{raddr})$*, a value* fetched*, and a proof* π*. It outputs a bit* b *and an updated digest* d*.*

The correctness requirement is that for any initial array D, and any (adaptively chosen) sequence of instructions $I_1, \ldots, I_m$, if we run

$$(\tilde{D}_0, d_0) \leftarrow \mathsf{Setup}(1^\lambda, D);\quad \forall\, i : (\tilde{D}_i, \mathsf{fetched}_i, \pi_i) \leftarrow \mathsf{Prove}(\tilde{D}_{i-1}, I_i); (b_i, d_i) \leftarrow \mathsf{Vrfy}(d_{i-1}, I_i, \mathsf{fetched}_i, \pi_i),$$

then $b_1 = \cdots = b_m = 1$ and $\mathsf{fetched}_1, \ldots, \mathsf{fetched}_m$ are all correct (as defined above).

Security requires that for all poly-time $\mathcal{A}$, any initial array D, and any (adaptively chosen) sequence of instructions $I_1, \ldots, I_m$, if we run

$$(\tilde{D}_0, d_0) \leftarrow \mathsf{Setup}(1^\lambda, D);\quad \forall\, i : (\tilde{D}_i, \mathsf{fetched}_i, \pi_i) \leftarrow \mathcal{A}(\tilde{D}_{i-1}, I_i); (b_i, d_i) \leftarrow \mathsf{Vrfy}(d_{i-1}, I_i, \mathsf{fetched}_i, \pi_i),$$

then the probability that $b_1 = \cdots = b_m = 1$ but $\mathsf{fetched}_m$ is not the correct answer is negligible.

If the above holds even when $\mathcal{A}$ is given d_0, then we say the scheme is publicly verifiable.

Succinct Non-Interactive Arguments of Knowledge (SNARKs). We present the standard definition of SNARKs.

Definition 7 (SNARK)**.** *Algorithms* (KeyGen, Prove, Verify, Extract) *give a* succinct non-interactive argument of knowledge (SNARK) *for an NP language* L *with corresponding NP relation* R_L *if:*

Completeness: *For all* $x \in L$ *with witness* $w \in R_L(x)$*:*

$$\Pr\left[\mathsf{Verify}(sk, x, \pi) = 0 \;\middle|\; \begin{array}{c}(pk, sk) \leftarrow \mathsf{KeyGen}(1^\lambda),\\ \pi \leftarrow \mathsf{Prove}(pk, x, w)\end{array}\right] = \mathsf{negl}(\lambda)$$

Adaptive soundness: *For any probabilistic polynomial-time algorithm* $\mathcal{A}$*,*

$$\Pr\left[\begin{array}{c}\mathsf{Verify}(sk, x, \pi) = 1\\ \wedge\ (x \notin L)\end{array} \;\middle|\; \begin{array}{c}(pk, sk) \leftarrow \mathsf{KeyGen}(1^\lambda),\\ (x, \pi) \leftarrow \mathcal{A}(1^\lambda, pk)\end{array}\right] = \mathsf{negl}(\lambda)$$

Succinctness: *The length of a proof is given by* $|\pi| = \text{poly}(\lambda)\text{poly}\log(|x| + |w|)$*.*

Extractability. *For any poly-size prover* Prove^**, there exists an extractor* $\mathsf{Extract}^*$ *such that for any statement* x*, auxiliary information* μ*, the following holds:*

$$\Pr\left[\begin{array}{l}(pk, sk) \leftarrow \mathsf{KeyGen}(1^\lambda)\\ \pi \leftarrow \mathsf{Prove}^*(pk, x, \mu)\\ \mathsf{Verify}(sk, x, \pi) = 1\end{array} \quad \wedge \quad \begin{array}{l}w \leftarrow \mathsf{Extract}^*(pk, sk, x, \pi)\\ w \notin R_L\end{array}\right] = \mathsf{negl}(\lambda)$$

We say that a SNARK is *publicly verifiable* if $sk = pk$. In this case, proofs can be verified by anyone with pk. Otherwise, we call it a *secretly-verifiable* SNARK, in which case only the party with sk can verify.

Lemma 2 (Efficient SNARKs [16])**.** *Assume that the q-PDH assumption and the q-PKE assumption hold in an appropriately chosen bilinear group. There exists a publicly verifiable SNARK for Circuit-SAT where circuits have size at most s, such that* KeyGen *takes* $\tilde{O}(s) \cdot O(\lambda)$ *time, prover computation takes* $\tilde{O}(s) \cdot O(\lambda)$ *time, verifier computation is* $O(|x|\lambda)$*, the size of pk is* $O(s\lambda)$*, and proof size is* $O(\lambda)$*. Furthermore, assuming the q-PDH, d-PKE and q-PKEQ assumptions, there is a secretly verifiable SNARK with the same asymptotic efficiency.*

A.2 The Random Access Machine (RAM) Model

RAMs and RAM programs. We are interested in modeling computation on a *von Neumann architecture.* Here, we have a CPU with random access to some memory D that stores n words each of length ℓ. We let NextIns denote the CPU's next-instruction function, which takes as input some small, local state $\mathsf{cpustate}$ along with the last-read word of memory, and outputs a write address waddr_t, a read address raddr_t, and a value data_t to be written to $D[\mathsf{waddr}_t]$. A RAM *program* f is a sequence of instructions (i.e., executable code). With D initialized as described, we may then run f on some small input x (also called a *query*) by setting.

```
$\mathsf{fetched}_0 = \perp$
For $t = 1, 2, \ldots$ :
    $(\mathsf{data}_t, \mathsf{waddr}_t, \mathsf{raddr}_t, \mathsf{cpustate}_t) := \textsc{NextIns}\left(\mathsf{fetched}_{t-1}, \mathsf{cpustate}_{t-1}\right)$
    $\mathsf{fetched}_t := D[\mathsf{raddr}_t]$
    $D[\mathsf{waddr}_t] := \mathsf{data}_t$
```

We assume that any program f we consider has associated with it a time bound τ such that the program runs for exactly τ steps for all queries. Thus, the execution above ends when $t = \tau$, at which point (by convention) the final output is data_τ.

Repeated queries. We are interested in the case where a RAM program is repeatedly executed on multiple queries, with the contents of D possibly being updated as the queries are answered. (E.g., some inputs might represent an update to the underlying database itself, or might update the data during the course of computing the output.) If D denotes the initial contents of the memory (including f itself, which we assume is not being modified), then we write $y_m \leftarrow f_D(x_1, x_2, \ldots, x_m)$ to denote that the result of answering the mth query in the sequence $x_1, \ldots, x_m$ is y_m.

A.3 VC-RAM Construction

A memory-checking scheme easily yields an interactive VC-RAM scheme. Say we want to execute queries starting from initial data array D (recall that D includes both the program f as well as any underlying data). The client simply outsources storage of D to a server using a memory-checking scheme. To answer query x, the client begins running the RAM program, making read/write requests to the server as needed during the course of this execution. Each time the server gives a response, the client first verifies the response (halting execution if verification fails), and then updates its digest appropriately. It is trivial to see that the resulting scheme satisfies verifiability.

The above approach can be made non-interactive if the memory-checking scheme is publicly verifiable – by having the client simply send x to the server, and then having the server simulate on its own the actions of the client and the server from the previous, interactive protocol. At the end, the server sends the final result back to the client along with the entire sequence of proofs that the client can verify all at once.

We describe this non-interactive scheme more formally since we will further modify it later below. Let $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Vrfy})$ be a memory-checking scheme. Given initial array D containing (in addition to data) a program f that runs for exactly τ steps, the client runs $(\tilde{D}_0, d_0) \leftarrow \mathsf{Setup}(1^\lambda, D)$ and sends $\tilde{D}_0, d_0, \tau$ to the server; the client stores only d_0. The client also sends to the server a description of the NextIns circuit. When the client later wants to compute the answer to some query x, it sends x to the server. The server sets $\mathsf{cpustate}_0 = x$ and $\mathsf{fetched}_0 = \perp$. Then for $t = 1, \ldots, \tau$ it does:

$$\begin{aligned}
&(\mathsf{data}_t, \mathsf{waddr}_t, \mathsf{raddr}_t, \mathsf{cpustate}_t) := \textsc{NextIns}\left(\mathsf{fetched}_{t-1}, \mathsf{cpustate}_{t-1}\right)\\
&I_t := (\mathsf{data}_t, \mathsf{waddr}_t, \mathsf{raddr}_t)\\
&(\tilde{D}_t, \mathsf{fetched}_t, \pi_t) \leftarrow \mathsf{Prove}(\tilde{D}_{t-1}, I_t)
\end{aligned}$$

Finally, it sends the result $y = \mathsf{data}_\tau$ along with the sequence $(\mathsf{fetched}_1, \pi_1, \ldots, \mathsf{fetched}_\tau, \pi_\tau)$ to the client. To verify, the client sets $\mathsf{cpustate}_0 = x$ and $\mathsf{fetched}_0 = \perp$. Then for $t = 1, \ldots, \tau$ it does:

$$\begin{aligned}
&(\mathsf{data}_t, \mathsf{waddr}_t, \mathsf{raddr}_t, \mathsf{cpustate}_t) := \textsc{NextIns}\left(\mathsf{fetched}_{t-1}, \mathsf{cpustate}_{t-1}\right)\\
&I_t := (\mathsf{data}_t, \mathsf{waddr}_t, \mathsf{raddr}_t)\\
&(b_t, d_t) \leftarrow \mathsf{Vrfy}(d_{t-1}, I_t, \mathsf{fetched}_t, \pi_t)
\end{aligned}$$

If $b_1 = \cdots = b_\tau = 1$ and $y = \mathsf{data}_\tau$ then the client accepts y as the correct result. The client updates its local digest to d_τ; by doing so, the client and server are now ready to evaluate the RAM program again on some other input x' (with the memory array updated appropriately).

The above scheme allows the client to outsource *storage*, but not *computation.* We can address this, however, using a SNARK. Define the following $\mathcal{NP}$ relation R:

$$\big((x, y, d_0, d_\tau),\ (\mathsf{fetched}_1, \pi_1, \ldots, \mathsf{fetched}_\tau, \pi_\tau)\big) \in R$$

iff the client verification described above succeeds, and furthermore the final state of the client's digest is d_τ. We now modify the previous scheme as follows: Rather than having the server send $(\mathsf{fetched}_1, \pi_1, \ldots, \mathsf{fetched}_\tau, \pi_\tau)$, and then having the client verify all these proofs, we instead have the server *prove* (using a SNARK) that a valid proof sequence exists. Finally, the server sends the client (y, d_τ), in addition to a succinct proof that a valid witness $(\mathsf{fetched}_1, \pi_1, \ldots, \mathsf{fetched}_\tau, \pi_\tau)$ exists for the NP statement (x, y, d_0, d_τ). Using the SNARK from [16] and any memory-checking scheme, we thus obtain:

Theorem 7 (Efficient Verifiable RAM Computation)**.** *The above gives a verifiable (but not private) VC-RAM where the server runs in time $\tilde{O}(\tau \log n) \cdot \mathrm{poly}(\lambda)$, the verifier runs in time $O((|x|+|y|) \cdot \lambda)$, and the proof size is $O(\lambda)$.*

The VC-RAM is publicly verifiable if the memory-checking scheme and SNARK are.

Proof. (sketch.) The correctness of the construction can be derived from the correctness of the underlying SNARK and memory checking scheme in a straightforward manner.

Verifiability of the client-efficient construction follows from verifiability of the client-inefficient construction plus extractability of the SNARK; namely, if there exists an adversary who can break verifiability in the client-efficient version, then by extracting from that adversary a witness $\mathsf{fetched}_1, \pi_1, \ldots, \mathsf{fetched}_\tau, \pi_\tau$ we can break verifiability of the client-inefficient version. □

B Generic ORAM-to-VOS Compiler in the Malicious Model

The RAM program f. Let f denote the RAM program that executes the server's algorithm for each round of the semi-honest VOS construction:

- Suppose the initial memory array is of the form $D := (pk, \overline{D_o}[1..n], \overline{st})$.
- The input to f contains $\overline{\mathsf{op}}$, and (optionally) a set of read addresses $\{\mathsf{raddr}\}$ and write addresses $\{\mathsf{waddr}\}$. Note that corresponding to the first round of every data access operation, the input contains only $\overline{\mathsf{op}}$, and no memory addresses.
- Unless the inputs are $\perp$, perform memory reads and writes: $\overline{D_o}[\{\mathsf{waddr}\}] := \{\overline{\mathsf{data}}\}$, and $\{\mathsf{fetched}\} := \overline{D_o}[\{\mathsf{raddr}\}]$.
- Homomorphically evaluate the $\mathsf{ORAM.Next}$ circuit:

  $$(\overline{\mathsf{out}}, \{\overline{\mathsf{raddr}}\}, \{\overline{\mathsf{waddr}}\}, \{\overline{\mathsf{data}}\}, \overline{st}) \leftarrow \mathsf{FHE.Eval}(\mathsf{ORAM.Next}(\overline{\mathsf{op}}, \overline{\mathsf{st}}, \{\overline{\mathsf{fetched}}\}))$$

  Note that the $\{\overline{\mathsf{fetched}}\}$ is set to empty if this is the first round.
- Output $(\overline{\mathsf{out}}, \{\overline{\mathsf{raddr}}\}, \{\overline{\mathsf{waddr}}\})$ to the client.

ORAM-to-VOS compiler in the malicious model. The ORAM-to-VOS compiler in the malicious model is described in Figure 4.

C Proofs of Security

C.1 Proof of Security for the Semi-honest Compiler

Proof. It suffices to prove security for a dummy adversary which simply passes messages to and from the environment. We now show that for a dummy real-world adversary, there exists an ideal-world simulator $\mathcal{S}$ such that no environment can distinguish whether it is in the real or ideal world. The simulation is described as below.

- In Setup,

- Setup: Client runs $(pk, sk) \leftarrow \mathsf{FHE.Setup}(1^\lambda)$. Client runs $(D_o, st) \leftarrow \mathsf{ORAM.Init}(1^\lambda, D)$.

  For $i = 1$ to $|D_o|$, the client computes $\overline{D_o}[i] := \mathsf{FHE.Enc}_{pk}(D_o[i])$. The client also computes $\overline{st} := \mathsf{FHE.Enc}_{pk}(st)$.

  The client calls $(z, Z) \leftarrow \mathsf{VCRAM.Setup}\left(1^\lambda, \left(pk, \{\overline{D_o}[i]\}_{i \in |D_o|}, \overline{st}\}\right), f\right)$ and sends Z to the server. The client retains the local state z, and the FHE secret key sk.

- Access:

  For the j-th operation op, let R_j denote the number of ORAM rounds necessary for the j-th operation.

  First, the client encrypts $\overline{\mathsf{op}} := \mathsf{FHE}_{pk}(\mathsf{op})$ and sends it to the server.

  For $\mathsf{rnd} = 1$ to R_j:

  - Let input $x := (\{\overline{\mathsf{op}}, \{\mathsf{raddr}\}, \{\mathsf{waddr}\})$. Server evaluates $(\overline{\mathsf{ans}}, Z) \leftarrow \mathsf{VCRAM.Compute}(x, Z)$. For $\mathsf{rnd} = 1$, the sets of memory addresses $\{\mathsf{raddr}\}, \{\mathsf{waddr}\}$ are set to empty. Server sends $\overline{\mathsf{ans}}$ back to the client.
  - Client computes $(\mathsf{ans}, b, z) \leftarrow \mathsf{VCRAM.Verify}(x, \overline{\mathsf{ans}}, z)$, and verifies that $b = 1$. In case not, the client outputs $\perp$.

    Parse $\mathsf{ans} := (\overline{\mathsf{out}}, \{\overline{\mathsf{raddr}}\}, \{\overline{\mathsf{waddr}}\})$.

    If this is the last round, client uses sk to decrypt the fetched block $\overline{\mathsf{out}}$. This marks the end of this operation. Otherwise, client decrypts $\{\overline{\mathsf{raddr}}\}$ and $\{\overline{\mathsf{waddr}}\}$ and sends back $\{\mathsf{raddr}\}$ and $\{\mathsf{waddr}\}$ to the server.

Figure 4: ORAM-to-VOS generic compiler: malicious model.

1. Run $(pk, sk) \leftarrow \mathsf{FHE.KeyGen}(1^\lambda)$.
2. For $i = 1$ to $|D_o|$, the client computes $\overline{D_o}[i] := \mathsf{FHE.Enc}_{pk}(0)$. The client also computes $\overline{st} := \mathsf{FHE.Enc}_{pk}(0)$. Finally, the client sends $\left(pk, \{\overline{D_o}[i]\}_{i \in |D_o|}, \overline{st}\}\right)$ to the server.

- For each Access operation $j = 1, \ldots, m$,

  1. Send $\overline{\mathsf{op}} := \mathsf{FHE}_{pk}(0)$ to the server.
  2. When the server sends $\{\overline{\mathsf{raddr}}\}, \{\overline{\mathsf{waddr}}\}$ for decryption, Send plain-texts $\{\mathsf{raddr}\}, \{\mathsf{waddr}\}$ to the server that are returned by the ORAM simulator $\mathsf{Sim}(1^\lambda, m)$ at step j.

It is not hard to see that due to the semantic security of the FHE encryption scheme and the security of the ORAM, no PPT environment $\mathcal{Z}$ can distinguish the above ideal world from the real world. □

D The Hierarchical VOS

Here, we propose a hierarchical VOS construction based on the Goodrich-Mitzenmacher ORAM scheme [22] and its variants [26]. Although this hierarchical VOS construction achieves worse asymptotics than the Path VOS mentioned in Section 4, it is necessary later for our dynamic proofs of retrievability scheme — since the Path VOS scheme does not satisfy the next-read pattern hiding property (NRPH) proposed by Cash et al. [9].

We first state the main theorem of this section before describing the construction.

Theorem 8. *Let $g(n)$ denote some function on n. Assume collision resistance hash functions, the ring LWE assumption with suitable parametrization [5, 18], and the q-PDH and q-PKE assumptions [16]. Then, there exists a VOS scheme for a reasonably large block size $\beta = \tilde{\Omega}(\lambda)$, with $O(\beta \log n / \log g(n))$ bandwidth cost, and $\tilde{O}(\beta g(n) \log^3 n / \log g(n)) \mathrm{poly}(\lambda)$ server computation (per data access), where n is the total number of blocks and λ is the security parameter.*

The following table shows some interesting special cases of Theorem 8.

$g(n)$	server computation	bandwidth overhead
n^ϵ for constant $\epsilon < 1$	$\tilde{O}(\beta n^\epsilon \log^2 n)\mathrm{poly}(\lambda)$	$O(\beta)$
$\log n$	$\tilde{O}(\beta \log^4 n / \log\log n)\mathrm{poly}(\lambda)$	$O(\beta \log n / \log \log n)$
constant $c > 1$	$\tilde{O}(\beta \log^3 n)\mathrm{poly}(\lambda)$	$O(\beta \log n)$

Intuition. Our main idea is as follows:

- *Rely on Fully Homomorphic Encryption (FHE) to outsource client computation to the server* whenever possible, and henceforth reduce communication overhead between the client and server. One technicality here is that to preserve bandwidth overhead, we need to use FHE ciphertext packing techniques such that we can encrypt each data block of size β using a $O(\beta)$-bit ciphertext, and still maintain the ability to perform operations on each individual bit. This can be achieved using techniques described in recent works, e.g. [5].

- *Balance reads and writes to achieve better bandwidth overhead.* If we apply the above FHE and VC-RAM idea to the ORAM construction by Goodrich and Mitzenmacher [22], we can easily obtain a VOS scheme with $O(\log n)$ overhead.

  To achieve lower bandwidth overhead, we can use the the balancing trick proposed by Kushilevitz, Lu, and Ostrovsky [26]. The idea is that for a scheme where reads and writes are not of equal cost, we can balance their cost to achieve better asymptotic bandwidth overhead. Interestingly, while writes are typically more expensive than reads in the non-FHE setting [22], it turns out that reads are more expensive (in terms of bandwidth overhead) under FHE, since the writes correspond to shuffling operations which the server can homomorphically evaluate on its own. Therefore, by balancing reads and writes under FHE, we are actually unbalancing them in the traditional non-FHE setting.

  To balance reads and writes, we will adjust the rate (denoted $g(n)$) at which adjacent levels in the storage hierarchy grows. When the next level grows faster, we get schemes with better bandwidth overhead – however, at the cost of more server computation.

- *Enforcing honest server behavior and efficiency optimizations.* Finally, we apply VC-RAM techniques to enforce honest server behavior.

D.1 Preliminary: The GM Hierarchical ORAM

As a starting point, consider the Oblivious RAM scheme by Goodrich and Mitzenmacher [22]. On a high level, their scheme [22] works as follows. The server-side storage is divided into $O(\log n)$ levels denoted $B_k, B_{k+1}, \dots B_L$, where k is an appropriately chosen initial starting point for the hierarchy. Each level B_i has capacity 2^i.

For technical reasons related to proving inverse superpolynomial (i.e., negligible) failure probabilities, levels $k+1, \dots, K$ are treated differently from levels $K+1, \dots, L$, where $K = O(\log \log n)$.

- B_k is a table that the client scans through on every data access.
- For a lower level $i \in \{k+1, \dots, K\}$, B_i contains 2^{i+1} hash buckets, each of size $O(\log n)$.
- For an upper level $i \in \{K+1, \dots, L\}$, B_i is a cuckoo hash table with $(1+\epsilon) \cdot 2^{i+2}$ cells, and a stash of size $s = O(\log n)$.

The data access operations are sketched in Figure D.1. Each data access request has a read and a write phase.

D.2 Hierarchical VOS

Step 1: Applying FHE. Suppose that all blocks are encrypted under FHE. Figure D.2 shows how to execute the read and write phases more efficiently by leveraging server-side FHE evaluations.

Using FHE to delegate the ORAM client's computation to the server, the read phase requires sending a single FHE-encrypted block at every level, introducing $O(\beta \log n)$ bandwidth overhead, and $O(\log n)$ rounds of interaction. The write phase requires no communication – other than the client sending to the server the FHE-encrypted new block and the necessary FHE-encrypted hash keys. Basically, the server is capable of performing reshuffling operations under FHE on its own. The server has $O(\beta \log^2 n)\text{poly}(\lambda)$ amortized computation overhead per data access in this scheme — there will be an additional $\tilde{O}(\log n)$ factor after compilation to the malicious model with the VC-RAM.

Step 2: Balance reads and writes. In the above scheme, the read phase requires $O(\beta \log n)$ overhead, while the write phase requires $O(\beta)$ overhead. To balance reads and writes, we can make the next level grow faster than a constant rate than the previous level – however, while this reduces bandwidth overhead, the tradeoff is server computation.

As an example, consider $g(n) = 2 \log n$, i.e., we make level B_{i+1} larger than level B_i by $2 \log n$ times. In this way, we have $O(\log n / \log \log n)$ levels. Every time $B_k, \dots, B_i$ are consecutively full, they will be shuffled into a subsequent non-full level B_{i+1}. Starting from an empty B_{i+1}, levels $B_k, \dots, B_i$ will be shuffled into B_{i+1} a total of $\log n$ times – at which point B_{i+1} is deemed full as well. Every time $B_k, \dots, B_i$ are shuffled into B_{i+1}, all existing blocks inside level $B_k, \dots, B_i$ and B_{i+1} are shuffled together, and written back to B_{i+1}.

It is easy to adjust the parameter K (separating the lower and upper levels) correspondingly such that the same failure probability analysis holds as in the GM-ORAM scheme [22].

It is not hard to see that with this new rebalanced scheme, the read phase now requires only $O(\beta \log n / \log \log n)$ bandwidth overhead, while write phase requires constant bandwidth overhead. The server has $O(\beta \log^3 n)\text{poly}(\lambda)$ amortized computation overhead per data access in this rebalanced scheme. It is also not hard to apply a similar deamortization trick described in [29] and [26], to

Read phase [22].

- Scan through B_k, if block *ind* is in there, $\mathsf{found} := \text{true}$; else $\mathsf{found} := \text{false}$.
- For each level $i \in \{k+1, \ldots, K\}$: if $\mathsf{found} = \text{true}$, read a random hash bucket; else look for block *ind* in hash bucket $B_i[h_i(ind)]$ in level B_i. If found, mark $\mathsf{found} := \text{true}$.
- For each level $i \in \{K+1, \ldots, L\}$:

  If $\mathsf{found} = \text{true}$: Read $B_i[h_{i,0}(\mathsf{nextdummy}_i)]$ $B_i[h_{i,1}(\mathsf{nextdummy}_i)]$, and let $\mathsf{nextdummy}_i \leftarrow \mathsf{nextdummy}_i + 1$.

  Else: Read $B_i[h_{i,0}(ind)]$ and $B_i[h_{i,1}(ind)]$, and if found, mark $\mathsf{found} := \text{true}$.

  No matter which case: read through the stash at level i. If found, mark $\mathsf{found} := \text{true}$.
- For all of the above: after reading any block, the block is reencrypted and written back. If the block is *ind*, mark the block as obsolete before reencryption.

Write phase [22].

- If B_k is not full: write the block *ind* back to level B_k, replace with updated block if necessary.
- If B_k is full, find consecutively full levels $B_k, B_{k+1}, \ldots, B_m$, such that B_{m+1} is the first empty level. Reshuffle levels $B_k, \ldots, B_m$ into level B_{m+1} – this involves obliviously rebuilding a regular hash table or a cuckoo hash table at B_{m+1} (see [22] for the detailed algorithm).

  New hashes $h_{m+1,0}$ and $h_{m+1,1}$ are chosen every time for a level being rebuilt (by choosing a new secret key freshly at random). $\mathsf{nextdummy}_{m+1}$ is reset to 0.

Figure 5: **Sketch of the GM-ORAM construction [22].** Each level has two secret hashes $h_{i,0}$ and $h_{i,1}$ that are freshly chosen every time a level is rebuilt. Each level also has a dummy block counter denoted $\mathsf{nextDummy}_i$, which is reset to 0 every time the level is rebuilt.

spread out the reshuffling work over time, such that the server computation is $O(\beta \log^3 n)\text{poly}(\lambda)$ per data access in the worst-case. Note that our rebalancing is in the opposite direction of the Kushilevitz, Lu, and Ostrovsky scheme [26]. In the FHE setting, reads are more expensive; while in the standard ORAM setting they consider, writes are more expensive.

More generally, we can set $g(n)$ to be other functions as shown in the table in Theorem 8. For example, when $g(n) = \sqrt{n}$, the bandwidth overhead is $O(1)$ – however, server computation is $O(\beta\sqrt{n} \log n)\text{poly}(\lambda)$ per data access. The scheme for $g(n) = \sqrt{n}$ is actually a variant of the Square-Root construction [21] – with an adjusted hashing strategy to achieve inverse superpolynomial failure probability.

Step 3: Applying VC-RAM to enforce honest server behavior. To force a malicious server to adhere to the prescribed protocol, we can rely on VC-RAM techniques as introduce in Section 3.3.1. Particularly, we will apply the VC-RAM techniques to the deamortized version of our scheme.

Read phase under FHE.

- *Client*: found := false. $\overline{ind} \leftarrow \mathsf{FHE.Enc}(ind)$. Send $\overline{ind}$ to server.
- **Level** k:

  Server: $\overline{\mathsf{block}} \leftarrow \mathsf{FHE.Eval}(\mathsf{find}(\overline{ind}, \overline{\mathsf{B}}_\mathsf{k}))$. where find is the function that looks for a block ind in a table, and returns the block found or $\perp$ upon failure. Send $\overline{\mathsf{block}}$ to client.

  Client: Decrypt $\overline{\mathsf{block}}$ and update found appropriately.
- **For each level** $i \in \{k+1, \ldots, K\}$:

  Client: if found = true, choose a at random; else choose $a \leftarrow h_i(ind)$. Send a to server.

  Server: $\overline{\mathsf{block}} \leftarrow \mathsf{FHE.Eval}(\mathsf{find}(\overline{ind}, \overline{\mathsf{B}}_\mathsf{i}[\mathsf{a}]))$. Send $\overline{\mathsf{block}}$ to client.

  Client: Decrypt $\overline{\mathsf{block}}$ and update found appropriately.
- **For each level** $i \in \{K+1, \ldots, L\}$:

  Client: if found = true, choose a_0, a_1 at random; else choose $a_0 \leftarrow h_{i,0}(ind)$, $a_1 \leftarrow h_{i,1}(ind)$. Send a_0, a_1 to server.

  Server: $\overline{\mathsf{block}} \leftarrow \mathsf{FHE.Eval}(\mathsf{find}(\overline{ind}, \overline{\mathsf{B}}_\mathsf{i}[\mathsf{a_0}], \overline{\mathsf{B}}_\mathsf{i}[\mathsf{a_1}], \overline{\mathsf{B}}_\mathsf{i}[\mathrm{stash}_\mathsf{i}]))$. Send $\overline{\mathsf{block}}$ to client.

  Client: Decrypt $\overline{\mathsf{block}}$ and update found appropriately.

Write phase under FHE.

- *Client*: $\overline{\mathsf{block}} \leftarrow \mathsf{FHE.Enc}(ind, \mathsf{block})$. Choose one more more fresh random hash keys. $\overline{\mathsf{keys}} \leftarrow \mathsf{FHE.Enc}(\mathsf{keys})$. Send $\overline{\mathsf{block}}, \overline{\mathsf{keys}}$ to server.
- *Server:* If B_k is not full (the server can know this by keeping a counter of total data requests): $\overline{B}_k \leftarrow \mathsf{FHE.Eval}(\mathsf{insert}(\overline{\mathsf{B}}_\mathsf{k}, \overline{\mathsf{block}}))$ where insert is the function that inserts a block into a table.

  If B_k is full, find consecutively full levels $B_k, B_{k+1}, \ldots, B_m$ (the server knows this by keeping a counter of total data requests), such that B_{m+1} is the first empty level. Homomorphically evaluate $\overline{B}_k, \ldots, \overline{B}_{m+1} \leftarrow \mathsf{FHE.Eval}(\mathsf{reshuffle}(\overline{\mathsf{keys}}, \overline{\mathsf{B}}_\mathsf{k}, \ldots, \overline{\mathsf{B}}_\mathsf{m}))$ where $\mathsf{reshuffle}$ is the function that empties levels B_k through B_m, and reshuffles them into level B_{m+1} – suppressing obsolete blocks in the meanwhile.

Figure 6: Hierarchical VOS secure in the semi-honest model.

E Round Complexity

This paper focuses on reducing the bandwidth overhead of VOS, showing how allowing server-side computation enables us to circumvent the ORAM lower bound on bandwidth cost.

Our constructions so far require that the server interacts with the client whenever it needs the client to decrypt FHE ciphertexts to obtain the next read and write addresses. In the case of our Path VOS scheme, we already achieve constant-round overhead, since the depth of the recursive construction is $O(1)$ for any poly(λ)-sized database. On the other hand, our Hierarchical VOS scheme requires $O(\log(n)/\log(g(n)))$ rounds of communication (one for each level) per original RAM instruction.

We sketch how to use a recent trick relying on Indistinguishability Obfuscation proposed by Garg et al. [15] to reduce the round complexity of the Hierarchical VOS to a single round per operation of the original RAM. (The same idea applied to Path VOS reduces the rounds-per-operation from $O(1)$ to 1 as well.) The idea is that the client can obfuscate the FHE decryption circuit with Indistinguishability Obfuscation ($i\mathcal{O}$) and send it to the server, such that the server can decrypt on its own. In order to prevent the "reset" attack where the server evaluates the $i\mathcal{O}$ on arbitrary inputs of its choice, we need to use statistically sound NIZKs in a similar manner as Garg et al. [15], to ensure that only one evaluation trace is acceptable. At first sight, it would seem like virtual black-box obfuscation might be required to ensure security. However, as Garg et al. showed, because there is only a single acceptable evaluation trace, in the proof, we can replace the $i\mathcal{O}$ with another one that simply hardcodes this exact evaluation trace in the obfuscated program.

Also, note that Garg et al. considers a multi-party setting, and each party uses CCA-secure encryption to commit to their inputs at the beginning of the protocol. In this way, the simulator in the proof can extract inputs from malicious parties. In our setting, only the honest client has input, and the potentially malicious server does not have input. Therefore, the proof does not have to rely on extracting malicious party's inputs.