Non-Interactive Verifiable RAM Computation

Abstract

Consider a client who wishes to outsource a large database to an untrusted server, and subsequently would like to make queries and updates to the database. The client wishes to guarantee both privacy and verifiability. Most existing verifiable computation fail to address this problem satisfactorily, due to the fact that the server's computation per query is linear in the size of the database.

We propose non-interactive verifiable RAM computation – since most real-world database queries (e.g., range queries, binary search, graph queries) can be implemented efficiently in the RAM model. We propose two constructions: one that guarantees verifiability but does not offer privacy; and a second one that guarantees both. Our constructions have near-optimal efficiency: the server's computation per query is not significantly more than the run-time of the RAM program; the client's online computation per query is independent of the run-time of the RAM, or the database size.

We show that our VC-RAM model is powerful, and give several applications. Particularly, we show how to rely on verifiable-only VC-RAM construction to build verifiable oblivious storage schemes (i.e., oblivious RAM with active server computation) that consume asymptotically less bandwidth overhead than existing oblivious RAM schemes, while ensuring verifiability against an arbitrarily malicious server.

Keywords: verifiable RAM computation; verifiable oblivious storage; oblivious RAM

1 Introduction

Cloud computing allows users and organizations to outsource both their *data* and *computation* to cloud servers. Imagine that a client C outsources a database DB (e.g., a SQL database, a key-value store, a graph, etc.) to an untrusted cloud server S. The client will then make a series of queries to the server. For example, the client can ask the server to compute a function over the outsourced DB, and return the answer; the client can also make update queries such as inserting, deleting, or modifying entries in the database. Since the cloud server is outside the client's control, a big challenge is how to guarantee the *privacy* of outsourced data, and the *integrity* of computation performed by the server.

One way to address this problem is to rely on Non-Interactive Verified Computation (NIVC), first proposed by Gennaro, Gentry, and Parno [20] (henceforth referred to as the GGP construction), and improved by subsequent works [1, 8, 14, 15, 20, 21, 25, 48]. At a high-level, NIVC schemes allow a client to outsource a database to a server such that it can verify the correctness of the subsequent query results efficiently. Some schemes additionally guarantee the privacy of the database, and/or the inputs and outputs to the queries.

There are, however, several drawbacks with most existing schemes: First, most schemes [1, 8, 14, 15, 20, 21, 25, 48] require the server to perform *linear* (in the database size) computation, even when the query takes *sublinear* time to run in the insecure (i.e., unencrypted and unauthenticated) setting. This can be a prohibitive overhead in real-world applications. In fact, most existing database implementations leverage efficient data structures such that answering queries (e.g., binary search, range queries) take sublinear time.

Second, while insertions, deletions, and updates, are common-place in practical applications, most existing NIVC schemes [1, 8, 14, 20, 21, 25, 48] do not allow efficient updates to the database. For example, with the GGP construction, updating the database would require rerunning the preprocessing algorithm, which takes at least time linear in the size of the database.

1.1 Our Main Results and Contributions

We consider the Random Access Machine (RAM) model of computation, since most real-world databases queries can be computed efficiently (i.e., in sublinear time) in the RAM model. We make the following contributions:

New definitions. We formally define the problem of Verifiable Computation in the RAM model (VC-RAM). We define the full security of VC-RAM using a simulation-based definition (equivalent to Universal Composability) assuming an honest client and a malicious server. Our formulation naturally supports *updates* to the database.

Constructions with near-optimal client and server computation. We propose two new VC-RAM constructions, one that is verifiable-only, but does not guarantee privacy; and a second construction that guarantees verifiability and privacy simultaneously.

Our constructions achieve *near-optimal* computation overhead for both the client and the server in the following sense. Let λ denote the security parameter, n denote the memory size, and $|x|$ and $|y|$ denote the input and output sizes respectively.

- The client's online computation for each query is $O(|x| + |y|) \cdot \mathrm{poly}(\lambda)$, and is independent of τ. Notice that the client has to pay at least $|x| + |y|$ cost to read the input and output.
- If a RAM program takes τ time to compute in the insecure setting, our VC-RAM construction requires the server to perform only $\tau \cdot \mathrm{poly}(\lambda, \log n)$ computation, This implies that for queries that take sublinear time, our server computation is also sublinear.

Additional implications of our results. Our results have several implications beyond VC-RAM, including (1) *authenticated data structures*; (2) *verifiable oblivious storage*; and (3) *verifiable "searchable encryption"* in the symmetric-key setting for *general queries*, *hiding access patterns*, and with *sublinear* server computation (for queries that run in sublinear time). We now elaborate on the applications and implications of our main results.

1.2 Efficient Verifiable Oblivious Storage and Complete Security Definitions

Our first application of VC-RAM is verifiable oblivious storage (VOS). VOS is related to, but different from, oblivious RAM (ORAM) [23]. Historically, in the context of ORAM the honest server only reads/writes blocks of data as instructed by the client, but does no computation. More recently [37, 53], researchers have considered a more general model in which the server performs local computation as well. We refer to the latter model as verifiable oblivious storage.

We observe that the VOS setting demands a different security definition as in the original ORAM setting considered by Goldreich and Ostrovsky [23] – particularly, if the server is required to perform computation as part of the protocol, our definition aims to achieve *verifiability* even when a malicious server may arbitrarily deviate from the prescribed behavior (other than simply corrupting the data [23]).

Our security definition for VC-RAM immediately implies a full-fledged security definition for VOS, since VOS is a special case of VC-RAM, in the sense that the RAM program considered simply reads or writes data.

We construct more efficient VOS schemes in comparison with known constructions in the ORAM model [23, 26, 27, 35, 54], and known schemes in the VOS model [37, 53]. Relying on Fully Homomorphic Encryption and our verifiable-only VC-RAM construction, assuming constant client-side storage and a reasonably large block size, we can construct a VOS scheme with $O(\log n/\log\log n)$ *bandwidth overhead* with $\text{poly}(\log n, \lambda)$ server computation; alternatively, we can achieve $O(1)$ bandwidth cost with $O(n^{\alpha}\log n)\text{poly}(\lambda)$ server computation (where $\alpha < 1$ is a constant). Our schemes are secure with respect to a *malicious* server. In comparison, the best known result in the constant client storage setting requires $O(\log^2 n/\log\log n)$ overhead [35] and works in the ORAM model.

Note that this result does NOT violate the known super-logarithmic lower bound for ORAM [5], since this lower bound only applies to the traditional ORAM setting, not to VOS.

1.3 Other Applications and Implications of Our Main Result

Verifiable and oblivious "searchable encryption" in the symmetric-key setting. Searchable encryption [10, 11, 16, 24, 32, 36, 50, 51, 52] allows a client to outsource a set of encrypted documents to a server, and later make search queries over the encrypted data. Searchable encryption has been studied first for keyword queries [10, 16, 31, 52], and later for conjunctive queries [11, 51], inner product queries [32, 36], and general queries [24]. Particularly, private-index functional encryption [24] implies a searchable encryption scheme for general queries.

Previous verifiable searchable encryption schemes suffer from the following drawbacks: *i)* Existing schemes leak access patterns, i.e., the server can learn which encrypted documents match the query. Such access pattern leakage has been shown to be harmful due to statistical attacks [30]. *ii)* Existing schemes (except schemes for keyword queries [16, 31]) require linear server computation, namely, the server needs to perform computation for every encrypted document in the dataset. *iii)* Most existing searchable encryption schemes (with the exception of Kurosawa et al. [34]) assume a semi-honest server, and do not provide verifiability against malicious servers. For example, a malicious server can potentially leave out encrypted documents that match the query, violating completeness.

The task that searchable encryption aims to achieve[1] can be regarded as a special case of VC-RAM where the queries are searches. Our construction therefore addresses all of the above drawbacks by providing a (non-interactive) *verifiable* searchable encryption scheme for *general queries* in the symmetric-key setting, allowing *sublinear* server computation for sublinear-time queries, and *hiding access patterns.*

[1] Technically, VC-RAM is not a searchable encryption scheme, but a different formulation which better captures the goal that searchable encryption tried to achieve. Searchable encryption is formulated the same way as private-index functional encryption, i.e., each token can be used to evaluate on *any* ciphertext.

Generalization of (private) authenticated data structures. Authenticated data structures [18, 28, 29, 39, 42, 44, 45, 46, 47] allow a user to outsource a dataset to a server while retaining only a small digest., Later the client can make queries or updates to the dataset, and additionally verify that the answers provided by the server are correct.

Authenticated data structures focus on designing efficient schemes for specific queries such as (non)-membership queries [28, 42], range queries [39], set operations [47], graph queries [29], etc. Unlike in the verifiable computation line of research, authenticated data structures allow the server to compute in sublinear time for sublinear-time queries. Existing authenticated data structures do not take privacy into account in their design. Even without the privacy consideration, how to construct efficient (i.e., sublinear server computation) authenticated data structures for arbitrary databases and queries remains an open question.

Our work generalizes authenticated data structures and answers this open question in the affirmative: 1) we support arbitrary queries and data structures; 2) like in existing authenticated data structure schemes, the server's computation is close to the insecure setting; 3) our client's verification cost is independent of the query's execution time; 4) we additionally offer privacy guarantees in the two-party setting (but not the three-party setting) – however, our verifiable-only scheme works both the two-party and three-party settings.

1.4 Other Related Work

Related work on Non-Interactive Verified Computation [1, 8, 14, 15, 20, 21, 25, 48] and authenticated data structures [18, 28, 29, 39, 42, 44, 45, 46, 47] has been mentioned earlier Section 1.

Garbled RAM, recently proposed by Lu and Ostrovsky [37] is very closely related to our work. Garbled RAM can be modified to work in our VC-RAM setting – however, the client's online computation overhead per query is proportional to the run-time of the RAM program. Our second construction, the verifiable and private scheme, builds on top of the garbled RAM work – for various technical reasons explained later, we need a *generalized* variant of the garbled RAM scheme that builds on *any* Oblivious RAM compiler, and proven secure under our new VC-RAM security model (a bit stronger than Lu and Ostrovsky's definition [37]). We refer to this scheme (described in Appendix **??**) as a verifier-inefficient VC-RAM scheme. In Section **??**, we show how to boost its efficiency such that the client's online computation is independent of the RAM's run-time.

We note that unlike the GGP construction which wraps Fully Homomorphic Encryption (FHE) around a garbled circuit, wrapping FHE around Garbled RAM does not work in our case – and the reason is similar to why the strawman scheme in Section **??** fails.

The techniques for our verifiable-only VC-RAM scheme are reminiscent of those used by Ben-Sasson et al. [6] for reductions from RAMs to delegatable succinct constraint satisfaction problems.

Our work is also related to PCPs [2, 3, 4, 19, 33**?**], and interactive proofs [38, 49**?** **?** **?**]. Although we mainly focus on the non-interactive setting. Our verifiably-only construction relies on non-interactive arguments (of knowledge) as a building block [7, 17, 21, 25, 40, 41**?**].

2 Preliminaries

2.1 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 $\textsc{NextIns}$ 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) which is stored in some pre-allocated portion of D. Other data (e.g., a database, a graph) may be stored in D as well before execution begins. With D initialized as described, we may then run f on some input x (also called a *query*) by setting[2] $\mathsf{cpustate}_0 = x$ and then running

```
$\mathsf{fetched}_0 = \perp$
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 a 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.

Assumptions. If λ denotes the security parameter, we assume that the number of memory words $n = \mathrm{poly}(\lambda)$, the total number of queries $Q = \mathrm{poly}(\lambda)$, and the program executes for $\tau = \mathrm{poly}(\lambda)$ steps. We also assume that the bit-length of each memory word and the CPU states $\ell = O(\log \lambda)$, $|\mathsf{cpustate}| = O(\log \lambda)$. Since the $\textsc{NextIns}$ function takes in $O(\log \lambda)$ bits and outputs $O(\log \lambda)$ bits, the $\textsc{NextIns}$ function can be expressed with a size-$\tilde{O}(\lambda)$ circuit.

2.2 Memory Checking

We recast memory checking [9] in a form better suited for our purposes, however it is clear that our definition is equivalent to that used in prior work. We use the same notation as in Section 2.1. 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 1. *A* memory-checking scheme *consists of algorithms* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Vrfy})$ *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*.*

[2]This assumes x is "short." If x is "large," we can store it in D; see Section **??**.

- 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* $\mathsf{fetched}$, *and a proof* π.
- Vrfy *takes as input a digest* d, *an instruction* $I = (\mathsf{data}, \mathsf{waddr}, \mathsf{raddr})$, *a value* $\mathsf{fetched}$, *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.

2.3 Succinct Non-Interactive Arguments of Knowledge (SNARKs)

Definition 2 (SNARK). *Algorithms* $(\mathsf{KeyGen}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Extract})$ *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* $|\pi| = \mathrm{poly}(\lambda)\mathrm{poly}\log(|x| + |w|)$.

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 3 (Efficient SNARKs [21]). *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.*

2.4 Verifiable RAM Computation

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

$(Z, z) \leftarrow \mathsf{Setup}(1^\lambda, D, \mathit{params})$: The Setup algorithm is a one-time setup algorithm run by the client. Setup takes in the security parameter 1^λ, initial memory array D, and parameters $\mathit{params} := (n, \ell, |\mathsf{cpustate}|)$ of the RAM; and outputs server initial state Z, and client state z. The client hands Z to the server, and retains state z locally.

$(\overline{x}, z) \leftarrow \mathsf{ProbGen}(x, z)$: Given input x, prepare the input and obtain the encoding $\overline{x}$. The client state z is updated[3].

$(\overline{y}, Z) \leftarrow \mathsf{Compute}(\overline{x}, Z)$: Given current server state Z and encoded input $\overline{x}$, the server computes an encoded answer $\overline{y}$, which typically embeds the output as well as a proof of correct computation. The server also updates its state Z as a result of $\mathsf{Compute}$.

$(y, b, z) \leftarrow \mathsf{Verify}(\overline{y}, z)$: Outputs the decoded answer y, a bit $b \in \{0,1\}$ indicating whether the answer is accepted, and updates the client local state z.

Correctness is defined in the obvious way. We require that for any *params*, for any initial memory array $D \in \{0,1\}^{\ell n}$, for any query sequence $x_1, x_2, \ldots, x_m$ where $m = \mathrm{poly}(\lambda)$,

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

Server efficiency and client efficiency. We say that a VC-RAM scheme is *verifier efficient*, if the client's online computation per query (including the cost of the $\mathsf{ProbGen}$ and Verify algorithms) is asymptotically smaller than the run-time of the RAM program. In our constructions, the client's online computation is independent of the time τ of running the RAM program and the memory size n. We say that a VC-RAM scheme is *server efficient*, if the server's online computation per query is sublinear in the size of D for queries that take sublinear time to execute in the RAM model.

We now define the security of VC-RAM. We give two security definitions: 1) what is a *verifiable* VC-RAM scheme; and 2) what is a *verifiable* and *private* VC-RAM scheme. In both definitions, we consider an honest client, and a *malicious* server.

2.5 Security Definitions: Verifiability of VC-RAM

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

[3] We use the notation $(\mathit{output}, \mathit{state}) \leftarrow \mathsf{Alg}(\mathit{input}, \mathit{state})$ to denote that a party runs algorithm Alg, which results in updating its local state.

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

Public vs. secret verifiability, two-party vs. three-party settings. In the above, we have defined a secretly verifiable VC-RAM, i.e., the client state z needs to be kept secret from the server.

We say that a VC-RAM scheme is *publicly verifiable* if the client state z necessary for the verification need not be kept secret from the server. More formally, in lines marked [*] in the above security definition, we supply the latest client state z to the adversary $\mathcal{A}$ as well.

Public verifiability is also referred to the *three-party* setting in the authenticated data structure literature. Imagine the scenario where a trusted data source (i.e., client) distributes a signed copy of the latest client state z (often referred to as a digest in the authenticated data structure literature). We can easily augment the definition to distinguish between two types of queries – *read-only* queries and *update* queries. Update queries (e.g., insertions and deletions to a database) should only be made by the trusted data source, and would write to memory as a result of the RAM execution. By contrast, a read-only query (e.g., performing a binary or keyword search) does not update the memory during the RAM execution, and anyone with the latest client state (i.e., digest) z should be able to issue read-only queries and verify the result. More formally, the $\mathsf{ProbGen}$ and Verify algorithms for read-only queries should not cause updates to the client state (i.e., digest) z. How to enforce such access control is an orthogonal issue, and outside the scope of this paper.

3 VC-RAM Construction

In this section we sketch a construction for verifiable-only VC-RAM construction. Our construction is based on any publicly-verifiable memory-checking scheme and SNARK; see Appendices 2.2 and 2.3. Our construction is publicly verifiable if the underlying SNARK is publicly verifiable; else it is secretly verifiable.

Although it may appear as if our verifiable and private construction (Section **??**) subsumes the verifiable-only scheme, the verifiable-only construction is actually worthy of independent interest, since 1) we can additionally achieve public verifiability in this setting; 2) the verifiable-only scheme is cheaper than our verifiable and private scheme – it turns out that this is necessary for our verifiable oblivious storage application (Section **??**).

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 beings running the RAM program as in Section 2.1, 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 [21] (see Appendix 2.3) and any memory-checking scheme, we thus obtain:

Theorem 6 (Efficient Verifiable RAM Computation)**.** *The above gives a verifiable (but not private) VC-RAM where the server runs in time $O(\tau \log n) \cdot \mathrm{poly}(\lambda)$, the verifier runs in time $O(|x| \cdot \lambda)$, and the proof size is $O(\lambda)$.*

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

Proof. 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. We defer the straightforward details to the full version. $\square$

4 Verifiable Oblivious Storage

VC-RAM is a very powerful primitive, and has immediate implications in several areas in cryptography. One application is to build efficient verifiable oblivious storage schemes. Oblivious RAM (ORAM) [23, 26, 27, 35, 43, 53, 54] was initially intended for obliviously simulating RAM programs and achieving software piracy. In this setting, the memory is considered *passive*, i.e., only capable of fetching and storing data, but not capable of performing computation.

As cloud computing gains popularity, ORAM was applied to the outsourced data setting, such that a client can store data with an untrusted server, such that the server learns nothing about the data or access patterns. In this setting, it is natural to consider *active* servers capable of performing computation. In fact, a couple recent constructions [37, 55] have leveraged server-side computation to achieve single-round ORAM.

4.1 Complete Security Definition for VOS

To differentiate, we use the terminology verifiable oblivious storage (VOS) to refer to the data outsourcing setting when the server actively performs computation. In fact, we observe that the VOS setting demands a new security definition from the traditional ORAM setting, and the security definition should take into account the fact that a dishonest server can arbitrarily deviate from the prescribed computation (other than simply corrupting data blocks as the model considered by Goldreich and Ostrovsky in their initial ORAM work [23]), We note that our VC-RAM security definition immediately implies a full-fledged security definition for VOS, since VOS is just special case of VC-RAM with a degenerated RAM program that simply reads or writes data.

4.2 More Efficient VOS Construction

Under the traditional ORAM (i.e., passive memory) setting, with $O(1)$ client private memory, the best known result is a recent scheme by Kushilevitz, Lu, and Ostrovsky [35], achieving $O((\log n)^2/\log\log n)$ bandwidth overhead. Meanwhile, lower bound results [5, 23] have suggested that there is an $O(\log n \log\log n)$ lower bound on the bandwidth overhead of any Oblivious RAM scheme.

However, we observe that this super-logarithmic lower bound does *not* apply to the VOS setting. By leveraging server-side computation, we can build VOS schemes that are more bandwidth efficient than traditional ORAM schemes, and meanwhile, ensure verifiability against an arbitrarily malicious server.

Theorem 7. *Let $g(n)$ denote some function on n. Assume collision resistance hash functions, the ring LWE assumption with suitable parametrization [13, 22], and the q-PDH and q-PKE assumptions [21]. Then, there exists a VOS scheme for a reasonably large block size*[4] *$\beta = \tilde{\Omega}(\lambda)$, with $O(\log n/\log g(n))$ bandwidth overhead, and $O(g(n)\log^2 n/\log g(n))\text{poly}(\lambda)$ server computation (per data access), where n is the total number of blocks and λ is the security parameter.*

[4] Note that the VOS data block size β is differet from the memory word size ℓ of the RAM. We assume $\beta = \tilde{\Omega}(\lambda)$, and $\ell = O(\log\lambda)$.

The following table shows some interesting special cases of Theorem 7.

$g(n)$	server computation	bandwidth overhead
n^α for constant $\alpha < 1$	$O(n^\alpha \log n)\mathrm{poly}(\lambda)$	$O(1)$
$(\log n)^{(\log\log n)^\alpha}$ for constant $\alpha \geq 0$	$O((\log n)^{(\log\log n)^\alpha+2}/(\log\log n)^{\alpha+1})\mathrm{poly}(\lambda)$	$O(\log n/(\log\log n)^{\alpha+1})$
$\log n$	$O(\log^3 n/\log\log n)\mathrm{poly}(\lambda)$	$O(\log n/\log\log n)$
constant $\alpha > 1$	$O(\log^2 n)\mathrm{poly}(\lambda)$	$O(\log n)$

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 [12, 13].

- *Use our verifiable-only VC-RAM construction (Section 3) to enforce honest server behavior*[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 [26], 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 [35]. 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 [26], 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.

Preliminary: the GM-ORAM scheme. As a starting point, consider the Oblivious RAM scheme by Goodrich and Mitzenmacher [26]. On a high level, their scheme [26] works as follows. The server-side storage is divided into $O(\log n)$ levels denoted $B_k, B_{k+1}, \ldots B_L$, 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 $k+1, \ldots, K$ are treated differently from levels $K+1, \ldots, L$, where $K = O(\log\log n)$.

[5] Although our verifiable and private VC-RAM construction implies a VOS scheme, it does not achieve the desired bandwidth overhead. We need the verifiable-only VC-RAM scheme here because it is cheaper than our verifiable and private VC-RAM scheme.

- B_k is a table that the client scans through on every data access.
- For a lower level $i \in \{k+1, \ldots, K\}$, B_i contains 2^{i+1} hash buckets, each of size $O(\log n)$.
- For an upper level $i \in \{K+1, \ldots, L\}$, 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 below. Each data access request has a read and a write phase.

Read phase [26].

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

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

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

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

Write phase [26].

- If B_k is not full: write the block u 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 [26] for the detailed algorithm) New hashes are chosen every time for level being rebuilt (by choosing a new secret key freshly at random).

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

Read phase under FHE.

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

  Server: $\overline{\mathsf{block}} \leftarrow \mathsf{FHE.Eval}(\mathsf{find}(\overline{\mathsf{u}}, \overline{B}_k))$. where find is the function that looks for a block u 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 $a \leftarrow h_i(\mathsf{u})$. Send a to server.

  Server: $\overline{\mathsf{block}} \leftarrow \mathsf{FHE.Eval}(\mathsf{find}(\overline{\mathsf{u}}, \overline{B}_i[a]))$. 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 $a_0 \leftarrow h_{i,0}(\mathsf{u})$, $a_1 \leftarrow h_{i,1}(\mathsf{u})$. Send a_0, a_1 to server.

  Server: $\overline{\mathsf{block}} \leftarrow \mathsf{FHE.Eval}(\mathsf{find}(\overline{\mathsf{u}}, \overline{B}_i[a_0], \overline{B}_i[a_1], \overline{B}_i[\text{stash}_i]))$. Send $\overline{\mathsf{block}}$ to client.

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

Write phase under FHE.

- *Client*: $\overline{\mathsf{block}} \leftarrow \mathsf{FHE.Enc}(\mathsf{u}, \mathsf{data})$. 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): $\overline{B}_k \leftarrow \mathsf{FHE.Eval}(\mathsf{insert}(\overline{B}_k, \overline{\mathsf{block}}))$ 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 $\overline{B}_k, \ldots, \overline{B}_{m+1} \leftarrow \mathsf{FHE.Eval}(\mathsf{reshuffle}(\overline{\mathsf{keys}}, \overline{B}_k, \ldots, \overline{B}_m))$ 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.

Using the above FHE idea, the read phase requires sending a single FHE-encrypted block at every level, introducing $O(\log n)$ 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 $O(\log^2 n)\text{poly}(\lambda)$ amortized computation overhead per data access in this scheme.

Step 2: Applying verifiable-only VC-RAM to enforce honest server behavior. The above gives us an $O(\log n)$ overhead VOS scheme assuming a semi-honest server. However, the server may be malicious, and may not perform the prescribed shuffling and homormorphic evaluations honestly.

To address this problem, we can apply our verifiable-only VC-RAM scheme. Observe that all the homomorphic evaluations the server performs can be modelled as RAM computation. During the setup phase, the client will upload the FHE-encrypted and initially shuffled data blocks to the server, and retain a digest of this initial storage snapshot. Later in the online phase, the server can always use the verifiable-only VC-RAM to prove to the client that it has performed the prescribed computation correctly, i.e., 1) the claimed result returned to the client is correct; and 2) the claimed new digest of the updated storage snapshot is correct.

Step 3: Balance reads and writes. In the above scheme, the read phase requires $O(\log n)$ overhead, while the write phase requires constant 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 $B_k, \ldots, B_i$ are consecutively full, they will be shuffled into a subsequent non-full level B_{i+1}. Starting from an empty B_{i+1}, levels $B_k, \ldots, B_i$ 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 $B_k, \ldots, B_i$ are shuffled into B_{i+1}, all existing blocks inside level $B_k, \ldots, B_i$ 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 [26].

It is not hard to see that with this new rebalanced scheme, the read phase now requires only $O(\log n/\log\log n)$ bandwidth overhead, while write phase requires constant bandwidth overhead. The server has $O(\log^3 n)\text{poly}(\lambda)$ amortized computation overhead per data access in this rebalanced scheme. It is also not hard to apply a similar deamortization trick described in [43] and [35], to spread out the reshuffling work over time, such that the server computation is $O(\log^3 n)$ per data access in the worst-case. Note that our rebalancing is in the opposite direction of the Kushilevitz, Lu, and Ostrovsky scheme [35]. 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 7. For example, when $g(n) = \sqrt{n}$, the bandwidth overhead is $O(1)$ – however, server computation is $O(\sqrt{n}\log n)\text{poly}(\lambda)$ per data access. The scheme for $g(n) = \sqrt{n}$ is actually a variant of the Square-Root construction [23] – with an adjusted hashing strategy to achieve inverse superpolynomial failure probability.

References

[1] B. Applebaum, Y. Ishai, and E. Kushilevitz. From secrecy to soundness: Efficient verification via secure computation. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 152–163. Springer, 2010.

[2] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, May 1998.

[3] S. Arora and S. Safra. Probabilistic checking of proofs: a new characterization of np. *J. ACM*, 45(1):70–122, Jan. 1998.

[4] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, STOC '91, pages 21–32, New York, NY, USA, 1991. ACM.

[5] P. Beame and W. Machmouchi. Making branching programs oblivious requires superlogarithmic overhead. In *Proceedings of the 2011 IEEE 26th Annual Conference on Computational Complexity*, CCC '11, pages 12–22, Washington, DC, USA, 2011. IEEE Computer Society.

[6] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. Fast reductions from rams to delegatable succinct constraint satisfaction problems: extended abstract. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, ITCS '13, pages 401–414, New York, NY, USA, 2013. ACM.

[7] 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*, pages 326–349, 2012.

[8] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. Succinct non-interactive arguments via linear interactive proofs. In *TCC*, 2013.

[9] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

[10] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *EUROCRYPT*, pages 506–522, 2004.

[11] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In *TCC*, pages 535–554, 2007.

[12] Z. Brakerski, C. Gentry, and S. Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In *Public Key Cryptography*, pages 1–13, 2013.

[13] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, 2012. ACM.

[14] K.-M. Chung, Y. Kalai, and S. Vadhan. Improved delegation of computation using fully homomorphic encryption. In *CRYPTO*, 2010.

[15] K.-M. Chung, Y. T. Kalai, F.-H. Liu, and R. Raz. Memory delegation. In *CRYPTO*, pages 151–168, 2011.

[16] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security*, pages 79–88, 2006.

[17] I. Damgård, S. Faust, and C. Hazay. Secure two-party computation with low communication. In *TCC*, pages 54–74, 2012.

[18] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. Stubblebine. Flexible authentication of XML documents. *Journal of Computer Security*, 6:841–864, 2004.

[19] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Interactive proofs and the hardness of approximating cliques. *J. ACM*, 43(2):268–292, Mar. 1996.

[20] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, pages 465–482, 2010.

[21] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. Cryptology ePrint Archive, Report 2012/215, 2012. `http://eprint.iacr.org/`.

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

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

[24] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich. Succinct functional encryption and applications: Reusable garbled circuits and beyond. IACR Cryptology ePrint Archive 2012/733, `http://eprint.iacr.org/2012/733`.

[25] S. Goldwasser, H. Lin, and A. Rubinstein. Delegation of computation without rejection problem from designated verifier CS-proofs. *IACR Cryptology ePrint Archive*, 2011:456, 2011.

[26] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 576–587, 2011.

[27] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012.

[28] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *Proc. DARPA Information Survivability Conference and Exposition II (DISCEX II)*, pages 68–82, 2001.

[29] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Efficient authenticated data structures for graph connectivity and geometric search problems. *Algorithmica*, 60(3):505–552, 2011.

[30] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[31] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security*, pages 965–976, 2012.

[32] J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *EUROCRYPT*, pages 146–162, 2008.

[33] J. Kilian. Improved efficient arguments (preliminary version). In *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '95, pages 311–324, London, UK, UK, 1995. Springer-Verlag.

[34] K. Kurosawa and Y. Ohtaki. Uc-secure searchable symmetric encryption. In *Financial Cryptography*, pages 285–298, 2012.

[35] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012.

[36] A. B. Lewko, T. Okamoto, A. Sahai, K. Takashima, and B. Waters. Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption. In *EUROCRYPT*, pages 62–91, 2010.

[37] S. Lu and R. Ostrovsky. How to garble ram programs. IACR Cryptology ePrint Archive 2012/601, `http://eprint.iacr.org/2012/601`.

[38] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39(4):859–868, Oct. 1992.

[39] C. U. Martel, G. Nuckolls, P. T. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[40] S. Micali. Cs proofs (extended abstracts). In *FOCS*, pages 436–453, 1994.

[41] S. Micali. Computationally sound proofs. *SIAM J. Comput.*, 30(4):1253–1298, Oct. 2000.

[42] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proc. 7th USENIX Security Symposium*, pages 217–228, Berkeley, 1998.

[43] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *ACM Symposium on Theory of Computing (STOC)*, pages 294–303, 1997.

[44] C. Papamanthou and R. Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *Proc. Int. Conference on Information and Communications Security (ICICS)*, volume 4861 of *LNCS*, pages 1–15. Springer, 2007.

[45] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 437–448. ACM, October 2008.

[46] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal authenticated data structures with multilinear forms. In *Proc. Int. Conference on Pairing-Based Cryptography (PAIRING)*, pages 246–264, 2010.

[47] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. In *CRYPTO*, pages 91–110, 2011.

[48] B. Parno, M. Raykova, and V. Vaikuntanathan. How to delegate and verify in public: Verifiable computation from attribute-based encryption. In *TCC*, pages 422–439, 2012.

[49] A. Shamir. IP = PSPACE. *J. ACM*, 39(4):869–877, Oct. 1992.

[50] E. Shen, E. Shi, and B. Waters. Predicate privacy in encryption systems. In *Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography*, TCC '09, pages 457–473, Berlin, Heidelberg, 2009. Springer-Verlag.

[51] E. Shi, J. Bethencourt, T.-H. H. Chan, D. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 350–364, Washington, DC, USA, 2007. IEEE Computer Society.

[52] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2000. IEEE Computer Society.

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

[54] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 139–148, New York, NY, USA, 2008. ACM.

[55] P. Williams, R. Sion, and A. Tomescu. Privatefs: a parallel oblivious file system. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 977–988, New York, NY, USA, 2012. ACM.