# Efficient Zero-Knowledge Proofs of Non-Algebraic Statements with Sublinear Amortized Cost

Zhangxiang Hu[1], Payman Mohassel[2], and Mike Rosulek[1]

[1] Oregon State University; {huz,rosulekm}@eecs.oregonstate.edu
[2] Yahoo Labs; pmohassel@yahoo-inc.com

**Abstract.** We describe a zero-knowledge proof system in which a prover holds a large dataset $M$ and can repeatedly prove NP relations about that dataset. That is, for any (public) relation $R$ and $x$, the prover can prove that $\exists w : R(M, x, w) = 1$. After an initial setup phase (which depends only on $M$), each proof requires only a constant number of rounds and has communication/computation cost proportional to that of a *random-access machine (RAM)* implementation of $R$, up to polylogarithmic factors. In particular, the cost per proof in many applications is sublinear in $|M|$. Additionally, the storage requirement between proofs for the verifier is constant.

## 1 Introduction

Zero-knowledge (ZK) proofs are a fundamental concept in cryptography and are used as a building block in numerous applications. ZK proofs allow a prover with the knowledge of a witness $w$ to prove statements of the form $\exists w : R(x, w) = 1$ to a verifier $V$, for a public NP statement $R$ and a public input $x$. The *soundness* of such a proof guarantees that a malicious prover cannot prove a false statement to a verifier, and the *zero-knowledge* property guarantees that a malicious verifier cannot learn any information about the witness except for validity of the proved statement.

Since the conception of zero-knowledge proofs [GMR89], a large body of work has focused on design of efficient constructions that are practical enough for use in practice. But until recently, all such constructions were practical only for proving statements about certain algebraic structures such as proving knowledge of and relations for discrete logarithms, RSA public keys, and bilinear equations [Sch90,CDS94,CM99,GS08].

The recent work of [JKO13] proposes a new approach based on garbled circuits (GC) that is suitable for general-purpose statements represented as boolean circuits. This is particularly powerful for proving non-algebraic statements, e.g., proving knowledge of $x$ such that $y = \mathsf{Sha256}(x)$ for a public value $y$. The construction is very efficient, only requiring a constant number of rounds and communication/computation cost that is similar to that of *semi-honest* 2PC based on garbled circuits (i.e., Yao's protocol). Given the recent advances in design &

implementation of circuit garbling techniques, these ZK proofs are scalable to statements with billions of gates.

*Need for ZK Proof of RAM Programs.* But the GC-based approach falls short when the statement being proven involves access to a large dataset committed by the prover. For instance, recall the problem solved by *zero-knowledge sets* [MRK03]: a prover commits to a set $S$ in an initial phase and is later able to prove membership and non-membership statements ($x \in S$, $x \notin S$) for any input $x$ without revealing additional information.

A natural extension is to prove membership for a (possibly private) value $x$ that satisfies a predicate $p$ without leaking any additional information about $x$ or the set $S$. For instance, the prover may need to prove knowledge of an $x \in S$ where $\mathsf{Sha256}(x) = y$ for a public $y$ in order to prove inclusion of a password in a password-file. Furthermore, to improve on storage cost, the prover may want to store his set $S$ in a *Bloom filter* [Blo70]. This would lead to major storage improvement, especially when considering the inevitable overhead caused by crypto for every bit of memory stored. Now, the prover needs to prove knowledge of an $x$ where $\mathsf{Sha256}(x) = y$ and where the Bloom filter stores a bit 1 at each of the locations $H_1(x), \ldots, H_k(x)$ (the $H_i$'s are the hash functions associated with the Bloom filter and can be public). Such a statement involves several hash evaluations and memory lookups. More generally, the prover may want to store its data in a data-structure of its own choice and still have efficient tools for proving statements about it.

In all of these scenarios, the statements being proven are naturally expressed as RAM programs whose running time is sublinear in the size of the large dataset. By comparison, directly applying a circuit-based approach (i.e., [JKO13]) would involve garbled circuits that are at least linear in the size of the large dataset.

*Existing Solutions for RAM-ZK.* One can combine the GC-based proof system of [JKO13] with the recent garbled RAM constructions [LO13,GHL+14] that directly garble RAM programs as opposed to circuits. But the existing constructions for garbled RAM are not efficient enough for practical use. In particular, one needs to perform cryptographic operations inside the garbled circuits for every step of RAM computation, which is a major bottleneck.

Finally, given that ZK proofs are a special case of secure two-party computation against malicious adversaries (*i.e.*, a malicious 2PC where one party, the verifier, has no input), we can obtain a solution by employing an efficient malicious 2PC for RAM programs [AHMR15] and not assigning one party any input. But for statistical security $2^{-s}$, such a proof would be a factor of $s$ more expensive than the semi-honest 2PC for the same RAM program, and the number of rounds would also be proportional to the running time of the RAM program.

### 1.1 Our Contribution

We propose a new solution for zero-knowledge proof of statements of the form $\exists w : R(M, x, w) = 1$ where $R$ is a RAM program and $M$ is its (large) memory.

Here, $M$ is committed upfront by the prover and can in general remain private from the verifier. Our construction is constant-round, and incurs online computation and communication cost that is linear in the running time of the RAM program (upto a polylogarithmic factor), competitive with the best *semi-honest* 2PC for RAM programs ([GKK+12]), and hence sublinear in $|M|$ for many applications of interest. Sublinear-time 2PC is not possible in general when expressing the NP relation as a boolean circuit. Furthermore, in our protocol the verifier maintains only constant storage space between multiple proofs.

Our construction combines an Oblivious RAM [GO96] and garbled circuits, but it avoids the use of cryptographic operations inside the garbled circuits as in current garbled-RAM constructions. Unlike previous 2PC constructions based on RAM computation [GKK+12,AHMR15], our construction requires only a constant number of rounds of interaction. We discuss the construction in more detail next.

## 2 Overview of the Protocol

*The JKO protocol.* Our starting point is the garbled-circuit-based ZK protocol of [JKO13], which we summarize here. To prove a statement $\exists w : R(x, w) = 1$ (for public $R$ and $x$), the protocol proceeds as follows:

1. The verifier generates a garbled circuit computing $R(x, \cdot)$. Using a committing oblivious transfer, the prover obtains the wire labels corresponding to his private input $w$. Then the verifier sends the garbled circuit to the prover.
2. The prover evaluates the garbled circuit, obtaining a single garbled output (wire label). He commits to this garbled output.
3. The verifier opens his inputs to the committing oblivious transfer, giving the prover all garbled inputs. From this, the prover can check whether the garbled circuit was generated correctly. If so, the prover opens his commitment to the garbled output; if not, the prover aborts.
4. The verifier accepts the proof if the prover's commitment holds the output wire label corresponding to TRUE.

Security against a cheating prover follows from the properties of the circuit garbling scheme. Namely, the prover commits to the output wire label before the circuit is opened, so the *authenticity* property of the garbling scheme ensures that he cannot predict the TRUE output wire label unless he knows a $w$ with $R(x, w) = \text{TRUE}$. Security against a cheating verifier follows from correctness of the garbling scheme. The garbled output of a *correctly generated* garbled circuit reveals only the output of the (plain) circuit, and this garbled output is not revealed until the garbled circuit was shown to be correctly generated.

Note that in this protocol, the prover evaluates the garbled circuit on an input which is completely known to him. This is the main reason that the garbled circuit used for evaluation can also be later opened and checked for correctness, unlike in the setting of cut-and-choose for general 2PC. Along the same lines, it was further pointed out in [FNO15] that the circuit garbling scheme need not

satisfy the *privacy* requirement of [BHR12], only the *authenticity* requirement. Removing the privacy requirement from the garbling scheme leads to a non-trivial reduction in garbled circuit size.

*Adapting to the ORAM setting, using constant rounds.* We follow roughly the RAM-2PC paradigm of [GKK+12,AHMR15], with some important differences. Let $\Pi$ be an Oblivious RAM program with memory $\widehat{M}$, that implements $R(M, x, \cdot)$.[3] We assume a trusted setup phase in which $\Pi$'s memory $\widehat{M}$ and state $st$ are initialized from $M$. The prover learns $\widehat{M}$, $st$, as well as a garbled encoding of these values (i.e., one wire label for each bit of memory & state); the verifier specifies the garbled encoding to be used (i.e., both wire labels for each bit). If we follow [GKK+12,AHMR15] strictly, we would have both parties repeatedly evaluate the next-memory-access circuit of $\Pi$, updating memory $\widehat{M}$, until it halts. However, this would result in a protocol with one round of interaction for each memory access of $\Pi$.

To see how to achieve the same effect in a constant number of rounds, imagine that when executing an ORAM program, the memory access pattern $\mathcal{I}$ is known in advance. Then it is possible to express the entire computation in a single circuit. The circuit includes many copies of the RAM program's next-memory-access circuit, but is wired together under the assumption that the memory accesses will be $\mathcal{I}$. For example, if $\mathcal{I}$ says that $\Pi$ writes to some memory block at time 2, and later reads from the same memory block at time 10, then the memory-output wires of subcircuit copy #2 will be connected to the memory-input wires of subcircuit copy #10, and so on.

We can leverage this optimization in our setting because the prover knows all (plaintext) inputs to $\Pi$, including the contents of memory and the ORAM state. Hence, the prover can execute $\Pi$ locally to determine the complete memory access pattern $\mathcal{I}$. Since $\Pi$ is an oblivious RAM, its access pattern $\mathcal{I}$ leaks no information about the inputs/memory/state, so the prover can safely send $\mathcal{I}$ to the verifier. Using $\mathcal{I}$, the verifier constructs a *single* garbled circuit $C_{x,\mathcal{I}}$ as described above. To prevent the prover from lying about the access pattern $\mathcal{I}$, the circuit recomputes the memory access pattern of $\Pi$ and compares it to (hard-coded) $\mathcal{I}$.

Hence, this setting admits a constant-round solution based on ORAM, but avoiding tools like garbled RAM [LO13,GHL+14] which incorporate expensive additional crypto circuitry into the garbled circuits.

*Reusing $M$ to perform many proofs.* We follow the approach of [AHMR15], where the prover stores the ORAM memory and ORAM state encoded as wire labels from the various garbled circuits. The idea is that these wire labels can be reused directly as inputs to subsequent circuits, avoiding oblivious transfers for garbled circuit input. However, some modifications are required to adapt this idea to our setting.

[3] We use $M$ to refer to the logical RAM memory, and $\widehat{M}$ to refer to the physical ORAM memory.

After evaluating a garbled circuit, the prover holds a garbled output encoding of ORAM state & memory. The *authenticity* property of the garbling scheme guarantees that the prover knows at most one valid label per wire. As soon as the garbled circuit is opened, however, the prover learns both labels for each wire and authenticity is lost. The output wire labels are no longer useful for input to subsequent circuits, as the prover can now feed arbitrary garbled state/memory into subsequent garbled circuits. We need a mechanism to restore authenticity on all wire labels that may be later used (this includes the ORAM internal state as well as all memory locations that are read or written by the garbled circuit).

Say the two wire labels on some output wire are $y_0$ and $y_1$, and that the prover knows only $y_b$. Let us call $y_0$ and $y_1$ the *temporary* wire labels, since they will soon be discarded. The verifier chooses a random function $h$ from a strongly universal hash family. Just before the garbled circuit is opened (clobbering wire-label authenticity), the parties perform a *private function evaluation (PFE)*, where the prover gives $y_b$, the verifier gives $h$, and the prover learns $h(y_b)$. After the PFE, the garbled circuit can be opened, revealing $y_0$ and $y_1$.

Define $y'_0 = h(y_0)$ and $y'_1 = h(y_1)$ to be the *permanent* wire labels for this wire. At the time of the PFE, the prover could not have guessed $y_{1-b}$, and so learned the output of $h$ on some point that was not $y_{1-b}$. From strong universality of $h$, even if $y_{1-b}$ is later revealed, $y'_{1-b} = h(y_{1-b})$ is still random from the prover's point of view. Hence the PFE "transfers" the authenticity guarantee from the temporary wire labels $y_0, y_1$ to the permanent ones $y'_0, y'_1$, preserving authenticity even after both of $y_0, y_1$ are revealed. Hence, $y'_0, y'_1$ are safe to use as input wire labels to a subsequent garbled circuit. We emphasize that all wire labels are used only in a single garbled circuit — we use the term "permanent" since these wire labels will be the long-term representation of the RAM program's memory between proof instances. (It may be many proof instances before a particular block of memory is next accessed.)

For technical reasons, the PFE needs to be committing with respect to the input $h$ (so that the verifier can later "open" the $h$ that was used). We suggest two efficient instantiations of committing-PFE for strongly universal families: one based on oblivious linear function evaluation (OLFE) [WW06] and one based on the string-select variant of OT presented in [KK12].

Note that all the PFE instances can be run in parallel hence, maintaining the constant round complexity of the overall protocol.

*Eliminating the verifier's storage requirement.* As described so far, the verifier is required to keep track of two wire labels for each bit of $\widehat{M}$, at all times. We can decrease this burden somewhat by letting the verifier derive these wire labels from a PRF. Let $s$ be a seed to a PRF. For simplicity, suppose a wire label encoding truth value $b$ on the $j$th bit of the $i$th memory block, last accessed at time $t$, is chosen as $\mathsf{PRF}(s, i\|j\|t\|b)$. In the actual protocol, the choice of wire labels is slightly more complicated.

Using this choice of wire labels, the verifier need only remember the last-access time of each block of $\widehat{M}$. However, this is still storage proportional to $|\widehat{M}|$. To reduce the storage even further, we "outsource" the maintenance of

these last-access times to the prover. Let $T[i]$ denote the last-access time of block $i$. We let the prover store the array $T$ authenticated by a Merkle tree for which the verifier remembers only the root node.[4]

Whenever the verifier is about to garble a circuit, he must be reminded of $T[i]$ for each memory block $i$ to be read by the RAM in its computation. We make the prover report each such $T[i]$ to the verifier, authenticating each value via the Merkle tree. The ORAM circuit performs some reads & writes in $\widehat{M}$, so $T$ and the Merkle tree are updated accordingly, for each memory block that was accessed. Note that all accesses to the Merkle tree are done at the same time (in parallel), and similarly for the updates at the end of the execution.

Overall, accessing/updating the authenticated array $T$ adds polylogarithmic (in $|\widehat{M}|$) communication/computation overhead and only a small constant number of rounds to the protocol. Instead of remembering two wire labels for each bit of $\widehat{M}$, the verifier need now remember only a PRF seed and the root of a Merkle tree.

## 3 Preliminaries

Throughout the paper, we let $k \in \mathbb{N}$ be the security parameter. We say a function $\epsilon : \mathbb{N} \to [0, 1]$ is negligible if for any polynomial $p$, there exists a large enough $k'$ such that for all $k > k'$, $\epsilon(k) < 1/p(k)$. Also, for a integer $n$, we define $[n] = \{1, 2, \ldots, n\}$.

### 3.1 ZK Proofs & Other Standard Functionalities

Here we define the variant of ZK proofs that we achieve, as well as other standard ideal functionalities used in our protocol.

*Zero-knowledge proofs:* Roughly speaking, a zero-knowledge proof is an interactive protocol in which a party $P$ (the prover) can prove to another party $V$ (the verifier) that some NP statement $x$ is true by using a valid witness $w$, leaking no information about $w$ (except that the statement $x$ is true).

More precisely, for any language $\mathcal{L} \in \mathsf{NP}$ with some binary relation $\mathcal{R}_{\mathcal{L}}$, for all valid instances $x \in \mathcal{L}$, there exists a string $w$ such that $\mathcal{R}_{\mathcal{L}}(x, w) = 1$. Otherwise, if $x \notin \mathcal{L}$, then for all string $w$ we have $\mathcal{R}_{\mathcal{L}}(x, w) = 0$.

The ideal functionality $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ is defined in figure 1, which allows for many proofs to reference a common (secret) value $M$.

*Commitment:* The commitment functionality $\mathcal{F}_{\mathsf{com}}$ is described in Figure 2. It allows a party to commit to a secret value at one time and reveal that value at a later time.

[4] More generally, $T$ can be stored in any authenticated data structure that provides small storage for the verifier.

$\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ is parametrized by a relation $\mathcal{R}$. It involves two parties: a prover $P$ and a verifier $V$.

- Setup: On input $(\textsc{init}, M)$ from $P$, if no previous INIT command has been given, then $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ stores $M$ internally.
- Proof: On input $(\textsc{prove}, sid, x, w)$ from $P$, if $\mathcal{R}(M, x, w) = 1$, output $(\textsc{accept}, sid, x)$ to $V$.

**Fig. 1.** Ideal functionality $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ for zero-knowledge proofs of NP-relation $\mathcal{R}$

Let $\mathcal{M}$ denote the space of valid messages. $\mathcal{F}_{\mathsf{com}}$ receives input from party $P$ and sends output message to party $V$. It consists of two phases: Commit and Open.

- Commit: On input $(\textsc{commit}, m)$ from $P$ with $m \in \mathcal{M}$, if there is no value $m$ already stored in memory, then $\mathcal{F}_{\mathsf{com}}$ stores $m$ internally and outputs COMMITTED to party $V$.
- Open: On input OPEN from $P$, if value $m$ exists in memory, then $\mathcal{F}_{\mathsf{com}}$ outputs $(\textsc{opened}, m)$ to party $V$.

**Fig. 2.** Ideal functionality $\mathcal{F}_{\mathsf{com}}$ for commitment

- Initialization: $\mathcal{F}_{\mathsf{otc}}$ takes private input $E$ (an $m \times 2$ array) from party $V$ and the private input $\sigma \in \{0,1\}^m$ from party $P$, then stores $(E, \sigma)$ internally and output COMMITTED.
- Transfer: On command TRANSFER from $V$, $\mathcal{F}_{\mathsf{otc}}$ sends $(\textsc{transferred}, E|_\sigma)$ to $P$.
- Open: On command OPEN from $V$, $\mathcal{F}_{\mathsf{otc}}$ sends $(\textsc{opened}, E)$ to $P$.

**Fig. 3.** Ideal functionality $\mathcal{F}_{\mathsf{otc}}$ for committing oblivious transfer. Notation $E|_\sigma$ is defined in Section 3.4.

- Initialization: On input $(\textsc{init}, N)$ from party $V$, $\mathcal{F}_{\mathsf{Aut}}$ initialize an array $T$ of size $N$. For each $T[i]$, $i \in \{1, \ldots, N\}$, set $T[i] = 0$.
- Update: On input $(\textsc{update}, id, data)$ from party $V$, set $T[id] = data$ and output $(\textsc{updated}, id, data)$ to both parties.
- Open: On input $(\textsc{access}, id)$ from party $V$, where $id \in \{1, \ldots, N\}$, send $(\textsc{accessed}, id, T[id])$ to $V$.

**Fig. 4.** Ideal functionality $\mathcal{F}_{\mathsf{Aut}}$ for authenticated array access.

*Committing Oblivious Transfer:* The definition of committing oblivious transfer was first given by Kiraz and Schoenmakers [KS06]. In the general OT protocol, party $V$ inputs a description of wire labels $E$ and party $P$ has input $\sigma$. After running oblivious transfer, $P$ receives a garbled encoding of $\sigma$ under the encoding $E$. See Section 3.4 for more details about the wire-label syntax used in the figure.

$\mathcal{F}_{\mathsf{cpfe}}$ is parametrized by a class of functions $\mathcal{H}$, with each $h \in \mathcal{H}$ having a common domain $A$.

- Evaluation: On input $h \in \mathcal{H}$ from party $V$ and input $x \in A$ from party $P$, give output $h(x)$ to party $P$. Remember $h$ internally.
- Open: On input OPEN from party $V$, give output $h$ to party $P$.

**Fig. 5.** Ideal functionality $\mathcal{F}_{\mathsf{cpfe}}$ for committing private function evaluation.

The "committing" aspect of committing OT allows party $V$ to reveal $E$ at a later time. The ideal functionality $\mathcal{F}_{\mathsf{otc}}$ is defined in Figure 3.

*Authenticated Array:* The functionality $\mathcal{F}_{\mathsf{Aut}}$ in Figure 4 simply provides storage of an array, in which the party $V$ has control over modifications. Such a functionality becomes interesting in our setting when it is realized by a protocol with minimal (constant) storage for party $V$. A simple approach is to use an authenticated Merkle-tree, with $V$ storing only the root of the tree.

### 3.2 Committing Private Function Evaluation (of a Strongly Universal Family)

Private function evaluation (PFE) takes input $h$ (a function) from a sender, input $x$ from a receiver, and gives output $h(x)$ to the receiver. We define and use a committing variant of PFE in which the sender can later reveal the $h$ that was used. The formal description is given in Figure 5.

In our final protocol, we require committing PFE supporting a **strongly universal** class $\mathcal{H}$ of functions. Suppose each function $h$ in $\mathcal{H}$ is of the form $h : A \to B$. Then $\mathcal{H}$ is strongly universal if for all distinct $a, a' \in A$ and all (possibly equal) $b, b' \in B$,

$$\Pr_{h \leftarrow \mathcal{H}}[h(a) = b \mid h(a') = b'] = 1/|B|$$

Below we suggest several efficient choices for PFE of strongly universal families:

*Using 1-out-of-2 OT:* Let $X$ be an $n \times 2$ matrix of length-$m$ strings. For such an $X$, define the function $h_X : \{0,1\}^n \to \{0,1\}^m$ via:

$$h_X(z) = \bigoplus_{i=1}^{n} X_{i,z_i}$$

Then the class $\mathcal{H} = \{h_X \mid X \in (\{0,1\}^m)^{n \times 2}\}$ is strongly universal.

A simple protocol for private function evaluation of $\mathcal{H}$ uses standard 1-out-of-2 oblivious transfer (of strings) in the following way: For $i = 1$ to $n$, the sender gives input $X_{i,0}$ and $X_{i,1}$ as input to an instance of OT. The receiver gives input $z_i$ and obtains $r_i = X_{i,z_i}$. Finally the receiver outputs $r_1 \oplus \cdots \oplus r_n$.

Technically, this protocol is *not* a secure PFE for the family $\mathcal{H}$, because the receiver learns more than $h_X(z)$. In particular, the receiver learns various $X_{i,z_i}$ values. However, the protocol suffices for our needs, by considering slightly relaxed definitions. Let $\mathcal{H}$ be a family of *pairs* of functions. We write $(h, \widehat{h}) \in \mathcal{H}$, where $h : A \to B$ and $\widehat{h} : A \to \widehat{B}$. Then we say that $\mathcal{H}$ is **modified strongly universal** if for all distinct $a, a' \in A$ and all (possibly equal) $b \in B$, $\widehat{b}' \in \widehat{B}$:

$$\Pr_{(h,\widehat{h}) \leftarrow \mathcal{H}} [h(a) = b \mid \widehat{h}(a') = \widehat{b}'] = 1/|B|$$

The family $\mathcal{H} = \{h_X\}$ we described above satisfies this definition, taking $\widehat{h}_X(z) = (X_{1,z_1}, \ldots, X_{n,z_n})$. That is, the value of $h_X(z')$ is distributed uniformly even after fixing the output of $\widehat{h}_X(z)$ for $z \neq z'$.

Then the protocol just described is secure for a variant of Figure 5 in which an adversarial receiver obtains not $h(x)$ but $\widehat{h}(x)$. It should be clear that such a modified functionality suffices for our eventual usage of $\mathcal{F}_{\mathsf{cpfe}}$ when the family $\mathcal{H}$ is *modified* strongly universal. For simplicity we write our eventual ZK protocol in terms of the simpler $\mathcal{F}_{\mathsf{cpfe}}$ defined in Figure 5.

Furthermore, when the underlying OT protocol is a committing OT, then the PFE protocol is also committing in a natural way (with the sender revealing all committed-OT inputs). We note that this protocol is essentially the "string-select oblivious transfer" protocol of [KK12] but without the final verification step which is not needed here.

*Using OLFE:* In a finite field $\mathbb{F}$, the class of functions of the form $x \mapsto ax + b$ is strongly universal (with $a, b \in \mathbb{F}$). A private function evaluation for this class therefore accepts $a, b \in \mathbb{F}$ from the sender, $x \in \mathbb{F}$ from the receiver, and gives output $ax + b$ to the receiver. Such a functionality is already known by the name of *oblivious linear function evaluation* (OLFE or OLE) [WW06].

The state of the art for malicious-secure OLFE is due to the general protocol of Ishai, Prabhakaran, and Sahai [IPS09] for evaluating arithmetic circuits in the OT-hybrid model. Since OLFE can be represented by an arithmetic circuit with just 2 gates, their construction yields an OLFE protocol with (amortized) constant number of field elements communicated per OLFE and computation roughly $O(\log k)$ field operations per OLFE.

The general construction of [IPS09] combines an outer MPC protocol among imaginary parties and an inner 2PC protocol between the real parties. It is easy to see that if the inner protocol is committing, so is the overall protocol.

### 3.3 Oblivious RAM program

Oblivious RAM (ORAM) programs were first introduced by Goldreich & Ostrosvsky [GO96]. ORAM allows a client to hide its access pattern and data to the server. In this work we freely identify a RAM program $\Pi$ with its deterministic *next-instruction* circuit. We use $M$ to represent the logical memory of a RAM program and $\widehat{M}$ to indicate the physical memory array in Oblivious RAM

program. We consider all memory to be split into **blocks**, where $M[i]$ denotes the $i$th block of $M$.

Without loss of generality, we assume that the RAM program is deterministic. Although constructions of oblivious RAM require randomness, we can allow the prover to provide that randomness as part of the witness $w$. Thus, we think of $w$ as $w = w_{\mathsf{real}} \| r$, where $w_{\mathsf{real}}$ is the actual witness to the statement and $r$ is randomness used by the ORAM. An honest prover will choose $r$ uniformly so that the ORAM memory access sequence hides private information. Allowing a corrupt prover to choose $r$ does not compromise soundness in practical ORAM constructions (e.g., [SvDS+13]) — it only affects the probability of an overflow error event (in which case we can have the ORAM circuit output FALSE).

Let the next-instruction circuit $\Pi$ have syntax:

$$(inst, st, block) \leftarrow \Pi(st, \Sigma, block)$$

where $\Sigma$ is external input, $st$ is the ORAM state, $block$ is the memory blocks and $inst$ represents a RAM memory access instruction, which must have one of the following forms: $(\text{READ}, i)$, $(\text{WRITE}, i)$, or $(\text{HALT}, z)$, where $i$ is the index of a memory block.

The execution of an ORAM program $\Pi$ on input $(x, w)$ using memory $\widehat{M}$ is as follows:

$\underline{\text{RAMEval}(\Pi, \widehat{M}, (x, w), st)}$
$\quad \mathcal{I} := \emptyset$
$\quad (inst, st, block) := \Pi(st, (x, w), \perp)$
$\quad$ do until inst has the form $(\text{HALT}, \text{TRUE})$:
$\qquad block :=$ [if $inst = (\text{READ}, id)$ then $\widehat{M}[id]$ else $\perp$]
$\qquad (inst, st, block) := \Pi(st, \perp, block)$
$\qquad$ if $inst = (\text{WRITE}, id)$ then $\widehat{M}[id] := block$
$\qquad \mathcal{I} := \mathcal{I} \| inst$
$\quad$ output $\mathcal{I}$

Note that we have RAMEval output the access sequence $\mathcal{I}$. We say $\mathcal{I}$ is an **accepting access sequence** if the last instruction in $\mathcal{I}$ is $(\text{HALT}, \text{TRUE})$.

We assume a function Initialize with syntax:

$$(\widehat{M}, st) \leftarrow \mathsf{Initialize}(1^k, M)$$

This function returns the initial value of $st$ and also the initialized physical memory array $\widehat{M}$ encoding the logical memory $M$.

The security definition of an oblivious RAM program $\Pi$ requires that the memory access sequence $\mathcal{I}$ does not leak information about the data set $M$ or witness $w_{\mathsf{real}}$. More formally:

**Definition 1.** *We say that $\Pi$ is a* **secure ORAM** *if there exists an efficent $\mathcal{S}$ such that, for all $M$, all $(\widehat{M}, st) \leftarrow \mathsf{Initialize}(1^k, M)$, all $(x, w_{real})$ such that $\mathcal{R}(M, x, w_{real}) = 1$ and for all PPT $\mathcal{A}$, the following difference:*

$$\left| \Pr[\mathcal{A}(\mathcal{S}(1^k, |\widehat{M}|, x) = 1] - \Pr_r[\mathcal{A}(RAMEval(\Pi, \widehat{M}, (x, w_{real} \| r), st)) = 1] \right|$$

*is negligible in $k$.*

Any RAM program can be converted into an oblivious one satisfying our definitions, using standard constructions [SvDS+13,CP13]. Note that $\mathcal{I}$ (the output of RAMEval) contains only the memory *locations* and not the *contents* of memory. Hence, we do not require the ORAM construction to encrypt/decrypt memory contents.

### 3.4 Garbling Scheme

We assume some familiarity with standard constructions of garbled circuits. We employ the abstraction of garbling scheme [BHR12] introduced by Bellare *et al.* , but we use a slightly different syntax for our needs.

We represent a set of wire labels on $m$ wires via a $m \times 2$ array $W$. For each wire $i$, $W[i,0] \in \{0,1\}^k$ and $W[i,1] \in \{0,1\}^k$ are two wire labels that encode FALSE and TRUE, respectively. For a truth value $x$, the corresponding wire labels are defined as $W|_x = (W[1,x_1], \ldots, W[m,x_m])$.

Our protocol adopts the idea of [MGFB14,AHMR15] of re-using wire labels between different garbled circuits. We require somewhat different syntax for the garbling scheme in order to facilitate this reuse.

For our purposes, a garbling scheme consists of the following algorithms:

- $\mathsf{Gb}(1^k, f, E, D) \rightarrow F$. Takes as input a boolean circuit $f$, descriptions of input wire labels $E$ and output wire labels $D$, and outputs a garbled circuit $F$.
- $\mathsf{En}(E, x) \rightarrow X = E|_x$. Takes as input description of input wire labels $E$, a plaintext input $x$ and outputs a garbled input $X$. In our schemes, encoding is always done via $E|_x$.
- $\mathsf{Ev}(F, X) \rightarrow Y$. Takes as input a garbled circuit $F$ and a garbled input $X$ and returns a garbled output $Y$.
- $\mathsf{Chk}(f, F, E) \rightarrow D$ or $\perp$. Takes as input a boolean circuit, a (purported) garbled circuit $F$ and input wire label desription $E$ and outputs either $D$ or an error indicator $\perp$.

The correctness and security condition of garbling scheme we require here is slightly different from those given in [BHR12], but any garbling scheme that meet the requirements in [BHR12] also works well for our definitions.

**Definition 2.** *A garbling scheme satisfies* **correctness** *if:*

1. *For all circuits $f$, circuit-inputs $x$, and valid wire label descriptions $E, D$,*

$$\mathsf{Chk}(f, F, E) = D \text{ whenever } F \leftarrow \mathsf{Gb}(1^k, f, E, D)$$

2. *For all circuits $f$, (possibly malicious) garbled circuits $F$ and wire-label descriptions $E$,*

$$\mathsf{Ev}(F, E|_x) = D|_{f(x)} \text{ whenever } \mathsf{Chk}(f, F, E) = D \neq \perp$$

**Definition 3.** *Let $\mathcal{W}$ denote the uniform distribution of $m \times 2$ matrices as described above. A garbling scheme has* **authenticity** *if for every circuit $f$, circuit-input $x$, and PPT algorithm $\mathcal{A}$, the following probability:*

$$\Pr[\exists y \neq f(x), \tilde{D} = D|_y : E \leftarrow \mathcal{W}, F \leftarrow \mathsf{Gb}(1^k, f, E, D), \tilde{D} = \mathcal{A}(F, E|_x)]$$

*is negligible in $k$.*

The above definition says that when given $F$ and $E|_x$, there is no efficient adversary that can forge valid output wire labels $\tilde{D}$ such that $\tilde{D} \neq D|_{f(x)}$.

We emphasize that the garbling scheme we use here only requires only the authenticity property and not any privacy property. Hence, the protocol may use a more efficient and simpler garbling scheme (*e.g.*, the "privacy-free" constructions of [FNO15,ZRE15]).

## 4 Zero-Knowledge by Oblivious RAM

### 4.1 Notation and Helper Routines

*ORAM components:* Let $\mathcal{I}$ be an ORAM memory access sequence. We define $\mathsf{read}(\mathcal{I}) = \{i \mid (\textsc{read}, i) \in \mathcal{I}\}$, $\mathsf{write}(\mathcal{I}) = \{i \mid (\textsc{write}, i) \in \mathcal{I}\}$, and $\mathsf{access}(\mathcal{I}) = \mathsf{read}(\mathcal{I}) \cup \mathsf{write}(\mathcal{I})$; *i.e.*, the indices of blocks that are read/write/accessed in $\mathcal{I}$. If $S = \{s_1, \ldots, s_n\}$ is a set of memory-block indices, then we define $M[S] = (M[s_1], \ldots, M[s_n])$.

Let $\Pi$ denote the next-instruction circuit of an ORAM. Given a zero-knowledge statement $x$ and ORAM access sequence $\mathcal{I}$, we let circuit $C_{x,\mathcal{I}}$ denote the following circuit:

$\underline{C_{x,\mathcal{I}}(st, w, \widehat{M}[\mathsf{read}(\mathcal{I})]):}$
$\quad (inst, st, block) := \Pi(st, (x, w), \bot)$
$\quad$ for $i = 1$ to $|\mathcal{I}| - 1$:
$\qquad$ if $\mathcal{I}[i] = (\textsc{read}, id)$ then:
$\qquad\quad (st, inst, \bot) \leftarrow \Pi(st, \bot, \widehat{M}[id])$
$\qquad$ if $\mathcal{I}[i] = (\textsc{write}, id)$ then:
$\qquad\quad (st, inst, block) \leftarrow \Pi(\mathsf{st}, \bot, \bot)$
$\qquad\quad \widehat{M}[id] = block$
$\qquad \mathcal{I}' := \mathcal{I}' \| inst$
$\quad z := [\mathcal{I} \stackrel{?}{=} \mathcal{I}']$
$\quad$ return $(st, z, \widehat{M}[\mathsf{access}(\mathcal{I})])$

As described in Section 2, $C_{x,\mathcal{I}}$ is the circuit that will be garbled in the protocol. Note that both $x$ and $\mathcal{I}$ are hard-coded into $C_{x,\mathcal{I}}$. Also, the circuit verifies that $\mathcal{I} = \mathcal{I}'$, and this entails checking the correctness of the witness since the final element of $\mathcal{I}$ is $(\textsc{halt}, \textsc{true})$.

*Garbling notation:* The circuit $C_{x,\mathcal{I}}$ has 3 logical inputs and 3 logical outputs, and we must distinguish among them. When garbling the circuit via $F \leftarrow \mathsf{Gb}(C_{x,\mathcal{I}}, E, D, 1^k)$, we denote by $E$ a description of input wire labels (*i.e.*, two labels per wire) and $D$ a description of output wire labels. We write $E = E_{\mathsf{st}} \| E_{\mathsf{wit}} \| E_{\mathsf{mem}}$, denoting the corresponding input wire labels for state, witness, and memory blocks, respectively. We define $D = D_{\mathsf{st}} \| D_{\mathsf{z}} \| D_{\mathsf{mem}}$ similarly. When referring to a specific memory block $i$, we use notation $E_{\mathsf{mem},i}$ and $D_{\mathsf{mem},i}$.

We use $X$ to denote the prover's garbled input, and $Y$ to denote the prover's garbled output (*i.e.*, one label per wire). As above, we define $X_{\mathsf{st}}$, $X_{\mathsf{wit}}$, $X_{\mathsf{mem}}$, $Y_{\mathsf{st}}$, $Y_{\mathsf{z}}$, $Y_{\mathsf{mem}}$. Finally, we have the prover maintain an array $R_{\mathsf{mem}}$ at all times, containing the current wire labels for all of the ORAM memory $\widehat{M}$.

For an overview of the notation used in the protocol, see Figure 6.

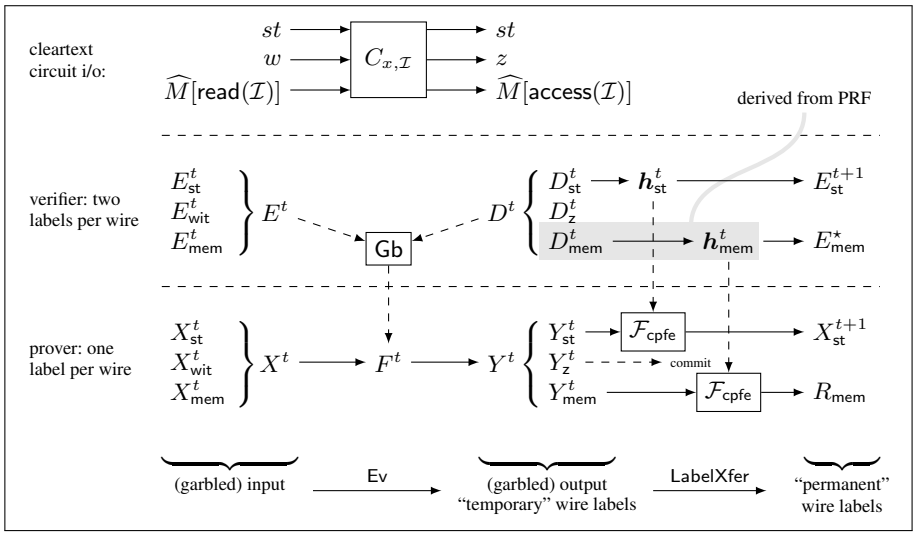


**Fig. 6.** Summary of variables and notation used in the protocol.

*Temporary & permanent wire labels.* Recall from Section 2 that the output wire labels of a circuit are "temporary" in the sense that their authenticity is lost when the garbled circuit is opened. We use PFE to transfer the authenticity property of these temporary wire labels to a different set of "permanent" wire labels.

We transfer authenticity with the $\mathsf{LabelXfer}$ subprotocol, where $Y$ is a list of "temporary" wire labels (*i.e.*, one label per wire), and $\boldsymbol{h}$ is a list of elements from a strongly universal hash family $\mathcal{H}$.

prot $\mathsf{LabelXfer}(Y, \boldsymbol{h})$:
  for $i = 1$ to $|Y|$ (in parallel):

$V$ sends $Y[i]$ and $P$ sends $\boldsymbol{h}[i]$ to an instance of $\mathcal{F}_{\mathsf{cpfe}}$
$P$ receives output $Z[i] := \boldsymbol{h}[i](Y[i])$
$P$ outputs $Z$

Note that all instances of $\mathcal{F}_{\mathsf{cpfe}}$ are run in parallel and hence the protocol remains constant-round given that $\mathcal{F}_{\mathsf{cpfe}}$ is itself constant-round.

*Selecting wire labels.* Now let's consider how the verifier generates wire labels for the circuit. Recall from Section 2 that the verifier uses a PRF to generate wire labels corresponding to the ORAM memory, in order to reduce storage.

Since permanent wire labels are derived by applying strongly universal functions to temporary wire labels, the verifier must also select strongly universal functions using the PRF to be able to reconstruct the choice of functions later.

Let $s$ be the seed to a $\mathsf{PRF}$. The verifier derives the *temporary* wire labels for a set $S$ of memory block indices, last updated at time $t$, via the subroutine $\mathsf{TempMemLabels}$. The verifier derives the choice of strongly universal functions via the subroutine $\mathsf{GenH}$.

Finally, the verifier derives the *current, permanent* wire labels for a set $S$ of memory block indices via the subprotocol $\mathsf{PermMemLabels}$. Since each block may have been last accessed a different time, the authenticated array $\mathcal{F}_{\mathsf{Aut}}$ is referenced. For each block, the most recent temporary wire labels and strongly universal functions are reconstructed to derive the permanent wire labels.

```
func TempMemLabels(S, t):
  D := ∅
  for i ∈ S:
    for j ∈ {1, ..., l}, b ∈ {0, 1}:
      D_i[j, b] = PRF(s, 0‖i‖j‖t‖b)
    D := D‖D_i
  return D

func GenH(S, t):
  h = ∅
  for i ∈ S:
    for j ∈ {1, ..., l}:
      h_i[j] = PRF(s, 1‖i‖j‖t)
    h := h‖h_i
  return h

prot PermMemLabels(S):
  E := ∅
  for all i in S (in parallel):
    send (ACCESS, i) to F_Aut
    receive t_i := T[i]
    D_i := TempMemLabels({i}, t_i)
    h_i := GenH({i}, t_i)
    E_i := h_i(D_i)
    E := E‖E_i
  return E
```

When $\boldsymbol{h}$ is an array of functions and $D$ is a matrix of wire labels, the notation $\boldsymbol{h}(D)$ refers to the matrix $E$ whose entries are $E[j,b] = \boldsymbol{h}[j](D[j,b])$.

### 4.2 Detailed protocol

Now we present the full protocol $\pi$. We refer to the prover as $P$ and the verifier as $V$. The setup phase uses the initialization functionality $\mathcal{F}_{\mathsf{init}}$ defined in Figure 7.

- Initialize: On command $(\textsc{init}, M)$ from $P$ and $(\textsc{init}, D_{\mathsf{st}}, D_{\mathsf{mem}})$, where $M$ is logical ORAM memory, and $D_{\mathsf{st}}$ & $D_{\mathsf{mem}}$ are wire label descriptions, run $(st, \widehat{M}) \leftarrow \mathsf{Initialize}(1^k, M)$. Give output $(st, \widehat{M}, D_{\mathsf{st}}|_{st}, D_{\mathsf{mem}}|_{\widehat{M}})$ to $P$.
- Open: On command $\textsc{open}$ from $V$, give output $(D_{\mathsf{st}}, D_{\mathsf{mem}})$ to $P$.

**Fig. 7.** Ideal functionality $\mathcal{F}_{\mathsf{init}}$ for initializing an ORAM program along with wire labels.

**Setup:** On input $M$ for prover $P$, let $N$ denote the number of blocks in the ORAM encoding of $M$. Then both parties do the following:

1. $V$ picks random wire label descriptions $D^0_{\mathsf{st}}$ and computes $D^0_{\mathsf{mem}} = \mathsf{TempMemLabels}([N], 0)$. $V$ also chooses a random PRF seed $s \leftarrow \{0,1\}^k$.
2. $P$ sends $(\textsc{init}, M)$ to $\mathcal{F}_{\mathsf{init}}$; $V$ sends $(\textsc{init}, D^0_{\mathsf{st}}, D^0_{\mathsf{mem}})$ to $\mathcal{F}_{\mathsf{init}}$. $P$ receives output $(st, \widehat{M}, Y^0_{\mathsf{st}} = D^0_{\mathsf{st}}|_{st}, Y^0_{\mathsf{mem}} = D^0_{\mathsf{mem}}|_{\widehat{M}})$.
3. **[Transfer wire-label authenticity]:**[5]
   (a) $V$ picks random vector $\boldsymbol{h}^0_{\mathsf{st}}$ of strongly universal functions and sets $E^1_{\mathsf{st}} = \boldsymbol{h}^0_{\mathsf{st}}(D^0_{\mathsf{st}})$. The parties perform subprotocol $\mathsf{LabelXfer}(Y^0_{\mathsf{st}}, \boldsymbol{h}^0_{\mathsf{st}})$, with $P$ obtaining output $\boldsymbol{h}^0_{\mathsf{st}}(Y^0_{\mathsf{st}})$ which he stores as $X^1_{\mathsf{st}}$.
   (b) $V$ picks vector $\boldsymbol{h}^0_{\mathsf{mem}} = \mathsf{GenH}([N], 0)$ and the parties perform subprotocol $\mathsf{LabelXfer}(Y^0_{\mathsf{mem}} \boldsymbol{h}^0_{\mathsf{mem}})$. $P$ receives output $\boldsymbol{h}^0_{\mathsf{mem}}(Y^0_{\mathsf{mem}})$ which he stores as $R_{\mathsf{mem}}$.
   (c) $V$ sends $\textsc{open}$ to $\mathcal{F}_{\mathsf{init}}$, and $P$ receives output $(D^0_{\mathsf{st}}, D^0_{\mathsf{mem}})$.
4. $P$ sends $(\textsc{init}, N)$ to $\mathcal{F}_{\mathsf{Aut}}$ to initialize authenticated array $T$ (with $T[i] = 0$ for all $i$).

**Proofs:** On input $(x, w)$ for the prover, let this be the $t$th such proof. The parties do the following:

4. **[ORAM Evaluation]:** $P$ runs $\mathcal{I} \leftarrow \mathsf{RAMEval}(\Pi, \widehat{M}, x, w, st)$, then sends $(x, \mathcal{I})$ to $V$. $V$ aborts if $\mathcal{I}$ is not an accepting access sequence. Note that $\mathsf{RAMEval}$ modifies $\widehat{M}$ for the prover.
5. **[Garbling the circuit]:** $V$ generates a garbled circuit as follows:
   (a) $V$ chooses input wire labels to the circuit as follows: $E^t_{\mathsf{wit}}$ are chosen randomly. $E^t_{\mathsf{mem}}$ are chosen as $E^t_{\mathsf{mem}} \leftarrow \mathsf{PermMemLabels}(\mathsf{read}(\mathcal{I}))$. Recall that $E^t_{\mathsf{st}}$ has been set previously.
   (b) $V$ chooses output wire labels $D^t_{\mathsf{z}}$ and $D^t_{\mathsf{st}}$ randomly, and chooses $D^t_{\mathsf{mem}} = \mathsf{TempMemLabels}(\mathsf{access}(\mathcal{I}), t)$.
   (c) $V$ sets $E^t = E^t_{\mathsf{st}} \| E^t_{\mathsf{wit}} \| E^t_{\mathsf{mem}}$, sets $D^t = D^t_{\mathsf{st}} \| D^t_{\mathsf{z}} \| D^t_{\mathsf{mem}}$, then invokes garbling algorithm $F^t \leftarrow \mathsf{Gb}(1^k, C_{x,\mathcal{I}}, E^t, D^t)$.
6. **[Evaluating garbled circuit]:**

[5] This step could be easily incorporated into $\mathcal{F}_{\mathsf{init}}$, but is written separately so that the remainder of the protocol has no edge-cases involving $t = 0$.

(a) The parties invoke $\mathcal{F}_{\mathsf{otc}}$ with $P$ giving input $w$ and $V$ giving input $E^t_{\mathsf{wit}}$. $P$ receives $X^t_{\mathsf{wit}} = E^t_{\mathsf{wit}}|_w$. Additionally, $P$ finds $X^t_{\mathsf{st}}$ in its memory and sets $X^t_{\mathsf{mem}} = R_{\mathsf{mem}}[\mathsf{read}(\mathcal{I})]$.
(b) $V$ sends $F^t$ to $P$, and $P$ evaluates the garbled circuit $Y^t \leftarrow \mathsf{Ev}(F^t, X^t)$.
(c) $P$ commits to $Y^t_{\mathsf{z}}$ (a single wire label) under $\mathcal{F}_{\mathsf{com}}$.

7. **[Transfer wire-label authenticity]:**
(a) $V$ picks random vector $\boldsymbol{h}^t_{\mathsf{st}}$ of strongly universal functions and sets $E^{t+1}_{\mathsf{st}} = \boldsymbol{h}^t_{\mathsf{st}}(D^t_{\mathsf{st}})$. The parties perform subprotocol $\mathsf{LabelXfer}(Y^t_{\mathsf{st}}, \boldsymbol{h}^t_{\mathsf{st}})$, with $P$ obtaining output $\boldsymbol{h}^t_{\mathsf{st}}(Y^t_{\mathsf{st}})$ which he stores as $X^{t+1}_{\mathsf{st}}$.
(b) $V$ picks vector $\boldsymbol{h}^t_{\mathsf{mem}} = \mathsf{GenH}(\mathsf{access}(\mathcal{I}), t)$ and the parties perform subprotocol $\mathsf{LabelXfer}(Y^t_{\mathsf{mem}}, \boldsymbol{h}^t_{\mathsf{mem}})$. $P$ receives output $\boldsymbol{h}^t_{\mathsf{mem}}(Y^t_{\mathsf{mem}})$ which he stores as $R_{\mathsf{mem}}[\mathsf{access}(\mathcal{I})]$.

8. **[Check garbled circuit]:**
(a) $V$ sends OPEN to the $\mathcal{F}_{\mathsf{otc}}$-instance from time $t$, and $P$ receives output $E^t_{\mathsf{wit}}$.
(b) $V$ sends OPEN to the PFE-instances used for the state wire labels in time $t-1$. The prover thus obtains $\boldsymbol{h}^{t-1}_{\mathsf{st}}$ and sets $E^t_{\mathsf{st}} = \boldsymbol{h}^{t-1}_{\mathsf{st}}(D^{t-1}_{\mathsf{st}})$.
(c) For each $i \in \mathsf{read}(\mathcal{I})$, verifier sends OPEN to the PFE-instances used for memory block $i$ in time $T[i]$. The prover thus obtains $\boldsymbol{h}^{T[i]}_{\mathsf{mem},i}$ and sets $E^t_{\mathsf{mem},i} = \boldsymbol{h}^{T[i]}_{\mathsf{mem},i}(D^{T[i]}_{\mathsf{mem},i})$.
(d) The verifier sets $E^t = E^t_{\mathsf{st}} \| E^t_{\mathsf{wit}} \| E^t_{\mathsf{mem}}$ and runs $D^t = \mathsf{Chk}(C_{x,\mathcal{I}}, F^t, E^t)$. If the result is $\perp$, then $V$ aborts. Otherwise, $V$ opens his commitment to $Y^t_{\mathsf{z}}$.

9. **[Check prover's output]:** $V$ checks whether $Y^t_{\mathsf{z}} = D^t_{\mathsf{z}}|_{\text{TRUE}}$. If not, then $V$ aborts the protocol. Otherwise, $V$ outputs $(\text{ACCEPT}, t, x)$.

10. **[Update $T$]:** For all $i \in \mathsf{access}(\mathcal{I})$ (in parallel), $V$ sends $(\text{UPDATE}, i, t)$ to $\mathcal{F}_{\mathsf{Aut}}$.

*Other discussion.* Our protocol is written in a hybrid model with access to various setup functionalities. In particular, $\mathcal{F}_{\mathsf{cpfe}}$ is a *reactive* functionality, and our protocol involves many ($O(|\widehat{M}|)$) instances of $\mathcal{F}_{\mathsf{cpfe}}$ that remain "active" between ZK proofs. We have shown how the verifier's *inputs* to the $\mathcal{F}_{\mathsf{cpfe}}$ instances can be derived from a PRF, eliminating the need to explicitly store them. However, when these $\mathcal{F}_{\mathsf{cpfe}}$ instances are realized by concrete protocols, both parties are required to keep internal state between the PFE phase and opening phase. Hence, the verifier's *random coins* for the $\mathcal{F}_{\mathsf{cpfe}}$-protocols should also be derived from a PRF. In that way, the verifier's entire view can be reconstructed as needed when it is time to OPEN each $\mathcal{F}_{\mathsf{cpfe}}$ instance.

### 4.3 Security Proof

**Theorem 1.** *The protocol $\pi$ presented in Section 4.2 is a secure realization of the $\mathcal{F}^{\mathcal{R}}_{\mathsf{ZK}}$ functionality.*

*Proof.* We describe two simulators, depending on which party is corrupted.

*Prover is corrupt:* The primary role of the simulator in this case is to extract the witness from $P$. We construct the simulator in a sequence of hybrid interactions:

$\mathcal{H}_0$: Simulator plays the role of an honest verifier $V$ (who has no input) and all ideal functionalities. In particular, the simulator obtains all of $P$'s inputs to the ideal functionalities. This interaction is identical to the real interaction with $\pi$.

$\mathcal{H}_1$: Same as $\mathcal{H}_0$ except that instead of using a PRF, the simulated verifier chooses output wire labels $D^t_{\mathsf{mem}}$ and $\boldsymbol{h}^t_{\mathsf{mem}}$ functions uniformly (in $\mathsf{TempMemLabels}$ and $\mathsf{GenH}$). We have $\mathcal{H}_1 \approx \mathcal{H}_0$ by the security of the PRF.

$\mathcal{H}_2$: Same as $\mathcal{H}_1$ except that the simulator aborts in certain cases as follows. The simulator has initially generated $\widehat{M}$ and $st$ (while simulating $\mathcal{F}_{\mathsf{init}}$) and obtains $w$ as $P$'s input to $\mathcal{F}_{\mathsf{otc}}$ in each step (6a). Hence, each time in step 6, the simulator executes $C_{x,\mathcal{I}}(st, w, \widehat{M}[\mathsf{read}(\mathcal{I})]) \rightarrow (st, z, \widehat{M}[\mathsf{access}(\mathcal{I})])$, updating its internal $st$ and $\widehat{M}$.

In the $\mathsf{LabelXfer}$ subprotocols in steps (3) and (7), $P$ is meant to provide his garbled output $Y^t_{\mathsf{mem}}$ and $Y^t_{\mathsf{st}}$ to the $\mathcal{F}_{\mathsf{cpfe}}$ functionalities. Similarly, in step (6c), the prover is expected to commit to $Y^t_{\mathsf{z}}|_{\text{TRUE}}$. In $\mathcal{H}_2$, the simulator artificially aborts if $P$ provides a *valid* encoding $D^t|_y$ for $y$ not equal to the simulated output of $C_{x,\mathcal{I}}$ at time $t$.

Now we claim that the simulator artificially aborts with only negligible probability (so $\mathcal{H}_1 \approx \mathcal{H}_2$) and that the prover's view of $E^t$ during step (7) in time $t$ can be simulated given only $E^t_{\mathsf{mem}}|_{\widehat{M}[\mathsf{read}(\mathcal{I})]}$ and $E^t_{\mathsf{st}}|_{st}$. This follows essentially from the authenticity property of the garbling scheme and the strong-universal hashing property of $\mathcal{H}$.

Consider the $\mathsf{LabelXfer}$ subprotocol in step (3) (i.e., time $t = 0$). At this time, all wire labels in $D^0$ besides $D^0_{\mathsf{mem}}|_{\widehat{M}}$ and $D^0_{\mathsf{mem}}|_{st}$ are independent of the adversary's view by definition of the $\mathcal{F}_{\mathsf{init}}$ functionality. Hence, the simulator artificially aborts with negligible probability during these steps. Conditioned on not aborting, the action of the strongly universal hash functions on the "wrong" wire labels of $D^0$ — and hence the value of the "wrong" input wire labels in $E^1$ — is distributed independently of $P$'s view. Thus $P$'s view in step (6) can be simulated given only the claimed subset of $E^1$. Inductively, the prover's view of $E^t$ at the time of the $\mathsf{LabelXfer}$ steps depends only on the "expected" input wire labels. Hence, the simulator artificially aborts with negligible probability, due to the authenticity property of the garbling scheme. As above, conditioned on not aborting, the strong universal hashing property ensures that the prover's view of $E^{t+1}$ depends only on the claimed subset of $E^{t+1}$.

$\mathcal{H}_3$: Same as $\mathcal{H}_2$ except that in step (2) the simulator sends $P$'s input $M$ to $\mathcal{F}^{\mathcal{R}}_{\mathsf{ZK}}$. In step (9), if the simulated verifier does not abort, then the simulator sends $(x, w_{\mathsf{real}})$ to $\mathcal{F}^{\mathcal{R}}_{\mathsf{ZK}}$ (where $w$ was extracted from the prover in step (6a). We claim that the output of the ideal verifier always matches that of the simulated verifier. The simulated verifier accepts the proof if $P$ has committed

to $D^t_{\mathsf{z}}|_{\text{TRUE}}$. Provided that the simulator has not artificially aborted, then it must be that the simulated $C_{x,\mathcal{I}}$ has output $z = \text{TRUE}$. By the correctness of the RAM program, it must be that $w_{\mathsf{real}}$ is a valid witness for $x$.

Hence, the simulator implicit in $\mathcal{H}_3$ is our final simulator.

*Verifier is corrupt:* In this case, the primary role of the simulator is to simulate its view without knowledge of the witness $w$. We note that the only information that needs to be simulated in each proof is the memory access sequence $\mathcal{I}$ and the opened commitment to output wire label $Y^t_{\mathsf{z}}$. Again we proceed in a sequence of hybrid interactions.

$\mathcal{H}_0$: Simulator plays the role of an honest prover $P$ (including $M$ and witnesses $w$ as input) and all ideal functionalities. Hence, the simulator obtains all of $V$'s inputs to the ideal functionalities. This interaction is identical to the real interaction with $\pi$.

$\mathcal{H}_1$: Same as $\mathcal{H}_0$ except for the following changes. An honest prover computes $D^t$ in step (8d) when the verifier decommits to certain inputs to ideal functionalities. Here we have the simulator perform the same computations, but as soon as possible given the ability to see the verifier's inputs to the functionalities. Hence, in step (6c), the simulator will know the entire contents of $D^t$. Instead of evaluating the garbled circuit to obtain garbled output $Y^t_{\mathsf{z}}$, we have the simulator simply commit to $D^t_{\mathsf{z}}|_{\text{TRUE}}$.
This commitment is only opened when the garbled circuit $F^t$ is shown to be correct. Hence, $\mathcal{H}_0 \equiv \mathcal{H}_1$.

$\mathcal{H}_2$: Same as $\mathcal{H}_1$ except for the following changes. Note that in $\mathcal{H}_1$ the simulator uses secret values $M$ and $w$ only to generate the memory access sequence $\mathcal{I}$. All of the simulated prover's other inputs to ideal functionalities can be set to dummy values, as $V$ gets no outputs. So in $\mathcal{H}_2$ we have the simulated prover generate $\mathcal{I}$ in step (4) using the ORAM simulator instead of actually executing the RAM program itself. We have $\mathcal{H}_1 \approx \mathcal{H}_2$ by the security of the ORAM.

The simulator implicit in $\mathcal{H}_2$ defines our final simulator, since it no longer requires the secret values $M$ and $w$ to operate.

This completes the security proof of our protocol.

## Acknowledgements

We thank anonymous CRYPTO reviewers for helpful feedback. Zhangxiang Hu and Mike Rosulek were supported by NSF award CCF-1149647.

## References

AHMR15. Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. How to efficiently evaluate RAM programs with malicious security. In Elisabeth

Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 702–729. Springer, 2015.

BHR12. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12: 19th Conference on Computer and Communications Security*, pages 784–796. ACM Press, October 2012.

Blo70. Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

CDS94. Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Yvo Desmedt, editor, *Advances in Cryptology – CRYPTO'94*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187. Springer, August 1994.

CM99. Jan Camenisch and Markus Michels. Proving in zero-knowledge that a number is the product of two safe primes. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 107–122. Springer, May 1999.

CP13. Kai-Min Chung and Rafael Pass. A simple ORAM. Cryptology ePrint Archive, Report 2013/243, 2013. http://eprint.iacr.org/2013/243.

FNO15. Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In *Eurocrypt*, 2015. To appear.

GHL+14. Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 405–422. Springer, May 2014.

GKK+12. S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12: 19th Conference on Computer and Communications Security*, pages 513–524. ACM Press, October 2012.

GMR89. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

GO96. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

GS08. Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In Nigel P. Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 415–432. Springer, April 2008.

IPS09. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In Omer Reingold, editor, *TCC 2009: 6th Theory of Cryptography Conference*, volume 5444 of *Lecture Notes in Computer Science*, pages 294–314. Springer, March 2009.

JKO13. Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 955–966. ACM Press, November 2013.

KK12. Vladimir Kolesnikov and Ranjit Kumaresan. Improved secure two-party computation via information-theoretic garbled circuits. In Ivan Visconti and Roberto De Prisco, editors, *SCN 12: 8th International Conference on Security in Communication Networks*, volume 7485 of *Lecture Notes in Computer Science*, pages 205–221. Springer, September 2012.

KS06. Mehmet Kiraz and Berry Schoenmakers. A protocol issue for the malicious case of Yao's garbled circuit construction. In *27th Symposium on Information Theory in the Benelux*, pages 283–290, 2006.

LO13. Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 719–734. Springer, May 2013.

MGFB14. Benjamin Mood, Debayan Gupta, Joan Feigenbaum, and Kevin Butler. Reuse It Or Lose It: More Efficient Secure Computation Through Reuse of Encrypted Values. In *ACM CCS*, 2014.

MRK03. Silvio Micali, Michael O. Rabin, and Joe Kilian. Zero-knowledge sets. In *44th Annual Symposium on Foundations of Computer Science*, pages 80–91. IEEE Computer Society Press, October 2003.

Sch90. Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252. Springer, August 1990.

SvDS+13. Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 299–310. ACM Press, November 2013.

WW06. Stefan Wolf and Jürg Wullschleger. Oblivious transfer is symmetric. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 222–232. Springer, May / June 2006.

ZRE15. Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 220–250. Springer, 2015.