How to Efficiently Evaluate RAM Programs with Malicious Security

Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek[2*]

[1] University of Calgary, aafshar@ucalgary.ca
[2] Oregon State University, {huz,rosulek}@eecs.oregonstate.edu
[3] Yahoo Labs, pmohassel@yahoo-inc.com

Abstract. Secure 2-party computation (2PC) is becoming practical for some applications. However, most approaches are limited by the fact that the desired functionality must be represented as a boolean circuit. In response, random-access machines (RAM programs) have recently been investigated as a promising alternative representation.

In this work, we present the first practical protocols for evaluating RAM programs with security against malicious adversaries. A useful efficiency measure is to divide the cost of malicious-secure evaluation of f by the cost of semi-honest-secure evaluation of f. Our RAM protocols achieve ratios matching the state of the art for circuit-based 2PC. For statistical security 2^{-s}, our protocol without preprocessing achieves a ratio of s; our online-offline protocol has a pre-processing phase and achieves online ratio $\sim 2s/\log T$, where T is the total execution time of the RAM program.

To summarize, our solutions show that the "extra overhead" of obtaining malicious security for RAM programs (beyond what is needed for circuits) is minimal and does not grow with the running time of the program.

1 Introduction

General secure two-party computation (2PC) allows two parties to perform "arbitrary" computation on their joint inputs without revealing any information about their private inputs beyond what is deducible from the output of computation. This is an extremely powerful paradigm that allows for applications to utilize sensitive data without jeopardizing its privacy.

From a feasibility perspective, we know that it is possible to securely compute any function, thanks to seminal results of [41,11]. The last decade has also witnessed significant progress in design and implementation of more practical/scalable secure computation techniques, improving performance by orders of magnitude and enabling computation of circuits with billions of gates.

These techniques, however, are largely restricted to functions represented as Boolean or arithmetic circuits, whereas the majority of applications we encounter

* Supported by NSF award CCF-1149647.

in practice are more efficiently captured using random-access memory (RAM) programs that allow constant-time memory lookup. Modern algorithms of practical interest (e.g., binary search, Dijkstra's shortest-paths algorithm, and the Gale-Shapely stable matching algorithm) all rely on fast memory access for efficiency, and suffer from major blowup in running time otherwise. More generally, a circuit computing a RAM program with running time T requires $\Theta(T^2)$ gates in the worst case, making it prohibitively expensive (as a general approach) to compile RAM programs into a circuit and then apply known circuit 2PC techniques.

A promising alternative approach uses the building block of *oblivious RAM*, introduced by Goldreich and Ostrovsky [12]. ORAM is an approach for making a RAM program's memory access pattern input-oblivious while still retaining fast (polylogarithmic) memory access time. Recent work in 2PC has begun to investigate direct computation of ORAM computations as an alternative to RAM-to-circuit compilation [13,28,19,10,27]. These works all follow the same general approach of evaluating a sequence of ORAM instructions using traditional circuit-based 2PC phases. More precisely, they use existing circuit-based MPC to (1) initialize and setup the ORAM, a one-time computation with cost proportional to the memory size, (2) evaluate the next-instruction circuit which outputs "shares" of the RAM program's internal state, the next memory operations (read/write), the location to access, and the data value in case of a write. All of these existing solutions provide security only against semi-honest adversaries.

Challenges for malicious-secure RAM evaluation. It is possible to take a semi-honest secure protocol for RAM evaluation (e.g., [13]) and adapt it to the malicious setting using standard techniques. Doing so naïvely, however, would result in several major inefficiencies that are avoidable. We point out three significant challenges for efficient, malicious-secure RAM evaluation:

1: Integrity and consistency of state information, by which we mean both the RAM's small internal state and its large memory both of which are passed from one CPU step to the next. A natural approach for handling internal state is to have parties hold secret shares of the state (as in [13]), which they provide as input to a secure evaluation of the next-instruction circuit. Using standard techniques for malicious-secure SFE, it would incur significant overhead in the form of oblivious transfers and consistency checks to deal with state information as inputs to the circuit.

A natural approach suitable for handling RAM memory is to evaluate an Oblivious RAM that encrypts its memory contents. In this approach, the parties must evaluate a next-instruction circuit that includes both encryption and decryption sub-circuits. Evaluating a block cipher's circuit securely against malicious adversaries is already rather expensive [22], and this approach essentially asks the parties to do so at every time-step, even when the original RAM's behavior is non-cryptographic. Additional techniques are needed to detect any tampering of data by either participant, such as computing/verifying a MAC of each memory location access inside the circuit or computing a "shared" Merkle-

tree on top of the memory in order to check its consistency after each access. All these solutions incur major overhead when state is passed or memory is accessed and are hence prohibitively expensive (see full version [1] for a concrete example).

2: Compatibility with batch execution and input-recovery techniques. In a secure computation, every input bit must be "touched" at some point. Oblivious RAM programs address this with a pre-processing phase that "touches" the entire (large) RAM memory, after which the computation need not "touch" every bit of memory. Since an offline phase is already inevitable for ORAMs, we would like to use such a phase to further increase the efficiency of the online phase of the secure evaluation protocol. In particular, recent techniques of [15,26] suggest that pre-processing/batching garbled circuits can lead to significant efficiency improvement for secure evaluation of circuits. The fact that the ORAM next-instruction circuits are used at every timestep and are known *a priori* makes the use of batch execution techniques even more critical.

Another recent technique, called input-recovery [23], reduces the number of garbled circuits in cut-and-choose by a factor of 3 by only requiring that at least one of the evaluated circuits is correct (as opposed to the majority). This is achieved by running an input-recovery step at the end of computation that recover's the garbler's private input in case he cheats in more than one evaluated circuit. The evaluator then uses the private input to do the computation on his own. A natural applications of this technique in case of RAM programs, would require running the input-recovering step after every timestep which would be highly inefficient (see full version [1] for a concrete example).

3: Run-time dependence. The above issues are common to any computation that involves persistent, secret internal state across several rounds of inputs/outputs (any so-called *reactive* functionality). RAM programs present an additional challenge, in that only part of memory is accessed at each step, and furthermore these memory locations are determined *only at run-time.* In particular, it is non-trivial to reconcile run-time data dependence with offline batching optimizations.

Our approach: In a RAM computation, both the memory and internal state need to be *secret* and *resist tampering* by a malicious adversary. As mentioned above, the obvious solutions to these problem all incur major overhead whenever state is passed from one execution to the next or memory is accessed. We bypass all these overheads and obtain secrecy and tamper-resistance essentially for free. Our insight is that these are properties also shared by wire labels in most garbling schemes — they hide the associated logical value, and, given only one wire label, it is hard to "guess" the corresponding complementary label.

Hence, instead of secret-sharing the internal state of the RAM program between the parties, we simply "re-use" the garbled wire labels from the output of one circuit into the input of the next circuit. These wire labels already inherit the required authenticity properties, so no oblivious transfers or consistency checks are needed.

Similarly, we also encode the RAM's memory via wire labels. When the RAM reads from memory location ℓ, we simply reuse the appropriate output wire labels from the most recent circuit to write to location ℓ (not necessarily the previous instruction, as is the case for the internal state). Since the wire labels already hide the underlying logical values, we only require an oblivious RAM that hides the memory access pattern and *not* the contents of memory. More concretely, this means that we do not need to add encryption/decryption and MAC/verify circuitry inside the circuit that is being garbled or perform oblivious transfers on shared intermediate secrets. Importantly, if the RAM program being evaluated is "non-cryptographic" (i.e., has a small circuit description) then the circuits garbled at each round of our protocols will be small.

Of course, it is a delicate task to make these intuitive ideas work with the state of art techniques for cut-and-choose. We present two protocols, which use different approaches for reusing wire labels.

The first protocol uses ideas from the LEGO paradigm [33,9] for 2PC and other recent works on batch-preprocessing of garbled circuits [15,26]. The idea behind these techniques is to generate all the necessary garbled circuits in an offline phase (before inputs are selected), open and check a random subset, and randomly assign the rest into buckets, where each bucket corresponds to one execution of the circuit. But unlike the setting of [15,26], where circuits are processed for many *independent* evaluations of a function, we have the additional requirement that the wire labels for memory and state data should be directly reused between various garbled circuits. Since we cannot know which circuits must have shared wire labels (due to random assignment to buckets and run-time memory access pattern), we use the "soldering" technique of [33,9] that directly transfers garbled wire labels from one wire to another, after the circuits have been generated. However, we must adapt the soldering approach to make it amenable to soldering entire circuits as opposed to soldering simple gates as in [33,9]. For a discussion of subtle problems that arise from a direct application of their soldering technique, see Section 3.

Our second approach directly reuses wire labels without soldering. As a result, garbled circuits cannot be generated offline, but the scheme does not require the homomorphic commitments required for the LEGO soldering technique. At a high level, we must avoid having the cut-and-choose phase reveal secret wire labels that are shared in common with other garbled circuits. The technique recently proposed in [31] allows us to use a single cut-and-choose for all steps of the RAM computation (rather than independent cut-and-choose steps for each time step), and further hide the set of opened/evaluated circuits from the garbler using an OT-based cut-and-choose [18,22]. We observe that this approach is compatible with the state of the art techniques for input-consistency check [30,38].

We also show how to incorporate the input-recovery technique of [23] for reducing the number of circuits by a factor of three. The naive solution of running the cheating recovery after each timestep would be prohibitively expensive since it would require running a malicious 2PC for the cheating recovery circuit

(and the corresponding input-consistency checks) at every timestep. We show a modified approach that only requires a final cheating recovery step at the end of the computation.

Based on some concrete measurements (see full version [1]), the "extra overhead" of achieving malicious security for RAM programs (i.e. the additional cost beyond what is needed for malicious security of the circuits involved in the computation), is at least an order of magnitude smaller than the naive solutions and this gap grows as the running time of the RAM program increases.

Related work. Starting with seminal work of [42,11], the bulk of secure multiparty computation protocols focus on functions represented as circuits (arithmetic or Boolean). More relevant to this work, there is over a decade's worth of active research on design and implementation of *practical* 2PC protocols with malicious security based on garbled circuits [29,20,24,25,37,38,23,14,30], based on GMW [32], and based on arithmetic circuits [8].

The work on secure computation of RAM programs is much more recent. [13] introduces the idea of using ORAM inside a Yao-based secure two-party computation in order to accommodate (amortized) sublinear-time secure computation. The work of [28,10] study non-interactive garbling schemes for RAM programs which can be used to design protocols for secure RAM program computation. The recent work of [19], implements ORAM-based computation using arithmetic secure computation protocol of [8], hence extending these ideas to the multiparty case, and implementing various oblivious data-structures. SCVM [27] and Obliv-C [44] provide frameworks (including programming languages) for secure computation of RAM programs that can be instantiated using different secure computation RAM programs on the back-end. The above work all focus on the semi-honest adversarial model. To the best of our knowledge, our work provides the first practical solution for secure computation of RAM program with malicious security. Our constructions can be used to instantiate the back-end in SCVM and Obliv-C with malicious security.

2 Preliminaries

2.1 (Oblivious) RAM Programs

A RAM program is characterized by a deterministic circuit Π and is executed in the presence of memory M. The memory is an array of *blocks*, which are initially set to 0^n. An execution of the RAM program Π on inputs (x_1, x_2) with memory M is given by:

$\underline{\text{RAMEval}(\Pi, M, x_1, x_2)}$
  $\mathsf{st} := x_1 \| x_2 \| 0^n$; $\mathsf{block} := 0^n$; $\mathsf{inst} := \bot$
  do until inst has the form $(\textsc{halt}, z)$:
    $\mathsf{block} :=$ [if $\mathsf{inst} = (\textsc{read}, \ell)$ then $M[\ell]$ else 0^n]
    $r \leftarrow \{0,1\}^n$; $(\mathsf{st}, \mathsf{inst}, \mathsf{block}) := \Pi(\mathsf{st}, \mathsf{block}, r)$
    if $\mathsf{inst} = (\textsc{write}, \ell)$ then $M[\ell] := \mathsf{block}$
  output z

Oblivious RAM, introduced in [12], is a technique for hiding all information about a RAM program's memory (both its contents and the data-dependent access pattern). Our constructions require a RAM program that hides only the memory access pattern, and we will use other techniques to hide the *contents* of memory. Throughout this work, when we use the term "ORAM", we will be referring to this weaker security notion. Concretely, such an ORAM can often be obtained by taking a standard ORAM construction (e.g., [40,7]) and removing the steps where it encrypts/decrypts memory contents.

Define $\mathcal{I}(\Pi, M, x_1, x_2)$ as the random variable denoting the sequence of values taken by the inst variable in $\text{RamEval}(\Pi, M, x_1, x_2)$. Our precise notion of ORAM security for Π requires that there exist a simulator $\mathcal{S}$ such that, for all x_1, x_2 and initially empty M, the output $\mathcal{S}(1^\lambda, z)$ is indistinguishable from $\mathcal{I}(\Pi, M, x_1, x_2)$, where z is the final output of the RAM program on inputs x_1, x_2.

2.2 Garbling Schemes

In this section we adapt the abstraction of *garbling schemes* [5] to our needs. Our 2PC protocol constructions re-use wire labels between different garbled circuits, so we define a specialized syntax for garbling schemes in which the input and output wire labels are pre-specified.

We represent a set of wire labels W as a $m \times 3$ array. Wire labels $W[i, 0]$ and $W[i, 1]$ denote the two wire labels associated with some wire i. We employ the point-permute optimization [34], so we require $\mathsf{lsb}(W[i, b]) = b$. The value $W[i, 2]$ is a single-bit *translation bit*, so that $W[i, W[i, 2]]$ is the wire label that encodes FALSE for wire i. For shorthand, we use $\tau(W)$ to denote the m-bit string $W[1, 2] \cdots W[m, 2]$.

We require the garbling scheme to have syntax $F \leftarrow \mathsf{Garble}(f, E, D)$ where f is a circuit, E and D represent wire labels as above.

For $v \in \{0,1\}^m$, we define $W|_v = (W[1, v_1], \ldots, W[m, v_m])$, i.e., the wire labels with *select bits* v. We also define $W|^*_x := W|_{x \oplus \tau(W)}$, i.e., the wire labels corresponding to *truth values* x. The correctness condition we require for garbling is that, for all f, x, and valid wire label descriptions E, D, we have:

$$\mathsf{Eval}(\mathsf{Garble}(F, E, D), E|^*_x) = D|^*_{f(x)}$$

If Y denotes a vector of output wire labels, then it can be decoded to a plain output via $\mathsf{lsb}(Y) \oplus \tau(D)$, where lsb is applied component-wise. Hence, $\tau(D)$ can be used as output-decoding information. More generally, if $\mu \in \{0,1\}^m$ is a mask value, then revealing $(\mu, \tau(D) \wedge \mu)$ allows the evaluator to learn only the output bits for which $\mu_i = 1$.

Let $\mathcal{W}$ denote the uniform distribution of $m \times 3$ matrices of the above form (wire labels with the constraint on least-significant bits described above). Then the security condition we need is that there exists an efficient simulator $\mathcal{S}$ such that for all f, x, D, the following distributions are indistinguishable:

$$\begin{array}{ll} \underline{\mathrm{Real}(f,x,D)\text{:}} & \underline{\mathrm{Sim}^{\mathcal{S}}(f,x,D)\text{:}} \\ E \leftarrow \mathcal{W} & E \leftarrow \mathcal{W} \\ F \leftarrow \mathsf{Garble}(f,E,D) & F \leftarrow \mathcal{S}(f, E|_x^*, D|_{f(x)}^*) \\ \text{return } (F, E|_x^*) & \text{return } (F, E|_x^*) \end{array}$$

To understand this definition, consider an evaluator who receives garbled circuit F and wire labels $E|_x^*$ which encode its input x. The security definition ensures that the evaluator learns no more than the correct output wires $D|_{f(x)}^*$.

Consider what happens when we apply this definition with D chosen from $\mathcal{W}$ and against an adversary who is given only partial decoding information $(\mu, \tau(D) \wedge \mu)$.[4] Such an adversary's view is then independent of $f(x) \wedge \overline{\mu}$. This gives us a combination of the *privacy* and *obliviousness* properties of [5]. Furthermore, the adversary's view is independent of the complementary wire labels $D|_{\overline{f(x)}}^*$, except possibly in their least significant bits (by the point-permute constraint). So the other wire labels are hard to predict, and we achieve an *authenticity* property similar to that of [5].[5]

Finally, we require that it be possible to efficiently determine whether F is in the range of $\mathsf{Garble}(f,E,D)$, given (f,E,D). For efficiency improvements, one may also reveal a seed which was used to generate the randomness used in Garble.

These security definitions can be easily achieved using typical garbling schemes used in practice (e.g., [21]). We note that the above arguments hold even when the distribution $\mathcal{W}$ is slightly different. For instance, when using the Free-XOR optimization [21], wire label matrices E and D are chosen from a distribution parameterized by a secret Δ, where $E[i,0] \oplus E[i,1] = \Delta$ for all i. This distribution satisfies all the properties of $\mathcal{W}$ that were used above.

Conventions for wire labels. We exclusively garble the ORAM circuit which has its inputs/outputs partitioned into several logical values. When W is a description of input wire labels for such a circuit, we let $\mathsf{st}(W)$, $\mathsf{rand}(W)$, $\mathsf{block}(W)$ denote the submatrices of W corresponding to the incoming internal state, random tape, and incoming memory block. When W describes output wires, we use $\mathsf{st}(W)$, $\mathsf{inst}(W)$ and $\mathsf{block}(W)$ to denote the outgoing internal state, output instruction (read/write/halt, and memory location), and outgoing memory data block. We use these functions analogously for vectors (not matrices) of wire labels.

2.3 (XOR-Homomorphic) Commitment

In addition to a standard commitment functionality $\mathcal{F}_{\mathsf{com}}$, one of our protocols requires an XOR-homomorphic commitment functionality $\mathcal{F}_{\mathsf{xcom}}$. This functionality allows P_1 to open the XOR of two or more commited messages without

[4] Our definition applies to this case, since a distinguisher for the above two distributions is allowed to know D which parameterizes the distributions.

[5] We stress that the evaluator *can indeed decode* the garbled output (using $\tau(D)$ and the select bits), yet *cannot forge* valid output wire labels in their entirety. This combination of requirements was not considered in the definitions of [5].

The functionality is initialized with internal value $i = 1$. It then repeatedly responds to commands as follows:

- On input (commit, m) from P_1, store (i, m) internally, set $i := i + 1$ and output $(\mathit{committed}, i)$ to both parties.
- On input (open, S) from P_1, where S is a set of integers, for each $i \in S$ find (i, m_i) in memory. If for some i, no such m_i exists, send $\perp$ to P_2. Otherwise, send $(\mathit{open}, S, \bigoplus_{i \in S} m_i)$ to P_2.

Fig. 1. XOR-homomorphic commitment functionality $\mathcal{F}_{\mathsf{xcom}}$.

leaking any other information about the individual messages. The funcionality is defined in Figure 1. Further details, including an implementation, can be found in [9].

3 Batching Protocol

3.1 High-level Overview

Roughly speaking, the LEGO technique of [33,9] is to generate a large quantity of garbled gates, perform a cut-and-choose on all gates to ensure their correctness, and finally assemble the gates together into a circuit which can tolerate a bounded number of faulty gates (since the cut-and-choose will not guarantee that all the gates are correct). More concretely, with sN gates and a cut-and-choose phase which opens half of them correctly, a statistical argument shows that permuting the remaining gates into **buckets** of size $O(s/\log N)$ each ensures that each bucket contains a majority of correct gates, except with negligible probability in s.

For each gate, the garbler provides a *homomorphic commitment* to its input/output wire labels, which is also checked in the cut and choose phase. This allows wires to be connected on the fly with a technique called **soldering.** A wire with labels (w_0, w_1) (here 0 and 1 refer to the public select bits) can be soldered to a wire with labels (w'_0, w'_1) as follows. If w_0 and w'_0 both encode the same truth value, then decommit to $\Delta_0 = w_0 \oplus w'_0$ and $\Delta_1 = w_1 \oplus w'_1$. Otherwise decommit to $\Delta_0 = w_0 \oplus w'_1$ and $\Delta_1 = w_1 \oplus w'_0$. Then when an evaluator obtains the wire label w_b on the first wire, $w_b \oplus \Delta_b$ will be the correct wire label for the second wire. To prove that the garbler hasn't inverted the truth value of the wires by choosing the wrong case above, she must also decommit to the XOR of each wire's *translation* bit (i.e., $\beta \oplus \beta'$ where w_β and $w'_{\beta'}$ both encode false).

Next, an arbitrary gate within each bucket is chosen as the **head**. For each other gate, we solder its input wires to those of the head, and output wires to those of the head. Then an evaluator can transfer the input wire labels to each of the gates (by XORing with the appropriate solder value), evaluate the gates, and

transfer the wire labels back. The majority value is taken to be the output wire label of the bucket. The cut-and-choose ensures that each bucket functions as a correct gate, with overwhelming probability. Then the circuit can be constructed by appropriately soldering together the buckets in a similar way.

For our protocol we use a similar approach but work with buckets of *circuits*, not buckets of gates. Each bucket evaluates a single timestep of the RAM program. To transfer RAM memory and internal state between timesteps, we solder wires together appropriately (i.e., state input of time t soldered to state output of time $t-1$; memory-block input t soldered to memory-block output of the previous timestep that wrote to the desired location). Additionally, the approach of using buckets also saves an asymptotic $\log T$ factor in the number of circuits needed for each timestep (i.e., the size of the buckets), where T is the total running time of the ORAM, a savings that motivates similar work on batch pre-processing of garbled circuits [15,26].

We remark that our presentation of the LEGO approach above is a slight departure from the original papers [33,9]. In those works, all gates were garbled using Free XOR optimization, where $w_0 \oplus w_1$ is a secret constant shared on all wires. Hence, we have only one "solder" value $w_0 \oplus w'_0 = w_1 \oplus w'_1$. If the sender commits to only the "false" wire label of each wire, then the sender is prevented from inverting the truth value while soldering ("false" is always mapped to "false"). However, to keep the offset $w_0 \oplus w_1$ secret, only one of the 4 possible input combinations of each gate can be opened in the cut-and-choose phase. The receiver has only a 1/4 probability of identifying a faulty gate. This approach does not scale to a cut-and-choose of entire circuits, where the number of possible input combinations is exponential. Hence our approach of forgoing common wire offsets $w_0 \oplus w_1$ between circuits and instead committing to the translation bits. As a beneficial side effect, the concrete parameters for bucket sizes are improved since the receiver will detect faulty circuits with probability 1, not 1/4.

Back to our protocol, P_1 generates $O(sT/\log T)$ garblings of the ORAM's next-instruction circuit, and commits to the circuits and their wire labels. P_2 chooses a random half of these to be opened and aborts if any are found to be incorrect.

For each timestep t, P_2 picks a random subset of remaining garbled circuits and the parties assemble them into a bucket $\mathcal{B}_t$ (this is the $\mathsf{MkBucket}$ subprotocol) by having P_1 open appropriate XORs of wire labels, as described above. We can extend the garbled-circuit evaluation function Eval to $\mathsf{EvalBucket}$ using the same syntax. Then $\mathsf{EvalBucket}$ inherits the correctness property of Eval with overwhelming probability, for each of the buckets created in the protocol.

After a bucket is created, P_2 needs to obtain garbled inputs on which to evaluate it. See Figure 3 for an overview. Let X_t denote the vector of input wire labels to bucket $\mathcal{B}_t$. We use $\mathsf{block}(X_t), \mathsf{st}(X_t), \mathsf{rand}(X_t)$ to denote the sets of wire labels for the input memory block, internal state, and shares of random tape, respectively. The simplest wire labels to handle are the ones for internal state, as they always come from the previous timestep. We solder the output internal

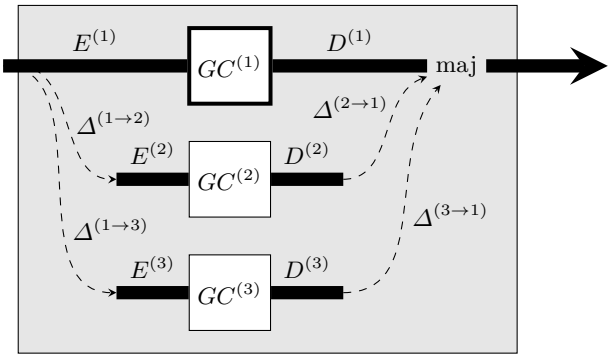

Fig. 2. Illustration of $\mathsf{MkBucket}(\mathcal{B} = \{1, 2, 3\}, \mathsf{hd} = 1)$.

state wires of bucket $\mathcal{B}_{t-1}$ to the input internal state wires of bucket $\mathcal{B}_t$. Then if Y_{t-1} were the output wire labels for bucket $\mathcal{B}_{t-1}$ by P_2, we obtain $\mathsf{st}(X_t)$ by adjusting $\mathsf{st}(Y_{t-1})$ according to the solder values.

If the previous memory instruction was a READ of a location that was last written to at time t', then we need to solder the appropriate output wires from bucket $\mathcal{B}_{t'}$ to the corresponding input wires of $\mathcal{B}_t$. P_2 then obtains $\mathsf{block}(X_t)$ by adjusting the wire labels $\mathsf{block}(Y_{t'})$ according to the solder values. If the previous memory instruction was a READ of an uninitialized block, or a WRITE, then P_1 simply opens these input wire labels to all zero values (see $\mathsf{GetInput}_{\mathsf{pub}}$).

To obtain wire labels $\mathsf{rand}(X_t)$, we have P_1 open wire labels for its shares ($\mathsf{GetInput}_1$) and have P_2 obtain its wire labels via a standard OT ($\mathsf{GetInput}_2$).

At this point, P_2 can evaluate the bucket ($\mathsf{EvalBucket}$). Let Y_t denote the output wire labels. P_1 opens the commitment to their translation values, so P_2 can decode and learn these outputs of the circuit. P_2 sends these labels back to P_1, who verifies them for authenticity. Knowing only the translation values and not the entire actual output wire labels, P_2 cannot lie about the circuit's output except with negligible probability.

3.2 Detailed Protocol Description

Let Π be the ORAM program to be computed. Define $\tilde{\Pi}(\mathsf{st}, \mathsf{block}, \mathsf{inp}_1, \mathsf{inp}_{2,1}, \ldots, \mathsf{inp}_{2,n}) = \Pi(\mathsf{st}, \mathsf{block}, \mathsf{inp}_1, \bigoplus_i \mathsf{inp}_{2,i})$. Looking ahead, during the first timestep, the parties will provide $\mathsf{inp}_1 = x_1$ and $\mathsf{inp}_2 = x_2$, while in subsequent timesteps they input their shares r_1 and r_2 of the RAM program's randomness. P_2's input is further secret shared to prevent a selective failure attack on both x_2 and his random input r_2. We first define the following subroutines / subprotocols:

prot $\mathsf{Solder}(A, A')$ // A, A' are wire labels descriptions
  P_1 opens $\mathcal{F}_{\mathsf{xcom}}$-commitments to $\tau(A)$ and $\tau(A')$
    so that P_2 receives $\tau = \tau(A) \oplus \tau(A')$
  for each position i in τ and each $b \in \{0, 1\}$:

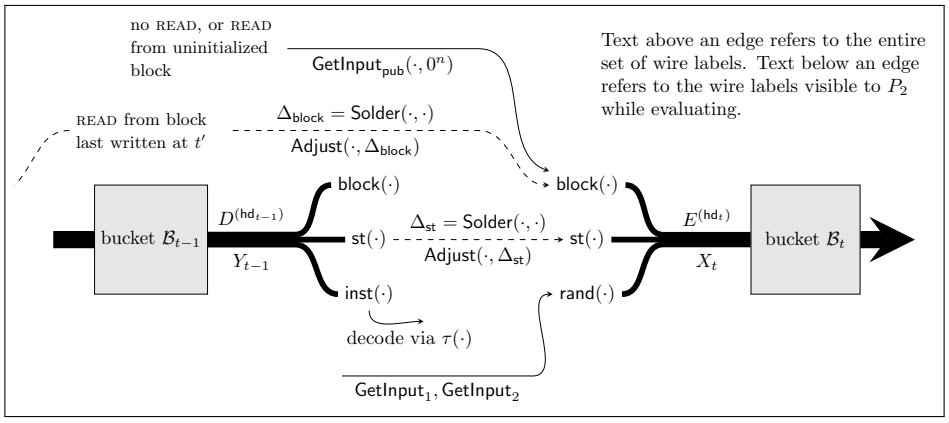

Fig. 3. Overview of soldering and evaluation steps performed in the online phase.

P_1 opens $\mathcal{F}_{\mathsf{xcom}}$-commitments to $A[i,b]$ and $A'[i,\tau_i \oplus b]$
so that P_2 receives $\Delta[i,b] = A[i,b] \oplus A'[i,\tau_i \oplus b]$
return Δ

prot $\mathsf{MkBucket}(\mathcal{B}, \mathsf{hd})$ // $\mathcal{B}$ is a set of indices
for each $j \in \mathcal{B} \setminus \{\mathsf{hd}\}$:
$\Delta^{(\mathsf{hd}\to j)} = \mathsf{Solder}(E^{(\mathsf{hd})}, E^{(j)})$
$\Delta^{(j\to\mathsf{hd})} = \mathsf{Solder}(D^{(j)}, D^{(\mathsf{hd})})$
$\Delta^{(\mathsf{hd}\to\mathsf{hd})} :=$ all zeroes // for convenience

func $\mathsf{Adjust}(X, \Delta)$ // X is a vector of wire labels
for each i do $\tilde{X}[i] = X[i] \oplus \Delta[i, \mathsf{lsb}(X[i])]$
return $\tilde{X}$

func $\mathsf{EvalBucket}(\mathcal{B}, X, \mathsf{hd})$
for each j in $\mathcal{B}$:
$\tilde{X}_j = \mathsf{Adjust}(X, \Delta^{(\mathsf{hd}\to j)})$
$Y_j = \mathsf{Adjust}(\mathsf{Eval}(GC^{(j)}, \tilde{X}_j), \Delta^{(j\to\mathsf{hd})})$
return the majority element of $\{Y_j\}_{j\in\mathcal{B}}$

prot $\mathsf{GetInput}_{\mathsf{pub}}(A, x)$ // A describes wire labels; x public
P_1 opens commitments of $A|_x^*$; call the result X
P_1 opens commitments of $\tau(A)$
P_2 aborts if $\mathsf{lsb}(X) \neq \tau(A) \oplus x$; else returns X

prot $\mathsf{GetInput}_1(A, x)$ // A describes wire labels; P_1 holds x
P_1 opens commitments of $A|_x^*$; return these values

prot $\mathsf{GetInput}_2(A, x)$ // A describes wire labels; P_2 holds x
for each position i in A, parties invoke an instance of $\mathcal{F}_{\mathsf{ot}}$:

P_1 uses input $(A[i, A[i,2]], A[i, 1 \oplus A[i,2]])$
P_2 uses input x_i
P_2 stores the output as $X[i]$
P_2 returns X

We now describe the main protocol for secure evaluation of Π. We let s denote a statistical security parameter, and T denote an upper bound on the total running time of Π.

1. **[Pre-processing phase] Circuit garbling:** P_1 and P_2 agree on the total number $N = O(sT/\log T)$ of garbled circuits to be generated. Then, for each circuit index $i \in \{1, \ldots, N\}$:
   (a) P_1 chooses random input/output wire label descriptions $E^{(i)}$, $D^{(i)}$ and commits to each of these values component-wise under $\mathcal{F}_{\mathsf{xcom}}$.
   (b) P_1 computes $GC^{(i)} = \mathsf{Garble}(\tilde{\Pi}, E^{(i)}, D^{(i)})$ and commits to $GC^{(i)}$ under $\mathcal{F}_{\mathsf{com}}$.
2. **[Pre-processing phase] Cut and choose:** P_2 randomly picks a subset S_c of $\{1, \ldots, N\}$ of size $N/2$ and sends it to P_1. S_c will denote the set of check circuits and $S_e = \{1, \ldots, N\} \setminus S_c$ will denote the set of evaluation circuits. For check circuit index $i \in S_c$:
   (a) P_1 opens the commitments of $E^{(i)}$, $D^{(i)}$, and $GC^{(i)}$.
   (b) P_2 checks that $GC^{(i)} \in \mathsf{Garble}(\tilde{\Pi}, E^{(i)}, D^{(i)})$; if not, P_2 aborts.
3. **Online phase:** For each timestep t:
   (a) **Bucket creation:** P_2 chooses a random subset of $\mathcal{B}_t$ of S_e of size $\Theta(s/\log T)$ and a random head circuit $hd_t \in \mathcal{B}_t$. P_2 announces them to P_1. Both parties set $S_e := S_e \setminus \mathcal{B}_t$.
   (b) **Garbled input: randomness:** P_1 chooses random $r_1 \leftarrow \{0,1\}^n$, and P_2 chooses random $r_{2,1}, \ldots, r_{2,n} \leftarrow \{0,1\}^n$. P_2 sets

$$\mathsf{rand}_1(X_t) = \mathsf{GetInput}_1(\mathsf{rand}_1(E^{(\mathsf{hd}_t)}), r_1)$$
$$\mathsf{rand}_2(X_t) = \mathsf{GetInput}_2(\mathsf{rand}_2(E^{(\mathsf{hd}_t)}), r_{2,1} \cdots r_{2,n})$$

   (c) **Garbled input: state:** If $t > 1$ then the parties execute:

$$\Delta_{\mathsf{st}} = \mathsf{Solder}(\mathsf{st}(D^{(\mathsf{hd}_{t-1})}), \mathsf{st}(E^{(\mathsf{hd}_t)}))$$

   and P_2 sets $\mathsf{st}(X_t) := \mathsf{Adjust}(\mathsf{st}(Y_{t-1}), \Delta_{\mathsf{st}})$.
   Otherwise, in the first timestep, let x_1 and x_2 denote the inputs of P_1 and P_2, respectively. For input wire labels W, let $\mathsf{st}_1(W), \mathsf{st}_2(W), \mathsf{st}_3(W)$ denote the groups of the internal state wires corresponding to the initial state $x_1 \| x_2 \| 0^n$. To prevent selective abort attacks, we must have P_2 encode his input as n-wise independent shares, as above. P_2 chooses random $r_{2,1}, \ldots, r_{2,n} \in \{0,1\}^n$ such that $\sum_i^n r_{2,i} = x_2$, and sets:[6]

$$\begin{aligned}\mathsf{st}(X_t) = \mathsf{GetInput}_1(\mathsf{st}_1(E^{(\mathsf{hd}_t)}), x_1) \\ \| \ \mathsf{GetInput}_2(\mathsf{st}_2(E^{(\mathsf{hd}_t)}), r_{2,1} \cdots r_{2,n}) \\ \| \ \mathsf{GetInput}_{\mathsf{pub}}(\mathsf{st}_3(E^{(\mathsf{hd}_t)}), 0^n)\end{aligned}$$

[6] We are slightly abusing notation here. More precisely, the parties are evaluating a slightly different circuit $\tilde{\Pi}$ in the first timestep than other timesteps. In the first

(d) **Garbled input: memory block:** If the previous instruction $\mathsf{inst}_{t-1} = (\text{READ}, \ell)$ and no previous (WRITE, ℓ) instruction has happened, or if the previous instruction was not a READ, then the parties do $\mathsf{block}(X_t) = \mathsf{GetInput}_{\mathsf{pub}}(\mathsf{block}(E^{(\mathsf{hd}_t)}), 0^n)$.
Otherwise, if $\mathsf{inst}_{t-1} = (\text{READ}, \ell)$ and t' is the largest time step with $\mathsf{inst}_{t'} = (\text{WRITE}, \ell)$, then the parties execute:

$$\Delta_{\mathsf{block}} = \mathsf{Solder}(\mathsf{block}(D^{(\mathsf{hd}_{t'})}), \mathsf{block}(E^{(\mathsf{hd}_t)}))$$

Then P_2 sets $\mathsf{block}(X_t) := \mathsf{Adjust}(\mathsf{block}(Y_{t'}), \Delta_{\mathsf{block}})$.

(e) **Construct bucket:** P_1 and P_2 run subprotocol $\mathsf{MkBucket}(\mathcal{B}_t, \mathsf{hd}_t)$ to assemble the circuits.

(f) **Circuit evaluation:** For each $i \in \mathcal{B}_t$, P_1 opens the commitment to $GC^{(i)}$ and to $\tau(\mathsf{inst}(D^{(i)}))$. P_2 does $Y_t = \mathsf{EvalBucket}(\mathcal{B}_t, X_t, \mathsf{hd}_t)$.

(g) **Output authenticity:** P_2 sends $\tilde{Y} = \mathsf{inst}(Y_t)$ to P_1. Both parties decode the output $\mathsf{inst}_t = \mathsf{lsb}(\tilde{Y}) \oplus \tau(\mathsf{inst}(D^{(\mathsf{hd}_t)}))$. P_1 aborts if the claimed wire labels $\tilde{Y}$ do not equal the expected wire labels $\mathsf{inst}(D^{(\mathsf{hd}_t)})|^*_{\mathsf{inst}_t}$. If $\mathsf{inst}_t = (\text{HALT}, z)$, then both parties halt with output z.

3.3 Security proof

Due to page limits, we give only an overview of the simulator $\mathcal{S}$ and security proof. The complete details are deferred to the full version [1].

Assumptions. The security of our protocol relies on the security underlying functionalities, i.e. $\mathcal{F}_{\mathsf{xcom}}, \mathcal{F}_{\mathsf{com}}, \mathcal{F}_{\mathsf{ot}}$, a garbling scheme satisfying properties discussed in Section 2.2, and an ORAM scheme satisfying standard properties discussed in Section 2.1. All the functionalities can be instantiated using standard number theoretic assumptions, and for UC security would be in the CRS model. The garbling scheme can be instantiated using a standard PRF, or using stronger assumptions such as correlation-secure hash functions for taking advantage of free-XOR. As noted earlier, we do not require the garbling scheme to be adaptively secure, but if so, we can simplify the protocol by not committing to the garbled circuits.

When P_1 is corrupted: The pre-processing phase does not depend on party's inputs, so it is trivial to simulate the behavior of an honest P_2. However, $\mathcal{S}$ can obtain P_1's commitments to all circuits and wire labels. Hence, it can determine whether each of these circuits is correct.

In each timestep t of the online phase, $\mathcal{S}$ can abort if an bucket is constructed with a majority of incorrect circuits; this happens with only negligible probability. $\mathcal{S}$ can abort just as an honest P_2 would abort if P_1 cheats in the Solder,

timestep, it is P_2's input x_2 that is encoded randomly, whereas in the other steps it is P_2's share r_2 of the random tape. However, the difference between these circuits is only in the addition of new XOR gates, and only at the input level. When using the Free-XOR optimization, these gates can actually be added after the fact, so the difference is compatible with our pre-processing.

$\mathsf{GetInput}_1$, or $\mathsf{GetInput}_{\mathsf{pub}}$ subprotocols. Using a standard argument from [24], $\mathcal{S}$ can also match (up to a negligible difference) the probability of an honest P_2 aborting due to cheating in the $\mathsf{GetInput}_2$ subprotocol. $\mathcal{S}$ can extract P_1's input x_1 in timestep $t = 1$ by comparing the sent wire labels to the committed wire labels extracted in the offline phase. $\mathcal{S}$ can send x_1 to the ideal functionality and receive the output z. Then $\mathcal{S}$ generates a simulated ORAM memory-access sequence. Each time in step (3g), $\mathcal{S}$ knows all of the relevant wire labels so can send wire labels $\tilde{Y}$ chosen to encode the desired simulated ORAM memory instruction.

When P_2 is corrupted: In the pre-processing phase, $\mathcal{S}$ simulates commit messages from $\mathcal{F}_{\mathsf{com}}$. After receiving S_c from P_2, it equivocates the opening of the check sets to honestly garbled circuits and wire labels.

In each timestep t of the online phase, $\mathcal{S}$ sends random wire labels in the $\mathsf{GetInput}_1$ and $\mathsf{GetInput}_{\mathsf{pub}}$ subprotocols, and also simulates random wire labels as the output of $\mathcal{F}_{\mathsf{ot}}$ in the $\mathsf{GetInput}_2$ subprotocols. These determine the wire labels that are "visible" to P_2. $\mathcal{S}$ also extracts P_2's input x_2 from its select bits sent to $\mathcal{F}_{\mathsf{ot}}$. It sends x_2 to the ideal functionality and receives the output z. Then $\mathcal{S}$ generates a simulated ORAM memory-access sequence.

In the Solder steps, $\mathcal{S}$ equivocates soldering values chosen to map visible wire labels to their counterparts in other circuits, and chooses random soldering values for the non-visible wire labels. When it is time to open the commitment to the garbled circuit, $\mathcal{S}$ chooses a random set of visible output wire labels and equivocates to a simulated garbled circuit generated using only these visible wire labels. $\mathcal{S}$ also equivocates on the decommitment to the decoding information $\tau(\mathsf{inst}(D^{(i)}))$, chosen so that the visible output wires will decode to the next simulated ORAM memory instruction. Instead of checking P_2's claimed wire labels in step (3g), the simulator simply aborts if these wire labels are not the pre-determined visible output wire labels.

3.4 Efficiency and Parameter Analysis

In the offline phase, the protocol is dominated by the generation of many garbled circuits, $O(sT/\log T)$ in all. In the full version [1], we describe computation of the exact constant. As an example, for $T = 1$ million, and to achieve statistical security 2^{-40}, it is necessary to generate $10 \cdot T$ circuits in the offline phase.

In the online phase, the protocol is dominated by two factors: the homomorphic decommitments within the Solder subprotocol, and the oblivious transfers (in $\mathsf{GetInput}_2$) in which P_2 receives garbled inputs. For the former, we require one decommitment for each input and output wire label (to solder that wire to another wire) of the circuit $\tilde{\Pi}$. Hence the cost in each timestep is proportional to the input/output size of the circuit and the size of the buckets. Continuing our example from above ($T = 10^6$ and $s = 40$), buckets of size 5 are sufficient.

In the full version [1], we additionally discuss parameter settings for when the parties open a different fraction (i.e., not 1/2) of circuits in the cut-and-choose phase. By opening a smaller fraction in the offline phase, we require fewer

circuits overall, at the cost of slightly more circuits per timestep (i.e., slightly larger buckets) in the online phase.

We require one oblivious transfer per input bit of P_2 per timestep (independent of the size of buckets). P_2's input is split in an s-way secret share to assure input-dependent failure probabilities, leading to a total of sn OTs per timestep (where n is the number of random bits required by $\tilde{\Pi}$). However, online oblivious transfers are inexpensive (requiring only few symmetric-key operations) when instantiated via OT extension [16,2], where the more expensive "seed OTs" will be done in the pre-processing phase. In Section 5 we suggest further ways to reduce the required number of OTs in the online phase.

Overall, the online overhead of this protocol (compared to the semi-honest setting) is dominated by the bucket size, which is likely at most 5 or 7 for most reasonable settings.

In terms of memory requirements, P_1 must store all pre-processed garbled circuits, and P_2 must store all of their commitments. For each bit of RAM memory, P_1 must store the two wire labels (and their decommitment info) corresponding to that bit, from the last write-time of that memory location. P_2 must store only a single wire label per memory bit.

4 Streaming Cut-and-choose Protocol

4.1 High-level Overview

The standard **cut-and-choose approach** is (for evaluating a *single circuit*) for the sender P_1 to garble $O(s)$ copies of the circuit, and receiver P_2 to request half of them to be opened. If all opened circuits are correct, then with overwhelming probability (in s) a majority of the unopened circuits are correct as well.

When trying to apply this methodology to our setting, we face the challenge of feeding past outputs (internal state, memory blocks) into future circuits. Naïvely doing a separate cut-and-choose for each timestep of the RAM program leads to problems when reusing wire labels. Circuits that are opened and checked in time step t must have wire labels independent of past circuits (so that opening these circuits does not leak information about past garbled outputs). Circuits used for evaluation must be garbled with input wire labels *matching* output wire labels of past circuits. But the security of cut and choose demands that P_1 cannot know, at the time of garbling, which circuits will be checked or used for evaluation.

Our alternative is to use a technique suggested by [31] to perform a single cut-and-choose that applies to all timesteps. We make $O(s)$ independent **threads** of execution, where wire labels are directly reused only within a single thread. A cut-and-choose step at the beginning determines whether each *entire thread* is used for checking or evaluation. Importantly, this is done using an oblivious transfer (as in [18,22]) so that P_1 does not learn the status of the threads.

More concretely, for each thread the parties run an oblivious transfer allowing P_2 to pick up either k_{check} or k_{eval}. Then at each timestep, P_1 sends the garbled circuit but also encrypts the *entire set* of wire labels under k_{check} and encrypts

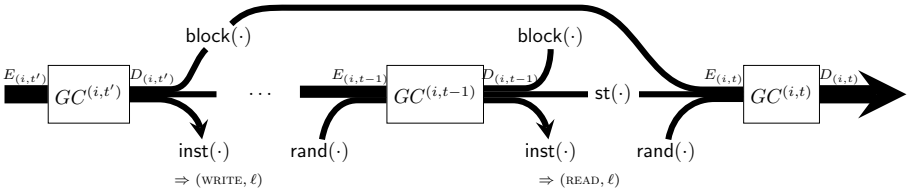

Fig. 4. Wire-label reuse within a single thread i, in the streaming cut-and-choose protocol.

wire labels for only her input under k_{eval}. Hence, in check threads P_2 receives enough information to verify correct garbling of the circuits (including reuse of wire labels — see below), but learns nothing about P_1's inputs. In evaluation threads, P_2 receives only P_1's garbled input and the security property of garbled circuits applies. If P_1 behaves incorrectly in a *check thread*, P_2 aborts immediately. Hence, it is not hard to see that P_1 cannot cause a majority of evaluation threads to be faulty while avoiding detection in *all* check threads, except with negligible probability.

Reusing wire labels is fairly straight-forward since it occurs only within a single thread. The next circuit in the thread is simply garbled with input wire labels matching the appropriate output wire labels in the same thread (i.e., the state output of the previous circuit, and possibly the memory-block output wires of an earlier circuit). We point out that P_1 must know the previous memory instruction before garbling the next batch of circuits: if the instruction was (READ, ℓ), then the next circuit must be garbled with wire labels matching those of the last circuit to write to memory location ℓ. Hence this approach is not compatible with batch pre-processing of garbled circuits.

For enforcing consistency of P_1's input, we use the approach of [38][7], where the very first circuit is augmented to compute a "hiding" universal hash of P_1's input. For efficiency purposes, the hash is chosen as $M \cdot (x_1 \| r)$, where M is a random binary matrix M of size $s \times (n + 2s + \log s)$ chosen by P_2. We prevent input-dependent abort based on P_2's input using the XOR-tree approach of [24], also used in the previous protocol.

We ensure authenticity of the output for P_1 using an approach suggested in [30]. Namely, wire labels corresponding to the same output wire and truth value are used to encrypt a random "output authenticity" key. Hence P_2 can compute these output keys only for the circuit's true output. P_2 is not given the information required for checking these ciphertexts until after he *commits* to the output keys. At the time of committing, he cannot guess complementary output keys, but he does not actually open the commitment until he receives the checking information and is satisfied with the check circuits.

The adaptation of the input-recovery technique of Lindell [23] is more involved and hence we discuss it separately in Section 4.5.

[7] although our protocol is also compatible with the solution of [30].

4.2 Detailed Protocol Description

We now describe the streaming cut-and-choose protocol for secure evaluation of Π, the ORAM program to be computed. Recall that $\tilde{\Pi}(\mathsf{st}, \mathsf{block}, \mathsf{inp}_1, \mathsf{inp}_{2,1}, \ldots, \mathsf{inp}_{2,n}) = \Pi(\mathsf{st}, \mathsf{block}, \mathsf{inp}_1 \bigoplus_i \mathsf{inp}_{2,i})$. We let s denote a statistical security parameter parameter, and T denote an upper bound on the total running time of Π. Here, we describe the majority-evaluation variant of the protocol and discuss how to integrate the input-recovery technique in Section 4.5.

1. **Cut-and-choose.** The parties agree on $S = O(s)$, the number of threads (see discussion below). P_2 chooses a random string $b \leftarrow \{0,1\}^S$. Looking ahead, thread i will be a check thread if $b_i = 0$ and an evaluation thread if $b_i = 1$.
   For each $i \in \{1, \ldots, S\}$, P_1 chooses two symmetric encryption keys $k_{(i,check)}$ and $k_{(i,eval)}$. The parties invoke an instance of $\mathcal{F}_{\mathsf{ot}}$ with P_2 providing input b_i and P_1 providing input $(k_{(i,check)}, k_{(i,eval)})$.
2. **RAM evaluation.** For each timestep t, the following are done in parallel for each thread $i \in \{1, \ldots, S\}$:
   (a) **Wire label selection.** P_1 determines the input wire labels $E_{(t,i)}$ for garbled circuit $GC_{(t,i)}$ as follows. If $t = 1$, these wire labels are chosen uniformly. Otherwise, we set $\mathsf{st}(E_{(t,i)}) = \mathsf{st}(D_{(t-1,i)})$ and choose $\mathsf{rand}_1(E_{(t,i)})$ and $\mathsf{rand}_2(E_{(t,i)})$ uniformly. If the previous instruction $\mathsf{inst}_{t-1} = (\text{READ}, \ell)$ and no previous (WRITE, ℓ) instruction has happened, or if the previous instruction was not a READ, then P_1 chooses $\mathsf{block}(E_{(t,i)})$ uniformly at random. Otherwise, we set $\mathsf{block}(E_{(t,i)}) = \mathsf{block}(D_{(t',i)})$, where t' is the last instruction that wrote to memory location ℓ.
   (b) **Input selection.** Parties choose shares of the randomness required for $\tilde{\Pi}$: P_1 chooses $r_1 \leftarrow \{0,1\}^n$, and P_2 chooses $r_{2,1}, \ldots, r_{2,n} \leftarrow \{0,1\}^n$.
   (c) **P_1's garbled input transfer.** P_1 sends the following wire labels, encrypted under $k_{(i,eval)}$:

$$\begin{cases} \mathsf{st}_1(E_{(t,i)})|^*_{x_1} & \text{if } t = 1 \\ \mathsf{rand}_1(E_{(t,i)})|^*_{r_1} \end{cases}$$

   The following additional wire labels are also sent in the clear:

$$\begin{cases} \mathsf{st}_3(E_{(t,i)})|^*_{0^n} & \text{if } t = 1 \\ \mathsf{block}(E_{(t,i)})|^*_{0^n} & \text{if WRITE or uninitialized READ} \end{cases}$$

   (d) **P_2's garbled input transfer.** P_2 obtains garbled inputs via calls to OT. To guarantee that P_2 uses the same input in all threads, we use a single OT across all threads for each input bit of P_2. For each input bit, P_1 provides the true and false wire labels for all threads as input to $\mathcal{F}_{\mathsf{ot}}$, and P_2 provides his input bit as the OT select bit.
   Note that P_2's inputs consist of the strings $r_{2,1}, \ldots, r_{2,n}$ as well as the string x_2 for the case of $t = 1$.

(e) **Input consistency.** If $t = 1$, then P_2 sends a random $s \times (n + 2s + \log s)$ binary matrix M to P_1. P_1 chooses random input $r \in \{0,1\}^{2s+\log s}$, and augments the circuit for $\tilde{\Pi}$ with a subcircuit for computing $M \cdot (x_1 \| r)$.

(f) **Circuit garbling.** P_1 chooses output wire labels $D_{(t,i)}$ at random and does $GC^{(t,i)} = \mathsf{Garble}(\tilde{\Pi}, E_{(t,i)}, D_{(t,i)})$, where in the first timestep, $\tilde{\Pi}$ also contains the additional subcircuit described above. P_1 sends $GC^{(t,i)}$ to P_2 as well as $\tau(\mathsf{inst}(D_{(t,i)}))$.
In addition, P_1 chooses a random Δ_t for this time-step and for each inst-output bit j, he chooses random strings $w_{(t,j,0)}$ and $w_{(t,j,1)}$ (the same across all threads) to be used for output authenticity, such that $w_{(t,j,0)} \oplus w_{(t,j,1)} = \Delta_t$. For each thread i, output wire j and select bit b corresponding to truth value b', let $v_{i,j,b}$ denote the corresponding wire label. P_1 computes $c_{i,j,b} = \mathsf{Enc}_{v_{i,j,b}}(w_{(t,j,b')})$ and $h_{i,j,b} = H(c_{i,j,b})$, where H is a 2-Universal hash function. P_1 sends $h_{i,j,b}$ in the clear and sends $c_{i,j,b}$ encrypted under $k_{(eval,i)}$.

(g) **Garbled input collection.** If thread i is an evaluation thread, then P_2 assembles input wire labels $X_{(t,i)}$ for $GC^{(t,i)}$ as follows:
P_2 uses $k_{(eval,i)}$ to decrypt wire labels sent by P_1. Along with the wire labels sent in the clear and those obtained via OTs in $\mathsf{GetInput}_2$, these wire labels will comprise $\mathsf{rand}(X_{(t,i)})$; $\mathsf{block}(X_{(t,i)})$ in the case of a WRITE or uninitialized READ; and $\mathsf{st}(X_{(t,i)})$ when $t = 1$.
Other input wire labels are obtained via:

$$\mathsf{st}(X_{(t,i)}) = \mathsf{st}(Y_{(t-1,i)})$$
$$\mathsf{block}(X_{(t,i)}) = \mathsf{block}(Y_{(t',i)})$$

where t' is the last write time of the appropriate memory location, and Y denote the output wire labels that P_2 obtained during previous evaluations.

(h) **Evaluate and commit to output.** If thread i is an eval thread, then P_2 evaluates the circuit via $Y_{(t,i)} = \mathsf{Eval}(GC^{(t,i)}, X_{(t,i)})$ and decodes the output $\mathsf{inst}_{(t,i)} = \mathsf{lsb}(Y_{(t,i)}) \oplus \tau(D_{(t,i)})$. He sets $\mathsf{inst}_t = \text{majority}_i\{\mathsf{inst}_{(t,i)}\}$. For each inst-output wire label j, P_2 decrypts the corresponding ciphertext $c_{i,j,b}$, then takes w'_j to be the majority result across all threads i. P_2 commits to w'_j.
If $t = 1$, then P_2 verifies that the output of the auxiliary function $M \cdot (x_1 \| r)$ is identical to that of all other threads; if not, he aborts.

(i) **Checking the check threads.** P_1 sends $\mathsf{Enc}_{k_{(i,check)}}(seed_{(t,i)})$ to P_2, where $seed_{(t,i)}$ is the randomness used in the call to Garble. Then if thread i is a check thread, P_2 checks the correctness of $GC^{(t,i)}$ as follows. By induction, P_2 knows all the previous wire labels in thread i, so can use $seed_{(t,i)}$ to verify that $GC^{(t,i)}$ is garbled using the correct outputs. In doing so, P_2 learns all of the output wire labels for $GC^{(t,i)}$ as well. P_2 checks that the wire labels sent by P_1 in the clear are as specified in the protocol, and that the $c_{i,j,b}$ ciphertexts and $h_{i,j,b}$ are correct and consistent. He also decrypts $c_{i,j,b}$ for

$b \in \{0,1\}$ with the corresponding output label to recover $w'_{(t,j,b)}$ and checks that $w'_{(t,j,0)} \oplus w'_{(t,j,1)}$ is the same for all j. Finally, P_2 checks that the wire labels obtained via OT in $\mathsf{GetInput}_2$ are the correct wire labels encoding P_2's provided input. If any of these checks fail, then P_2 aborts immediately.

(j) **Output verification.** P_2 opens the commitments to values w'_j and P_1 uses them to decode the output inst_t. If a value w'_j does not match one of $w_{(t,j,0)}$ or $w_{(t,j,1)}$, then P_1 aborts.

4.3 Security Proof

Again we only give a brief overview of the simulator, with the details deferred to the full version [1].

The security of the protocol relies on functionalities $\mathcal{F}_{\mathsf{com}}, \mathcal{F}_{\mathsf{ot}}$ which can both be instantiated under number theoretic assumptions in the CRS model, a secure garbling scheme and an ORAM scheme satisfying standard properties discussed earlier. More efficiency can be obtained using RO or correlation-secure hash functions, to take advantage of the free-XOR technique for garbling (and faster input-consistency checks), or the use of fast OT extension techniques.

When P_1 is corrupt: In the cut-and-choose step, the simulator $\mathcal{S}$ extracts both encryption keys $k_{(i,eval)}$ and $k_{(i,check)}$. Just as P_2, the simulator designates half of the threads to be check threads and half to be eval threads, and aborts if a check thread is ever found to be incorrect. However, the simulator can perform the same check for all threads, and keeps track of which eval threads are correct. A standard argument shows that if all check threads are correct, then a majority of eval threads are also correct, except with negligible probability. Without loss of generality, we can have $\mathcal{S}$ abort if this condition is ever violated.

Knowing both encryption keys, $\mathcal{S}$ can associate P_1's input wire labels with truth values (at least in the correct threads). If P_1 provides disagreeing inputs x_1 among the correct eval threads, then $\mathcal{S}$ aborts, which is negligibly close to P_2's abort probability (via the argument regarding the input-consistency of [38]). Otherwise, this determines P_1's input x_1 which $\mathcal{S}$ sends to the ideal functionality, receiving output z in return. $\mathcal{S}$ generates a simulated ORAM memory access pattern.

In the output commitment step, $\mathcal{S}$ simulates a commit message. Then after the check phase, $\mathcal{S}$ learns all of the output-authenticity keys. So $\mathcal{S}$ simply equivocates the opening of the output keys to be the ones encoding the next ORAM memory instruction.

When P_2 is corrupt: In the cut-and-choose phase, $\mathcal{S}$ extracts P_2's selection of check threads and eval threads. In check threads, $\mathcal{S}$ always sends correctly generated garbled circuits, following the protocol specification and generates dummy ciphertexts for the encryptions under $k_{(i,eval)}$. Hence, these threads can be simulated independently of P_1's input.

In each eval thread, $\mathcal{S}$ maintains visible input/output wire labels for each circuit, chosing new output wire labels at random. $\mathcal{S}$ ensures that P_2 picks up

these wire labels in the input collection step. $\mathcal{S}$ also extracts P_2's input x_2 in this phase, from its select bit inputs to $\mathcal{F}_{\mathsf{ot}}$. $\mathcal{S}$ sends x_2 to the ideal functionality and receives output z. Then $\mathcal{S}$ generates a simulated ORAM memory access pattern.

At each timestep, for each eval thread, $\mathcal{S}$ generates a simulated garbled circuit, using the appropriate visible input/output wire labels. It fixes the decoding information τ so that the visible output wire labels will decode to the appropriate ORAM instruction. In the output reveal step, $\mathcal{S}$ aborts if P_2 does not open its commitment to the expected output keys. Indeed, P_2's view in the simulation is independent of the complementary output keys.

4.4 Efficiency and Parameter Analysis

At each timestep, the protocol is dominated by the generation of S garbled circuits (where S is the number of threads) as well as the oblivious transfers for P_2's inputs. As before, using OT extension as well as the optimizations discussed in Section 5, the cost of the oblivious transfers can be significantly minimized. Other costs in the protocol include simple commitments and symmetric encryptions, again proportional to the number of threads. Hence the major computational overhead is simply the number of threads. An important advantage of this protocol is that we avoid the soldering and the "expensive" xor-homomorphic commitments needed for input/outputs of each circuit in our batching solution. On the other hand, this protocol always require $O(s)$ garbled circuit executions regardless of the size of the RAM computation, while as discussed earlier, our batching protocol can require significantly less garbled circuit execution when the running time T is large. The choice of which protocol to use would then depend on the running time of the RAM computation, the input/output size of the next-instruction circuits as well as practical efficiency of xor-homomorphic commitment schemes in the future.

Compared to our other protocol, this one has a milder memory requirement. Garbled circuits are generated on the fly and can be discarded after they are used, with the exception of the wire labels that encode memory values. P_1 must remember $2S$ wire labels per bit of memory (although in Section 5 we discuss a way to significantly reduce this requirement). P_2 must remember between S and $2S$ wire labels per bit of memory (1 wire label for evaluation threads, 2 wire labels for check threads).

Using the standard techniques described above, we require $S \approx 3s$ threads to achieve statistical security of 2^{-s}. Recently, techniques have been developed [23] for the SFE setting that require only s circuits for security 2^{-s} (concretely, s is typically taken to be 40). We now discuss the feasibility of adapting these techniques to our protocol:

4.5 Integrating Cheating Recovery

The idea of [23] is to provide a mechanism that would detect inconsistency in the output wire labels encoding the final output of the computation. If P_2 receives output wire labels for two threads encoding disparate values, then a secondary

computation allows him to recover P_1's input (and hence compute the function himself). This technique reduces the number of circuits necessary by a factor of 3 since we only need a single honest thread among the set of evaluated threads (as opposed to a majority). We refer the reader to [23] for more details. We point out that in some settings, recovering P_1's input may not be enough. Rather, if P_2 is to perform the entire computation on his own in the case of a cheating P_1, then he also needs to know the contents of the RAM memory!

Cheating recovery at each timestep. It is possible to adapt this approach to our setting, by performing an input-recovery computation at the end of each timestep. But this would be very costly, since each input-recovery computation is a maliciously secure 2PC that requires expensive input-consistency checks for both party's inputs, something we worked hard to avoid for the state/memory bits. Furthermore, each cheating-recovery garbled circuit contains non-XOR gates that need to be garbled/evaluated $3s$ times at each timestep. These additional costs can become a bottleneck in the computation specially when the next-instruction circuit is small.

Cheating recovery at the end. It is natural to consider delaying the input-recovery computation until the last timestep, and only perform it once. If two of the threads in the final timestep (which also computes the final output of computation) output different values, the evaluator recovers the garbler's input. Unfortunately, however, this approach is not secure. In particular, a malicious P_1 can cheat in an intermediate timestep by garbling one or more incorrect circuits. This could either lead to two or more valid memory instruction/location outputs, or no valid outputs at all. It could also lead to a premature "halt" instruction. In either case, P_2 cannot yet abort since that would leak extra information about his private input. He also cannot continue with the computation because he needs to provide P_1 with the next instruction along with proof of its authenticity (i.e. the corresponding garbled labels) but that would reveal information about his input.

We now describe a solution that avoids the difficulties mentioned above and at the same time eliminates the need for input-consistency checks or garbling/evaluating non-XOR gates at each timestep. In particular, we delay the "proof of authenticity" by P_2 for all the memory instructions until after the last timestep. Whenever P_2 detects cheating by P_1 (i.e. more than two valid memory instructions), instead of aborting, he pretends that the computation is going as planned and sends "dummy memory operations" to P_1 but does not (and cannot) prove the authenticity of the corresponding wire labels yet. For modern tree-based ORAM constructions ([40,7], etc) the memory access pattern is always uniform, so it is easy for P_2 to switch from reporting the real memory access pattern to a simulated one. Note that in step (h) of the protocol, P_2 no longer needs to commit to the majority w'_j. As a result, step (j) of the protocol will be obsolete. Instead, in step (h), P_2 sends the inst_t in plaintext. This instruction is the single valid instruction he has recovered or a dummy instruction (if P_2 has attempted to cheat).

After the evaluation of the final timestep, we perform a fully secure 2PC for an input-recovery circuit that has two main components. The first one checks if P_1 has cheated. If he has, it reveals P_1's input to P_2. The second one checks the proofs of authenticity of the inst instructions P_2 reveals in all timesteps and signals to P_1 to abort if the proof fails.

First cheating recovery, then opening the check circuits. For this cheating recovery method to work, we perform the evaluation steps (step (h)) for all time-steps first (at this stage, P_2 only learns the labels for the final output but not the actual value), then perform the cheating recovery as described above, and finally perform all the checks (step (i)) for all time-steps.

We now describe the cheating recovery circuit which consists of two main components in more detail.

- The first component is similar to the original cheating recovery circuit of [23]. P_2's input is the XOR of two valid output authenticity labels for a wire j at step t for which he has detected cheating (if there is more than one instance of cheating he can use the first occurrence). Lets denote the output authenticity labels for jth bit of $\mathsf{block}(Y_{(t,i)})$ at time-step t with $w_{(t,j,b)}, b \in \{0,1\}$. Then P_2 will input $w_{(t,j,0)} \oplus w_{(t,j,1)}$ to the circuit. If there is no cheating, he inputs garbage. Notice that $w_{(t,j,0)} \oplus w_{(t,j,1)} = \Delta_t$ for valid output authenticity values, as described in the protocol (note that we assume that all output authenticity labels in timestep t use the same offset Δ_t).
  P_1 inputs his input x_1. He also hardcodes Δ_t. For timestep t (as shown in Figure 5) the circuit compares P_2's input against the hardcoded Δ_t. If P_2's input is the same as the Δ_t, cheating is detected and the circuit outputs 1. To check that P_2's input is the same as at least one of the hard-coded Δs, in the circuit of Figure 6 we compute the OR of all these outputs. Thus, if the output of this circuit is 1, it means that P_1 has cheated in at least one timestep.
  To reveal P_1's input, we compute the AND of output of circuit of Figure 6 with each bit of P_1's input as depicted in Figure 7. This concludes the description of the first component for cheating recovery.
- In the second component, we check the authenticity of the memory instructions P_2 provided in all timesteps. In particular, he provides the hash of concatenation of all output authentication labels he obtained during the evaluation corresponding to inst in all timesteps (P_2 uses dummy labels if he does not have valid ones due to P_1's cheating), while P_1 does the same based on the plaintext instructions he received from P_2 and the labels which he knows. The circuit then outputs 1 if the two hash values match. The circuit structure is therefore identical to that of Figure 5, but the inputs are the hash values. An output of 0 would mean that P_2 does not have a valid proof of authenticity.
  As shown in the final circuit of Figure 7 then, if P_1 was not already caught cheating in the previous step, and P_2's proof of authenticity fails, the circuit

outputs a 1 to signal an abort to P_1. This is a crucial condition, i.e., it is important to ensure P_1 did not cheat (the output of circuit of Figure 6) before accusing P_2 of cheating, since in case of cheating by P_1 say in timestep t, P_2 may be able to prove authenticity of the instructions for timestep t or later.

Efficiency: Following the techniques of [23], all the gates of Figures 5, and 6 can be garbled using non-cryptographic operations (XORs) and only the circuit of Figure 7 has non-XOR gates. More precisely it requires $|x_1|$ ANDs and a NOT gate.

Of course, the final circuit will be evaluate using a basic maliciously secure 2PC. Thus, we need to add a factor of $3s$ to the above numbers which results in garbling a total of $3s(|x_1|+1)$ non-XOR gates which is at most $12s(|x_1|+1)$ symmetric operations.

The input consistency checks are also done for P_1's input x_1 and P_2's input which is a proof of cheating of length $|\Delta|$ and a proof of authenticity which is the output of a hash function (both are in the order of the computational security parameter). We stress that the gain is significant since both the malicious 2PC and the input consistency cheks are only done once at the end.

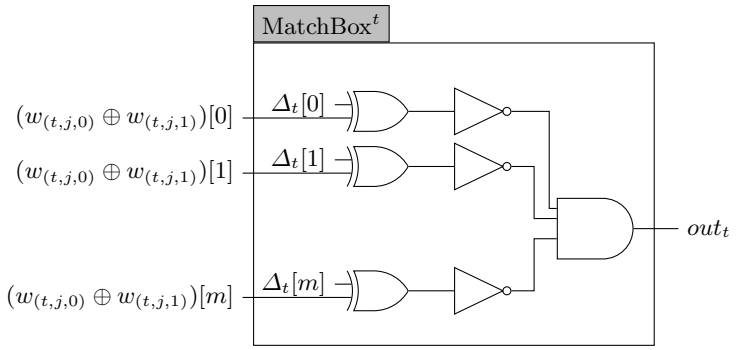

Fig. 5. Cheating recovery component 1: MatchBox. Where $\Delta_t[i]$ denotes the ith bit of Δ_t and $m = |\Delta_t|$.

5 Optimizations

Here we present a collection of further optimizations compatible with our 2PC protocols:

5.1 Hide only the input-dependent behavior

Systems like SCVM [27] use static program analysis to "factor out" as much input-independent program flow as possible from a RAM computation, leaving

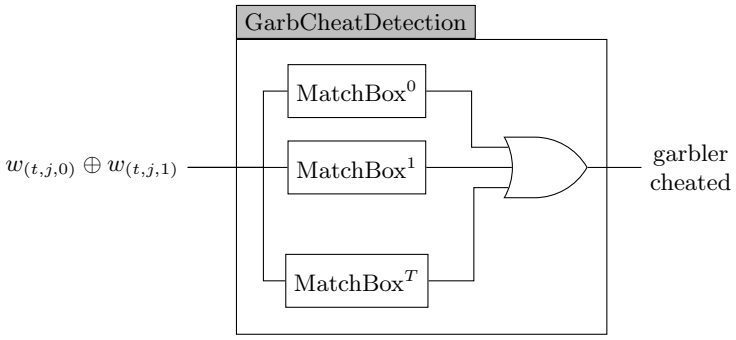

Fig. 6. Cheating Recovery component 1: Garbler Cheating Detection.

significantly less residual computation that requires protection from the 2PC mechanisms.

The backend protocol currently implemented by SCVM achieves security only against semi-honest adversaries. However, our protocols are also compatible with their RAM-level optimizations, which we discuss in more detail:

Special-purpose circuits. For notational simplicity, we have described our RAM programs via a *single* circuit Π that evaluates each timestep. Then Π must contain subcircuits for every low-level instruction (addition, multiplication, etc) that may ever be needed by this RAM program.

Instruction-trace obliviousness means that the choice of low-level instruction (e.g., addition, multiplication) performed at each time t does not depend on private input. The SCVM system can compile a RAM program into an instruction-trace-oblivious one (though one does not need full instruction-trace obliviousness to achieve an efficiency gain in 2PC protocols). For RAM programs with this property, we need only evaluate an (presumably much smaller) instruction-specific circuit Π_t at each timestep t.

It is quite straight-forward to evaluate different circuits at different timesteps in our cut-and-choose protocol of Section 4. For the batching protocol of Section 3, enough instruction-specific circuits must be generated in the pre-processing phase to ensure a majority of correct circuits in each bucket. However, we point out that buckets at different timesteps could certainly be different sizes! One particularly interesting use-case would involve a very aggressive pre-processing of the circuits involved in the ORAM construction (i.e., the logic translating logical memory accesses to physical accesses), since these will dominate the computation and do not depend on the functionality being computed.[8] The bucket size / replication factor for these timesteps could be very low (say, 5), while the less-aggressively pre-processed instructions could have larger buckets. In this case, the plain-RAM internal state could be kept separate from the ORAM-specific internal state, and only fed into the appropriate circuits.

[8] Such pre-processing yields an instance of *commodity-based MPC* [3].

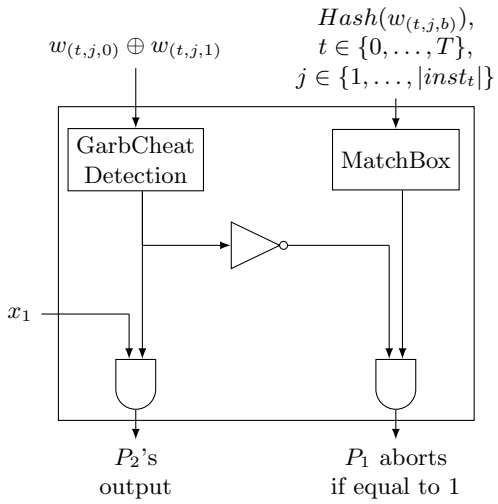

Fig. 7. Final Circuit

Along similar lines, we have for simplicity described RAM programs that require a random input tape at each timestep. This randomness leads to oblivious transfers within the protocol. However, if it is known to both parties that a particular instruction does not require randomness, then these OTs are not needed. For example, deterministic algorithms require randomness only for the ORAM mechanism. Concretely, tree-based ORAM constructions [39,40,7] require only a small amount of randomness and at input-indepenent steps.

Memory-trace obliviousness. Due to their general-purpose nature, ORAM constructions protect *all* memory accesses, even those that may already be input-independent (for example, sequantial iteration over an array). One key feature of SCVM is detecting which memory accesses are already input-independent and not applying ORAM to them. Of course, such optimizations to a RAM program would yield benefit to our protocols as well.

5.2 Reusing memory

We have described our protocols in terms of a single RAM computation on an initially empty memory. However, one of the "killer applications" of RAM computations is that, after an initial quasi-linear-time ORAM initialization of memory, future computations can use time sublinear in the total size of data (something that is impossible with circuits). This requires an ORAM-initialized memory to be reused repeatedly, as in [13].

Our protocols are compatible with reusing garbled memory. In particular, this can be viewed as a single RAM computation computing a reactive functionality (one that takes inputs and gives outputs repeatedly).

5.3 Other Protocol Optimizations

Storage requirements for RAM memory. In our cut-and-choose protocol, P_1 chooses random wire labels to encode bits of memory, and then has to remember these wire labels when garbling later circuits that read from those locations. As an optimization, P_1 could instead choose wire labels via $F_k(t, j, i, b)$, where F is a suitable PRF, t is the timestep in which the data was written, j is the index of a thread, i is the bit-offset within the data block, and b is the truth value. Since memory *locations* are computed at run-time, P_1 cannot include the memory location in the computation of these wire labels. Hence, P_1 will still need to remember, for each memory location ℓ, the last timestep t at which location ℓ was written.

Adaptive garbling. In the batching protocol, P_1 must commit to the garbled circuits and reveal them only after P_2 obtains the garbled inputs. This is due to a subtle issue of (non)adaptivity in standard security definitions of garbled circuits; see [4] for a detailed discussion. These commitments could be avoided by using an adaptively-secure garbling scheme.

Online/offline tradeoff. For simplicity we described our online/offline protocol in which P_1 generates many garbled circuits and P_2 opens exactly half of them. Lindell and Riva [26] also follow a similar approach of generating many circuits in an offline phase and assigning the remainder to random buckets; they also point out that changing the fraction of opened circuits results in different tradeoffs between the amount of circuits used in the online and offline phases. For example, checking 20% of circuits results in fewer circuits overall (i.e., fewer generated in the offline phase) but larger buckets (in our setting, more garbled circuits per timestep in the online phase).

References

1. A. Afshar, Z. Hu, P. Mohassel, and M. Rosulek. How to efficiently evaluate ram programs with malicious security. Cryptology ePrint Archive, Report 2014/759, 2014. http://eprint.iacr.org/.
2. G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Sadeghi et al. [35], pages 535–548.
3. D. Beaver. Commodity-based cryptography (extended abstract). In *29th Annual ACM Symposium on Theory of Computing*, pages 446–455. ACM Press, May 1997.
4. M. Bellare, V. T. Hoang, and P. Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In X. Wang and K. Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 134–153. Springer, Dec. 2012.
5. M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In Yu et al. [43], pages 784–796.
6. R. Canetti and J. A. Garay, editors. *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*. Springer, Aug. 2013.

7. K.-M. Chung and R. Pass. A simple ORAM. Cryptology ePrint Archive, Report 2013/243, 2013. http://eprint.iacr.org/2013/243.
8. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [36], pages 643–662.
9. T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, P. S. Nordholt, and C. Orlandi. MiniLEGO: Efficient secure two-party computation from general assumptions. In Johansson and Nguyen [17], pages 537–556.
10. C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs. Garbled RAM revisited. In *EUROCRYPT*, 2014.
11. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In A. Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229. ACM Press, May 1987.
12. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.
13. S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In Yu et al. [43], pages 513–524.
14. Y. Huang, J. Katz, and D. Evans. Efficient secure two-party computation using symmetric cut-and-choose. In Canetti and Garay [6], pages 18–35.
15. Y. Huang, J. Katz, V. Kolesnikov, R. Kumaresan, and A. J. Malozemoff. Amortizing garbled circuits. In *Advances in Cryptology – CRYPTO 2014.*, 2014.
16. Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In D. Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, Aug. 2003.
17. T. Johansson and P. Q. Nguyen, editors. *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*. Springer, May 2013.
18. S. Kamara, P. Mohassel, and B. Riva. Salus: a system for server-aided secure function evaluation. In Yu et al. [43], pages 797–808.
19. M. Keller and P. Scholl. Efficient, oblivious data structures for MPC. Cryptology ePrint Archive, Report 2014/137, 2014. http://eprint.iacr.org/.
20. M. Kiraz and B. 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.
21. V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, July 2008.
22. B. Kreuter, a. shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 14–14. USENIX Association, 2012.
23. Y. Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In Canetti and Garay [6], pages 1–17.
24. Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In M. Naor, editor, *Advances in Cryptology – EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 52–78. Springer, May 2007.

25. Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Y. Ishai, editor, *TCC 2011: 8th Theory of Cryptography Conference*, volume 6597 of *Lecture Notes in Computer Science*, pages 329–346. Springer, Mar. 2011.
26. Y. Lindell and B. Riva. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In J. A. Garay and R. Gennaro, editors, *CRYPTO (2)*, volume 8617 of *Lecture Notes in Computer Science*, pages 476–494. Springer, 2014.
27. C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks. Automating efficient RAM-model secure computation. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, May 2014.
28. S. Lu and R. Ostrovsky. How to garble RAM programs. In Johansson and Nguyen [17], pages 719–734.
29. P. Mohassel and M. Franklin. Efficiency tradeoffs for malicious two-party computation. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 458–473. Springer, Apr. 2006.
30. P. Mohassel and B. Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In Canetti and Garay [6], pages 36–53.
31. B. Mood, D. Gupta, J. Feigenbaum, and K. Butler. Reuse It Or Lose It: More Efficient Secure Computation Through Reuse of Encrypted Values. In *ACM CCS*, 2014.
32. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [36], pages 681–700.
33. J. B. Nielsen and C. Orlandi. LEGO for two-party secure computation. In O. Reingold, editor, *TCC 2009: 6th Theory of Cryptography Conference*, volume 5444 of *Lecture Notes in Computer Science*, pages 368–386. Springer, Mar. 2009.
34. B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In M. Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, Dec. 2009.
35. A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors. *ACM CCS 13: 20th Conference on Computer and Communications Security*. ACM Press, Nov. 2013.
36. R. Safavi-Naini and R. Canetti, editors. *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*. Springer, Aug. 2012.
37. a. shelat and C.-H. Shen. Two-output secure computation with malicious adversaries. In K. G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 386–405. Springer, May 2011.
38. a. shelat and C.-H. Shen. Fast two-party secure computation with minimal assumptions. In Sadeghi et al. [35], pages 523–534.
39. E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In D. H. Lee and X. Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 197–214. Springer, Dec. 2011.
40. E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Sadeghi et al. [35], pages 299–310.

41. A. C.-C. Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164. IEEE Computer Society Press, Nov. 1982.
42. A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE Computer Society Press, Oct. 1986.
43. T. Yu, G. Danezis, and V. D. Gligor, editors. *ACM CCS 12: 19th Conference on Computer and Communications Security*. ACM Press, Oct. 2012.
44. S. Zahur. Obliv-c: A lightweight compiler for data-oblivious computation. Workshop on Applied Multi-Party Computation. Microsoft Research, Redmond, 2014.