Multi-Party Oblivious RAM based on Function Secret Sharing and Replicated Secret Sharing Arithmetic

Marina Blanton and Chen Yuan
Department of Computer Science and Engineering
University at Buffalo
{mblanton,chyuan}@buffalo.edu

Abstract

In this work, we study the problem of constructing oblivious RAM for secure multi-party computation to obliviously access memory at private locations during secure computation. We build on recent two-party Floram construction [14] that uses function secret sharing for a point function and incurs $O(\sqrt{N})$ secure computation and $O(N)$ local computation per ORAM access for an N-element data set. Our new construction, Top ORAM, is designed for multi-party computation with $n \geq 3$ parties and uses replicated secret sharing. We reduce secure computation component to $O(\log N)$, which has notable effect on performance. As a result, when Top ORAM is instantiated with $n = 3$ parties, it outperforms all other 2- and 3-party ORAM constructions that we tested for datasets up to a few million (at which point $O(N)$ local work becomes the bottleneck).

To be able to accomplish the above, we design a number of secure n-party protocols for semi-honest adversaries in the setting with honest majority for replicated secret sharing. They are suitable to be instantiated over any finite ring, which has the advantage of using native hardware arithmetic with rings $\mathbb{Z}_{2^k}$ for some k. We also provide conversion procedures between other, more common types of secret sharing and replicated secret sharing to enable integration of Top ORAM with other secure computation frameworks.

As an additional contribution of this work, we show how our ORAM techniques can be used to realize private binary search at the cost of only a single ORAM access and $\log N$ comparisons, instead of conventional $O(\log N)$ ORAM accesses and comparisons. Because of this property, performance of our binary search is significantly faster than binary search using other ORAM schemes for all ranges of values that we tested.

1 Introduction

Secure multi-party computation refers to the ability of a number of participants to evaluate a function of their choice on private data without disclosing unintended information about the private data to the computation participants. It has been the subject of research for many years with its performance experiencing tremendous progress during the last decade. Such techniques are now suitable for data and computations of significant sizes. Furthermore, they can be increasingly applied to perform analysis of large private data sets distributed among a number of participants, as well as data analytics and decision making using private distributed data (including medical, financial, and other domains). Of particular interest to the research community in recent years has been privacy-preserving machine learning, which uses non-trivial algorithms to analyze large volumes of data. Data used in such analyses are often sparse, while it is important to protect information about the structure of the data in privacy-preserving applications. This means that straightforward translation of the conventional machine learning or other algorithms to the privacy-preserving setting not only incurs the cost of secure multi-party computation techniques, but also often increases the asymptotic complexity of the algorithms due to the need to employ data-oblivious (i.e., data-independent) algorithms and data structures in the privacy-preserving setting. For example, if data is represented as a sparse vector or matrix and one wishes to access its (non-zero) element without revealing its location, a naive way to accomplish this is to touch all elements of the data representation. This necessitates the use of data oblivious algorithms that offer better performance than scanning the entire dataset when an element of the set at a private location needs to be retrieved. The most general way to achieve this is to use oblivious RAM (ORAM) techniques, which is the subject of this work.

The original ORAM constructions [19, 27, 20] and the work that followed (see, e.g., [30, 33] among many others) were designed for the client-server setting, where the client is entitled to have access to the entire memory in the clear and outsources it in encrypted form to a server. In the context of secure multi-party computation, there is no single party with full access to the data and the client's work needs to be simulated by multiple participants in a privacy-preserving way. This places different constraints on ORAM constructions compared to the client-server setting in order to achieve efficient performance. For that reason, multiple constructions have been developed to optimize performance of ORAM in the secure two-party and multi-party settings (e.g., [35, 24, 34, 15, 37] and others). We follow this line of work and design a novel ORAM construction for secure multi-party computation, called Top ORAM, which offers attractive performance.

Our construction is built using the high-level structure of the two-party ORAM of Doerner and shelat [14], in which the participants perform $O(N)$ local work and $O(\sqrt{(N)})$ secure computation per ORAM access for an N-element data set. Our setting and the underlying secure computation arithmetic, however, are fundamentally different: We work in the setting with honest majority, where the computation is being carried out by $n \geq 3$ parties, at most $t < n/2$ of which can be corrupt and colluding. The setting of primary interest to us is the three-party setup with at most one corrupt participant (i.e., $n = 3$ and $t = 1$), as commonly employed in secure multi-party tools and compilers (e.g., [5, 38]).

Using a careful choice of secret sharing, we are able to reduce the secure computation component from $O(\sqrt{N})$ in [14] to only $O(\log N)$ per ORAM access, while retaining $O(N)$ local work per participant. As a result, our solution exhibits excellent performance and, when instantiated with $n = 3$, outperforms all ORAM constructions we are aware of (including those that work with only three parties or only two parties) for a wide range of ORAM sizes N. In particular, because the overall work has linear dependency on N, our construction (as well as [14]) is not competitive for very large data sizes (e.g., $N \geq 2^{25}$), where constructions with overall work being polylogarithmic in N provide better performance. However, for moderate data sizes, Top ORAM is faster than other two- and three-party schemes by up to a factor of 13 in the LAN setting and up to a factor of 3.1 in the WAN setting.

Our observation that no single ORAM construction is expected to outperform all other schemes on all values of N and other parameters is consistent with a recently published result [9]. We also note that performance of Top ORAM can quickly degrade as n increases. For that reason, in the rest of the paper we describe the general solution and use the three-party setting as a running example and for evaluation.

We summarize performance of recent two- and multi-party ORAM constructions in Table 1, all are secure in the semi-honest adversarial model. A more detailed description of the prior schemes and comparison with our work are given in Section 8.

Our contributions. We build a new ORAM scheme for secure multi-party computation in the

ORAM	n	t	Access			
			Sec. Comp.	Local Comp.	Comm.	Rounds
Square-Root ORAM [37]	2	1	$O(\sqrt{N}\log^3 N)$	$O(\sqrt{N}\log N)$	$O(\sqrt{N}\log^3 N)$	$O(\log N)$
Circuit ORAM [34]	2	1	$O(\log^3 N)$	$O(1)$	$O(\log^3 N)$	$O(\log N)$
3PORAM [15]	3	1	$O(Z\log^2 N)$	$O(Z\log^2 N)$	$O(Z\log^3 N)$	$O(\log N)$
Floram [14]	2	1	$O(\sqrt{N})$	$O(N)$	$O(\sqrt{N})$	$O(1)$
Floram CPRG [14]	2	1	$O(\sqrt{N})$	$O(N)$	$O(\sqrt{N})$	$O(\log N)$
Top ORAM	≥ 3	$< n/2$	$O(\log N)$	$O(N)$	$O(\log N)$	$O(\log N)$

ORAM	n	t	Initialization			
			Sec. Comp.	Loc. Comp.	Comm.	Rounds
Square-Root ORAM [37]	2	1	$O(N\log^2 N)$	$O(N\log N)$	$O(N\log^2 N)$	$O(\log N)$
Circuit ORAM [34]	2	1	$O(\log^3 N)$	$O(1)$	$O(\log^3 N)$	$O(\log N)$
3PORAM [15]	3	1	Unknown			
Floram [14]	2	1	0	$O(N)$	$O(N)$	$O(1)$
Floram CPRG [14]	2	1	0	$O(N)$	$O(N)$	$O(1)$
Top ORAM	≥ 3	$< n/2$	0	0	0	0

Table 1: Comparison of secure two- and multi-party ORAM constructions. N is the size of the data set in elements or blocks; n is the number of computation parties; t is the maximum number of corrupt parties; Z is the size of a bucket in blocks.

setting with honest majority, called Top (func**t**i**o**n & re**p**licated secret sharing) ORAM. Its main building blocks are function secret sharing (FSS) for a point function and secure arithmetic using replicated secret sharing (RSS) for threshold structures. Its main novelty is eliminating the use of a stash, the need for which was the bottleneck in the performance of FSS-based Floram construction [14] and resulted in $O(\sqrt{N})$ access time. For that reason, both read and write ORAM access in our scheme requires only $O(\log N)$ secure computation work for generating FSS for a point function. The remaining work is local to each party.

To realize the structure above, a significant portion of this work is dedicated to the design of the necessary secure computation arithmetic using replicated secret sharing (Section 4). While the techniques are information-theoretic in their nature, we use computational security to reduce communication cost. The techniques can be instantiated to work in any ring or field, and therefore can enjoy improved performance of native hardware operations when instantiated in a ring $\mathbb{Z}_{2^k}$ for some k. Because the underlying secret sharing scheme is linear, any linear combination of secret shared values is computed locally, while operations such as multiplication and opening a secret-shared value use one round of communication and involve each party transmitting t ring elements in the (n, t) setting.

Our next contribution is customizing the techniques underlying our ORAM construction to perform binary search (Section 5). By capitalizing on the tree-like evaluation employed in FSS generation and evaluation, we are able to reduce the cost of binary search to that of about a single ORAM access with $\log N$ secure comparison operations from the conventional $\log N$ ORAM accesses and comparison operations.

Lastly, because replicated secret sharing is relatively unexplored compared to several other types of secret sharing, we provide conversion procedures between different types of secret sharing (Section 6). In particular, we consider Shamir secret sharing (SSS) and additive secret sharing used in Sharemind and discuss conversion between RSS and each of these secret sharing types. We consequently use RSS-to-SSS and SSS-to-RSS conversion together with SSS-based implementation of secure comparison in our implementation of binary search.

We implement ORAM operations and the binary search and evaluate their performance on different ORAM sizes in both LAN and WAN settings. Our results (Section 7) indicate that Top ORAM is the best performing ORAM among recent 2- and 3-party ORAM constructions for a wide range of data set sizes N. Our binary search outperforms all other implementations that we tested for all values of N by a significant amount: it is faster than other alternatives by an order of magnitude or more.

2 Preliminaries

2.1 Secure Multi-Party Computation

We consider the conventional secure multi-party setting with n computational parties, out of which at most t can be corrupt. Our primary focus is on the security against semi-honest participants and simulation-based security, formulated as follows:

Definition 1 *Let parties* $P_1, \ldots, P_n$ *engage in a protocol* Π *that computes function* $f(\mathsf{in}_1, \ldots, \mathsf{in}_n) = (\mathsf{out}_1, \ldots, \mathsf{out}_n)$*, where* in_i *and* out_i *denote the input and output of party* P_i*, respectively. Let* $\mathrm{VIEW}_{\Pi}(P_i)$ *denote the view of participant* P_i *during the execution of protocol* Π*. More precisely,* P_i*'s view is formed by its input and internal random coin tosses* r_i*, as well as messages* $m_1, \ldots, m_k$ *passed between the parties during protocol execution:* $\mathrm{VIEW}_{\Pi}(P_i) = (\mathsf{in}_i, r_i, m_1, \ldots, m_k)$*. Let* $I = \{P_{i_1}, P_{i_2}, \ldots, P_{i_t}\}$ *denote a subset of the participants for* $t < n$*,* $\mathrm{VIEW}_{\Pi}(I)$ *denote the combined view of participants in* I *during the execution of protocol* Π *(i.e., the union of the views of the participants in* I*), and* $f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ *denote the projection of* $f(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ *on the coordinates in* I *(i.e.,* $f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ *consists of the* i_1*th, ...,* i_t*th element that* $f(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ *outputs). We say that protocol* Π *is* t*-private in the presence of semi-honest adversaries if for each coalition of size at most* t *there exists a probabilistic polynomial time (PPT) simulator* S_I *such that* $\{S_I(\mathsf{in}_I, f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)), f(\mathsf{in}_1, \ldots, \mathsf{in}_n)\} \equiv \{\mathrm{VIEW}_{\Pi}(I), (\mathsf{out}_1, \ldots, \mathsf{out}_n)\}$*, where* $\mathsf{in}_I = \bigcup_{P_i \in I} \{\mathsf{in}_i\}$ *and* $\equiv$ *denotes computational or statistical indistinguishability.*

As customary with techniques based on secret sharing, the set of computational parties does not have to coincide with (and can be formed independently from) the set of parties supplying inputs in the computation (input providers) and the set of parties receiving output of the computation (output recipients). Then if a computational party learns no output, the computation should not reveal any information to that party. Consequently, if we wish to design a functionality that takes input in the secret-shared form and produces shares of the output, any computational party should learn nothing from protocol execution.

2.2 Replicated Secret Sharing

Our techniques utilize replicated secret sharing (RSS) [22] which has an associated access structure Γ. An access structure is defined by qualified sets $Q \in \Gamma$, which are the sets of participants who are granted access, and the remaining sets of the participants are called unqualified sets. In the context of this work we only consider threshold structures in which any set of t or fewer participants is not authorized to learn information about private values (i.e., they form unqualified sets), while any $t+1$ or more participants are able to jointly reconstruct the secret (and thus form qualified sets). To secret-share private x using RSS, we treat x as an element of a finite ring $\mathcal{R}$ and additively split it into shares x_T such that $x = \sum_{T \in \mathcal{T}} x_T$ (in $\mathcal{R}$), where $\mathcal{T}$ consists of all maximal unqualified set of Γ (i.e., all sets of t parties in our case). Then each party $p \in [1, n]$ stores shares x_T for all $T \in \mathcal{T}$

subject to $p \notin T$. We use notation $[x]$ to mean that (private) x is secret shared among the parties using RSS.

Example. In the $(3, 1)$ setting, $\mathcal{T}$ consists of 3 sets $\mathcal{T} = \{\{1\}, \{2\}, \{3\}\}$ and thus there are 3 shares such that $x = x_{\{1\}} + x_{\{2\}} + x_{\{3\}}$. Then party 1 stores shares $x_{\{2\}}, x_{\{3\}}$, party 2 stores $x_{\{1\}}, x_{\{3\}}$, and party 3 stores $x_{\{1\}}, x_{\{2\}}$.

In the general case of (n, t)-threshold RSS, the total number of shares is $\binom{n}{t}$ with $\binom{n-1}{t}$ shares stored by each party, which can become large as n and t grow. For convenience and without loss of generality, we let $n = 2t + 1$. In the case when $n > 2t + 1$, $2t + 1$ parties can carry out the computation on a reduced set of shares in such a way that there is no need to involve the remaining parties in the computation.

The parties will need to perform computation on secret shared values. The first important property of RSS is that it is linear. That is, given shares of private values, each party can locally compute the corresponding shares of a linear combination of the values. For example, to add $[a]$ and $[b]$, party p computes $a_T + b_T$ (in $\mathcal{R}$) for each $T \in \mathcal{T}$ that p stores. A number of other operations, such as multiplications, reconstructing a value from its shares, etc., are interactive. We consequently describe in Section 4 the way we realize these operations for the purposes of this work. An important optimization on which we rely is non-interactive evaluation of a pseudorandom function (PRF) using RSS in the computational (as opposed to number-theoretic) setting as proposed in [12]. We detail our use of this functionality in Section 4.

2.3 Function Secret Sharing

Recent literature [7] introduced the notion of function secret sharing (FSS), which allows n parties to split a function f into concisely described shares f_i for $1 \leq i \leq n$ such that (i) $\sum_{i=1}^{n} f_i = f$ and (ii) any strict subset of the f_is hides f. A special class of functions for which efficient constructions of FSS are known are point functions.

A point function is a function $f_{\alpha,\beta} : [1, N] \to \mathcal{R}$[1] such that $f_{\alpha,\beta}(x) = \beta$ if $x = \alpha$ and 0 otherwise. An optimized realization of point functions for the two-party case was given in [8], while the n-party solution remains slower. It is possible to construct shares of function $f_{\alpha,\beta}$ in time logarithmic in n.

In seeking an efficient solution for our setting, we realized that the n-party formulation of FSS is too strong for our setup. Instead, the notion that we require is that of threshold FSS, which to the best of our knowledge has not been considered before. That is, we desire that $t + 1$ or more shares f_i can be used to reconstruct f, while t or fewer shares f_i reveal no information about f. We realize this functionality by generalizing the two-party FSS interface and the solution from [8, 14] to our (n, t) setting. We also modify the construction to operate over a finite ring $\mathcal{R}$ to be compatible with our solution. Lastly, for clarity of presentation, we explicitly mark values as protected (secret shared) and not protected (available to the parties in the clear).

A multi-party *threshold function secret sharing* scheme of a point function $f_{\alpha,\beta}$ is then a pair of polynomial-time algorithms $(\mathsf{Gen}, \mathsf{Eval})$ such that:

- $\mathsf{Gen}(1^\kappa, [\alpha_1][\alpha_2] \ldots [\alpha_\ell], [\beta])$ is a multi-party key generation protocol, which on input a security parameter 1^κ and a private function description (α, β), where $\alpha = \alpha_1\alpha_2 \ldots \alpha_\ell$ is represented in the bit-decomposed form with $\ell = \lceil \log N \rceil$, computes key $[\mathsf{fk}]$ and public portion $\mathsf{fk}_{\mathsf{pub}}$, where each party learns its shares of fk (i.e., party p learns shares fk_T for each $T \in \mathcal{T}$ subject to $p \notin T$) and all parties learn $\mathsf{fk}_{\mathsf{pub}}$.
- $\mathsf{Eval}(\mathsf{fk}_T, \mathsf{fk}_{\mathsf{pub}}, \mathsf{ind})$ is an evaluation algorithm run locally by a party, which on input a key share fk_T, public key portion $\mathsf{fk}_{\mathsf{pub}}$, and an evaluation point $\mathsf{ind} \in [1, N]$, outputs elements

[1] We define the function to return an element of a ring to be compatible with our setting.

$y_T^{\mathsf{ind}}, t_T^{\mathsf{ind}} \in \mathcal{R}$ such that y_T^{ind} is a share of $f_{\alpha,\beta}(\mathsf{ind})$ (associated with the unqualified set T) and t_T^{ind} is a share of bit 0 if $f_{\alpha,\beta}(\mathsf{ind}) = 0$, and share of bit 1 otherwise.

We use notation $\leftarrow$ to denote output of randomized algorithms, while notation $=$ refers to deterministic assignment.

3 Multi-Party ORAM

In this section we describe the structure of our Top ORAM, the algorithms for the read and write operations, and provide their analysis. The read and write operations rely on a number of building blocks using RSS, which we consequently describe in Section 4.

The high-level idea behind our solution is as follows. It was previously known that two-party FSS can be used to realize read-only ORAM (or Private Information Retrieval) with $O(N)$ secure computation and $O(N)$ local work or write-only ORAM with $O(N)$ secure computation and $O(N)$ local work, but not both because of the differences in data representation (see [14] and references therein). To combine the two operations in a single scheme, Floram [14] used a stash to reconcile the two representations when they start diverging following writes. The optimal stash size was $O(\sqrt{N})$, which resulted in $O(\sqrt{N})$ secure computation per ORAM access (and $O(N)$ local time). We were, however, able to get around this computation bottleneck. Our observation is that the use of replicated secret sharing in the multi-party setting allows for both read and write operations to be carried out on the same representation, thus removing the need for the stash. In particular, to realize a write-only ORAM in the two-party setting, each data item X^i was secret-shared among the two parties. However, to realize a read-only ORAM, both parties stored the same value for each data item, namely the data item blinded by a pseudorandom value (not know to either, but jointly computable by them). Here we observe that it is possible to simultaneously have both representations in the same construction if we use replicated secret sharing. For example, in the three-party case, we store the data item in a secret shared form (as in write), any pair of the participants have a share in common (as in read), and any participant can only learn a protected value of the data item (as required for read), with the last, missing share serving the role of the blinding factor.

This gives us a conceptually simple realization of ORAM accesses, primarily consisting of FSS generation and evaluation. As a result, we obtain that each ORAM access costs $O(\log N)$ secure computation due to FSS generation and $O(N)$ local work due to FSS evaluation.

Let $X = (X^0, X^2, \ldots, X^{N-1})$ denote the data items (or blocks), i.e., the memory is represented as sequence of N blocks. In Top ORAM this dataset is stored among the parties using RSS, i.e., each party p stores $X_T = (X_T^0, \ldots, X_T^{N-1})$ for each $T \in \mathcal{T}$ such that $p \notin T$. Each share of a block is represented as one or more ring elements depending on the original size of each block, but for simplicity of exposition we treat each block share X_T^i as a single field element. We require that $\mathcal{R}$ has characteristic $\geq 2^\kappa$.

The above means that the ORAM has a straightforward initialization procedure: If data block X^i is contributed by a single party (input provider), that party produces secret shares of X^i and communicates them to the parties running the computation. Alternatively, X^i may be a result of prior secure multi-party computation, in which case it may need to be converted to RSS from a different representation, e.g., Shamir secret sharing. We treat the topic of share conversion in more detail in Section 6.

Algorithm 1 $\langle[\hat{X}^0], \ldots, [\hat{X}^{N-1}]\rangle \leftarrow \mathsf{Write}([\mathsf{ind}_1] \ldots [\mathsf{ind}_\ell],\ [\Delta v], \langle[X^0] \ldots [X^{N-1}]\rangle)$

```
Execute $([\mathsf{fk}], \mathsf{fk}_{\mathsf{pub}}) \leftarrow \mathsf{Gen}(1^\kappa, [\mathsf{ind}_1] \ldots [\mathsf{ind}_\ell], [\Delta v])$.
Each $p \in [1, n]$ locally computes for $T \in \mathcal{T}$ s. t. $p \notin T$ and $\forall i \in [0, N-1]$
    $(y_T^i, t_T^i) = \mathsf{Eval}(\mathsf{fk}_T \mathsf{fk}_{\mathsf{pub}}, i)$
    $\hat{X}_T^i = X_T^i + y_T^i$ (in $\mathcal{R}$)
end for
return $\langle[\hat{X}^0], \ldots, [\hat{X}^{N-1}]\rangle$
```

Algorithm 2 $[v] \leftarrow \mathsf{Read}([\mathsf{ind}_1] \ldots [\mathsf{ind}_\ell], \langle[X^0], \ldots [X^{N-1}]\rangle)$

```
$([\mathsf{fk}], \mathsf{fk}_{\mathsf{pub}}) \leftarrow \mathsf{Gen}(1^\kappa, [\mathsf{ind}_1] \ldots [\mathsf{ind}_\ell], 0)$.
Each $p \in [1, n]$ locally computes for $T \in \mathcal{T}$ s. t. $p \notin T$ and $i \in [0, N-1]$
    $(y_T^i, t_T^i) = \mathsf{Eval}(\mathsf{fk}_T, \mathsf{fk}_{\mathsf{pub}}, i)$
end for
$[v] \leftarrow \mathsf{DotProd}(\langle[X^0], \ldots, [X^{N-1}]\rangle, \langle[t^0], \ldots, [t^{N-1}]\rangle)$.
return $[v]$
```

3.1 ORAM Read and Write Operations

In what follows, we describe read and write operations separately. Note that in the context of secure multi-party computation, the function being executed is normally known and thus there is no need to hide the operation type (i.e., read or write). For that reason, our read and write operations execute different instructions and are distinguishable. When, however, it is necessary to hide the type of operation being executed, it is not difficult to merge the functionalities at a minimal cost and have a single access function capable of executing both operations. Read and write operations are simple and are given in Algorithms 1 and 2.

Both of them first involve calling FSS Gen on shares of the index and the dataset, which the parties execute jointly. Afterwards, each party locally executes Eval for each element of the dataset and each share it stores. The Write operation then updates each share with the computed values, while Read obliviously selects the desired element using secure dot-product computation.

FSS Gen and Eval are provided as Algorithms 3 and 4, respectively. They are based on the original protocols in [8, 14], but we modify the way the arithmetic is carried out as well as the way the multi-party operations are realized, which we consequently discuss. All arithmetic is in $\mathcal{R}$.

Recall that with this type of secret sharing, any linear combination of secret shared values can be performed locally by the parties without interaction. Other operations, on the other hand, may require interaction. Besides multiplication of secret shared values, we also utilize the computation of the dot product of two secret shared values DotProd as part of Read. Gen additionally uses reconstruction of a value from shares Open, generation of shares of a random field element RandF, and evaluation of a pseudo-random generator on a secret shared seed PRG.

Before we proceed with describing the building blocks, we note that Gen also computes values $[s^{j,\alpha_j}]$, $[s^{j,\overline{\alpha_j}}]$, $[\tau^{j,\alpha_j}]$, $[v^{j,\alpha_j}]$ that depend on private bit α_j. Thus, their computation corresponds to branching on conditional statements with private conditions α_j. This means that we can rewrite the computation using conventional evaluation of conditional statements with private conditions. For example, we compute $[v^{j,\alpha_j}] = [\alpha_j]([v^{j,1} - [v^{j,0}]) + [v^{j,0}]$ using a single interactive operation (multiplication). Computation of $[\tau^{j,\alpha_j}] = [\alpha_j](\tau^{j,1} - \tau^{j,0}) + \tau^{j,0}$, on the other hand, is local. Lastly, computing $[s^{j,\alpha_j}]$ and $[s^{j,\overline{\alpha_j}}]$ can be accomplished using the total of one multiplication (i.e., by first computing $[u] = [\alpha_j]([s^{j,1}] - [s^{j,0}])$ and then setting $[s^{j,\alpha_j}] = [u] + [s^{j,0}]$ and $[s^{j,\overline{\alpha_j}}] = [s^{j,1}] - [u]$).

Algorithm 3 $([\mathsf{fk}], \mathsf{fk}_{\mathsf{pub}}) \leftarrow \mathsf{Gen}(1^\kappa, [\alpha_1][\alpha_2]\dots[\alpha_\ell], [\beta])$

```
$[w^0] \leftarrow \mathsf{RandF}()$
$[t^0] \leftarrow 1$
for $j \in [1, \ell]$ do
    $[s^{j,0}]||[v^{j,0}] \ || \ [s^{j,1}]||[v^{j,1}] = \mathsf{PRG}([w^{j-1}])$
    $\sigma^j = \mathsf{Open}([s^{j,\overline{\alpha_j}}])$
    $\tau^{j,0} = \mathsf{Open}([v^{j,0}] + [\alpha_j] + 1)$
    $\tau^{j,1} = \mathsf{Open}([v^{j,1}] + [\alpha_j])$
    $[w^j] = [s^{j,\alpha_j}] + [t^{j-1}] \cdot \sigma^j$
    $[t^j] = [v^{j,\alpha_j}] + [t^{j-1}] \cdot [\tau^{j,\alpha_j}]$
end for
$\gamma := \mathsf{Open}([w^\ell] + [\beta])$
$[\mathsf{fk}] = ([w^0], [t^0])$
$\mathsf{fk}_{\mathsf{pub}} = ((\sigma^j, \tau^{j,0}, \tau^{j,1})_{j\in[1,\ell]}, \gamma)$
return $[\mathsf{fk}], \mathsf{fk}_{\mathsf{pub}}$
```

Algorithm 4 $(y_T, t_T) := \mathsf{Eval}(\mathsf{fk}_T = (w_T^0, t_T^0),\ \mathsf{fk}_{\mathsf{pub}} = (\{\sigma^j, \tau^{j,0}, \tau^{j,1}\}_{j\in[1,\ell]}, \gamma),\ \mathsf{ind} = \mathsf{ind}_1 \dots \mathsf{ind}_\ell)$

```
for $j \in [1, \ell]$ do
    $s_T^{j,0}||v_T^{j,0} \ || \ s_T^{j,1}||v_T^{j,1} = \mathsf{PRG}(w_T^{j-1})$
    $w_T^j = s_T^{j,\mathsf{ind}_j} + t_T^{j-1} \cdot \sigma^j$
    $t_T^j = v_T^{j,\mathsf{ind}_j} + t_T^{j-1} \cdot \tau^{j,\mathsf{ind}_j}$
end for
$y_T = w_T^\ell + t_T^\ell \cdot \gamma$
$t_T = t_T^\ell$
return $y_T, t_T$
```

We also note that in our Gen the value of $s^{j,\overline{\alpha_j}}$ gets open after its computation and in our setting it is possible to perform multiplication and open together (see MulPub in section 4.3). This means that we compute $s^{j,\overline{\alpha_j}}$ in one step and locally set $[s^{j,\alpha_j}] = [s^{j,0}] + [s^{j,1}] - s^{j,\overline{\alpha_j}}$.

3.2 Complexity of Operations

We summarize performance of the secure computation building blocks in Table 2. The table reports numbers for the three-party setting as well as the general n-party case. Communication is measured in the number of ring elements sent by each party. The number of cryptographic operations is measured in the number of pseudo-random ring elements which each each party generates via calls to PRG/PRF.

We also include the cost of Gen in the table. It is computed based on Algorithm 3 with small optimizations in the first and last loop iterations (e.g., skipping the computation of unused $[t^\ell]$) and executing the operations in parallel when possible. Recall that the only interactive computation in ORAM Write is a single call to Gen, while executing Read additionally involves a call to DotProd, which requires the same amount of interaction as a single multiplication.

3.3 ORAM Initialization

If we, similar to prior work [14], assume that the computational parties hold shares of the data set X, then no initialization is needed in Top ORAM. To best of our knowledge, this is not the case for all other ORAM constructions for secure multi-party computation. Floram [14] was the first to have a light-weight initialization procedure, but it still requires a number of cryptographic operations and communication linear in the data set size N (even assuming that the participants hold shares of the data set X). Comparison of initialization costs for different ORAM schemes is given in Table 1.

In more detail, if the data set X is used as input into secure computation, the input party in possession of item X^i locally produces its (RSS) shares X^i_T and securely communicates them to the appropriate computational parties. ORAM operations may be executed immediately with no additional initialization procedure. An item X^i may also be produced as a result of secure computation (other than an ORAM write). In that case, if the secure computation uses a different type of secret sharing than RSS, a conversion procedure is needed, which we discuss in Section 6.

4 Building Blocks: Operations on Replicated Shares

Recall that RSS enjoys the linear property. For example, we can add secret-shared $[a]$ and $[b]$ by adding a_T and b_T for each $T \in \mathcal{T}$ that a party possesses. We also use the ability to add/subtract known integers to a secret-shared value $[a]$ and multiply a secret-shared value $[a]$ by a known integer. In the former case, we realize $[a] + b$ by converting b to $[b]$ and performing the addition, where shares b_T use no randomness (e.g., we could set one share to b and the remaining shares to 0 to maintain that $\sum_{T \in \mathcal{T}} b_T = b$). In the latter case, $c = [a] \cdot b$ is realized by setting $c_T = a_T \cdot b$ (in $\mathcal{R}$) for each $T \in \mathcal{T}$.

4.1 Random Number Generation

Our solution relies on two types random number generation, which we discuss here.

4.1.1 PRG

Invocation of $[a_1], [a_2], \ldots \leftarrow \mathsf{PRG}([s])$ is realized by independently executing a PRG algorithm on each share of s without interaction between the parties. Because the output of $\mathsf{PRG}([s])$ is private, we expect it to produce a sequence of secret-shared values (represented as ring elements). Furthermore, in our construction we only call the PRG to obtain random (secret-shared) ring elements. This means that calling $\mathsf{PRG}(s_T)$ to produce pseudo-random a_T will result in $\mathsf{PRG}([s])$ generating $[a]$, where a is pseudo-random as well because $a = \sum_{T \in \mathcal{T}} a_T$ (in $\mathcal{R}$). This is similar to evaluating a PRF on a secret-shared key in the RSS setting without interaction in [12].

$\mathsf{PRG}(s_T)$ can be realized internally using any suitable algorithm, as long as it is consistent among the computational parties. For example, because of the speed of AES encryption on modern processors, one might implement $\mathsf{PRG}(s_T) = \mathsf{PRF}(s_T, 0)||\mathsf{PRF}(s_T, 1)||\ldots$, where $\mathsf{PRF} : \mathcal{R} \times \{0, 1\}^\kappa \to \mathcal{R}$ is a PRF instantiated with AES.

Let $\mathsf{G} = \mathsf{PRG}([s])$. When the output of G is not consumed all at once, we use notation $\mathsf{G.next}$ to retrieve the next (secret-shared) element from G. Similarly, if $\mathsf{G}_T = \mathsf{PRG}(s_T)$, notation $\mathsf{G}_T.\mathsf{next}$ refers to the next pseudo-random share output by G_T.

4.1.2 RandF

$[a] \leftarrow \mathsf{RandF}()$ computes a secret-shared random element. We implement this function by executing $\mathsf{PRG}([k]).\mathsf{next}$, where k is a system-wide key. It is set up at the system initialization time (in the form of secret shares) and does not change throughout program execution.

4.2 Multiplication

Multiplication $[c] \leftarrow [a] \cdot [b]$ can be realized using the fact that $a = \sum_{T \in \mathcal{T}} a_T$ and $b = \sum_{T \in \mathcal{T}} b_T$ and thus $[a] \cdot [b] = \sum_{T_1, T_2 \in \mathcal{T}} a_{T_1} \cdot b_{T_2}$ (in $\mathcal{R}$). Note that for any (T_1, T_2) pair, there will be a party holding shares T_1 and T_2, and thus performing this operation involves local multiplication and addition over different choices of T_1, T_2. More formally, let mapping $\rho : \mathcal{T} \times \mathcal{T} \rightarrow [1, n]$ denote a function that for each pair $(T_1, T_2) \in \mathcal{T}^2$ dedicates a party $p \in [1, n]$ responsible for computing the product $a_{T_1} \cdot b_{T_2}$ (clearly, p must possess shares T_1 and T_2). For performance reasons, we also desire that ρ distributes the load among the parties as fairly as possible.

Example. With 3 parties, we could have party 1 (in possession of shares $\{2\}$ and $\{3\}$) compute (and add) products $a_{\{2\}}b_{\{2\}}$, $a_{\{2\}}b_{\{3\}}$, and $a_{\{3\}}b_{\{2\}}$, party 2 (in possession of shares $\{1\}$ and $\{3\}$) compute products $a_{\{3\}}b_{\{3\}}$, $a_{\{1\}}b_{\{3\}}$, and $a_{\{3\}}b_{\{1\}}$, and party 3 (in possession of shares $\{1\}$ and $\{3\}$) compute products $a_{\{1\}}b_{\{1\}}$, $a_{\{1\}}b_{\{2\}}$, and $a_{\{2\}}b_{\{1\}}$.

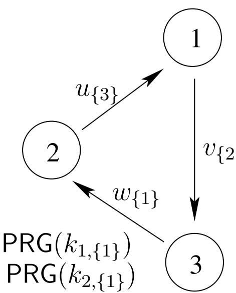
As a result of this (local) computation, the parties hold additive shares of the product $a \cdot b = c$, which needs to be converted to RSS for consecutive computation. This conversion was realized in early publications [26, 4] by having each party create replicated secret shares of their result and distribute each share to the parties entitled to knowing it (i.e., party p receives shares from each party for each $T \in \mathcal{T}$ subject to $p \notin \mathcal{T}$). This results in each participant creating $\binom{n}{t}$ shares and sending $\binom{n-1}{t}$ of them to each party. Consequently, each participant adds the values received for share T and stores the sum as c_T, for each T in its possession.

More recent work such as [3] and others traded information-theoretic security (in the presence of secure channels) for communication efficiency by having the parties generate shared (pseudo-)random values. We apply this idea to our setting as well and obtain the following solution: for each participant $i \in [1, n]$ we let i and all parties $p \notin T$ (i.e., those with access to the share T) have access to a secret $k_{i,T} \in \{0, 1\}^\kappa$. Then when the multiplication algorithm calls for re-distributing the locally computed sum of products, each party p generates the shares of its value v by choosing $\binom{n}{t} - 1$ pseudo-random shares v_T for each $T \in \mathcal{T}$ except one and setting the last share to be consistent with v and all other (pseudo-random) shares. Now each party entitled to having v_T (except for one pre-selected T in $\mathcal{T}$, which we denote by T^* in the current description) will generate this value locally using a pseudo-random generator with seed $k_{p,T}$. T^* is chosen for each party p to be a share to which p ordinarily has access as to minimize the amount of communication. To capture this formally, we define another mapping $\gamma : [1, n] \rightarrow \mathcal{T}$, which for each p returns the party's value of T^* (subject to $p \notin T^*$). As before, we want to choose the values of T^*s as to evenly distribute the load.

Example. In our three-party setup, we could use a mapping $\gamma(1) = \{2\}$, $\gamma(2) = \{3\}$, and $\gamma(3) = \{1\}$, as additionally illustrated in Figure 1. Consequently, the parties need to pre-distribute keys $k_{1,\{1\}}$ (known to parties 1 through 3), $k_{1,\{3\}}$ (known to parties 1 and 2), $k_{2,\{1\}}$ (known to parties 2 and 3), $k_{2,\{2\}}$ (known to all parties), $k_{3,\{2\}}$ (known to parties 3 and 1), and $k_{3,\{3\}}$ (known to all parties). Party 1 first computes $v = a_{\{2\}}b_{\{2\}} + a_{\{2\}}b_{\{3\}} + a_{\{3\}}b_{\{2\}}$ (in $\mathcal{R}$). Because $\gamma(1) = \{2\}$, shares $v_{\{1\}}$ and $v_{\{3\}}$ are computed by drawing the next portion of (pseudo-)random bits from the PRGs seeded with $k_{1,\{1\}}$ and $k_{1,\{3\}}$, respectively ($v_{\{1\}}$ is stored by parties 2 and 3 and $v_{\{3\}}$ is stored by parties 1 and 2). Consequently, party 1 computes $v_{\{2\}} = v - v_{\{1\}} - v_{\{3\}}$ (in $\mathcal{R}$) and communicates mappings:

$$\begin{array}{ll} \rho : \{1\},\{1\} \to 3 & \gamma : 1 \to \{2\} \\ \quad \{1\},\{2\} \to 3 & \quad 2 \to \{3\} \\ \quad \{1\},\{3\} \to 2 & \quad 3 \to \{1\} \\ \quad \{2\},\{1\} \to 3 & \\ \quad \{2\},\{2\} \to 1 & \\ \quad \{2\},\{3\} \to 1 & \\ \quad \{3\},\{1\} \to 2 & \\ \quad \{3\},\{2\} \to 1 & \\ \quad \{3\},\{3\} \to 2 & \end{array}$$

input:
$a_{\{2\}}, a_{\{3\}}, b_{\{2\}}, b_{\{3\}}, \mathsf{G}_{1,1} = \mathsf{PRG}(k_{1,\{1\}})$
$\mathsf{G}_{1,3} = \mathsf{PRG}(k_{1,\{3\}}), \mathsf{G}_{2,2} = \mathsf{PRG}(k_{2,\{2\}})$
$\mathsf{G}_{3,2} = \mathsf{PRG}(k_{3,\{2\}}), \mathsf{G}_{3,3} = \mathsf{PRG}(k_{3,\{3\}})$
computation:
$v = a_{\{2\}}b_{\{2\}} + a_{\{2\}}b_{\{3\}} + a_{\{3\}}b_{\{2\}}$ (in $\mathcal{R}$)
$v_{\{1\}} = \mathsf{G}_{1,1}.\mathsf{next}$, $v_{\{3\}} = \mathsf{G}_{1,3}.\mathsf{next}$
$v_{\{2\}} = v - v_{\{1\}} - v_{\{3\}}$ (in $\mathcal{R}$)
output:
$c_{\{2\}} = v_{\{2\}} + \mathsf{G}_{2,2}.\mathsf{next} + \mathsf{G}_{3,2}.\mathsf{next}$ (in $\mathcal{R}$)
$c_{\{3\}} = v_{\{3\}} + u_{\{3\}} + \mathsf{G}_{3,3}.\mathsf{next}$ (in $\mathcal{R}$)



input:
$a_{\{1\}}, a_{\{3\}}, b_{\{1\}}, b_{\{3\}}, \mathsf{G}_{1,1} = \mathsf{PRG}(k_{1,\{1\}})$
$\mathsf{G}_{1,3} = \mathsf{PRG}(k_{1,\{3\}}), \mathsf{G}_{2,1} = \mathsf{PRG}(k_{2,\{1\}})$
$\mathsf{G}_{2,2} = \mathsf{PRG}(k_{2,\{2\}}), \mathsf{G}_{3,3} = \mathsf{PRG}(k_{3,\{3\}})$
computation:
$u = a_{\{3\}}b_{\{3\}} + a_{\{1\}}b_{\{3\}} + a_{\{3\}}b_{\{1\}}$ (in $\mathcal{R}$)
$u_{\{1\}} = \mathsf{G}_{2,1}.\mathsf{next}$, $u_{\{2\}} = \mathsf{G}_{2,2}.\mathsf{next}$
$u_{\{3\}} = u - u_{\{1\}} - u_{\{2\}}$ (in $\mathcal{R}$)
output:
$c_{\{1\}} = \mathsf{G}_{1,1}.\mathsf{next} + u_{\{1\}} + w_{\{1\}}$ (in $\mathcal{R}$)
$c_{\{3\}} = \mathsf{G}_{1,3}.\mathsf{next} + u_{\{3\}} + \mathsf{G}_{3,3}.\mathsf{next}$ (in $\mathcal{R}$)

input:
$a_{\{1\}}, a_{\{2\}}, b_{\{1\}}, b_{\{2\}}, \mathsf{G}_{1,1} = \mathsf{PRG}(k_{1,\{1\}})$
$\mathsf{G}_{2,1} = \mathsf{PRG}(k_{2,\{1\}}), \mathsf{G}_{2,2} = \mathsf{PRG}(k_{2,\{2\}})$
$\mathsf{G}_{3,2} = \mathsf{PRG}(k_{3,\{2\}}), \mathsf{G}_{3,3} = \mathsf{PRG}(k_{3,\{3\}})$
computation:
$w = a_{\{1\}}b_{\{1\}} + a_{\{1\}}b_{\{2\}} + a_{\{2\}}b_{\{1\}}$ (in $\mathcal{R}$)
$w_{\{2\}} = \mathsf{G}_{3,2}.\mathsf{next}$, $w_{\{3\}} = \mathsf{G}_{3,3}.\mathsf{next}$
$w_{\{1\}} = w - w_{\{2\}} - w_{\{3\}}$ (in $\mathcal{R}$)
output:
$c_{\{1\}} = \mathsf{G}_{1,1}.\mathsf{next} + \mathsf{G}_{2,1}.\mathsf{next} + w_{\{1\}}$ (in $\mathcal{R}$)
$c_{\{2\}} = v_{\{2\}} + \mathsf{G}_{2,2}.\mathsf{next} + w_{\{2\}}$ (in $\mathcal{R}$)

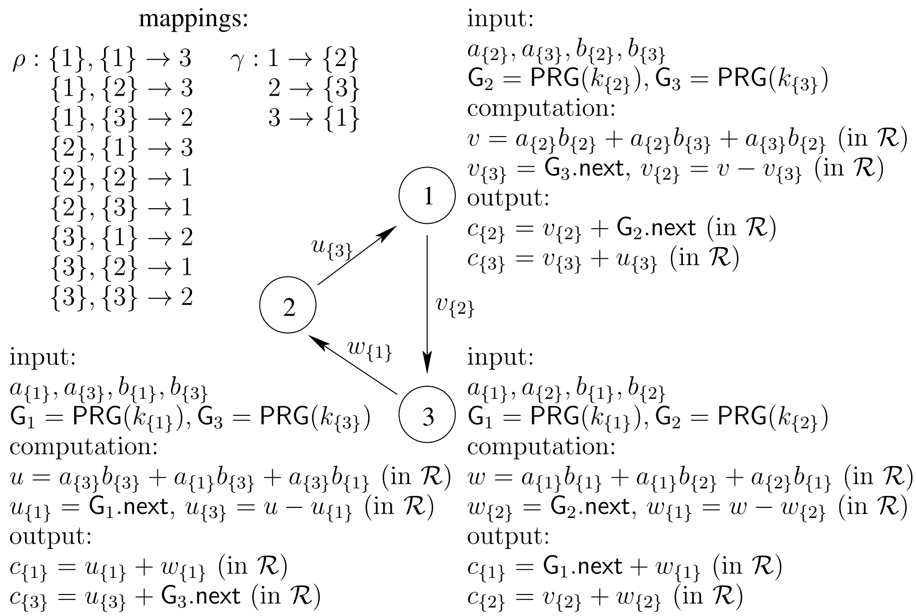
Figure 1: Illustration of initial three-party multiplication $[a] \cdot [b]$.

$v_{\{2\}}$ to party 3. The mappings ρ and γ also define the computation and communication associated with parties 2 and 3. Note that the mappings ρ and γ define the computation and communication associated with each party. Each participant communicates a single element and executes 5 PRG evaluations.

In the n-party case, we obtain that each party only needs to communicate a single element to t parties instead of communicating $\binom{n-1}{t}$ shares to each party.

By examining the computation in this initial solution, we can see that it is not ideal because some seeds (i.e., $k_{1,\{1\}}$, $k_{2,\{2\}}$, and $k_{3,\{3\}}$ in the three-party case) are known by all parties and thus do not contribute to secrecy. We eliminate such redundant cases from the computation to obtain more efficient computation, which still maintains the desired level of security. It is clear that in the three-party case the seeds known to all parties need to be eliminated, while in the more general n-party case we remove the seeds $k_{p,T}$ such that $p \in T$, i.e., party p normally does not have access to shares T. This creates consistency of key distribution in the sense that for $p \neq p'$, keys $k_{p,T}$ and $k_{p',T}$ are known to the same set of participants according to the formulation of T.

Next, we observe that the resulting construction still has inefficiencies because multiple keys $k_{p,T}, k_{p',T}$ for $p \neq p'$ are maintained by the same set of participants. While it is safe to merge keys with the same permissions into a single key k_T and thus obtain a single secret-shared key $[k]$, there might be a need to generate multiple pseudo-random elements using the same key as we explain below. Let us first finish describing the three-party construction before we return to the general case. Because we already have a setup for $\mathsf{RandF}()$, we let the parties generate new pseudo-random values using the same key shares. The resulting solution for three parties with a symmetric communication pattern is shown in Figure 2. This reduces the number of cryptographic operations (i.e., calls to PRG/PRF) per party from 5 in the initial solution to only 2 per multiplication in the three-party case, while communication remains to be a single element per party. Compared to other



mappings:

$\rho: \{1\},\{1\} \to 3$
$\{1\},\{2\} \to 3$
$\{1\},\{3\} \to 2$
$\{2\},\{1\} \to 3$
$\{2\},\{2\} \to 1$
$\{2\},\{3\} \to 1$
$\{3\},\{1\} \to 2$
$\{3\},\{2\} \to 1$
$\{3\},\{3\} \to 2$

$\gamma: 1 \to \{2\}$
$2 \to \{3\}$
$3 \to \{1\}$

input:
$a_{\{2\}}, a_{\{3\}}, b_{\{2\}}, b_{\{3\}}$
$\mathsf{G}_2 = \mathsf{PRG}(k_{\{2\}}), \mathsf{G}_3 = \mathsf{PRG}(k_{\{3\}})$
computation:
$v = a_{\{2\}}b_{\{2\}} + a_{\{2\}}b_{\{3\}} + a_{\{3\}}b_{\{2\}}$ (in $\mathcal{R}$)
$v_{\{3\}} = \mathsf{G}_3.\mathsf{next}$, $v_{\{2\}} = v - v_{\{3\}}$ (in $\mathcal{R}$)
output:
$c_{\{2\}} = v_{\{2\}} + \mathsf{G}_2.\mathsf{next}$ (in $\mathcal{R}$)
$c_{\{3\}} = v_{\{3\}} + u_{\{3\}}$ (in $\mathcal{R}$)

input:
$a_{\{1\}}, a_{\{3\}}, b_{\{1\}}, b_{\{3\}}$
$\mathsf{G}_1 = \mathsf{PRG}(k_{\{1\}}), \mathsf{G}_3 = \mathsf{PRG}(k_{\{3\}})$
computation:
$u = a_{\{3\}}b_{\{3\}} + a_{\{1\}}b_{\{3\}} + a_{\{3\}}b_{\{1\}}$ (in $\mathcal{R}$)
$u_{\{1\}} = \mathsf{G}_1.\mathsf{next}$, $u_{\{3\}} = u - u_{\{1\}}$ (in $\mathcal{R}$)
output:
$c_{\{1\}} = u_{\{1\}} + w_{\{1\}}$ (in $\mathcal{R}$)
$c_{\{3\}} = u_{\{3\}} + \mathsf{G}_3.\mathsf{next}$ (in $\mathcal{R}$)

input:
$a_{\{1\}}, a_{\{2\}}, b_{\{1\}}, b_{\{2\}}$
$\mathsf{G}_1 = \mathsf{PRG}(k_{\{1\}}), \mathsf{G}_2 = \mathsf{PRG}(k_{\{2\}})$
computation:
$w = a_{\{1\}}b_{\{1\}} + a_{\{1\}}b_{\{2\}} + a_{\{2\}}b_{\{1\}}$ (in $\mathcal{R}$)
$w_{\{2\}} = \mathsf{G}_2.\mathsf{next}$, $w_{\{1\}} = w - w_{\{2\}}$ (in $\mathcal{R}$)
output:
$c_{\{1\}} = \mathsf{G}_1.\mathsf{next} + w_{\{1\}}$ (in $\mathcal{R}$)
$c_{\{2\}} = v_{\{2\}} + w_{\{2\}}$ (in $\mathcal{R}$)

Figure 2: Illustration of three-party multiplication $[a] \cdot [b]$.

multiplication protocols in the literature, this matches communication of recent multiplication from [3], which (unlike this work) is available only for three parties. This also improves on communication of standard multiplication using Shamir secret sharing from [17] (information-theoretically secure in the presence of secure channels) by a factor of 2 and improves on communication of Sharemind's multiplication from [6] by a factor of 5 (communication in the latter can be reduced my means of shared PRG seeds as in this work).

Returning to the general case, we note that a participant may receive multiple messages. This is the case for three-party computation with an asymmetric communication pattern or when the number of parties is higher (with any communication pattern). In such cases it is not safe to protect multiple transmissions with the same pseudo-random value and the need to draw multiple elements from a PRG arises. To demonstrate this, we provide an additional example of three-party multiplication with asymmetric communication pattern in Figure 3. In that example, it is not safe to set $u_{\{1\}}$ and $w_{\{1\}}$ to the same values because they are used to protect the shares sent by parties 2 and 3, respectively, to party 1. Therefore, in this example G_1 would need to be queried twice to set two different pseudo-random values $v_{\{1\}}$ and $w_{\{1\}}$. The same reasoning applies to settings with $t \geq 2$, where even with symmetric communication patterns a participant receives t values.

The above gives us the optimized multiplication protocol for n parties below, where each party on average[2] draws $\binom{n-1}{t} - 1 + 2t\binom{n-2}{t} - t$ pseudo-random ring elements from a PRG, which is equal to $(t+1)(\binom{n-1}{t} - 1)$ when $n = 2t+1$, and on average communicates t ring elements. The reasoning for this performance numbers is as follows: First, consider an instance of the protocol with a symmetric communication pattern. Then any given party $p \in [1, n]$ maintains $\binom{n-1}{t}$ shares of a secret. During multiplication, the party will create $\binom{n-1}{t} - 1$ pseudo-random shares for its value being re-shared, computes the last share, and communicates it to t other parties who are entitled to that share. Party p also has $\binom{n-2}{t}$ shares in common with any other party $p' \neq p$. It thus will need to compute pseudo-random shares used by all $2t$ parties except t shares that it receives from them. When the communication pattern is not symmetric, the overall amount of work and

[2] It is possible to distribute the load evenly among the parties by appropriately setting the γ function.

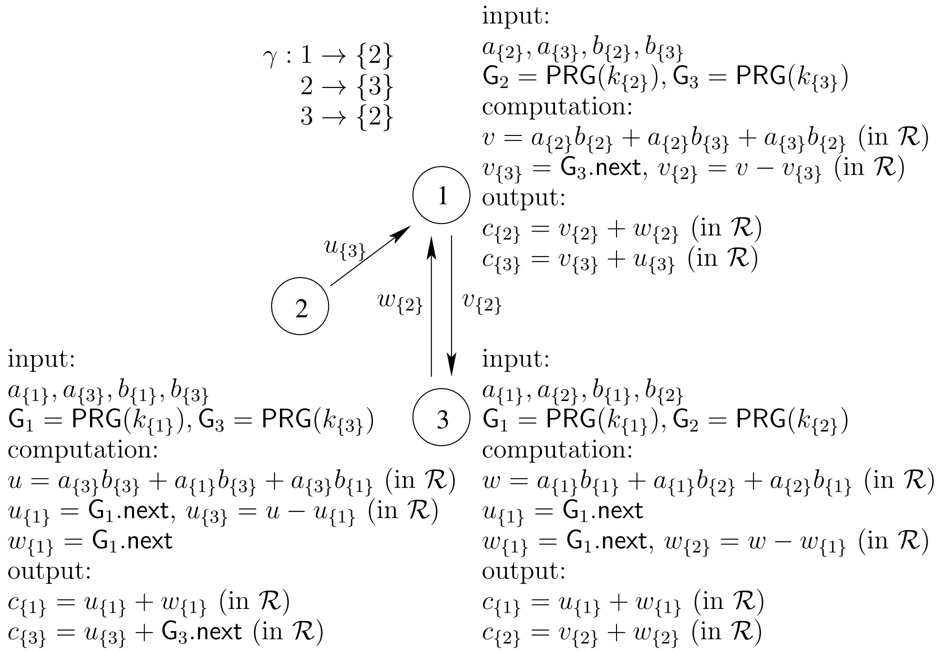

$\gamma : 1 \to \{2\}$
$2 \to \{3\}$
$3 \to \{2\}$
input:
$a_{\{2\}}, a_{\{3\}}, b_{\{2\}}, b_{\{3\}}$
$\mathsf{G}_2 = \mathsf{PRG}(k_{\{2\}}), \mathsf{G}_3 = \mathsf{PRG}(k_{\{3\}})$
computation:
$v = a_{\{2\}}b_{\{2\}} + a_{\{2\}}b_{\{3\}} + a_{\{3\}}b_{\{2\}}$ (in $\mathcal{R}$)
$v_{\{3\}} = \mathsf{G}_3.\mathsf{next}$, $v_{\{2\}} = v - v_{\{3\}}$ (in $\mathcal{R}$)
output:
$c_{\{2\}} = v_{\{2\}} + w_{\{2\}}$ (in $\mathcal{R}$)
$c_{\{3\}} = v_{\{3\}} + u_{\{3\}}$ (in $\mathcal{R}$)
1
$u_{\{3\}}$
2
$w_{\{2\}}$
$v_{\{2\}}$
3
input:
$a_{\{1\}}, a_{\{3\}}, b_{\{1\}}, b_{\{3\}}$
$\mathsf{G}_1 = \mathsf{PRG}(k_{\{1\}}), \mathsf{G}_3 = \mathsf{PRG}(k_{\{3\}})$
computation:
$u = a_{\{3\}}b_{\{3\}} + a_{\{1\}}b_{\{3\}} + a_{\{3\}}b_{\{1\}}$ (in $\mathcal{R}$)
$u_{\{1\}} = \mathsf{G}_1.\mathsf{next}$, $u_{\{3\}} = u - u_{\{1\}}$ (in $\mathcal{R}$)
$w_{\{1\}} = \mathsf{G}_1.\mathsf{next}$
output:
$c_{\{1\}} = u_{\{1\}} + w_{\{1\}}$ (in $\mathcal{R}$)
$c_{\{3\}} = u_{\{3\}} + \mathsf{G}_3.\mathsf{next}$ (in $\mathcal{R}$)
input:
$a_{\{1\}}, a_{\{2\}}, b_{\{1\}}, b_{\{2\}}$
$\mathsf{G}_1 = \mathsf{PRG}(k_{\{1\}}), \mathsf{G}_2 = \mathsf{PRG}(k_{\{2\}})$
computation:
$w = a_{\{1\}}b_{\{1\}} + a_{\{1\}}b_{\{2\}} + a_{\{2\}}b_{\{1\}}$ (in $\mathcal{R}$)
$u_{\{1\}} = \mathsf{G}_1.\mathsf{next}$
$w_{\{1\}} = \mathsf{G}_1.\mathsf{next}$, $w_{\{2\}} = w - w_{\{1\}}$ (in $\mathcal{R}$)
output:
$c_{\{1\}} = u_{\{1\}} + w_{\{1\}}$ (in $\mathcal{R}$)
$c_{\{2\}} = v_{\{2\}} + w_{\{2\}}$ (in $\mathcal{R}$)

Figure 3: Illustration of three-party multiplication $[a] \cdot [b]$ with asymmetric communication pattern.

communication remains unchanged, but it is distributed differently. Thus, we refer to the average work and communication in that case.

Correctness follows from the fact that each private value $v^{(p)}$ computed by party p is distributed into $\binom{n-1}{t}$ additive shares (one of which is communicated while others are computed using PRGs). Afterwards, each party sets its c_T as a sum of $t+1$ shares (computed or received) of values $v^{(p')}$ for each party p' entitled to shares c_T. This matches the fact that each share a_T of secret a is maintained by $t+1$ parties. We ensure that in Algorithm 5 two different participants p and p' with access to shares T consistently associate the values that they draw from G_T with shares belonging to different parties by always processing the values in the increasing order of participants' IDs. Preparation of the shares in Algorithm 5 is done on lines 10–16, where a participant either masks its share with a pseudo-random value because it is used by another party or forms its own shares and the value to be transmitted.

We state the security result as follows:

Theorem 1 *Multiplication $[c] \leftarrow [a] \cdot [b]$ is secure in the (n, t) setting with $t = (n-1)/2$ according to definition 1.*

Proof. Let I denote the set of corrupt parties. We consider the maximal amount of corruption with $|I| = t$. Because the computation proceeds on secret shares and the parties do not learn the result, no information should be revealed to the computational parties as a result of protocol execution.

We build a simulator S_I that interacts with the parties in I as follows: when a party $p \in I$ expects to receive a value from another party $p' \notin I$ in step 5 of the computation according to function γ, S_I chooses a random element of $\mathcal{R}$ and sends it to p. S_I preserves consistency of the view and ensures that when the same value is to be sent by p' to multiple parties in I, all of them receive the same random value. This is the only portion of the protocol where corrupt parties can receive values (that the simulator produces), and the only portion of the protocol when a corrupt

Algorithm 5 $[c] \leftarrow [a] \cdot [b]$

```
// pre-distributed values are $[k]$ and public maps $\rho$ and $\gamma$
// define $\mathsf{G}_T = \mathsf{PRG}(k_T)$
 1: each $p \in [1, n]$ does the following
 2:    let $S_p = \{T \in \mathcal{T} \mid p \notin T\}$
 3:    $v^{(p)} = \sum_{T_1, T_2 \in \mathcal{T}, \rho(T_1, T_2) = p} a_{T_1} b_{T_2}$ (in $\mathcal{R}$)
 4:    for $T \in S_p$ do
 5:       $c_T = 0$
 6:       $v^{(p)}_{\gamma(p)} = v^{(p)}$
 7:    end for
 8:    for $p' \in [1, n]$ in order do
 9:       for $T \in S_p$ do
10:          if $(p' \neq p) \wedge (p' \notin T) \wedge (\gamma(p') \neq T)$ then
11:             $c_T = c_T + \mathsf{G}_T.\mathsf{next}$ (in $\mathcal{R}$)
12:          else if $(p' = p) \wedge (\gamma(p) \neq T)$ then
13:             $z = \mathsf{G}_T.\mathsf{next}$
14:             $c_T = c_T + z$ (in $\mathcal{R}$)
15:             $v^{(p)}_{\gamma(p)} = v^{(p)}_{\gamma(p)} - z$ (in $\mathcal{R}$).
16:          end if
17:       end for
18:    end for
19:    send $v^{(p)}_{\gamma(p)}$ to each $p' \notin \gamma(p)$ (other than itself)
20:    for $p' \in [1, n]$ such that $p \notin \gamma(p')$ do
21:       after receiving $v^{(p')}_{\gamma(p')}$ from $p'$, set $c_{\gamma(p')} = c_{\gamma(p')} + v^{(p')}_{\gamma(p')}$ (in $\mathcal{R}$)
22:    end for
23:    $c_{\gamma(p)} = c_{\gamma(p)} + v^{(p)}_{\gamma(p)}$ (in $\mathcal{R}$)
24: return $[c]$
```

party p may send a value to an honest party p' is step 4, which S_I receives on behalf of p'. All other computation is local, in which S_I does not participate.

We next argue that the simulated view is computationally indistinguishable from the real view. First, note that the corrupt parties in I collectively hold shares a_T, b_T and keys k_T (and thus can compute values $\mathsf{G}_T.\mathsf{next}$) for each $T \in \mathcal{T}$ such that $\exists p \in I$ and $p \notin T$. This entitles the corrupt parties to computing the corresponding shares c_T, but the rest of the shares must remain unknown, so that they are unable to compute c. Next, notice that when $|I| = t$, there is only one share $T^* = I$ such that all parties $p \in I$ have no access to k_{T*} and c_{T^*}, while all parties $p' \notin I$ store those values. Then there are two cases to consider: (1) If one or more parties $p \in I$ receive $\gamma(p')$'s share of $v^{p'}$ from another party $p' \notin I$ (it must be the case that $\gamma(p') \neq T^*$), the received share has been masked by a fresh pseudo-random element from G_{T^*}, is therefore pseudo-random and indistinguishable from random by any $p \in I$. (2) If no party $p \in I$ receives a value from any given $p' \notin I$, indistinguishability is trivially maintained. □

The computation associated with multiplication can be generalized to compute the dot-product of two secret-shared vectors $\mathsf{DotProd}(\langle a^1, \ldots, a^N \rangle, \langle b^1, \ldots, b^N \rangle)$ (as used in Read), or evaluate any other multi-variate polynomial of degree 2, using the same communication and the same number of cryptographic operations as in multiplication. For that purpose, we only need to change the computation in step 3 of the multiplication protocol. For example, for $\mathsf{DotProd}$, we modify step

Operation	Rounds	$(3,1)$ setting		(n,t) setting	
		Comm	Crypto ops	Comm	Crypto ops
$\mathsf{PRG}([s]).\mathsf{next}$ and $\mathsf{RandF}()$	0	0	2	0	$\binom{n-1}{t}$
$\mathsf{Mul}([a],[b])$	1	1	2	t	$(t+1)\left(\binom{n-1}{t}-1\right)$
$\mathsf{Open}([a])$	1	1	0	t	0
$\mathsf{MulPub}([a],[b])$	1	2	2	$n-1$	$t\binom{n-1}{t}$
$\mathsf{DotProd}(\langle[a^1],\ldots,[a^N]\rangle,\langle[b^1],\ldots,[b^N]\rangle)$	1	1	2	t	$(t+1)\left(\binom{n-1}{t}-1\right)$
$\mathsf{Gen}([\alpha_1]\ldots[\alpha_\ell],[\beta])$	$\ell+1$	$6\ell-1$	$6\ell-4$	$(6\ell-1)t$	$\binom{n-1}{t}(3\ell t+2\ell-2t-2)$ $-(2\ell-2)(t+1)$

Table 2: Performance of building blocks.

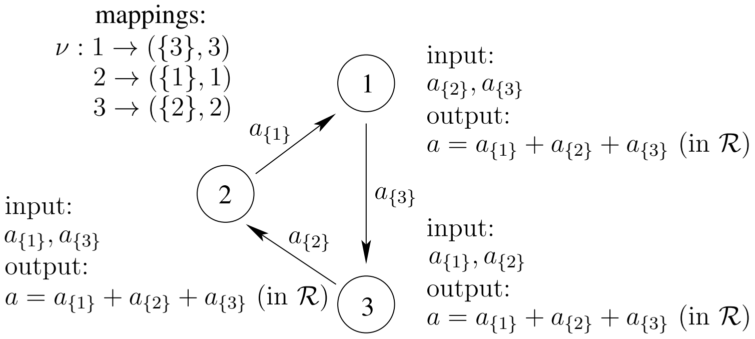


Figure 4: Illustration of three-party $\mathsf{Open}([a])$.

3 to compute $v^{(p)} = \sum_{T_1,T_2\in\mathcal{T},\rho(T_1,T_2)=p}\sum_{i=1}^{N} a^i_{T_1} b^i_{T_2}$ (in $\mathcal{R}$), while the rest of the steps remain unchanged.

4.3 Revealing Private Values

4.3.1 Open

Reconstruction of a secret shared value $a = \mathsf{Open}([a])$ amounts to communicating missing shares to each party so that the value could be reconstructed locally from all shares. Recall that there are $\binom{n}{t}$ total shares and each party holds $\binom{n-1}{t}$ of them. Thus, during this operation each party is to receive $d = \binom{n}{t} - \binom{n-1}{t}$ missing shares.

Our next observation is that when n is not small, the value of d will exceed n and transmitting d messages to each party is not needed. Because the value is reconstructed as the sum of all shares, it is sufficient to communicate sums of shares instead of the individual shares themselves. Recall that $[a]$ can be reconstructed by $t+1$ parties. This means that it is sufficient for a participant to receive one element (i.e., a sum of the necessary shares) from t other parties.

As before, we would like to balance the load between the parties and ideally have each party transmit the same amount of data. This means that we instruct each party to send information to t other parties according to another agreed upon mapping $\nu : [1,n] \to (\mathcal{T},[1,n])^d$. For each party p, this mapping will specify which of p's shares should be communicated to which other party.

Example. With $n = 3$, we could have $\nu(1) = (\{3\},3)$, $\nu(2) = (\{1\},1)$, and $\nu(3) = (\{2\},2)$, which corresponds to $\nu(p) = (\{p-1\},p-1)$ (where $p-1 = 3$ for $p = 1$), which corresponds to the communication pattern in Figure 4.

The mapping ν will then define computation associated with this operation: each p computes $\sum_{T,\nu(p)=T,p'} a_T$ (in $\mathcal{R}$) for each $p' \neq p$ present in the mapping and sends the result to p'.

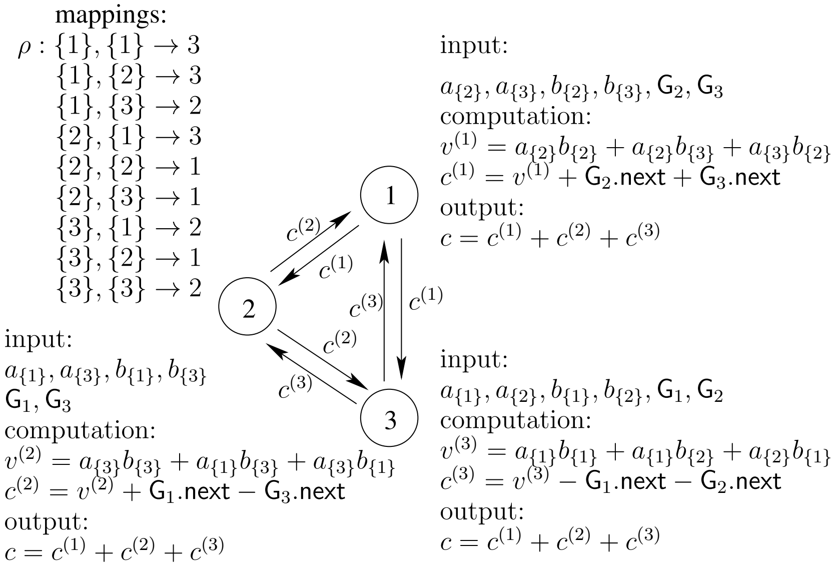

mappings:
$\rho : \{1\}, \{1\} \to 3$
$\{1\}, \{2\} \to 3$
$\{1\}, \{3\} \to 2$
$\{2\}, \{1\} \to 3$
$\{2\}, \{2\} \to 1$
$\{2\}, \{3\} \to 1$
$\{3\}, \{1\} \to 2$
$\{3\}, \{2\} \to 1$
$\{3\}, \{3\} \to 2$
input:
$a_{\{2\}}, a_{\{3\}}, b_{\{2\}}, b_{\{3\}}, \mathsf{G}_2, \mathsf{G}_3$
computation:
$v^{(1)} = a_{\{2\}}b_{\{2\}} + a_{\{2\}}b_{\{3\}} + a_{\{3\}}b_{\{2\}}$
$c^{(1)} = v^{(1)} + \mathsf{G}_2.\mathsf{next} + \mathsf{G}_3.\mathsf{next}$
output:
$c = c^{(1)} + c^{(2)} + c^{(3)}$
1
2
3
$c^{(2)}$
$c^{(1)}$
$c^{(3)}$
$c^{(1)}$
$c^{(2)}$
$c^{(3)}$
input:
$a_{\{1\}}, a_{\{3\}}, b_{\{1\}}, b_{\{3\}}$
$\mathsf{G}_1, \mathsf{G}_3$
computation:
$v^{(2)} = a_{\{3\}}b_{\{3\}} + a_{\{1\}}b_{\{3\}} + a_{\{3\}}b_{\{1\}}$
$c^{(2)} = v^{(2)} + \mathsf{G}_1.\mathsf{next} - \mathsf{G}_3.\mathsf{next}$
output:
$c = c^{(1)} + c^{(2)} + c^{(3)}$
input:
$a_{\{1\}}, a_{\{2\}}, b_{\{1\}}, b_{\{2\}}, \mathsf{G}_1, \mathsf{G}_2$
computation:
$v^{(3)} = a_{\{1\}}b_{\{1\}} + a_{\{1\}}b_{\{2\}} + a_{\{2\}}b_{\{1\}}$
$c^{(3)} = v^{(3)} - \mathsf{G}_1.\mathsf{next} - \mathsf{G}_2.\mathsf{next}$
output:
$c = c^{(1)} + c^{(2)} + c^{(3)}$

Figure 5: Illustration of three-party $\mathsf{MulPub}([a], [b])$. All arithmetic is in $\mathcal{R}$.

Similar to other secret sharing frameworks, simply opening the shares of a maintains security of the computation (in the sense that no information about private values is revealed beyond the opened value a). This is because we maintain that at the end of each operation secret-shared values are represented using random shares. In particular, it is clear that the result of $\mathsf{PRG}([s]).\mathsf{next}$ and $\mathsf{RandF}()$ produces random shares; shares are properly re-randomized during multiplication of $[a]$ and $[b]$, and shares of $[a] + [b]$ and $[a] - [b]$ are random if the shares of $[a]$ and $[b]$ are random themselves.

4.3.2 MulPub

Functionality $c = \mathsf{MulPub}([a], [b])$ refers to multiplying two secret-shared $[a]$ and $[b]$ and opening their product c. It can be invoked multiple times in our Gen algorithm, e.g., on line 5, where the computation of $[s^{j,\overline{\alpha_j}}]$ involves multiplication. The reason why we are discussing this functionality is because in the past this operation could be implemented more efficiently than multiplication followed by an opening in alternative SS frameworks (e.g., see [11]), and we pursue a similar direction here. In the protocol we present here, MulPub is realized using a single round without increasing communication cost. Executing multiplication followed by Open would double the number of rounds.

In multiplication, after computing a product, each locally computed value is no longer random and needs to be re-randomized prior to opening it. In our RSS setting, we realize this by relying on pseudo-random values locally computed by the parties. In particular, similar to other building blocks, we associate a secret key k_T with each $T \in \mathcal{T}$ (i.e., this is the same key shares used with $\mathsf{RandF}()$ and multiplication) and use pseudo-random values $\mathsf{G}_T.\mathsf{next}$ to protect the share of the product that each party locally computes, prior to that party revealing its randomized value to all others. To ensure that the product reconstructed by the parties is correct, we need to make sure that the sum of all blinding pseudo-random values equals to 0. In the three-party case, this can be accomplished by adding some pseudo-random values and subtracting others, as illustrated in Figure 5.

With larger n and t we must be careful to draw new elements from each PRG to ensure that

Algorithm 6 $c = \mathsf{MulPub}([a],[b])$

```
// pre-distributed values are $[k]$ and public map $\rho$
each $p \in [1,n]$ does the following
    $v^{(p)} = c^{(p)} = \sum_{T_1,T_2 \in \mathcal{T}, \rho(T_1,T_2)=p} a_{T_1} b_{T_2}$ (in $\mathcal{R}$)
    for $T \in S_p$ do
        let $j$ be the number of parties $p' < p$ such that $p' \notin T$
        for $i = 0$ to $t-1$ do
            $z = \mathsf{G}_T.\mathsf{next}$
            if $j = t$ then
                $c^{(p)} = c^{(p)} - z$ (in $\mathcal{R}$)
            else if $i = j$ then
                $c^{(p)} = c^{(p)} + z$ (in $\mathcal{R}$)
            end if
        end for
    end for
    send $c^{(p)}$ to all other parties
    $c = c^{(p)}$
    for $i = 1$ to $n-1$ do
        upon receiving $c^{(p')}$ from distinct $p'$, set $c = c + c^{(p')}$ (in $\mathcal{R}$)
    end for
return $c$
```

values released by different parties are protected using proper randomness without reusing them. This is similar to the logic used in multiplication. Then to realize this logic and ensure that all blinding factors add to 0, when multiple values are sampled from G_T, the last blinding value is set to the sum of all previously drawn elements multiplied by -1 (in $\mathcal{R}$). We provide a detailed description of MulPub in Algorithm 6. G_T and S_p are defined as in multiplication.

In this algorithm, each party draws the same number of elements from each G_T in its possession to ensure that after single execution of this algorithm all parties are in the same state (by any given party may discard some of the computed values). Similar to the computation in multiplication, we order the parties based on the values of their IDs. Because any given share T is stored at $t+1$ parties, there are t calls to each G_T per invocation of this operation. Then the participant with the lowest ID among the parties with access to T ($j = 0$) uses the first element of G_T to protect its value $v^{(p)}$ and disregards the $t-1$ other elements, the participant with the next lowest ID uses the second element, etc. The participant with the highest ID among those with access to T ($j = t$) computes the sum of all t elements drawn from G_T and subtracts the sum from its $v^{(p)}$. Correctness follows from the fact that the sum of all blinding values over all parties and all shares is equal to 0, i.e., $c = \sum_p c^{(p)} = \sum_p v^{(p)}$ (in $\mathcal{R}$).

To show security, we prove the following result:

Theorem 2 *The protocol* $c = \mathsf{MulPub}([a],[b])$ *is secure in the* (n,t) *setting with* $t = (n-1)/2$ *according to definition 1.*

Before proceeding with the proof, we demonstrate intuition behind it on the three-party example in Figure 5. Let z_T denote the output of $\mathsf{G}_T.\mathsf{next}$. Then party 1 transmits $c^{(1)} = v^{(1)} + z_{\{2\}} + z_{\{3\}}$, party 2 transmits $c^{(2)} = v^{(2)} + z_{\{1\}} - z_{\{3\}}$, and party 3 transmits $c^{(p)} = v^{(3)} - z_{\{1\}} - z_{\{2\}}$, where $c = v^{(1)} + v^{(2)} + v^{(3)}$ and each $v^{(i)}$ needs to be protected (all arithmetic is in $\mathcal{R}$). Without loss of generality, let party 3 be corrupt. Then party 3 (with access to $z_{\{1\}}$ and $z_{\{2\}}$) can compute $v^{(1)} + z_{\{3\}}$, $v^{(2)} - z_{\{3\}}$, and the output of the computation c, but no information about $v^{(1)}$ or $v^{(2)}$ (assuming security of the PRG) other than their sum $v^{(1)} + v^{(2)}$. The latter, however, is already computable by party 3 using the output c and its share $v^{(3)}$, which reveals no extra information about a and b beyond their product.

Proof. As before, let I denote the set of corrupt parties with $|I| = t$. We build a simulator S_I that interacts with the parties in I as follows: after S_I extracts shares a_T, b_T, k_T ($T \in \mathcal{T}$ such that $\exists p \in I$ and $p \notin T$) from the corrupt parties and receives the output c from the trusted party, S_I computes $v^{(p)}$ as prescribed by the protocol for each $p \in I$ and also their sum $v_I = \sum_{p \in I} v^{(p)}$ (in $\mathcal{R}$). S_I sets $v^{(p)}$ values for the remaining $n - t$ parties to random elements of $\mathcal{R}$ subject to $\sum_{p \notin I} v^{(p)} = c - v_I$ (in $\mathcal{R}$). S_I, acting on behalf of party $p \notin I$, sends the corresponding $v^{(p)}$ to each party in I.

To show that this simulation is indistinguishable from the real protocol execution, recall that there will be at least one T, denoted by $T^* = I$, to which the parties in I have no access (and thus correspondingly cannot distinguish the output G_{T^*} from random elements of the ring). During real protocol execution the parties in I receive $t + 1$ values $c^{(p)}$, one per $p \notin I$. With the knowledge that the corrupt parties collectively have, they can remove the effect of all randomization except the use of the output of G_{T^*}. If we let z_{i,T^*} denote the ith call to $\mathsf{G}_{T^*}.\mathsf{next}$ during the execution of MulPub in Algorithm 6, then the corrupt parties can recover t values of the form $v^{(p)} + z_{i,T^*}$ with unique p and i and one value of the form $v^{(p)} - \sum_{i=1}^{t} z_{i,T^*}$ for another p. The next thing to notice is that any t (out of $t + 1$) of these values are pseudo-random and computationally protect the corresponding $v^{(p)}$ values. The introduction of the remaining value reveals the sum of all $v^{(p)}$s, but not other information (i.e., the last value corresponds to the difference to make the sum equal to $c - v_I$). This means that substituting these values with random elements subject to $\sum_{p \notin I} v^{(p)} = c - v_I$ provides the same information to the corrupt parties and achieves computational indistinguishability of the views. □

Similar to multiplication, MulPub can be generalized to evaluate any (multi-variate) polynomial of degree 2 and open the result. For example, in Gen we compute and open $s^{j,\overline{\alpha_j}} = s^{j,1} + \alpha_j(s^{j,0} - s^{j,1})$ using this operation.

5 Binary Search

Unlike conventional ORAM where the client can hide information about the program it runs on external memory, when using ORAM for secure computation, the computational parties know the program and individual instructions being executed. Therefore, primary uses of ORAM in this setting is for accessing an element of an array (or another data structure) at a private location. ORAM capabilities are of particular importance for working with sorted data, where naive techniques for making computation data-oblivious such as linear scan result in unsatisfactory solutions (from both complexity and performance point of view) when the data set size is not small. For that reason, it is not surprising that ORAM designs for secure computation are often benchmarked on binary search. In this section we show that Top ORAM can be further optimized for performing binary search. A standard implementation of binary search on an N-element data set in the context of secure computation involves $\log N$ accesses to elements at private locations (i.e., $\log N$ ORAM reads) with the same number of private comparisons between the retrieved and searched values. Our binary search optimization, on the other hand, reduces the work associated with ORAM reads to only that associated with a single ORAM read (while retaining $\log N$ secure comparisons). What enables us to do so is the fact that FSS generation and evaluation use a tree-like structure, which we can execute a portion at a time.

Consider an array of N sorted data items $X = (X^0, X^1, \ldots, X^{N-1})$. (Each item may consist of a numeric key according to which the elements are sorted and associated data, but for the purposes of our current discussion, the exact composition of X^is is not significant as long as we can compare X^is to each other and to the query.) Then binary search for private value s consists of first comparing s to the element $X^{N/2}$. This involves access at a known location and does not require the use ORAM operations. Let α_1 denote the result of comparison $s \geq X^{N/2}$; i.e., we execute $[\alpha_1] \leftarrow \mathsf{GE}([s], [X^{N/2}])$, where GE denotes secure implementation of the $\geq$ operation. The next access is to read either $X^{N/4}$ or $X^{3N/4}$ based on α_1, i.e., perform a 2-element ORAM read access to $(X^{N/4}, X^{3N/4})$ at private location α_1 and compare it to s. Let α_2 denote the outcome of this comparison. If we continue in this manner, we observe that the next operation consists of a 4-element ORAM access to $(X^{N/8}, X^{3N/8}, X^{5N/8}, X^{7N/8})$ at a private location, the binary representation of which is $\alpha_1\alpha_2$, and computes the next private bit α_3. Then the $(\log N)$th iteration makes an ORAM access to a data set of size $2^{\log N - 1} \approx N/2$ and determines $\alpha_{\log N}$, where $\alpha_1\alpha_2 \ldots \alpha_{\log N}$ represents the location of the searched element s in the set X. Unlike many other realizations of ORAM, with our construction it is possible to extract a subset of the element of X and run an ORAM operation directly on that set. This is because our ORAM design does not need to maintain any state (in the form of stashes or any other form) and it is sufficient to have secret shares of the data items themselves.

Now let us go back to the Gen procedure in Algorithm 3. Recall that it processes one bit of location $\alpha = \alpha_1\alpha_2 \ldots \alpha_N$ at a time, in a single loop iteration. Because in the above binary search procedure one bit of α is computed at a time and the same α is used for all accesses, it is possible to incrementally execute Gen as we compute bits of α. Note that generally it is not safe to reuse randomness generated by Gen for multiple ORAM accesses if all information about private α is to be protected. However, in this case we know that (i) α does not change (in fact, we are building a single value of α) and (ii) these are read accesses which do not need to execute line 11 of Algorithm 3. The latter means that, even if we use a binary search to update an item, until we determine the entire value of α, all accesses are read accesses that do not take any private β (it would not be secure to release multiple γs after incremental executions of Gen).

Algorithm 7 specifies our binary search construction. Note that because the size of the data set N is not necessarily a power of 2, while the location α is represented as a binary integer, the first access is to $X^{2^{\ell-1}}$ (where $\ell = \lceil \log N \rceil$) instead of $X^{N/2}$ and consecutive accesses are also to indices based on the powers of 2.

We place the elements of X which contribute to the ORAM read in iteration j of the algorithm into a virtual set Y. These are elements of X at positions $(2i+i)2^{\ell-j-1}$ for $i = 0, 1, \ldots$ satisfying that $(2i+i)2^{\ell-j-1} < N$. The algorithm first computes the number of the elements in X satisfying this condition, forms Y, and consequently runs Eval on each element of Y. We also use notation Eval_R to denote a variant of Eval for reading, which takes $\mathsf{fk}_{\mathsf{pub}}$ of the form of triples $(\sigma^k, \tau^{k,0}, \tau^{k,1})$, i.e., without γ, and outputs no y_T.

Note that, while Algorithm 7 specifies to execute Eval_R in each loop iteration j of the algorithm with key $\mathsf{fk}_{\mathsf{pub}} = ((\sigma^k, \tau^{k,0}, \tau^{k,1})_{k=1}^{j})$ and j-bit index i (line 18), we should think of it as performing incremental execution, similar to the way it is done for Gen. In particular, in the first loop iteration, we execute Eval_R with a single bit index ind and store the computed values w_T^1, t_T^1 for both $\mathsf{ind} = 0$ and $\mathsf{ind} = 1$. In the second loop iteration, we resume from w_T^1, t_T^1 computed for $\mathsf{ind} = 0$ to compute w_T^2, t_T^2 for both $\mathsf{ind} = 00$ and $\mathsf{ind} = 01$ and from w_T^1, t_T^1 computed for $\mathsf{ind} = 1$ to compute w_T^2, t_T^2 for $\mathsf{ind} = 10$ and $\mathsf{ind} = 11$. In other words, we execute one loop iteration of Eval at a time without repeating the computation of the prior iterations. This gives us the desired performance:

Algorithm 7 $\mathsf{BinarySearch}([X] = ([X^0], \ldots, [X^{N-1}]), [s])$

```
$\ell \leftarrow \lceil \log N \rceil$
$[w^0] \leftarrow \mathsf{RandF}()$
$[t^0] \leftarrow 1$
$[\mathsf{fk}] = ([w^0], [t^0])$
$\mathsf{fk}_{\mathsf{pub}} = \emptyset$
$[\alpha_1] \leftarrow \mathsf{GE}([s], [X^{2^{\ell-1}}])$
for $j \in [1, \ell]$ do
    $[s^{j,0}]||[v^{j,0}] \mid\mid [s^{j,1}]||[v^{j,1}] := \mathsf{PRG}([w^{j-1}])$
    $\sigma^j = \mathsf{Open}([s^{j,\overline{\alpha_j}}])$
    $\tau^{j,0} = \mathsf{Open}([v^{j,0}] + [\alpha_j] + 1)$
    $\tau^{j,1} = \mathsf{Open}([v^{j,1}] + [\alpha_j])$
    $[w^j] = [s^{j,\alpha_j}] + [t^{j-1}] \cdot \sigma^j$
    $[t^j] = [v^{j,\alpha_j}] + [t^{j-1}] \cdot [\tau^{j,\alpha_j}]$
    $\mathsf{fk}_{\mathsf{pub}} = \mathsf{fk}_{\mathsf{pub}}||(\sigma^j, \tau^{j,0}, \tau^{j,1})$
    if $(j < \ell)$ then
        $N' = \lfloor N/2^{\ell-j} + 1/2 \rfloor$
        $[Y] = ([Y^0], \ldots, [Y^{N'-1}]) = ([X^{(2i+1)2^{\ell-j-1}}])_{i=0}^{N'-1}$
        Party $p \in [1, n]$ executes $t_T^i = \mathsf{Eval}_R(\mathsf{fk}_T, \mathsf{fk}_{\mathsf{pub}}, i)$ for each $i \in [0, N'-1]$ and $T \in \mathcal{T}$ subject
        to $p \notin T$
        $[v] \leftarrow \mathsf{DotProd}(\langle [Y^0], \ldots, [Y^{N'-1}] \rangle, \langle [t^0], \ldots, [t^{N'-1}] \rangle)$
        $[\alpha_{j+1}] \leftarrow \mathsf{GE}([s], [v])$
    end if
end for
return $[\mathsf{fk}], \mathsf{fk}_{\mathsf{pub}}, [\alpha] = [\alpha_1] \ldots [\alpha_\ell]$
```

the computation is similar to a single ORAM access plus ℓ executions of GE.

6 Share Conversion

As mentioned earlier, other types of secret sharing have received a lot of attention in the research literature and have efficient protocols for commonly used operations such as comparisons, bit decomposition, etc., as well as more complex fixed-point or floating-point arithmetic. For that reason, when our ORAM is used as part of general secure function evaluation, for performance reasons it might be desirable to be able to convert between RSS and other types of secret sharing. This is the topic that we treat in this section and discuss conversion to and from Shamir SS, as well as three-party additive SS as used in Sharemind.

6.1 Shamir Secret Sharing

Shamir secret sharing [29] (SSS) is an (n, t)-linear SS scheme with $t < n/2$, in which each secret value s is represented by a random polynomial of degree t with the free coefficient set to s. Each share of s corresponds to the evaluation of the polynomial on a unique non-zero point. Then given $t + 1$ or more shares, the parties can reconstruct the polynomial and learn s using Lagrange interpolation. Possession of t or fewer shares, however, information-theoretically reveals no information about s. With this representation, computation of any linear combination of secret-shared values is

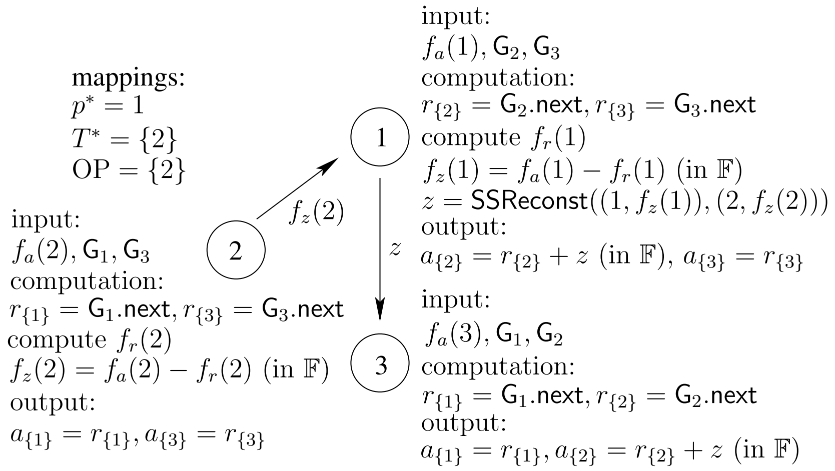

mappings:
$p^* = 1$
$T^* = \{2\}$
$\mathrm{OP} = \{2\}$
input:
$f_a(2), \mathsf{G}_1, \mathsf{G}_3$
computation:
$r_{\{1\}} = \mathsf{G}_1.\mathsf{next}, r_{\{3\}} = \mathsf{G}_3.\mathsf{next}$
compute $f_r(2)$
$f_z(2) = f_a(2) - f_r(2)$ (in $\mathbb{F}$)
output:
$a_{\{1\}} = r_{\{1\}}, a_{\{3\}} = r_{\{3\}}$
2
$f_z(2)$
1
z
3
input:
$f_a(1), \mathsf{G}_2, \mathsf{G}_3$
computation:
$r_{\{2\}} = \mathsf{G}_2.\mathsf{next}, r_{\{3\}} = \mathsf{G}_3.\mathsf{next}$
compute $f_r(1)$
$f_z(1) = f_a(1) - f_r(1)$ (in $\mathbb{F}$)
$z = \mathsf{SSReconst}((1, f_z(1)), (2, f_z(2)))$
output:
$a_{\{2\}} = r_{\{2\}} + z$ (in $\mathbb{F}$), $a_{\{3\}} = r_{\{3\}}$
input:
$f_a(3), \mathsf{G}_1, \mathsf{G}_2$
computation:
$r_{\{1\}} = \mathsf{G}_1.\mathsf{next}, r_{\{2\}} = \mathsf{G}_2.\mathsf{next}$
output:
$a_{\{1\}} = r_{\{1\}}, a_{\{2\}} = r_{\{2\}} + z$ (in $\mathbb{F}$)

Figure 6: Illustration of three-party share conversion from Shamir SS to Replicated SS.

performed locally by each party using its shares, while multiplication requires interaction.

All operations are performed in a finite field $\mathbb{F}$, and having a ring $\mathbb{Z}_{2^k}$ is not sufficient. For that reason, if Top ORAM is used as part of secure function evaluation realized using SSS, we instantiate the ORAM using the same field $\mathbb{F}$.

Let $\langle i, f_a(i)\rangle$ denote a Shamir secret share of $a \in \mathbb{F}$ evaluated on point $i \geq 1$. We assume that each party's ID i (for $1 \leq i \leq n$) is used as the first component of the share and is not explicitly stored.

Cramer at al. [12] showed how to convert an integer represented using RSS into SSS with no interaction. They also proved that local conversion from SSS to RSS is not possible. Thus, we design an interactive solution that allows n parties to convert their Shamir secret share of a to the replicated form.

Our solution proceeds by first generating RSSs of a pseudo-random value r and converting it to SSS (which the parties can do without interaction). Next, the parties can compute SSSs of $z = a - r$ (in $\mathbb{F}$) by locally setting $f_z(i) = f_a(i) - f_r(i)$ and open the value of z, which does not reveal information about private a. Now the knowledge of z can be used to construct RSSs of a from those of r. Note that this is easy to do by replacing one (replicated share of r) r_Ts, denoted by r_{T^*}, with $r_{T^*} + z$. In other words, we use $r = \sum_{T \in \mathcal{T}} r_T$ to obtain $a = \sum_{T \in \mathcal{T}} a_T = z + \sum_{T \in \mathcal{T}} r_T$.

As with other protocols, we use pre-distributed key shares k_T for the parties to agree on new pseudo-random values. Furthermore, to minimize communication, we instruct only one party to reconstruct z from shares, which makes it asymmetric with respect to the protocol participants. For that reason, they need to agree on the communication pattern; namely, whose (Shamir) shares will be used in reconstruction and who will be reconstructing the shares. This also influences which value of $T^* \in \mathcal{T}$ will be used to convert (replicated) shares of r to those of a. We call the party $p^* \in [1, n]$ performing share reconstruction as the assembling party and desire that $p^* \notin T^*$ (i.e., p^* has access to shares T^*). An example of this solution for three parties is given in Figure 6 with the general solution described next.

To distinguish between replicated and Shamir secret shares, we use notation $[x]_S$ to denote that x is secret shared using SSS. We also denote the (Shamir) share reconstruction procedure by $\mathsf{SSReconst}$, which is parameterized by $t+1$ and takes (at least) $t+1$ pairs $\langle i, f_a(i)\rangle$ and reconstruct the value of a.

This protocol involves sending the total $2t = n - 1$ elements in 2 rounds, which amounts to < 1

Algorithm 8 $[a] \leftarrow \mathsf{S2RConv}([a]_S)$

```
// pre-distributed values are $[k]$ and agreed upon $T^* \in \mathcal{T}$, assembling party $p^*$ with $p^* \notin T^*$, and
// $t$ originating parties other than $p^*$ denoted by OP
Compute $[r] = \mathsf{PRG}([k]).\mathsf{next}$.
Each $p \in \mathrm{OP} \cup \{p^*\}$ converts shares $r_T$ to SSS as $f_r(p) = \sum_{T \in S_p} r_T \cdot g_T(p)$ (in $\mathbb{F}$), where $g_T$ is
  the unique degree-$t$ polynomial such that $g_T(0) = 1$ and $g_T(i) = 0 \ \forall i \in T$.
Each $p \in \mathrm{OP}$ computes $f_z(p) = f_a(p) - f_r(p)$ (in $\mathbb{F}$) and sends it to $p^*$.
Party $p^*$ computes its own $f_z(p^*)$ and runs $\mathsf{SSReconst}_{t+1}$ on the $t+1$ shares $f_z(p)$ for $p \in$
  $\mathrm{OP} \cup \{p^*\}$.
Party $p^*$ sends the reconstructed $z$ to each $p \notin T^*$.
Each $p$ sets $a_T = r_T$ for each $T \neq T^* \in S_p$, while each $p \notin T^*$ sets $a_{T^*} = r_{T^*} + z$ (in $\mathbb{F}$).
return $[a]$
```

amortized element per participant. Each participant locally performs $\binom{n-1}{t}$ cryptographic operations. It is possible to reduce the number of rounds to 1 at the cost of increasing communication. In particular, an alternative solution is to simultaneously reconstruct z from shares at $t+1$ parties, after which not further communication is needed. This results in the total of $t(t+1)$ elements being sent in the first round. Note that this solution is attractive in the three-party case, where both alternatives yield the same volume of communication.

Theorem 3 *Protocol* $[a] \leftarrow \mathsf{S2RConv}([a]_S)$ *is secure in the* (n, t) *setting with* $t = (n-1)/2$ *according to definition 1.*

Proof. Let $|I| = t$. We consider two cases: (1) at least of the assembling and originating parties is corrupt, and (2) none of the assembling and original parties are corrupt. The second case is straightforward because the corrupt parties do not receive any information and both the simulator and security are trivial. Thus, we concentrate on the first case.

We build the simulator S_I as follows: if p^* is not corrupt, S_I receives values $f_z(p)$ from each $p \in I$ in step 3 on behalf of p^* and in step 5 sends a random element z to each $p \in I$ subject to $p \notin T^*$. Otherwise, if p^* is corrupt, S_I sends a random element to p^* on behalf of each $p \notin I$ in step 3 and receives p^*'s communication in step 5 on behalf of each honest $p \notin T^*$.

To demonstrate indistinguishability, we also divide the analysis based on whether p^* is corrupt or not. When p^* is not corrupt, each corrupt $p \notin T^*$ receives a random element in the ideal world, while in the real world it receives $z = a - r$. Then because r is pseudo-random and the corrupt parties do not possess enough shares to recover its value, z is pseudo-random as well. Furthermore, assuming the security of the PRG, the real and simulated executions are indistinguishable. When on the other hand p^* is corrupt, it receives random elements in place of $f_z(p)$ in the simulated view. This will result in p^* reconstructing a random element of $\mathbb{F}$ in the simulated view. However, as before we have that z is pseudo-random to the corrupt parties in the real protocol execution and due to security of the PRG, the simulated and real views are indistinguishable. □

6.2 Additive Secret Sharing

In the Sharemind framework, there are three participants and each party holds share a_i such that $a_1 + a_2 + a_3 = a$ in ring $\mathbb{Z}_{2^k}$ [5, 6]. Then because both RSS and Sharemind's additive SS work over a ring and are represented similarly, the conversion procedures are simple. In particular, to convert from three-party RSS where each participant stores 2 shares to Sharemind's SS, each party simply "forgets" one of its shares in such a way that each party keeps a unique share. This will result in a correctly formed Sharemind's secret sharing of the same value, but party p's share is known by another party. Because in the operations that follow the expectation might be that each share is not known by others, to maintain the desired level of security we might want to re-share (re-randomize) the obtained representation. This operation is one of the basic building blocks in Sharemind.

Conversion from three-party arithmetic SS to RSS is also straightforward: each party sends its share to another in such a way that each party ends up with a unique set of two shares. Then if the original shares were properly formed shares, the resulting representation will also comply with the security requirements of the RSS scheme. This costs communicating 1 ring element by each participant in a single round of communication.

7 Performance Evaluation

We have implemented Top ORAM access and binary search algorithms in C using GNU Multiple Precision Arithmetic Library (GMP) to realize secure channels between the participants (128-bit security) and OpenMP for managing threads. Other ORAM constructions that we use in our evaluation include Floram CPRG [14][3], Square-root ORAM [37], and Circuit ORAM [34]. All of these are two-party schemes and we use their implementation from [1]. Among three-party schemes, we evaluate 3PORAM [15] using its implementation from [2]. We did not run the recent ORAM from [23], which could be considered as an improvement of [15]. This is primarily because no implementation is available, but we also do not anticipate it to be competitive (its CPU time for the dataset sizes that we measure was reported to be 100ms or higher). We also did not include SPDZ-based [24, 25] (which can be instantiated with two or more parties) because of the differences in the adversarial model and thus incomparable run time. Lastly, we evaluate a two-party version of linear scan as implemented in [1]. We anticipate that this operation is more efficient in the two-party setting, in part due to its use of a constant number of communication rounds.

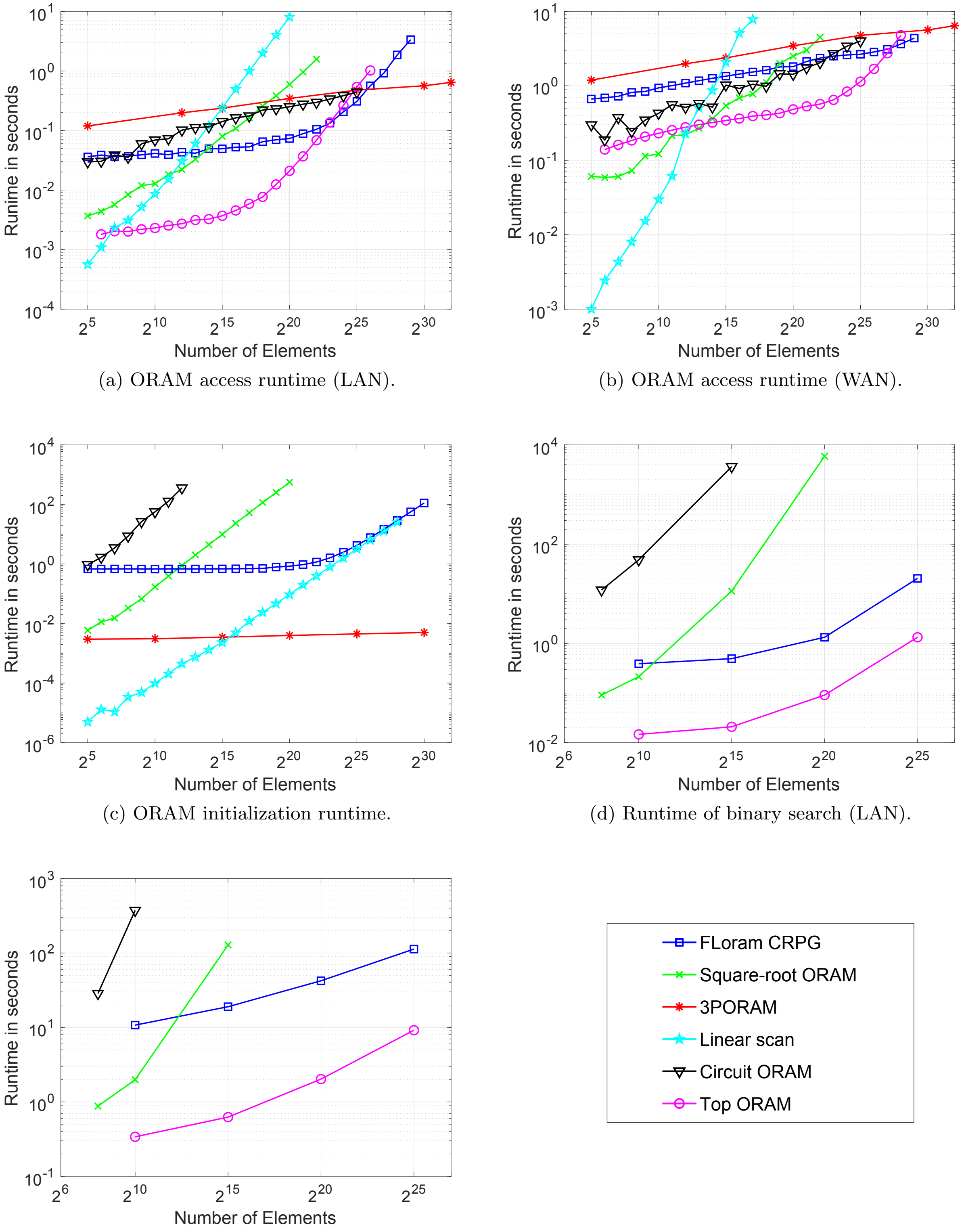
Our LAN experiments were carried out on two or three local machines with identical hardware connected via 1Gbs Ethernet with one-way latency of 0.15ms. Each machine was running CentOS 6.9 on an Intel Xeon CPU E5-2620 v4 processor (2.10 GHz, 16 cores). Our WAN experiments used one or two local machines and a remote machine running Ubuntu 16 with Intel Xeon CPU E5-2630 v3 processors (2.40 GHz, 32 cores). The one-way latency between our local and the remote machines was approximately 23ms. Although the machine configurations are slightly different between local and remote machines, this differences will not affect our evaluation results due to the interactive inter-party computations.

In our implementation, we use hardware-accelerated fixed-key AES implementation from [1] to realize PRG operations, similar to [14]. All experiments were executed with the block size of 16 bytes, and we instantiate arithmetic using the ring $\mathbb{Z}_{2^{128}}$ in Top ORAM. We varied the number of threads between 2 and 16, but only report the results of the optical configuration. For example, 2 threads give the best performance when the number of blocks is small, but 16 threads are preferred when the number of blocks is large. Similar to the experiments conducted in [14], we only use threads to parallelize local computation, but not interactive secure multi-party computation.

The results of our experiments are given in Figure 7, on which we elaborate below.

ORAM initialization. We measured the initialization time of ORAM schemes in Figure 7c as a function of data set size N. Recall that our ORAM requires no initialization as long as the data

[3] While [14] described two constructions Floram and Floram CPRG, we only evaluate the optimized version Floram CPRG which was shown to provide superior performance in [14].



(a) ORAM access runtime (LAN).

(b) ORAM access runtime (WAN).

(c) ORAM initialization runtime.

(d) Runtime of binary search (LAN).

(e) Runtime of binary search (WAN).

(f) Legend.

Figure 7: Comparison of execution times.

set is available in the secret shared form, and thus its cost is 0. This value could not be plotted in the figure with a logarithmic Y-scale, and therefore Top ORAM is not included. Linear scan pre-computes labels and thus the cost is not zero. As we can see from the figure, the initialization costs vary greatly. While the cost of ORAM initialization can often be considered one-time, it plays a more important role in secure multi-party computation. Depending on the function being evaluated, the number of accesses with private indices to a dataset may be small. In that case the initialization cost may dominate the overall time and it may be preferable to use an alternative solution to ORAM techniques.

ORAM access. Performance of a single ORAM access as a function of N is shown in Figures 7a and 7b for the LAN and WAN settings, respectively. As expected, liner scan offers the best performance when the number of elements is small. In the LAN setting, Top ORAM starts outperforming linear scan at the low data set size of $N = 2^7$. It has the best performance across all constructions when N is in the range $[2^7, 2^{24}]$. The largest difference in performance is at $N = 2^{14}$ when Top ORAM is 13 times faster than Floram CPRG, the second fastest construction for that size. We can see that both Floram and Top ORAM have similar curves, which demonstrate that for lower values of N the time is dominated by secure computation, while for higher values of N it is dominated by local $O(N)$ work. When N is high, performance of Top ORAM is expected to be about twice as high as that of Floram because of the need to repeat $O(N)$ computation for each of the two shares that each party possesses.

In the WAN setting, the runtime of all ORAM schemes increases with linear scan being the best option up to $N = 2^{12}$. Now Square-root ORAM is competitive with Top ORAM near that size, and Top ORAM is the best performing option for N in the range $[2^{14}, 2^{27}]$. The largest difference in performance of Top ORAM and the next best performing scheme is at 2^{22}, where Top ORAM is faster by a factor of 3.1.

Binary search. Our last experiments evaluate performance of binary search. Our construction is realized as described in Section 5, while the implementation of binary search for other ORAM constructions (Floram CPRG, Square-root ORAM, and Circuit ORAM) are from [1]. We do not evaluate binary search based on 3PORAM because 3PORAM uses custom techniques not compatible with common secure multi-party computation frameworks. This prevent 3PORAM from being used in binary search because of the need perform secure comparisons. Top ORAM's implementation used the implementation of GE from [11] which is based on Shamir SS. Consequently, all arithmetic was carried over field, and we used $\mathbb{Z}_p$ with a 128-bit prime p. Because ORAM's computation uses RSS, we convert between RSS and SSS shares. The comparisons were realized using 32-bit keys, while the ORAM elements in all constructions were 16 bytes. The results are given in Figures 7d and 7e for the LAN and WAN settings, respectively. As evident from the figures, our construction significantly outperforms other alternatives for all tested values of N, with the largest speedup being by a factor of 25 in the LAN setting and by a factor of 30 in the WAN setting near size $N = 2^{15}$.

8 Related Work

Oblivious RAM was introduced by Goldreich and Ostrovsky [19, 27, 20] to address the problem of software protection. In the last decade, ORAM has attracted a lot of interest and inspired a series of subsequent works (e.g., [30, 32, 30, 33, 28, 16] and others) with improved performance. In the original formulation, a client outsources its private memory to a remote server without revealing any information including access patterns to the server. There are also a number of publications in the multi-server setting (e.g., [31, 21]), where multiple servers store the client's private memory and serve its queries to reduce client's overhead. The closest among them to our work is recent S^{3}ORAM [21], which uses Shamir SS and secure multi-party computation techniques for processing the client's queries to achieve $O(1)$ client-server and $O(\log N)$ server-server communication. While relying on secret sharing, this construction is not suitable for our problem because we have to distribute the client's work and protect all data.

In the context of ORAM for secure computation, SCORAM [35] was among the early constructions in the two-party setting. The observation was that circuit complexity was an important measure besides bandwidth and storage overheads. SCORAM used an efficient eviction algorithm, which reduced the circuit size by a factor of 10 compared to prior work and consequently achieved better performance. Following this path, Circuit-ORAM [34] further reduced the circuit for eviction algorithm and reached the optimal circuit size for realistic choices of block sizes. Square-root ORAM [37] followed a somewhat different approach with higher asymptotic complexity, but improved performance which was competitive with linear scan even for small data sizes. More recently, Floram [14] built on FSS in the two-party setting and improved performance of prior schemes. As previously discussed, we build on this construction.

In the multi-party setting, Keller and Scholl [24] designed and implemented ORAM constructions on top of SPDZ [13], security of which holds in the malicious model with no honest majority with $n \geq 2$ parties. Despite much stronger adversarial model, the construction achieved better online performances than SCORAM. A recent improvement [25] builds ORAM in the same security model using efficient garbled-circuit-based secure RAM computation and SPDZ authenticated shares. Their protocol achieved lower circuit complexity and constant communication rounds. 3PORAM [15] is a scheme that works only for 3 parties with custom techniques based on a variant of [30], and we use this scheme in our evaluation. Most recently, Jarecki and Wei [23] re-designed Circuit ORAM for the three-party setting by using customized asymptotically bandwidth-optimal and constant-round protocols.

Concurrently with our work, Bunn et al. [10] designed a 3-party ORAM based on a distributed point function. Their ORAM relies on a different approach and has different characteristics: constant-round access, $O(\sqrt{N})$ communication, and $O(N)$ local computation. No performance data for this ORAM is available at this time.

We also mention publications related to other aspects of our work. The idea of realizing binary search at the cost of approximately one ORAM access has been known before. In particular, it was one of the optimizations proposed by Gentry et al. [18] for a tree-based ORAM. Also, Wang et al. [36] designed a number of oblivious data structures, some of which enjoyed a similar property, but were implemented in a different way. Both of them use very different underlying techniques from those of Top ORAM. Lastly, there has been recent work on three-party RSS [3] which uses custom share representation, different from the original formulation of RSS. In the three-party setting, our PRG and multiplication protocols achieve the same performance, and we are not aware of generalization of that work to $n > 3$.

9 Conclusions

In this work we make several contributions to the field of secure multi-party computation, as summarized below. We introduce a new multi-party oblivious RAM scheme called Top ORAM, which for an N-element data set uses secure computation of size $O(\log N)$ in $O(\log N)$ rounds has $O(N)$ local work per ORAM access. The ORAM can be instantiated with any number of participants n in the setting with honest majority and semi-honest adversaries. Its performance is particularly attractive for $n = 3$ and outperforms other 2- and 3-party alternatives for many choices of N, but is expected to significantly degrade as n increases. To enable this functionality, we develop arithmetic for computing with replicated secret shares using computational security to reduce communication. The protocols can be instantiated over any finite ring.

The structure of our ORAM operations are such that they are amenable to be partially executed to enable realization of binary search on a N-element sorted data set at the cost of only 1 ORAM access and $\log N$ secure comparisons to guide the search. For $n = 3$ parties, this results in a construction that outperforms all 2- and 3-party alternatives that we are aware of by an order of magnitude or more. Lastly, we discuss conversion between replicated and other types of secret sharing such as Shamir and Sharemind's SS to enable integration of Top ORAM with other secure multi-party computation frameworks.

Acknowledgments

We would like to thank Jack Doerner and abhi shelat for useful clarifications about Floram and its implementation. We also acknowledge the NSF-sponsored Global Environment for Network Innovations (GENI) test bed, which allowed us to run WAN experiments.

References

[1] The floram oblivious ram implementation for secure computation. `https://gitlab.com/neucrypt/floram/tree/floram-release`. [Online; accessed 20-Nov-2018].

[2] ORAM3P: Three-party oram implementation. `https://github.com/Boyoung-/oram3pc`. [Online; accessed 20-Nov-2018].

[3] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *ACM Conference on Computer and Communications Security (CCS)*, pages 805–817, 2016.

[4] D. Beaver and A. Wool. Quorum-based secure multi-party computation. In *EUROCRYPT*, pages 375–390, 1998.

[5] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security (ESORICS)*, pages 192–206, 2008.

[6] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

[7] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In *EUROCRYPT*, pages 337–367, 2015.

[8] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing: Improvements and extensions. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1292–1303, 2016.

[9] N. Buescher, A. Weber, and S. Katzenbeisser. Towards practical RAM based secure computation. In *European Symposium on Research in Computer Security (ESORICS)*, pages 416–437, 2018.

[10] P. Bunn, J. Katz, E. Kushilevitz, and R. Ostrovsky. Efficient 3-party distributed ORAM. Cryptology ePrint Archive Report 2018/706, 2018.

[11] O. Catrina and S. De Hoogh. Improved primitives for secure multiparty integer computation. In *International Conference on Security and Cryptography for Networks (SCN)*, pages 182–199, 2010.

[12] R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Theory of Cryptography Conference (TCC)*, pages 342–362, 2005.

[13] I. Damgard, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.

[14] J. Doerner and A. Shelat. Scaling ORAM for secure computation. In *ACM Conference on Computer and Communications Security (CCS)*, pages 523–535, 2017.

[15] S. Faber, S. Jarecki, S. Kentros, and B. Wei. Three-party ORAM for secure computation. In *ASIACRYPT*, pages 360–385, 2015.

[16] C. W. Fletcher, M. Naveed, L. Ren, E. Shi, and E. Stefanov. Bucket ORAM: Single online roundtrip, constant bandwidth oblivious RAM. *IACR Cryptology ePrint Archive*, 2015:1065, 2015.

[17] R. Gennaro, M. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *PODC*, pages 101–111, 1998.

[18] C. Gentry, K. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs. Optimizing ORAM and using it efficiently for secure computation. In *International Symposium on Privacy Enhancing Technologies Symposium (PETS)*, pages 1–18, 2013.

[19] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *ACM Symposium on Theory of Computing (STOC)*, pages 182–194, 1987.

[20] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[21] T. Hoang, C. D Ozkaptan, A. A Yavuz, J. Guajardo, and T. Nguyen. S^3ORAM: A computation-efficient and constant client bandwidth blowup ORAM with Shamir secret sharing. In *ACM Conference on Computer and Communications Security (CCS)*, pages 491–505, 2017.

[22] M. Ito, A. Saito, and T. Nishizeki. Secret sharing schemes realizing general access structures. In *IEEE Global Telecommunication Conference (Globecom)*, pages 99–102, 1987.

[23] S. Jarecki and B. Wei. 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. In *Applied Cryptography and Network Security (ANCS)*, 2018.

[24] M. Keller and P. Scholl. Efficient, oblivious data structures for MPC. In *ASIACRYPT*, pages 506–525, 2014.

[25] M. Keller and A. Yanai. Efficient maliciously secure multiparty computation for RAM. In *EUROCRYPT*, pages 91–124, 2018.

[26] U. Maurer. Secure multi-party computation made simple. In *Security in Communication Networks (SCN)*, pages 14–28, 2002.

[27] R. Ostrovsky. Efficient computation on oblivious RAMs. In *ACM Symposium on Theory of Computing (STOC)*, pages 514–523, 1990.

[28] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Ring ORAM: Closing the gap between small and large client storage oblivious ram. *IACR Cryptology ePrint Archive*, 2014:997, 2014.

[29] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

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

[31] E. Stefanov and E. Shi. Multi-cloud oblivious storage. In *ACM Conference on Computer and Communications Security (CCS)*, pages 247–258, 2013.

[32] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. *arXiv preprint arXiv:1106.3652*, 2011.

[33] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *ACM Conference on Computer and Communications Security (CCS)*, pages 299–310, 2013.

[34] X. Wang, H. Chan, and E. Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *ACM Conference on Computer and Communications Security (CCS)*, pages 850–861, 2015.

[35] X. Wang, Y. Huang, TH H. Chan, A. Shelat, and E. Shi. SCORAM: oblivious ram for secure computation. In *ACM Conference on Computer and Communications Security (CCS)*, pages 191–202, 2014.

[36] X. Wang, K. Nayak, C. Liu, T.H. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *ACM Conference on Computer and Communications Security (CCS)*, pages 215–226, 2014.

[37] S. Zahur, X. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz. Revisiting square root ORAM: Efficient random access in multi-party computation. In *IEEE Symposium on Security and Privacy*, pages 218–234, 2016.

[38] Y. Zhang, A. Steele, and M. Blanton. PICCO: A general-purpose compiler for private distributed computation. In *ACM Conference on Computer and Communications Security (CCS)*, pages 813–826, 2013.