Improved Building Blocks for Secure Multi-Party Computation based on Secret Sharing with Honest Majority

Marina Blanton[1], Ahreum Kang[2], and Chen Yuan[1]

[1] Department of Computer Science and Engineering, University at Buffalo (SUNY), Buffalo, USA, {mblanton,chyuan}@buffalo.edu
[2] SCH Media Labs, Soonchunhyang University, Asan-si, South Korea, armk@arkang.net

Abstract. Secure multi-party computation permits evaluation of any desired functionality on private data without disclosing the data to the participants. It is gaining its popularity due to increasing collection of user, customer, or patient data and the need to analyze data sets distributed across different organizations without disclosing them. Because adoption of secure computation techniques depends on their performance in practice, it is important to continue improving their performance. In this work, we focus on common non-trivial operations used by many types of programs, where any advances in their performance would impact the runtime of programs that rely on them. In particular, we treat the operation of reading or writing an element of an array at a private location and integer multiplication. The focus of this work is on secret sharing setting with honest majority in the semi-honest security model. We demonstrate improvement of the proposed techniques over prior constructions via analytical and empirical evaluation.

Keywords: Secure multi-party computation · Secret sharing · Array access at private location · Multiplication

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 significant 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. Computation used in such analyses often requires access to data at private locations, be it due to the nature of data representation, e.g., in the form of sparse data sets or due to the nature of the algorithm itself. When such operations are executed as part of secure computation on private data, we must employ data-oblivious (i.e., data-independent) constructions for realizing the operations to eliminate leakage of private information. In the case of accessing memory at a private location, we could either access each location of the data set or array or employ more complex randomized techniques such are Oblivious RAM (ORAM) for secure computation. The latter has lower asymptotic complexities as a function of the memory size, but are more complex to set up and invoke. As a result, the former approach known as linear scan outperforms known ORAM constructions for memory of small to medium sizes. While improved secure computation ORAM techniques in both two-party and multi-party settings are an active area of research, in this work we focus on improving performance of linear scan in the multi-party setting, which is also not an entirely straightforward operation.

In addition, we revisit the multiplication operation with optimizations in the computational (as opposed to information-theoretic) setting. While information-theoretic security might be considered stronger than computational security, all known secure multi-party computation frameworks rely on secure channels for communications, which are instantiated with algorithms secure only in the computational setting. This makes any invocation of a secure multiplication protocol only computationally secure. Multiplication is a fundamental building blocks of many secure computation frameworks based on arithmetic circuits and is ubiquitously used for realizing more complex operations including linear scan.

Motivating example. Consider the problem of building an exact machine learning model such as a Bayesian network from a distributed data set located at different organizations (for instance, patient information located at different hospitals). A data set includes a number of attributes or features (e.g., age, gender, medical diagnosis, BMI, etc. in the case of medical data) with an instance of the data set corresponding to a user, customer, or patient. Constructing a model consists of determining correlation between different attributes based on instances located at different locations. This can be accomplished using the so-called variable assignment or parent assignment problem [23], which is the critical component of Bayesian network learning, Markov blankets identification and, more generally, feature selection [23, 19]. Correlation between different variables is commonly computed using the MDL score [28] which uses conditional entropy computation as part of the score calculation. This computation is heavy on the use of logarithms, with in this context must be evaluated on private inputs. In particular, the computation takes the form of $\log(X)$, where X corresponds to the number of instances with a combination of specific values for some features (and thus is private), and the function is called extensively on different values of X. However, evaluating the logarithm function within a secure multi-party computation framework is expensive.

Our observation is that, instead of evaluating the logarithm function directly, we could build an alternative solution of much higher speed. In particular, because X is integer and ranges between 0 and the (combined) data set size, we can pre-compute the values of $\log(i)$ for each i in that range and store the result in array A. Then executing $\log(X)$ would translate into retrieving the value of stored at the private location X of array A. When the size of the array (which is proportional to the number of data set instances) is not very large, a solution based on a linear scan would outperform other alternatives such as using ORAM or evaluating the logarithm function itself. In this work we thus revisit existing solutions to implementing read or write operations at private locations in the multi-party setting based on secret sharing (SS) and show that significant optimizations are possible.

Our contributions. In this work we develop new constructions for access to an array at a private location (read or write) that significantly outperform conventional implementations of this operations in the setting with honest majority based on secret sharing. We present a general construction which works for any number of computation participants n that uses conventional Shamir secret sharing. We also present a custom construction for the common case of three parties, which outperforms the general construction. Because it uses 2-out-of-2 additive secret sharing, we show how to convert between that representation and conventional three-party Shamir SS.

We also develop optimizations for the multiplication operation based on Shamir SS in the computational setting. We provide two constructions: the first has communication complexity linear in the number of parties n (i.e., constant per party) and practical performance. The second construction has communication complexity quadratic in the number of parties, but offers lower communication complexity for the important case of $n = 3$ parties with a single field element transmitted by each party. This matches communication of the best known custom three-party multiplication protocols (designed for custom replicated secret sharing) from [3]. Our optimizations are tailored to the setting where the number of computational parties is small.

We implement our constructions on operations of varying sizes in the setting of the PICCO compiler [37] and show that they significantly outperform the previous implementations adopted by PICCO.

2 Related Work

Conventional implementations of performing an array access at a private location via a linear scan can be comparison-based or multiplexer-based as we further discuss in section 4. Optimizations to the simple solutions are available in both two-party setting based on garbled circuits (which does not directly apply to the content of this work) and in the multi-party setting. The closest to our work is the construction due to Laud [24] for array read, which is applicable to both Shamir SS and Sharemind framework. The goal of that work was to minimize the online work (which depends on the private inputs), while our goal is to minimize the overall work. As a result, the proposed solution from [24] has large round complexity. It also offers optimizations, the most effective of which is applicable only to the Sharemind framework. We draw a more detailed comparison to the construction from [24] and our solutions in section 4.

Laud [25] proposed efficient protocols for reading and writing elements of an array at private locations in parallel. The solution is based on sorting and for ℓ parallel read requests to an array of size m has complexity $O((m+\ell)\log(m+\ell))$. Because the solution is non-trivial and was implemented in the Sharemind framework, we are unable to empirically compare its runtime to our constructions designed for Shamir SS. However, in section 6 we still provide a detailed comparison based on published timings, which indicates that the solution of [25] is benefitial only when both m and ℓ are large.

Oblivious RAM [14, 26, 15] can also be used to realize array read or write at a private location, where a client outsources its private memory to a remote server without revealing any information including access patterns to the server. There are many results (see, e.g., [30, 32, 30, 33, 27, 12]) including publications in the multi-server setting (e.g., [31, 16]), but they are still not applicable to secure multi-party computation because the client's work is not distributed and the client has access to the data in the clear. In the context of ORAM for secure computation, SCORAM [35] was among the early constructions in the two-party setting, after which a number of improvements such as [34, 36, 10] followed. There are also multi-party or three-party constructions in different setting including [20, 21, 11, 18, 6]. These constructions become competitive with approaches based on a linear scan only for rather large array or dataset sizes. We compare performance of our solutions to the state-of-the-art ORAM in section 6.

Integer multiplication is a fundamental operation of any secure computation framework based on arithmetic circuits. Its original efficient implementation in Shamir SS is due to Gennaro et al. [13], while most recent version of multiplication for Sharemind can be found in [22]. The best known multiplication we are aware of is in a custom three-party replicated SS from [3], which we match with an n-party construction based on Shamir SS in this work. Additional information is provided in section 5.

3 Preliminaries

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. We work in the setting with honest majority, i.e., $t < n/2$ and focus on security against semi-honest participants, in which the participants are trusted to follow the prescribed computation, but might attempt to learn unauthorized information based on the information they possess. We use the standard simulation-based security definition that requires that the participants do not learn any information beyond their intended output. We provide the formal security proofs of our protocols in the full version [5].

As customary with techniques based on SS, 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 (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.

Secret sharing. A secret sharing scheme allows one to produce shares of secret x such that access to a predefined number of shares reveals no information about x. In the case of (n, t) threshold secret sharing schemes, there are n participants, each of whom receive their own shares. The security requirement is that possession of shares stored at any t or fewer parties reveals no information about x, while access to shares stored at $t + 1$ or more parties allows for efficient reconstruction of x. We refer to this type of secret sharing as t-sharing. Of particular importance to secure multi-party computation is linear secret sharing schemes, which have the property that a linear combination of secret shared values can be performed locally on the shares.

Shamir secret sharing. Shamir secret sharing [29] (SSS) is an (n, t)-linear SS scheme with $t < n/2$, where computation takes places over a finite field $\mathbb{F}$. A secret value $s \in \mathbb{F}$ 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. Consequently, given $t+1$ or more shares, the parties can reconstruct the polynomial and learn s using Lagrange interpolation. Possession of t or fewer shares, on the other hand, information-theoretically reveals no information about s. With this representation, computation of any linear combination of secret-shared values is performed locally by each party using its shares, while multiplication requires interaction.

In what follows, we use notation $[x]$ to denote that the value of x is secret shared among the participants using (Shamir) t-sharing. We also let $[x]_p$ denote the share of x stored at party $p \in [1, n]$. Because each secret-shared value is a field element, the size of the field $\mathbb{F}$ needs to be large enough to be able to represent values in the desired range. For example, to be able to support computation on k-bit integers, we must have that $|\mathbb{F}| \geq 2^k$. Furthermore, because we rely on certain building blocks from [7], some of them may place additional constraints on the field size (to increase the size by a certain amount) as specified in [7].

As customary in the literature, we measure performance in the number of elementary interactive operations (such as multiplication, opening the shares of private value, etc.) and the number of sequential operations, i.e., rounds. Local operations are not included in the cost due to their speed.

Replicated secret sharing. Replicated secret sharing (RSS) [17] is another type of linear secret sharing that can be used to realize (n, t)-threshold secret sharing (and can be defined for more general access structures Γ, but we limit our use to threshold structures only). RSS can be defined for any $n \geq 2$ and any $t < n$ and works over any finite ring. The RSS access structure uses the notion of qualified sets, which are all subsets of the participants who are permitted to reconstruct the secret (i.e., all subsets of $t+1$ or more parties in our case), while all other subsets are called unqualified. To secret-share private $x \in \mathbb{F}$ using RSS, we additively split it into shares x_T such that $x = \sum_{T \in \mathcal{T}} x_T$ (in $\mathbb{F}$), where $\mathcal{T}$ consists of all maximal unqualified sets (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 $p \notin T$. 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. However, for small n, the number of shares is small (e.g., with both (3, 1) and (3, 2) RSS, there are the total of 3 shares).

An important optimization on which we rely is non-interactive evaluation of a pseudo-random function (PRF) using RSS in the computational (as opposed to information-theoretic) setting as proposed in [8]. In particular, [8] provide a mechanism for non-interactive generation of Shamir secret shares from RSSs as follows: suppose that shares of secret key k have been distributed to the parties according to a (n, t)-threshold RSS and let $\mathsf{PRF} : \{0, 1\}^\kappa \times \{0, 1\}^* \to \mathbb{F}$ denote a pseudo-random function that takes a sufficiently large κ-bit key (κ is a security parameter). On input a, the parties collectively compute shares of $x = \mathsf{PRF}_k(a) = \sum_{T \in \mathcal{T}} \mathsf{PRF}_{k_T}(a)$ (in $\mathbb{F}$), where each party $p \in [1, n]$ computes its (Shamir) share of x as $[x]_p = \sum_{T \in \mathcal{T}, p \notin T} \mathsf{PRF}_{k_T}(a) \cdot f_T(p)$. Here, for each $T \in \mathcal{T}$, f_T refers to the unique polynomial of degree t such that $f_T(0) = 1$ and $f_T(p) = 0$ for each $p \in T$. We note that each (replicated) share k_T needs to be sufficiently long, while the computed (Shamir) shares $[x]_p$ can be within a smaller field. We denote this operation by PRSS (pseudo-random secret sharing).

4 Array Access at a Private Location

We next proceed with our constructions, which are in the honest majority setting based on Shamir SS. In this section we treat optimizations to array access at a private location, while the next section discusses integer multiplication.

Assume that we are given an array of m (private or public) elements $a_0, \ldots, a_{m-1}$ and would like to retrieve the element a_j at a private index j. Conventional implementations of this functionality via linear scan include (i) privately comparing j to every integer in the range $[0, m - 1]$ to compute m bits and computing the dot product of the resulting bits and the array elements and (ii) bit-decomposing the index and using a multiplexer to retrieve the desired element. The latter approach was implemented in the PICCO compiler [37] using conventional Shamir secret sharing arithmetic, while the former was later shown to be slightly faster for this setting [4]. Array write is implemented similarly, where instead of computing the dot product (i.e., a sum of products), we update each element of the array based on the result of an individual product. A similar logic is used for the multiplexer-based approach as well.

4.1 General Construction

Our starting point for improving the general solution was the first traditional approach above where we privately compare j to each position of the array and retrieve the element for which the result of the comparison was true. If we let

EQ denote the operation of privately comparing two integers for equality with at least one of them being private, this operation can be represented as follows:

```
$[b] \leftarrow \mathsf{ArrayRead}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [j])$
 1. for $i = 0$ to $m-1$, compute in parallel $[c_i] \leftarrow \mathsf{EQ}([j], i)$;
 2. $[b] \leftarrow \sum_{i=0}^{m-1} [c_i] \cdot [a_i]$;
 3. return $[b]$;
```

This computation is written for an array of private elements, but when the elements are public, the computation proceeds similarly. To turn this into the write operation where we write value w at private location j, one would use:

```
$[b] \leftarrow \mathsf{ArrayWrite}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [j], [w])$
 1. for $i = 0$ to $m-1$, compute in parallel $[c_i] \leftarrow \mathsf{EQ}([j], i)$;
 2. for $i = 0$ to $m-1$, compute in parallel $[b_i] \leftarrow [c_i]([w] - [a_i]) + [a_i]$;
 3. return $[b_0], \ldots, [b_{m-1}]$;
```

The second line here implements branching based on the value of c_i to use either w or a_i, as in $[b_i] \leftarrow [c_i] \cdot [w] + (1 - [c_i])[a_i]$, and is rewritten to lower the number of multiplications. The cost of both ArrayRead and ArrayWrite is heavily dominated by the cost of comparison EQ.

To optimize performance of this operation, our first observation stems from the fact that j is compared to all index values between 0 and $m-1$ and, as a result, part of the computation might be redundant. To determine whether this might be the case, let us look at the details of the secure equality operation EQ. The most efficient constant-round equality protocol in our setting is due to Catrina and de Hoogh [7], which we specify below. It proceeds by comparing a single private integer a to 0 and is denoted by EQZ. To compare a to b, one would enter their difference $a - b$ as the input to the protocol. The algorithm also takes a second argument, which is the bitlength k of the first operand a.

```
$[b] \leftarrow \mathsf{EQZ}([a], k)$
 1. $([r'], [r], [r_{k-1}], \ldots, [r_0]) \leftarrow \mathsf{PRandM}(k, k)$;
 2. $c \leftarrow \mathsf{Open}([a] + 2^k [r'] + [r])$;
 3. $(c_{k-1}, \ldots, c_0) \leftarrow Bits(c, k)$;
 4. for $i = 0$ to $k-1$ do $[d_i] \leftarrow c_i + [r_i] - 2c_i[r_i]$;
 5. $[b] \leftarrow 1 - \mathsf{KOr}([d_{k-1}], \ldots, [d_0])$;
 6. return $[b]$;
```

Here, the operation $\mathsf{PRandM}(k, \alpha)$ assumes that we work with k-bit integers and generates a $(k+\rho)$-bit random integer for a statistical security parameter ρ, the α least significant bits of which are available in the bit-decomposed form. The returned result is the shares of α random bits $r_0, \ldots, r_{\alpha-1}$, α-bit $r = \sum_{i=0}^{\alpha-1} 2^i r_i$, and $(k + \rho - \alpha)$-bit integer r'. The Open function reveals the value of its private argument. $Bits(c, \alpha)$ simply returns the α least significant bits of its public argument c. Lastly, KOr computes the k-ary OR of its k private input bits.

This operation hides the value of a by adding large random $2^k \cdot r' + r$ to it and opening the sum.[3] Because the bits of r are available (as r_0 through r_{k-1}),

[3] Because the original EQZ in [7] was designed for signed k-bit integers, it also specified to add 2^{k-1} to the value being opened, to move the input into the positive range.

the remaining computation can efficiently compute the bits of a (in step 4) and consequently test whether at least one of them is 1 (in step 5) using k-ary OR of k bits. The cost of this operation is dominated by PRandM which contributes k (parallel) interactive operations, while KOr costs $4\log(k)$ and Open costs 1 interactive operation, respectively. The overall number of rounds is 4.

When we compare private j to all possible indices i in the set, we invoke EQZ on inputs $j-i$, the adjacent values of which differ by 1. This introduces significant inefficiencies because expensive generation of random bits is invoked for each i to protect related values with a known difference. This means that, instead of generating independent random bits for each $j-i$ via a new call to PRandM, we could execute this function once, protect j using the random values as in step 2 above, and open this protected value as c. Given the protected value c of j, we can then form protected values of $j-i$ by computing $c-0, c-1, \ldots, c-(m-1)$ if we assume that i ranges from 0 to $m-1$. In other words, the computation for array read with a private index becomes:

```
$[b] \leftarrow \mathsf{ArrayRead}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [j])$
1. $([r'], [r], [r_{\log m-1}], \ldots, [r_0]) \leftarrow \mathsf{PRandM}(\log m, \log m)$;
2. $c \leftarrow \mathsf{Open}([j] + 2^{\log m}[r'] + [r])$;
3. for $i = 0$ to $m-1$, compute in parallel
   (a) $v \leftarrow c - i$;
   (b) $(v_{\log m-1}, \ldots, v_0) \leftarrow Bits(v, \log m)$;
   (c) for $\ell = 0$ to $\log m - 1$, compute in parallel $[d_\ell] \leftarrow v_\ell + [r_\ell] - 2v_\ell[r_\ell]$;
   (d) $[b_i] \leftarrow 1 - \mathsf{KOr}([d_{\log m-1}], \ldots, [d_0])$;
4. $[b] \leftarrow \sum_{i=0}^{m-1} [b_i] \cdot [a_i]$;
5. return $[b]$;
```

This optimization reduces the cost of array read from $m(\log m + 4\log\log m + 1) + 1$ interactive operations in 5 rounds to $4m\log\log m + \log m + 2$ in 5 rounds. Alternatively, we could use a simple tree-like implementation of KOr with $\log m - 1$ interactive operations in $\log\log m$ rounds, which makes the complexity of ArrayRead be $m(\log m - 1) + \log m + 1$ in $\log\log m + 2$ rounds.

This, however, still appears redundant because the bits of v, and consequently bits d provided as input into the k-ary OR in step 3(d), are often reused from one loop iteration i to another. For example, we know that c and $c-1$ are going to differ in their least significant bits, but a number of most significant bits might be the same. Also, because the bitlength of j is $\log m$, we know that most of (or all) possible combinations of $\log m$ bits will be used in KOr across all i. In other words, for any given v, its ith bit will be either the ith bit of c or its complement, and most of all possible $2^{\log m}$ combinations of bits will be used across all is to form vs. To combat this inefficiency, we design a new efficient mechanism for computing OR of all possible combinations of bits and then incorporate it in the private lookup protocol.

In our application, we use only non-negative values and let the entire k-bit space be occupied by them. For that reason, one should omit adding 2^{k-1}.

Our algorithm for computing ORs of bits uses a divide-and-conquer approach, where we split the original size into two halves, recurse on each half, and then assemble the result. It is denoted as AllOr and given below. On input k bits d_i, it computes 2^k k-ary ORs of the form $\bigvee_{i=0}^{k-1} c_i$, where c_i is either d_i or its complement $\neg d_i$.

```
$\langle [b_0], \ldots, [b_{2^k-1}] \rangle \leftarrow \mathsf{AllOr}([d_{k-1}], \ldots, [d_0])$
1. if $(k = 1)$ return $\langle [d_0], 1 - [d_0] \rangle$;
2. else
3.     $\ell \leftarrow \lfloor k/2 \rfloor$;
4.     $[u_0], \ldots, [u_{2^\ell-1}] \leftarrow \mathsf{AllOr}([d_{\ell-1}], \ldots, [d_0])$;
5.     $[v_0], \ldots, [v_{2^{k-m}-1}] \leftarrow \mathsf{AllOr}([d_{k-1}], \ldots, [d_\ell])$;
6.     for $i = 0$ to $2^{k-\ell} - 1$ and $j = 0$ to $2^\ell - 1$, compute in parallel $[b_{2^\ell i+j}] \leftarrow$
       $[v_i] + [u_j] - [v_i] \cdot [u_j]$;
7.     return $\langle [b_0], \ldots, [b_{2^k} - 1] \rangle$;
```

To integrate this solution into our array read protocol, we apply AllOr to the bits r_is computed in step 1 of the last variant of ArrayRead and, as before, reveal the value of j protected by r; let the $\log m$ least significant bits of the protected value be denoted by c'. The intuition is now that the computed k-ary ORs correspond to all possible k-ary ORs over all k-bit integers "shuffled" based on the value of r and the only OR that evaluates to 0 will be at position r. This means that if we would like to know whether, e.g., $j = 0$, we need to test whether $c' = r$ or, equivalently, whether the c'th position in the array of k-ary ORs corresponds to 0. Similarly, for testing whether $j = i$, we test whether $c' = r + i$ (or, equivalently, whether $r = c' - i$) and retrieve the $(c' - i)$th value in the returned array. Lastly, because we need a single OR evaluate to 1 with the remaining values being 0, we complement the result of the AllOr operation. (Note that the original implementation of EQZ from [7] computes $c \oplus r$ instead of $c - r$ prior to calling KOr using a more complex logic to show correctness of the algorithm, but the same approach does not work in our case.) We obtain the following solution:

```
$[b] \leftarrow \mathsf{ArrayRead}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [j])$
1. $([r'], [r], [r_{\log m-1}], \ldots, [r_0]) \leftarrow \mathsf{PRandM}(\log m, \log m)$;
2. $\langle [b_0], \ldots, [b_{2^{\log m}-1}] \rangle \leftarrow \mathsf{AllOr}([r_{\log m-1}], \ldots, [r_0])$;
3. for $i = 0$ to $2^{\log m} - 1$, $[b_i] = 1 - [b_i]$;
4. $c \leftarrow \mathsf{Open}([j] + 2^{\log m}[r'] + [r])$;
5. $c' \leftarrow c \bmod 2^{\log m}$;
6. $[b] \leftarrow \sum_{i=0}^{m-1} [b_{c'-i \bmod 2^{\log m}}] \cdot [a_i]$;
7. return $[b]$;
```

To realize the write operation with private index j, we replace line 6 of ArrayRead above with the computation $[d_i] \leftarrow [b_{c'-i \bmod 2^{\log m}}]([w] - [a_i]) + [a_i]$ for $i = 0, \ldots, m-1$, where, as before, $[w]$ corresponds to the value being written, and return the updated array $[d_0], \ldots, [d_{m-1}]$.

The cost of ArrayRead is dominated by that of the AllOr protocol. The recurrence in AllOr can be specified as $T(k) = 2T(k/2) + 2^k$. Thus, the function has complexity $\Theta(2^k)$ or, equivalently, $\Theta(m)$ where $k = \log m$. Furthermore, the constant behind the asymptotic notation is low and the number of interactive operations per array element reduces as the array size increases. For example, with $m = 2^4$, AllOr executes 1.5 multiplications per array element (i.e., 24), with $m = 2^8$, it is < 1.19 multiplications per array element, and with $m = 2^{16}$, it is < 1.01 per array element. The remaining steps in $\mathsf{ArrayRead}$ contribute $\log m + 2$ interactive operations. The round complexity of AllOr with $\log m$-bit argument is $\log \log m$, which means that the overall number of rounds of $\mathsf{ArrayRead}$ is $\log \log m + 3$. Furthermore, the first three steps can be precomputed, which makes the online number of rounds to be 2 and the online number of interactive operations is also 2. Implementing array write at a private location increases the total (and online) number of interactive operations by $m - 1$ without affecting round complexity.

An alternative solution for this operation developed by Laud in [24] uses $m+3$ interactive operations in $m + 3$ rounds[4] in the Shamir SS setting, where most of the work can be carried offline with the online work being 3 interactive operations in 3 rounds. The linear round complexity is however prohibitive, especially in the WAN setting. The round complexity of the array read from [24] can be reduced to a constant at the cost of increasing the number of multiplications by several times, at which point our construction is attractive and uses only a fraction of that cost. Thus, we offer practical performance improvement over known results.

To demonstrate security, we note that all instructions are input-independent and follow a similar structure to that of EQZ from [7]. All steps operate on shares except step 4, in which the value of c is revealed. The value of c corresponds to private j protected by a random value at least ρ bits longer than j. This means that the probability that any information is revealed about j is negligible in the security parameter ρ and is therefore acceptable. This implies that we are able to simulate the adversarial view without access to the inputs; see [5] for detail.

4.2 Custom Three-Party Construction

We also provide a second construction which is designed to work only with $n = 3$ parties using custom computation, but offers superior performance compared to the general construction. Our second construction uses 2-out-of-2 additive secret sharing, which means that if we would like to use it together with a standard SS framework such as Shamir SS, we need to convert between the two representations. We provide the conversion procedures in the full version [5].

In what follows, we use notation $[\![x]\!]$ to denote that the value of $x \in \mathbb{F}$ is secret shared using 2-out-of-2 additive secret sharing. We note that this solution works over any finite ring, which has performance benefits such as using native hardware implementations of arithmetic in $\mathbb{Z}_{2^k}$ for some k. For the purposes of this work, we let computation to be over a finite field to be compatible with other constructions we propose.

[4] This information is not explicitly provided in [24], but rather is deduced by us.

Because in this representation the shares are held by two parties out of three, for concreteness of the presentation, we let the notation include the parties holding the shares. Thus, we use $[\![x]\!]_{p_1 p_2}$ to indicate that the value is split between parties $p_1, p_2 \in [1, 3]$ with $p_1 \neq p_2$. For example, we might use $[\![x]\!]_{12}$. Then notation $[\![x]\!]_{p_1}$ and $[\![x]\!]_{p_2}$ denotes the shares when x is secret shared as $[\![x]\!]_{p_1 p_2}$.

In our construction, the data set is originally additively shared between parties 1 and 2 (i.e., we have $[\![a_0]\!]_{12}, \ldots, [\![a_{m-1}]\!]_{12}$). The private index j can be secret-shared using any linear SS scheme and for simplicity we assume it is shared using Shamir SS as $[j]$. The intuition behind our solution is that the data set is rotated by a private number of positions and the value of j gets adjusted by that value. Then the parties who do not have information about the entire amount of rotation learn the modified value of j and read the element at that position. To implement this idea, we need to be careful to ensure that reading the element is performed on the shares to prevent any single party from having access to the read element. And at the same time we must enforce that the parties with cleartext access to the modified j do not know by which value j was modified from its original value.

To realize this intuition, we instruct parties 1 and 2 to rotate their shares of the data set by random amount $r_1 \in \mathbb{Z}_m$ known only to the two of them. Next, party 1 re-shares its shares of the data set between parties 2 and 3, which makes the rotated data set to be shared between these two parties. Now parties 2 and 3 again rotate the shared data set by random amount r_2 known only to the two of them, after which party 2 re-shares its data set shares among parties 1 and 3. At this point, the data set has been rotated by $r_1 + r_2$ and is shared between parties 1 and 3, neither of whom knows the value of $r_1 + r_2$. Thus, we open $h = (j + r_1 + r_2) \bmod m$ to parties 1 and 3 who consequently retrieve the element at position h in their data sets and return their share as the output.

In our solution, we propose that the parties generate r_1 and r_2 non-interactively using a shared seed to a pseudo-random generator. That is, parties 1 and 2 share key k_{12}, while parties 2 and 3 share key k_{23}. Because generation of r_1 and r_2 is a one-time cost independent of the set size, any other suitable mechanism for agreeing on these values will work (e.g., if one wants to maintain information-theoretic security of the protocol). The computation then proceeds as follows:

$[\![b]\!] \leftarrow \mathsf{ArrayRead}(\langle [\![a_0]\!]_{12}, \ldots, [\![a_{m-1}]\!]_{12} \rangle, [j])$

1. Parties 1 and 2 agree on random $r_1 \in \mathbb{Z}_m$ and locally rotate their shares as $\langle [\![a_{r_1}]\!]_p, \ldots, [\![a_{m-1}]\!]_p, [\![a_0]\!]_p, \ldots, [\![a_{r_1-1}]\!]_p \rangle \leftarrow \langle [\![a_0]\!]_p, \ldots, [\![a_{m-1}]\!]_p \rangle$, where $p \in [1, 2]$, and also let $[h] \leftarrow [j] + r_1$.
2. Party 1 randomly generates $s_i \in \mathbb{F}$ for $i \in [0, m-1]$ and sends $\langle s_0, \ldots, s_{m-1} \rangle$ to party 2, who consequently sets $[\![a'_i]\!]_2 = [\![a_i]\!]_2 + s_i$ for $i \in [0, m-1]$.
3. Party 1 sets $[\![a'_i]\!]_3 = [\![a_i]\!]_1 - s_i$ for $i \in [0, m-1]$ and sends $\langle [\![a'_0]\!]_3, \ldots, [\![a'_{m-1}]\!]_3 \rangle$ to party 3.
4. Parties 2 and 3 agree on random $r_2 \in \mathbb{Z}_m$, locally rotate shares $\langle [\![a'_{r_2}]\!]_p, \ldots, [\![a'_{m-1}]\!]_p, [\![a'_0]\!]_p, \ldots, [\![a'_{r_2-1}]\!]_p \rangle \leftarrow \langle [\![a'_0]\!]_p, \ldots, [\![a'_{m-1}]\!]_p \rangle$, and let $[h] \leftarrow [h] + r_2$.
5. Party 2 randomly generates $s'_i \in \mathbb{F}$ for $i \in [0, m-1]$ and sends $\langle s'_0, \ldots, s'_{m-1} \rangle$ to party 3, who consequently sets $[\![a''_i]\!]_3 = [\![a'_i]\!]_3 + s'_i$ for $i \in [0, m-1]$.

6. Party 2 sets $[\![a''_i]\!]_1 = [\![a'_i]\!]_2 - s'_i$ for $i \in [0, m-1]$ and sends $\langle [\![a''_0]\!]_1, \ldots, [\![a''_{m-1}]\!]_1 \rangle$ to party 1.
7. Open $h \bmod m$ to parties 1 and 3 who set $[\![b]\!]_p = [\![a''_h]\!]_p$ for $p \in [1, 3]$.
8. Return $[\![b]\!]_{13}$.

This computation is dominated by communicating $4m$ elements in two rounds, i.e., similar to that of executing m multiplications in parallel. There might also be communication for computing h or $h \bmod m$ depending on the underlying SS scheme. In particular, if h is secret-shared using additive SS in $\mathbb{Z}_m$, no additional communication is needed. That is, with additive SS, we would need to modify only one of the shares to perform addition of r_1 or r_2, and the opened value will be in $\mathbb{Z}_m$, as desired, because the arithmetic is in $\mathbb{Z}_m$. With a different type of SS such as Shamir SS, the parties need to update h and re-share its value across all parties with fresh randomness. Similarly, when computation is not in $\mathbb{Z}_m$, computing $h \bmod m$ is needed prior to opening the value. For example, with SSS, one might invoke efficient Mod protocol from [7] (integer division with public divisor). This is a one-time operation of cost at most $O(\log m)$ and does not have a significant impact on the performance of the overall protocol.

If the parties would like to execute the write operation and store value $[\![w]\!]$ at private index j, we modify the protocol above to have parties 1 and 3 update the element at position h with shares of w in step 7. This is sufficient for this operation. However, if the values are to be opened instead of being used in consecutive computation, they would need to be re-randomized.

To show security in the three-party setting with a single corrupt party, we argue that the data set remains information-theoretically protected from any participant. In particular, it is always secret-shared among two parties. Furthermore, the value of j is also information-theoretically protected from the parties if r_1 and r_2 are chosen randomly (and otherwise is computationally protected). Thus, it can be shown that the simulated view with no access to real data is indistinguishable from a real run of the protocol. We provide a formal proof in the full version [5].

5 Multiplication

In this section, we design and present two new multiplication protocols suitable for use with Shamir SS that lower communication cost of prior protocols. In particular, the conventional multiplication protocol for SSS from [13] results in communicating the total of $n(n-1)$ field elements in the n-party setting, with each party sending $n-1$ field elements. This means that in the 3-party setting, the total of 6 elements are transmitted. Sharemind's multiplication protocol from [22] also results in communicating 6 elements with 3 computational parties and only works when $n = 3$; it is designed for additive SS. What we achieve is that our first multiplication protocol communicates at most $2(n-1)$ field elements and thus has lower communication cost than the protocol from [13] for any n, and in particular communicates 4 field elements with $n = 3$. Our second protocol, when instantiated with any n, has communication cost quadratic in it (specifically,

Protocol	n-party			3-party		
	comm.	rounds	comp.	comm.	rounds	comp.
Mult1 (section 5.1)	$1 + \frac{2t-1}{n}$	2	$O(n^t)$	$1\frac{1}{3}$	2	$O(1)$
Mult2 (section 5.2)	$n - t - 1$	1	$O(n)$	1	1	$O(1)$

Table 1. Summary of proposed multiplication protocols.

it is nt), but for $n = 3$ communicates only 3 field elements. It also uses fewer local operations for larger n than our first construction. Our optimizations are tailored to the settings when the number of parties n is not large. Both of our multiplication protocols are secure in the computational setting (as opposed to the information-theoretic setting in the presence of secure channels in [13]). We do not view this as a disadvantage because information-theoretically secure protocols rely on secure channels for communication, which are also built on computational assumptions.

A summary of our proposed multiplication protocols is given in Table 1. Communication refers to the average number of field elements transmitted by a party (i.e., all communication divided by the number of parties) and computation refers to the average work performed by a party including local and communication work. Performance is dominated by communication and round complexity unless local work is excessive.

5.1 Linear-Communication Multiplication

Our starting point was the multiplication protocol from [9] (figure 4 in section 3.3). The high-level structure of the computation is as follows: On input shares of a and b, each participant performs local multiplication of its shares (which raises the degree of the resulting polynomial to $2t$) and sends the result protected by a random element for reconstruction to a dedicated party (called the king). The king performs the reconstruction and announces the result to all other parties who use the opened value to adjust their respective shares. The protocol can be specified as given and uses two different types of sharings of the same field element. Namely, we have conventional t-sharing of x denoted by $[x]$ and $2t$-sharing of x denoted by $\langle x\rangle$, where shares are computed using a polynomial of degree $2t$ and at least $2t+1$ different shares are required for reconstruction of x.

```
$[c] \leftarrow \mathsf{Mult}([a], [b])$
1. $([r], \langle R\rangle) \leftarrow \mathsf{DRand}()$;
2. Each $p \in [1, n]$ computes $\langle D\rangle_p = [a]_p \cdot [b]_p + \langle R\rangle_p$ and sends $\langle D\rangle_p$ to the king;
3. The king reconstructs $D \leftarrow \mathsf{Open2}(\langle D\rangle)$ and sends $D$ to each party;
4. $[c] = D - [r]$;
5. return $[c]$;
```

Operation DRand (double random) refers to generation of a random value under two different types of secret sharing: t-sharing and $2t$-sharing. In other words, the execution of $([r], \langle R\rangle) \leftarrow \mathsf{DRand}()$ produces two different sharings of the same value: $[r]$ and $\langle R\rangle$ reconstruct to the same field element, but each sharing uses its own randomness. Open2 is similar to Open that reconstructs a value from its shares, but Open2 takes its input represented using $2t$-sharing and thus requires at least $2t+1$ shares for reconstruction.

The conventional implementation of Open (or Open2) involves parties sending their shares to others, after which each party reconstructs the value locally using its own and received shares. This requires $O(n^2)$ communication for any $t = O(n)$. However, with the use of a dedicated king, the overall communication can be lowered to $O(n)$, where the value is reconstructed only by the king. To realize Open2 in this way, we need $2t$ participants to communicate their share to the king, who reconstructs the value and consequently communicates it to all other $n-1$ participants. With $n = 2t+1$, we obtain $2n-2 = 4t$ transmitted field elements, which for the (3,1) setting corresponds to communicating the total of 4 elements. When $n > 2t+1$, still only $2t$ parties send their shares to the king, and the total number of communicated elements is $2t+n-1$.

Our main optimization consists of computing double randoms as in DRand non-interactively. While the goal of [9] was to design protocols secure in the stronger, malicious model, even their preliminary construction secure in the semi-honest security setting was not very cheap. Performing double random generation in a batch of size $\ell = n-t$ required $O(n\ell+n^2)$ communication measured in field elements. We can entirely eliminate this communication by utilizing replicated secret sharing and using computational security.

We start by saying that it is possible to generate pseudo-random $[r]$ non-interactively using RSS as described in section 3. Then if the same key shares are used in a related setup with a threshold set to $2t$, we would be able to non-interactively generate $\langle R\rangle$, where $R = r$. This, however, leads to the use of correlated randomness in the generation of $[r]$ and $\langle R\rangle$, which is not sufficient to provide the necessary security guarantees for our use of these shares. Instead, our approach is as follows: we first generate $[r]$ non-interactively using RSS. To create a $2t$-sharing of r using fresh randomness, we first raise the degree of r's secret sharing representation to $2t$ by multiplying it by another degree-t polynomial corresponding to [1]. Lastly, we randomize the resulting shares by adding fresh $\langle 0\rangle$ to the result. The last step is accomplished by calling the protocol for pseudo-random zero sharing from [8], denoted as PRZS. Luckily, that construction is already given for creating $\langle 0\rangle$ where the representation uses a polynomial of degree $2t$. We obtain the following construction for DRand that assumes pre-distributed shares k_T and a fixed representation of $[1]$[5]:

```
$([r], \langle R\rangle) \leftarrow$ DRand()
1. $[r] \leftarrow$ PRSS();
2. $\langle 0\rangle \leftarrow$ PRZS();
3. Each $p \in [1, n]$ computes $\langle u\rangle_p = [r]_p \cdot [1]_p$;
4. $\langle R\rangle = \langle u\rangle + \langle 0\rangle$;
5. return $([r], \langle R\rangle)$;
```

[5] Note that it is very easy to generate a fixed representation of [1] by choosing any degree-t polynomial that evaluates to 1 at 0, e.g., by setting all of its coefficients to 1. Each party computes $[1]_p$ using that polynomial and uses it in all calls to DRand().

Returning to the performance of our multiplication operation, we obtain communication of $2t + n - 1 \leq 2n - 2$ field elements, which we can contrast with $n(n-1)$ field elements in the solution of [13]. For a $(3, 1)$-sharing, the reduction is by a factor of $6/4 = 1.5$; for a $(5, 2)$-sharing, it is by a factor of 2.5, and the difference continues to grow with n.

To demonstrate security, we note that we only modified the DRand functionality from that of the multiplication protocol from [9]. Our DRand protocol, however, only invokes secure building blocks (PRSS and PRZS) and only operates on shares for the remaining computation without disclosing any values. This means that we can easily create a simulator which will not be able to distinguish between the real and simulated views. See the full version for a detailed proof.

5.2 Alternative Multiplication

As mentioned before, we present another multiplication protocol that outperforms the protocol above in terms of communication only when $n = 3$. However, it can still be useful for higher values of n because the total work is limited by $O(n)$ per party and does not require the use of replicated secret sharing.

The idea behind this solution is that the parties locally multiply their shares, which, as before, raises the polynomial degree to $2t$ and results in a $2t$-sharing of the product. To convert the product to a t-sharing, each participant re-shares its value using t-sharing and uses interpolation to compute the result similar to [13]. The difference is that instead of choosing a new random polynomial to do re-sharing, each party uses t pseudo-random points to create the polynomial. These points, together with the party's secret, define the polynomial and allow for the evaluation of the polynomial on other points. Then the pseudo-random points serve the role of the shares for t out of n participants, while the remaining shares are computed by the owner of the secret and are communicated to the remaining parties. The idea is that a pseudo-random value can be generated by two participants without communication and this approach reduced overall communication from $n(n-1)$ to $n(n-t-1)$ field elements, which is a factor of 2 with $n = 2t + 1$. In particular, in the case of (3, 1) secret sharing, we have each party transmitting 1 field element, for the total of 3 field elements and 25% bandwidth reduction compared to the previous multiplication protocol in section 5.2. This also matches best-known 3-party multiplication communication cost based on custom replicated secret sharing arithmetic from [3].

Before we proceed with the algorithm specification, we need to define additional notation. For a secret-shared $[x]$, we let $f_x()$ denote the underlying polynomial according to which the shares of x were computed (i.e., $[x]_p$ corresponds to $f_x(p)$ and $[x]_0 = f_x(0) = x$). We also denote the procedure of reconstructing the polynomial f_x from at least $t + 1$ shares of x by $\mathsf{SSReconst}_{t+1}$. In addition, we let λ_p denote polynomial interpolation constants as defined in [13].

We define mapping γ, which for each participant p specifies t other parties with whom p shares PRG seeds for the purpose of non-interactive share computation of its secret. Specifically, for each $\gamma(p, p') = 1$ we let $k_{p,p'}$ be the seed shared by parties p and p' and let $\mathsf{PRG}(k_{p,p'}).\mathsf{next}()$ denote retrieval of the next field element from the PRG seeded by $k_{p,p'}$. Our multiplication protocol then proceeds as follows:

```
$[c] \leftarrow \mathsf{Mult}([a],[b])$
1. Each $p \in [1,n]$ computes $\langle c\rangle_p = [a]_p \cdot [b]_p$;
2. Each $p \in [1,n]$ sets $t$ shares $[d_p]_{p'} \leftarrow \mathsf{PRG}(k_{p,p'}).\mathsf{next}()$ for each $\{p' \mid \gamma(p,p') = 1\}$ and one more share $[d_p]_0 = \langle c\rangle_p$;
3. Each $p \in [1,n]$ executes $f_{\langle c\rangle_p} \leftarrow \mathsf{SSReconst}_{t+1}([d_p])$;
4. Each $p \in [1,n]$ evaluates $[d_p]_{p'} = f_{\langle c\rangle_p}(p')$ for each $\{p' \mid \gamma(p,p') \neq 1\}$ and sends $[d_p]_{p'}$ to party $p'$ (other than $p' = p$).
5. Each $p \in [1,n]$ computes $[c]_p = \sum_{p'=1}^{n} \lambda_{p'}[d_{p'}]_p$, where $[d_{p'}]_p$ was either received in step 4 or set as $[d_{p'}]_p \leftarrow \mathsf{PRG}(k_{p',p}).\mathsf{next}()$ (for $\{p' \mid \gamma(p',p) = 1\}$);
6. return $[c]$;
```

As discussed before, this protocol communicates $n(n-t-1)$ fields elements across all parties in a single round, and the local work per party is $O(n)$.

Security follows from the fact all computation proceeds on secret-shared values and no intermediate values get revealed. Conceptually this construction follows the structure of the multiplication protocol from [13], where we replace a number of shares to be pseudo-random instead of chosen at random. Thus, while the construction of [13] is secure against unbounded adversaries (assuming secure channels), our security holds in the computational setting. A complete proof is given in the full version.

6 Performance Evaluation

We have implemented the proposed array read and multiplication operations in C using single invocation as well as batched execution. Because the custom 3-party array read is asymmetric, our batched execution of that protocol used 3 threads, each taking on the role of a different party and with the workload divided evenly across the threads. We used the GNU Multiple Precision Arithmetic Library (GMP) [2] for field arithmetic and executed SSS constructions within the PICCO compiler framework [37]. We also execute original array read with private index and multiplication operations as previously implemented in PICCO. All of our protocols are evaluated in the three-party setting with a single corrupt party. For comparison, we also include runtimes of two-party Floram CPRG [10] using their implementation from [1]. This is one of the best performing ORAM constructions among two- and three-party implementations and its performance tells us at which array sizes ORAM techniques outperform linear scan. Note that ORAM use might involve additional overhead beyond what we report, e.g., for initializing ORAM or converting between different data representations.

We provide experiments in the LAN and WAN configurations. Our LAN experiments were carried out on identical machines with a 2.1GHz processor connected via 1Gbps Ethernet with one-way latency of 0.15ms. Our WAN experiments used local machines and one remote machine with a 2.4GHz processor. One-way latency between the remote and local machines was 23ms. We note that

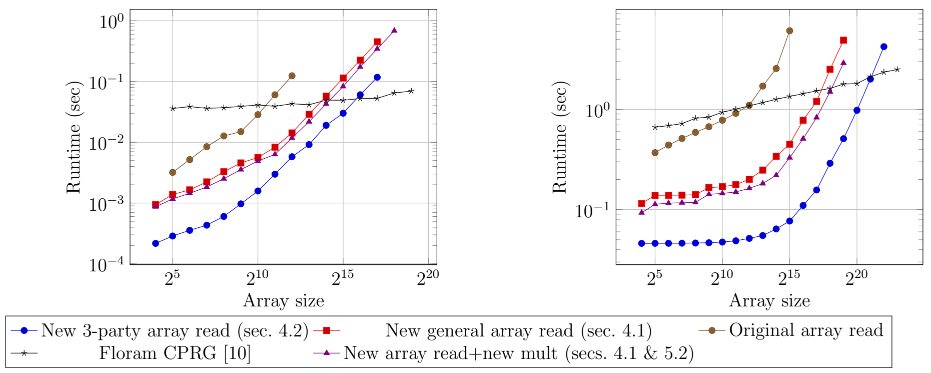

Fig. 1. Performances of array read with private index on a LAN (left) and WAN (right).

although the machine configurations were slightly different, we do not expect this to introduce inconsistencies in the experiments. In particular, computation time is dictated by the slower machines which do not change across our experiments and the introduced slowdown is attributed to the longer round-trip times and lower bandwidth in WAN experiments. All experiments except Floram used a single core and all experiments (except Floram) were executed over a 64-bit finite field and averaged over 100 executions.

Performance of array read is shown in Figures 1 in both LAN and WAN settings. We see that the custom three-party construction significantly outperforms other options and further improvements are possible with parallel execution (which we discuss later in this section). We also see that linear scan constructions outperform ORAM-based solutions for arrays of size up to 2^{16} in the LAN setting and up to 2^{21} in the WAN setting. The figure also shows the difference in the performance of our general array read protocol using the original multiplication protocol as implemented in PICCO (with 6 field elements communicated per multiplication) and the new multiplication protocol from section 5.2 (with 3 field elements per multiplication).

The difference between the two multiplication protocols is further detailed in Table 2, which shows that improved multiplication protocol provides up to over 30% and 70% runtime reduction in the LAN and WAN settings, respectively.

We further note that a flatter curve in Figure 1 indicates that round complexity or another portion of the computation sub-linear in the array size (Floram or linear scans for arrays of small sizes in the WAN setting) is the bottleneck. A steeper curve indicates that work linear in the array size (e.g., $O(m)$ communication in the case of linear scans) is the bottleneck.

We also provide measurement results for parallel execution of array read in Table 3. We compare the original PICCO multiplexer-based implementation with (i) our new general array read with new multiplication from section 5.2 and (ii) our custom 3-party array read from section 4.2. Substantial runtime reduction over single execution is observed for arrays of relatively small size and

Setting	LAN							WAN						
Batch size	1	10	10^2	10^3	10^4	10^5	10^6	1	10	10^2	10^3	10^4	10^5	10^6
Orig. mult.	0.139	0.169	0.521	2.24	23.5	246	2,520	22.9	23.03	23.8	29.0	178	1119	6760
New mult.	0.121	0.144	0.482	1.59	15.5	170	1,720	15.39	15.43	15.8	18.9	53.55	365	3,750

Table 2. Performance of the original [13] and new multiplication protocols (section 5.2) in the (3,1) setting on a LAN and WAN in batches of varying sizes in milliseconds.

	Original array read				New array read (sec. 4.1)				New 3-party array read (sec. 4.2)			
	1	10	10^2	10^3	1	10	10^2	10^3	1	10	10^2	10^3
2^4	0.0022	0.0058	0.025	0.24	0.00087	0.0021	0.0095	0.096	0.00022	0.00039	0.00084	0.0069
2^7	0.0085	0.028	0.26	2.33	0.0018	0.0071	0.044	0.46	0.00043	0.00075	0.0057	0.048
2^{10}	0.029	0.28	2.9	27.2	0.0049	0.028	0.29	2.98	0.0016	0.0039	0.036	0.37
2^{13}	0.27	2.77	28.8	276	0.022	0.22	2.2	22.5	0.0092	0.027	0.28	3.21
2^{16}	2.67	27.8	267	2,689	0.174	1.75	17.6	180	0.061	0.23	2.41	26.1

Table 3. Performance of array read with private index for varying array sizes and in batches of varying size (from 1 to 10^3) on a LAN in seconds. General constructions used (3, 1) setting.

improvement is present for all sizes in the case of the custom 3-party array read. The largest difference between the original and our general solution is by a factor of 16 with array size of 2^{16} and batch size of 10 and the largest difference between the original and our custom 3-party solution is by a factor of over 120 for the same configuration.

We also attempted to compare performance of our array read protocols with that of the parallel array access protocols from [25], which is designed to do many simultaneous read or write operations in a batch. Because the protocols were implemented in the Sharemind setting using different underlying arithmetic and building blocks, a direct comparison is not possible. Furthermore, the results were plotted in the log-scale and therefore extracted precise numbers is difficult and we can only offer approximate insights. The experiments in [25] were run on a cluster of three 12-core 3GHz computers on a 1Gbps LAN. Our conclusion was that our solutions significantly outperform that from [25] when either the array size is rather small or when the number of parallel invocations is low (or both). For example, performing 5 parallel reads from an array of size 5 costs $>$ 10ms in [25], which is 5 and 25 times slower than executing 10 reads from an array of size 2^4 in our general and 3-party solutions, respectively (recall that Sharemind-based implementation in [25] also works only with three parties). Performing 100 and 1 simultaneous reads from an array of size of 100 takes around 100ms and 50ms, respectively, which is 2 and respectively $>$ 25 times slower than the same number of reads from an array of 2^7 in our general protocol, and $>$ 17 and 115 times slower than our 3-party protocol. Executing a single read is always faster in our solution for all available data points by a significant amount (1–3 orders of magnitude). Where the construction of [25] can offer advantage over our solutions is when both the number of parallel reads and the array size are large. The largest advantage we can observe for 1000 simultaneous reads from an array of size 2^{16}, where our general construction is slower than the results from [25] by about a factor of 18 while our three-party construction is only slower by about 2.5 times.

7 Conclusions

In this work we study performance improvements to certain common building blocks in secure multi-party computation based on secret sharing. We present optimized protocols for reading or writing an element of an array at a private index and for integer multiplication. Most of our constructions are based on Shamir secret sharing with the exception of one array access construction. The latter uses 2-out-of-2 additive secret sharing in the three-party setting with honest majority, but offers superior performance compared to general constructions. To be compatible with computation based on Shamir secret sharing, we provide conversion procedures to convert between the two representations. We implement the presented constructions in the setting with three computational parties and show that they offer attractive performance in both LAN and WAN settings.

Acknowledgments

This work was supported in part by grant CNS-1705262 from the National Science Foundation, Google Faculty Research Award, and grant 2018R1A6A3A01011337 from the National Research Foundation of Korea. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies. We also acknowledge the NSF-sponsored Global Environment for Network Innovations (GENI) test bed, which allowed us to run WAN experiments.

References

1. Floram implementation. https://gitlab.com/neucrypt/floram/tree/floram-release.
2. The GNU multiple precision arithmetic library. https://gmplib.org/.
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 CCS*, pages 805–817, 2016.
4. F. Bayatbabolghani, M. Blanton, M. Aliasgari, and M. Goodrich. Secure fingerprint alignment and matching protocols. arXiv Report 1702.03379, 2017.
5. M. Blanton, A. Kang, and C. Yuan. Improved building blocks for secure multi-party computation based on secret sharing with honest majority. ePrint Archive Report 2019/718, 2019.
6. P. Bunn, J. Katz, E. Kushilevitz, and R. Ostrovsky. Efficient 3-party distributed ORAM. ePrint Archive Report 2018/706, 2018.
7. O. Catrina and S. De Hoogh. Improved primitives for secure multiparty integer computation. In *SCN*, pages 182–199, 2010.

8. R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *TCC*, pages 342–362, 2005.
9. I. Damgård and J. Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, pages 572–590, 2007.
10. J. Doerner and A. Shelat. Scaling ORAM for secure computation. In *ACM CCS*, pages 523–535, 2017.
11. S. Faber, S. Jarecki, S. Kentros, and B. Wei. Three-party ORAM for secure computation. In *ASIACRYPT*, pages 360–385, 2015.
12. C. W. Fletcher, M. Naveed, L. Ren, E. Shi, and E. Stefanov. Bucket ORAM: Single online roundtrip, constant bandwidth oblivious RAM. ePrint Archive Report 2015/1065, 2015.
13. 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.
14. O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *ACM STOC*, pages 182–194, 1987.
15. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.
16. 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 CCS*, pages 491–505, 2017.
17. M. Ito, A. Saito, and T. Nishizeki. Secret sharing schemes realizing general access structures. In *IEEE Globecom*, pages 99–102, 1987.
18. S. Jarecki and B. Wei. 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. In *ANCS*, pages 360–378, 2018.
19. S. Karan and J. Zola. Scalable exact parent sets identification in Bayesian networks learning with Apache Spark. In *IEEE HiPC*, pages 33–41, 2017.
20. M. Keller and P. Scholl. Efficient, oblivious data structures for MPC. In *ASIACRYPT*, pages 506–525, 2014.
21. M. Keller and A. Yanai. Efficient maliciously secure multiparty computation for RAM. In *EUROCRYPT*, pages 91–124, 2018.
22. L. Kerik, P. Laud, and J. Randmets. Optimizing MPC for robust and scalable integer and floating-point arithmetic. In *FC Workshops*, pages 271–287, 2016.
23. M. Koivisto. Parent assignment is hard for the MDL, AIC, and NML costs. In *International Conference on Computational Learning Theory*, pages 289–303, 2006.
24. P. Laud. A private lookup protocol with low online complexity for secure multiparty computation. In *ICICS*, pages 143–157, 2014.
25. P. Laud. Parallel oblivious array access for secure multiparty computation and privacy-preserving minimum spanning trees. *PoPETs*, 2015(2):188–205, 2015.
26. R. Ostrovsky. Efficient computation on oblivious RAMs. In *ACM STOC*, pages 514–523, 1990.
27. 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. ePrint Archive Report 2014/997, 2014.
28. G. Schwarz. *The Annals of Statistics*, 6:461–464, 1978.
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 CCS*, pages 247–258, 2013.

32. E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. arXiv Report 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 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 CCS*, pages 850–861, 2015.
35. X. Wang, Y. Huang, T-H. Chan, A. Shelat, and E. Shi. SCORAM: Oblivious RAM for secure computation. In *ACM CCS*, pages 191–202, 2014.
36. 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 S&P*, pages 218–234, 2016.
37. Y. Zhang, A. Steele, and M. Blanton. PICCO: A general-purpose compiler for private distributed computation. In *ACM CCS*, pages 813–826, 2013.