Binary Search in Secure Computation

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

Abstract

Binary search is one of the most popular algorithms in computer science. Realizing it in the context of secure multiparty computation which demands data-oblivious execution, however, is extremely non-trivial. It has been previously implemented only using oblivious RAM (ORAM) for secure computation and in this work we initiate the study of this topic using conventional secure computation techniques based on secret sharing. We develop a suite of protocols with different properties and of different structure for searching a private dataset of m elements by a private numeric key. Our protocols result in $O(m)$ and $O(\sqrt{m})$ communication using only standard and readily available operations based on secret sharing. We further extend our protocols to support write operations, namely, binary search that obliviously updates the selected element, and realize two variants: updating non-key fields and updating the key field. Our implementation results indicate that even after applying known and our own optimizations to the fastest ORAM constructions, our solutions are faster than optimized ORAM schemes for datasets of up to 2^{30} elements and by up to two orders of magnitude. We hope that this work will prompt further interest in seeking efficient realizations of this important problem.

1 Introduction

Secure multi-party computation is a mature research area which allows for any functionality to be evaluated on private data without having to disclose the data to the computation participants. While work in this direction originated in the 80s [56], the past two decades resulted in significant progress on two fronts: (i) secure computation techniques experienced notable performance advances and (ii) availability and expressiveness of existing implementations greatly improved. Today such techniques are performant enough to privately analyze large data sets with reasonable overhead. We also witness a growing number of compilers that automatically generate secure multi-party protocols for any desired functionality.

In addition to improving performance of underlying techniques and commonly used operations, there is a need to continue research on data-oblivious (i.e., data-independent) algorithms and data structures which permit higher-level functionalities to be executed within secure multi-party computation frameworks, ideally with asymptotic complexities of data-oblivious constructions being as close as possible to those of their non-oblivious counterparts. One particular direction that would benefit from additional exploration and research is algorithms for working with sorted data. Prior work [58] demonstrated that generically building data structures for working with sorted sets using common techniques such as linked lists or binary trees not only predictably significantly increases the asymptotic complexity of the resulting protocols, but performs even worse than using simple linear-scan-type techniques where the data is not sorted at all.

In this work, we look at the binary search problem in the context of secure multi-party computation and study algorithms suitable for its data-oblivious realization. We are aware of binary search discussed in prior literature only as an application of oblivious RAM (ORAM) and are not aware of any work that specifically studied this algorithm in the context of secure computation or, more generally, data-oblivious computation.[1] This work initiates such a study.

Binary search is one of fundamental algorithms in computer science with applications in all areas of computing. With the maturity of secure computation tools today that permit evaluation of any desired functionality and the greatly improved speed of secure computation techniques that process large data sets in record times, it is of great importance to provide efficient tools for common algorithms and data structures. Improving performance of secure binary search would have immediate impact on secure computation applications in use today, for example, for utilizing it in database queries over private data. Sharemind, a heavily developed secure multi-party computation compiler and toolkit [5], is advertised as a secure database and analytics system and offers a database module for manipulating private data stored in a database (see also [7, 14, 42, 46]). Naive linear search is not practical for typical database sizes and secure data-oblivious binary search would offer a superior solution. While data-oblivious and secure protocols have been developed for many other data structures and algorithms (e.g., [51, 54, 39, 37, 6, 12, 41, 11, 26]), a lack of sufficient exploration of the binary search problem can be explained by the difficulty of the problem and in particular the difficulty of developing a data-oblivious solution for it.

Problem statement. We are interested in the computation associated with a binary search when we search a private sorted dataset using a private key. In more detail, we are given a data set consisting of m private items $a_0, \ldots, a_{m-1}$, possibly consisting of multiple fields, one of which is the key field. The items are sorted by their keys. On input private search value b, we want to retrieve the element a_i whose key is equal to the searched key b or the smallest one that bigger than b. We also consider update operations where instead of retrieving an element, we update the chosen item.

Security requirements. The above formulation informs the security properties that our solution is to achieve. While the problem size and the computation to be performed are known, no information about the private inputs is to be revealed to any party including a party that controls a subset of the computation participants. In particular, the size of the data set m, the number of fields in each a_i, and size of each field is accessible during the computation, but no information about the a_is and b can be revealed. This, in particular, requires that the search procedure is data oblivious, which dictates that the executed instructions and accessed memory locations are independent of the private data.

Contributions. Our contributions are as follows:

- We design a suite of binary search protocols with different properties and structure in the multi-party setting based on secret sharing. In the process we also develop an optimization for use of generic ORAM schemes in binary search. These results are presented in sections 4 and 5.
- We combine and further improve our solutions to obtain hybrid schemes which outperform the individual constructions and lower binary search communication from $O(m)$ in our prior constructions to $O(\sqrt{m})$ for a dataset of size m.[2] Section 6 covers our hybrid protocols.
- We extend our binary search techniques to also support write operations. That is, instead of retrieving the element that the search selected, we update that element. We distinguish between updating non-key fields and the key field because the latter requires restructuring the dataset to preserve its sorted order. These techniques can be found in section 7.
- We implement several of our constructions in the semi-honest and malicious models and

[1] While [45] is said to provide oblivious binary search, it fails to meet the expectations; see section 2 for additional information.

[2] Sublinear communication is achievable when the underlying framework supports dot products with communication independent of the input size.

compare them to the state of the art on LAN and WAN. Our best result shows performance improvement over ORAM-based solutions (even with our ORAM optimizations) for dataset sizes up to 2^{27} on LAN and 2^{30} on WAN; the largest improvements are by 49× and 240× in the semi-honest and malicious models, respectively.

2 Related Work

All prior work that securely performs binary search in the context of secure multi-party computation we are aware of [50, 25, 57, 35, 43, 53] focuses on leveraging ORAM operations to hide search patterns, typically invoking a logarithmic number of ORAM accesses per binary search. With the ability to conceal access patterns, ORAM is often used as a general-purpose solution for constructing oblivious algorithms. The notion of ORAM was originally proposed [32, 34] for the client-server setting and allows a client to obliviously access data from an encrypted dataset held by a server. A popular type of ORAM is tree-based [49, 50, 35, 30], with constructions capable of achieving communicating $O(\log m)$ blocks of sufficient size per access with constant client storage [50]. Because the original ORAM formulation is not suitable for use in secure computation where no party is permitted to access the data in the clear, Gordon et al. [35] built on [49] to design ORAM for secure two-party computation with amortized sublinear complexity, where the parties jointly perform the work of the client without learning the access patterns. Additional results improving the efficiency of ORAM schemes for secure computation followed [25, 28, 38, 15].

Floram [25] is one of the state-of-the-art ORAM constructions for secure computation, which reported the best binary search performance among [57, 35, 43, 54], and thus we empirically compare performance of our constructions to that of Floram. Bunn et al.'s ORAM [15] has similar properties to Floram; i.e., it is based on a distributed point function [31], has the same round, communication, and local work complexities as Floram, but uses three parties instead of two. While its performance can be competitive to other ORAM schemes, it comes without an implementation and it is not possible for us to do meaningful comparisons of our solutions to that ORAM, especially because the underlying techniques differ (e.g., it uses oblivious transfer). We can only say that our constant round binary search construction is expected to outperform binary search based on Bunn et al.'s ORAM for small datasets. Its (linear per access) local work will also be the bottleneck when m is large and our optimization from section 4.3 reduces it from $O(m \log m)$ to $O(m)$ per search, which would match $O(\sqrt{m})$ communication and $O(m)$ local work of our best construction. In addition, 3PC ORAM [38] is a tree-based three-party ORAM based on two-party ORAM [52]. 3PC ORAM was reported to have a lower bandwidth than Floram when the dataset size is $< 2^{16}$ and thus is also competitive. Earlier, Gentry et al. [30] proposed an ORAM optimization for binary search for their tree-based construction, which enables binary search to have the asymptotic cost of a single (recursive) ORAM access. This optimization is also applicable to 3PC ORAM and therefore we apply the optimization to 3PC ORAM and include the optimized scheme in the discussion as well. The above constructions were developed for the semi-honest adversarial model, and the only ORAM secure against malicious adversaries we are aware of is by Keller and Yanai [40] which can be based on Circuit ORAM [52] or Path ORAM [50]. Although other recent ORAM constructions exist [36, 44, 19, 8, 16], they are designed for client-server environments and are not applicable to secure computation.

In addition to the above generic solutions, there have been efforts to improve efficiency of oblivious computation by designing oblivious data structures. Toft [51] proposed an oblivious secure priority queue with an amortized cost of $O(\log^2 n)$ per insertion and removal operation. Wang et al. [54] presented oblivious data structures that include priority-queues and stacks building upon techniques of [50] and [30]. Keller and Scholl [39] proposed oblivious data structures using two ORAM schemes [49, 50] and basic secure multi-party operations from [23]. The work realized oblivious arrays, dictionaries, and priority-queues in the multi-party setting using secret sharing. Shi [48] and Jafargholi et al. [37] proposed oblivious priority queues that achieve optimal $O(\log m)$ complexity. However, most of these data structures cannot be directly used for binary search operations in the context of secure multi-party computation. Among these oblivious data structures, the closest to our work is the oblivious dictionary construction for secure computation from [39]. It is capable of performing record search in a binary-search-like manner with communication complexity of $O(m)$, while the best of our protocols has sublinear communication complexity.

Lastly, a recent article by Rao et al. [45] claims to achieve oblivious binary search based on secret sharing. However, the protocol in [45] is not data-oblivious, simply invokes $O(\log m)$ comparisons the way a regular binary search would, and has to disclose the locations used in the comparisons. Furthermore, there are other significant discrepancies in that work. For example, the protocol's complexity is not analyzed in the text, but is said to be $O(m \log m)$ communication in $O(1)$ rounds in the abstract, which disagrees with the protocol itself.

3 Preliminaries

We consider binary search computation on private data, which can be a stand-alone operation or a part of larger computation. The parties carrying out the computation obtain a data set $[a_0], \ldots, [a_{m-1}]$, where notation $[x]$ means that the value of x is protected and not known to the participants in the clear. The items may consist of multiple fields, but there has to be a value used as the key, which we denote as $a_i.key$ for item a_i. The set is sorted in the increasing order by the key field of the elements, i.e., for each i $a_i.key < a_{i+1}.key$. The search uses keys $a_i.key$, but all fields associated with the chosen a_i are returned by the search. The dataset and other private inputs, e.g., the search key, can be contributed by one or more parties or be the result of preceding secure computation.

Our focus is secure multi-party computation based on linear secret sharing. That is, when we discuss non-ORAM-based approaches, our optimizations target, and performance improvements are measured, using secret sharing techniques, but the high-level ideas can likely apply to other types of techniques. With secret sharing, computation is carried out by n parties and there is a corruption threshold $t < n$, indicating that at most t out of n participants can be corrupt and conspiring (controlled by the same adversary) under which the techniques remain secure. Then the security expectations are that no information about any of the private data can be revealed to the corrupt participants, when the number of corrupt participants is below t. In particular, we use the standard simulation-based definition of security, which can be found, e.g., in [33], and requires that the view of the adversary controlling t parties during the protocol execution can be simulated using only the input and output of the corrupt parties and the simulated view is indistinguishable from an actual run of the protocol. Note that the functionality we target takes shares of the input and produces shares of the output and thus the parties performing the computation contribute no private input and learn no output. This means that they learn nothing as a result of protocol execution and the simulation proceeds without access to any data.

In light of the above, $[x]$ corresponds to a secret-shared value x. All inputs are provided in the secret-shared form and the outputs are also produced in the form of private shares. Also note that while each a_i may consist of multiple fields, we do not explicitly define them except the key field $a_i.key$. Thus, we use notation $[a_i]$ to refer to all fields of a_i, but it should be understood that different fields of each a_i might be secret shared separately. In particular, the computation uses shares of the keys $[a_i.key]$.

With linear secret sharing, a linear combination of shared values is performed locally, without communication, while multiplication is an elementary building block that requires a unit of communication. Because local operations are typically fast, relevant performance metrics are the total number of interactive operations and the number of rounds (i.e., sequential interactive operations).

Our constructions expect the availability of secure protocols for certain integer operations in the chosen secret sharing framework. For example, we rely on multiplication of private integers $[x]$ and $[y]$, less-than-or-equal comparison of integers $[x]$ and $[y]$ (denoted $\mathsf{LE}([x],[y])$), a dot product of two arrays of private integers, and generation of a (possibly pseudo-)random integer of a predefined bitlength k (denoted $\mathsf{RandInt}(k)$). Then any type of secret sharing for which these building blocks are available would be suitable for instantiating our constructions. Examples include Shamir secret sharing [47], which provides security in the semi-honest setting with honest majority, and SPDZ [24], which achieves security in the presence of malicious participants with dishonest majority. Both of these example carry out computation over a finite field, but our techniques would also apply to computation over ring $\mathbb{Z}_{2^k}$ for some k, e.g., as used in Sharemind [13] and SPD$\mathbb{Z}_{2^k}$ [20].

For concreteness, in our presentation we assume classical Shamir secret sharing. This will allow us to report concrete costs of the protocols which guide our optimizations and argue security for a fixed adversarial model (namely, in the presence of semi-honest participants with $t < n/2$). Computation based on secret sharing is information-theoretically secure in its nature, but for performance reasons building blocks can rely on computational security, therefore achieving computational indistinguishability.

Some of our constructions are based on oblivious RAM (ORAM), which comes with certain security guarantees. In particular, we require that an ORAM initialized over a private data set of size m allows us to read and write an element at a private location j without revealing any information about either the location or the element. We use ORAM for secure computation, where the computational parties perform the necessary computation on private data without access to the data or accessed location in the clear.

In the rest of this work, we use notation "$\leftarrow$" to indicate randomized functionalities (where randomization might only be due to share randomization and not randomization of the values to which the shares reconstruct) and use assignment notation "$=$" for deterministic computation.

4 Initial Constructions

In the current discussion, we focus on the read operation, i.e., we are interested in retrieving element a_i with the closest key to the searched value b. Discussion of the write operation is deferred to section 7 and specified as two variants where we update non-search fields based on the result of the search or update the search key itself. Note that the logic of the protocols for the read operation will apply to non-key field updates to a large extent, while updating the search key of an element requires different computation and restructuring of the dataset representation.

There can be two natural, and known, ways of performing a binary search operation in the context of secure computation: (i) comparing the search key to every element of the set to locate the desired element and (ii) using oblivious RAM (ORAM). The approach with a linear number of comparisons is expected to have favorable performance when the data set size m is small. ORAM, on the other hand, uses sub-linear (polylogarithmic) in m work to perform a read or write access at a hidden location and can result in solutions of sub-linear communication and computation. ORAM, however, usually requires large setup costs and has larger constants. Thus, it would be faster for data sets of large size and when the initialization cost could be amortized across many

Protocol 1 $[z] \leftarrow \mathsf{CompBS}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [b])$

```
 1: for $i = 0, \ldots, m-1$ in parallel do
 2:     $[c_i] \leftarrow \mathsf{LE}([b], [a_i.key])$;
 3: end for
 4: $[d_0] = [c_0]$;
 5: for $i = 1, \ldots, m-2$ in parallel do
 6:     $[d_i] \leftarrow [c_i] \cdot (1 - [c_{i-1}])$;
 7: end for
 8: $[d_{m-1}] = 1 - [c_{m-2}]$;
 9: $[z] \leftarrow \sum_{i=0}^{m-1} [a_i] \cdot [d_i]$
10: return $[z]$;
```

operations.

This section focuses on the above two ways of performing an oblivious search on sorted data, which we denote as initial constructions. While the structure of the computation in each of them is "common sense", we design the corresponding protocols to efficiently implement the functionalities in our setting using available building blocks. This section also describes our optimization to ORAM-based binary search, which is a new contribution of this work. All protocols and constructions described after this section are likewise new.

4.1 Linear Number of Comparisons

Recall that we are given a set $[a_0], \ldots, [a_{m-1}]$ sorted in the increasing order by their keys (we assume no duplicate key values) and would like to search for value $[b]$. In a linear-scan-type solution, we compare b to each $a_i.key$ by executing a protocol for $[b] \leq [a_i.key]$, which results in an array of m bits. We then search for the position j in the array where the bits switch from 0s to 1s and return the corresponding item a_j. A possible comparison-based binary search CompBS is given as Protocol 1, with less or equal comparisons LE being called on k-bit arguments. In this protocol, after performing the comparisons on line 2, we compute d_i as $c_i \wedge \neg c_{i-1}$. Note that $d_j = 1$ indicates that $b > a_{j-1}.key$ and $b \leq a_j.key$ and thus a_j is the desired element that we want to retrieve. Because there is only one index j such that $d_j = 1$, we retrieve the corresponding element a_j using computation $z = \bigvee_{i=0}^{m-1} (d_i \wedge a_i) = \sum_{i=0}^{m-1} a_i \cdot d_i$ on line 9.

CompBS is written to return a single element of the set even if the searched value is outside of the key range for all a_is. In particular, if $b \leq a_0.key$, a_0 is returned; if $a_{i-1}.key < b \leq a_i.key$ for some i, a_i is returned; and if $b > a_{m-1}.key$, a_{m-1} is returned (this means that a_{m-1} is returned if b is greater than the key of a_{m-2}). However, any desired variant of the algorithm can be easily supported.

Because round complexity is crucial to performance of secure protocols based on secret sharing, our protocol is written to run as many interactive operations in parallel as possible. The overall cost is dominated by m LE comparisons. Detailed costs of this and other protocols are provided in Table 1 assuming classical building blocks. In particular, the dot product costs 1 interactive operation with communication independent of the input size (e.g., implemented as a generalization of multiplication from [29]), and LE comparisons are instantiated as in [18, 17] on k-bit inputs $a_i.key$ and b, which cost $4k - 2$ interactive operations in 4 rounds, one of which can be precomputed.[3] Security analysis of this and other protocols is provided in Appendix C.

[3] Note that there are newer comparison protocols such as those that use edaBits for random bit generation [27] of noticeably lower communication, but higher round complexity. As will be seen later, our final solution uses a relatively small number of comparison operations and a lower-round version of the comparison operation is preferred.

Protocol 2 $[z] \leftarrow \mathsf{OramBS}(\mathsf{ORAM}([a_0], \ldots, [a_{m-1}]), [b])$

```
1:  $[c] \leftarrow \mathsf{LE}([b], [a_{\lfloor m/2 \rfloor}.key])$;
2:  $[d] = \lfloor m/2 \rfloor$;
3:  $[p] \leftarrow [c] \cdot ([a_{\lfloor m/2 \rfloor}] - [a_{m-1}]) + [a_{m-1}]$;
4:  for $i = 0, \ldots, \log(m) - 1$ do
5:      $[d] = [d] + (1 - 2[c]) \cdot \lfloor m/2^{i+2} \rfloor$;
6:      $[a] \leftarrow \mathsf{ORAMR}([d])$;
7:      $[c] \leftarrow \mathsf{LE}([b], [a.key])$;
8:      $[p] \leftarrow [c] \cdot ([a] - [p]) + [p]$;
9:  end for
10: return $[z] = [p]$;
```

4.2 ORAM-Based Search

We next discuss binary search that uses ORAM. Unless noted otherwise, we let the array size be $m = 2^k - 1$ for some integer k, i.e., the set can be represented as a perfect binary tree. Also, we use notation $\log(\cdot)$ to denote $\lceil \log_2(\cdot) \rceil$.

The algorithm for performing an ORAM-based binary search follows the same structure as that of a conventional binary search with the difference that the array access to a known index is replaced with ORAM access to a secret index. Due to the access pattern hiding properties of ORAM, no information about the decision (i.e., go left or right) will be leaked after each round of comparisons. Given an ORAM set up over sorted dataset $[a_0], \ldots, [a_{m-1}]$, we denote an ORAM read and write access to a logical address $[d]$ as $\mathsf{ORAMR}([d])$ and $\mathsf{ORAMW}([d])$, respectively. Note that for an entry $[a_i]$, the index i represents its logical address rather than its physical address in the ORAM. Furthermore, typical realizations of ORAM hide the type of operation (read or write) on each access, but in our context the operation type is public knowledge. Thus, we reveal the type of the operation in the ORAM notation (in the case that knowledge of the operation can permit performance optimizations).

The ORAM-based binary search, OramBS, is given as Protocol 2. It starts by comparing the search key to the element at position $\lfloor m/2 \rfloor$. Because the location of the first access is fixed and known, we store $[a_{\lfloor m/2 \rfloor}]$ outside of ORAM to save a costly ORAM access. The value of d stores the (private) location which should be read in each round, and after the total of j comparisons moves $\lfloor m/2^j \rfloor$ positions left or right depending on the last comparison result c.

In the last iteration, the search key b is compared to the element at position d to determine whether to return a_d or a_{d+1}. While the element at position d was just retrieved, retrieving a_{d+1} requires another ORAM access. We, however, observe that this extra ORAM read is not needed because a_{d+1} is guaranteed to be retrieved in an earlier ORAM access. That is, because we know $b > a_d$ based on the last comparison and $b \leq a_{d+1}$ based on correctness of the search, the element at position $d+1$ has to reside on the path from the root of the binary search tree $a_{\lfloor m/2 \rfloor}$ to a_d (except when $d = m - 1$ is the last position). This means that a_{d+1} has been previously read and all we need is to maintain a copy of it instead of invoking another ORAM access. This is what OramBS does: variable p stores the last element on the path from the root when the search proceeded left, i.e., the searched value was smaller than the element on the path (p is conditionally updated on line 8 during each loop iteration). This guarantees that p will be equal to a_{d+1} if the comparison in the last round results in incrementing d. In the event that the path never goes left, p is initialized to a_{m-1} on line 3.

The computation on lines 3 and 8 uses conditional statements of the type `if` $([c])$ `then` $[p] = [x]$ `else` $[p] = [y]$ expressed as $[p] = [c] \wedge [x] \vee [\neg c] \wedge [y] = [c] \cdot [x] + (1 - [c]) \cdot [y]$. We rewrite them as $[p] = [c]([x] - [y]) + [y]$ to save one multiplication each. Furthermore, because updating the value of p can be done together with the next interactive operation, updating p contributes to the round complexity only in the last round. Thus, the cost of the ORAM-based binary search is heavily dominated by $\log m$ ORAM accesses and $\log m$ LE comparisons. Security is discussed in Appendix C.

Gentry et al. [30] suggested a formula for performing greater-than comparisons, $x > y$, on bit-decomposed k-bit values written as $x = x_{k-1} \ldots x_0$ and $y = y_{k-1} \ldots y_0$. The computation is

$$g(x_{k-1} \ldots x_0, y_{k-1} \ldots y_0) = (x_{k-1} - y_{k-1})x_{k-1} + (x_{k-1} - y_{k-1} + 1)g(x_{k-2} \ldots x_0, y_{k-2} \ldots y_0). \quad (1)$$

Note that a straightforward implementation of this function would result in k rounds of computation, which becomes prohibitive in an application like binary search where comparisons are executed sequentially. We notice that the formula can be rewritten in a different form to support constant-round evaluation as $g(x, y) = \sum_{i=0}^{k-1} x_i z_i \prod_{j=i+1}^{k-1} w_j$, where $z_i = x_i - y_i$ and $w_i = z_i + 1$. However, note that the original formula was written for computation in $\mathbb{Z}_2$ and does not produce correct output when the computation is over a larger field. Furthermore, if we rewrite the formula to be correct, it can no longer be represented in a compact form suitable for constant-round evaluation. This means that we do not further consider it as competitive for our application compared to other options (such as LE from [18] mentioned above, which also avoids bit decomposition).

4.3 ORAM-based Optimizations

Gentry et al. [30] suggested an optimization to tree-based ORAM constructions that allows ORAM-based binary search to have asymptotically the same cost as that of a recursive tree-based ORAM access. Briefly, the optimization eliminates the need for a recursive position map to translate a logical address to its physical address in ORAM by adding pointers to each block that store the locations of the next access in the ORAM. As a result, the total cost of a binary search is equal to that of a single recursive ORAM access instead of a logarithmic number of full ORAM accesses in general tree-based ORAM protocols. The optimized solution was not empirically evaluated in [30] and therefore it is difficult to tell how its performance might compare to other constructions in the literature. We, however, note that this optimization is expected to apply to other tree-based ORAM constructions which recursively outsource the position map. In particular, we determined that one of the latest efficient ORAMs, 3PC ORAM [38], is tree-based and uses recursive position maps for each ORAM access. Thus, the optimization is applicable to 3PC ORAM and we use the resulting optimized 3PC ORAM in the performance evaluation of our constructions.

We also note that it is possible to optimize performance of ORAM-based binary search regardless of the internal structure of the underlying ORAM construction. The general idea behind our optimization is that we partition the original dataset $a_0, \ldots, a_{m-1}$ into $\log m$ layers of exponentially increasing size. The ith layer will correspond to the elements which can be accessed during the ith step of the binary search computation. That is, for m of the form $2^k - 1$, layer 0 contains only a single element $a_{\lfloor m/2 \rfloor}$, layer 1 contains two elements $a_{\lfloor m/4 \rfloor}$ and $a_{\lfloor 3m/4 \rfloor}$, etc., and layer $\log(m) - 1$ contains $(m+1)/2$ elements at even positions i. With this division, we can set up a separate ORAM for each layer, significantly decreasing the work associated with the first accesses to ORAM and thus leading to practical improvements in the performance of binary search.

This optimization can lead to varying impacts on the asymptotic complexity of the resulting binary search for different ORAM constructions. In particular, for constructions with polylogarithmic (in dataset size) complexities of ORAM access, there might be no asymptotic gain in applying this optimization. For example, if ORAM access costs $O(\log^2 N)$ for an ORAM set up for N elements, then making $\log m$ accesses of cost $O(\log^2 m)$ and making $\log m$ accesses with exponentially increasing sizes from $O(1)$ to $O(m)$ will both result in $O(\log^3 m)$ overall cost. On the other hand, ORAM constructions of larger asymptotic complexities can see pronounced improvements in the asymptotic cost. For example, we can look at Floram [25], which is one of the fastest two-party ORAM constructions. Its asymptotic complexity per access is $O(\sqrt{N})$ communication and $O(N)$ local work. After applying our optimization, the total work associated with binary search decreases from $O(m \log m)$ using full-size ORAM for each access to only $O(m)$. We empirically evaluate the associated performance gain and report it in section 8.

5 Hierarchical Constructions

In this section we describe two new approaches to binary search, both of which are hierarchical and use only a logarithmic number of comparisons. The high-level structure is similar to the one used with ORAM-based search, but we replace the mechanism for protecting true accesses at each iteration. Because of the cost of an ORAM access, it can be beneficial to replace it with alternative computation, including solutions of higher asymptotic complexity linear in the size of a layer. Our findings are in line with those in [10] that demonstrated that linear-time constructions for accessing an element at a private location outperform ORAM performance in practice unless the size of the array becomes very large.

In this section, we offer two solutions: The first one is based on array rotation to hide accessed locations and the second generates tags for all elements in a layer to mark the true path. As with the previous, ORAM-based, construction, we divide the array into layers and process one layer at a time. The conceptual difference is how true accesses are protected for each layer.

5.1 Rotation-based Construction

Our first construction rotates all elements in a layer to hide the true access pattern and is based on the following high-level idea: after comparing an element to the search key, we privately determine whether we are jumping left or right and compute the location d to read in the next layer. We next rotate the next layer of size 2^i by a random private amount $r \in [0, 2^i - 1]$ and disclose the protected location $(d + r) \bmod 2^i$. Once we know the location, we can retrieve the desired element and perform the next comparison without knowing what the true index d was. Note that the rotation operation should be oblivious and performed once per search for each layer to ensure that no information about the accessed locations is revealed.

Our algorithm for rotation-based binary search, RotBS, is given as Protocol 3. As before, we are given an ordered set $[a_0], \ldots, [a_{m-1}]$, where m is of the form $2^k - 1$, and divide it into layers as in the optimized ORAM-based solution, i.e., with layer 0 consisting of only one element $a_{\lfloor m/2 \rfloor}$ and layer $\log(m) - 1$ containing all elements at even indices. Let $[\ell^{(i)}]$ denote the elements stored at layer i and $[\ell^{(i)}_j]$ denote the jth element stored at layer i. We first randomly generate a secret random offset $r^{(i)} \in [0, 2^i)$ for each layer $i > 0$ and rotate all layers by those offsets in parallel to minimize round complexity (lines 2–4). This is illustrated in Figure 1. The rotation algorithm Rotate, which also chooses the amount of rotation, is described afterwards.

After rotating all layers, we proceed with comparisons and first retrieve the (only) element from the top layer $[\ell^{(0)}]$ and compare it with the target value b (line 5). We continue by jumping left or right as before and privately computing the index d to access next using local computation (line 9). Note that unlike OramBS that computed this index d as an index in the entire array, this time we compute it as a position in a layer. This means that we start from two possible positions in layer 1 and update d in the next layer as $2[d] + [c]$, where $[c]$ is the result of the current comparison.

Protocol 3 $[z] \leftarrow \mathsf{RotBS}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [b])$

```
 1: let $[\ell^{(i)}] = \langle [a_{\lfloor m/2^{i+1} \rfloor}], [a_{\lfloor 3m/2^{i+1} \rfloor}], \ldots, [a_{\lfloor (2^{i+1}-1)m/2^{i+1} \rfloor}] \rangle$ for $i = 1, \ldots, \log(m) - 1$;
 2: for $i = 1, \ldots, \log(m) - 1$ in parallel do
 3:     $\langle [\hat{\ell}^{(i)}], [r^{(i)}] \rangle \leftarrow \mathsf{Rotate}([\ell^{(i)}])$;
 4: end for
 5: $[c] \leftarrow \mathsf{LE}([b], [a_{\lfloor m/2 \rfloor}.key])$;
 6: $[p] \leftarrow [c] \cdot ([a_{\lfloor m/2 \rfloor}] - [a_{m-1}]) + [a_{m-1}]$;
 7: $[d] = 0$;
 8: for $i = 1, \ldots, \log(m) - 1$ do
 9:     $[d] = 2[d] + [c]$;
10:     $[w] \leftarrow \mathsf{RandInt}(\kappa + 1)$;
11:     $s' = \mathsf{Open}([d] + [r^{(i)}] + 2^i[w])$;
12:     $s = s' \bmod 2^i$;
13:     $[a] = [\hat{\ell}^{(i)}_s]$;
14:     $[c] \leftarrow \mathsf{LE}([b], [a.key])$;
15:     $[p] \leftarrow [c] \cdot ([a] + [p]) + [p]$;
16: end for
17: return $[z] = [p]$;
```

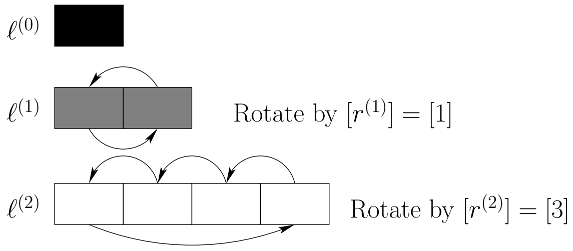

Figure 1: Illustration of the rotation-based construction.

To safely disclose the desired (protected) position in the rotated array, we must open $(d + r^{(i)}) \bmod 2^i$ instead of simply $d + r^{(i)}$. While the latter is a location operation, computing the remainder modulo a power of 2 in this framework is a rather expensive operation, with the same number of rounds as in comparisons and a number of interactive operations linear in the size of the modulus, i.e., in i (see $\mathsf{Mod2m}$ in [18]). Fortunately, we were able to get around this cost and safely disclose $(d + r^{(i)}) \bmod 2^i$ using only local computation (prior to the opening). In particular, we mask the $(i+1)$st bit of the sum $d + r^{(i)}$, i.e., the carry bit, by a randomly chosen integer, which allows us to safely open the result. To achieve this, we rely on statistical secrecy and choose an integer of κ bits longer than the value we are protecting, where κ is a statistical security parameter (line 10). The corresponding protocol is called $\mathsf{RandInt}$ and takes an argument that specifies the bitlength of an integer to generate. It can be realized non-interactively as described in [18]. The reader may notice that this approach requires that the field can represent integers κ bits longer than the bitlength of the key values. This, however, is already required by the LE protocol.

Because each layer has been rotated by a one-time random offset, we can safely proceed by revealing the value of $(d + r^{(i)}) \bmod 2^i$ in each layer, retrieving that element in the rotated layer, and performing the comparison until we reach the last layer. In addition to maintaining the current element used in the comparison, we also keep track of the last retrieved element p which was $\leq$ the target b, in the same way as in OramBS. It will be retrieved in the last round if the returned element should not be the one used in the last comparison. As before, the default value of p is

Protocol 4 $(\langle[\hat{a}_0],\ldots,[\hat{a}_{2^u-1}]\rangle,[r]) \leftarrow \mathsf{Rotate}([a_0],\ldots,[a_{2^u-1}])$

```
1:  for $i = 0,\ldots,u-1$ in parallel do
2:      $[r_i] \leftarrow \mathsf{RandBit}()$;
3:  end for
4:  let $[a_j^{(0)}] = [a_j]$ for $j \in [0, 2^u - 1]$;
5:  for $i = 0,\ldots,u-1$ do
6:      for $j = 0,\ldots,2^u-1$ in parallel do
7:          $[a_j^{(i+1)}] \leftarrow [a_j^{(i)}] - ([a_j^{(i)}] - [a^{(i)}_{(j-2^i) \bmod 2^u}]) \cdot [r_i]$;
8:      end for
9:  end for
10: $[r] = \sum_{i=0}^{u-1} 2^i [r_i]$;
11: let $[\hat{a}_j] = [a_j^{(u)}]$ for $j \in [0, 2^u - 1]$;
12: return $(\langle[\hat{a}_0],\ldots,[\hat{a}_{2^u-1}]\rangle,[r])$;
```

a_{m-1}.

The cost of this protocol consists of $\log(m)$ comparisons and other cheaper operations (i.e., multiplications and openings) and is dominated by the cost of rotating all layers. Because some interactive operations could be combined and executed in the same round, the overall round complexity is that of rotation (see below) plus $4\log m$. Security is shown in Appendix C.

We next describe our Rotate protocol. It takes as input an array $[a_0],\ldots,[a_{2^u-1}]$ of size 2^u, generates a random u-bit integer r, circularly rotates the elements of the array by r positions, and outputs r together with the rotated array. Our solution is conceptually simple and is given as Protocol 4. As the first step, we generate u random bits using protocol RandBit (line 2) and use them to assemble u-bit offset r (line 10). RandBit can be implemented using 1 interactive operation [18]. Despite having a higher cost than RandInt(u), generating r from random bits is important for two reasons: (i) we use the bits in the computation that follows and (ii) it allows us to generate a value in the exact range $[0, 2^u - 1]$, while RandInt generates values slightly larger than of the specified bitlength.

The next step is to rotate the array elements by the generated offset, which we do in u iterations: in iteration i, the elements are shifted by $2^i \cdot r_i$ positions right, i.e., we conditionally perform the shift if the ith bit of r is set. After u iterations, the array elements are shifted by the value of r, as desired. In more detail, on line 7 we either keep the current element a_j at position j or replace it with the element at position $a_{j-2^i \bmod 2^u}$ based on the value of r_i.

When $u = \log(m)$, Rotate requires $m \log m$ interactive operations in $\log(m) + 1$ rounds. Recall that we use it in RotBS and execute rotations for all layers in parallel in the beginning. However, rotation of only the smallest layer 1 needs to finish prior to its use in the first loop iteration on line 13. This means layer rotations do not increase the round complexity of RotBS. As before, we summarize performance of our binary search constructions in Table 1.

On using ring $\mathbb{Z}_{2^k}$. Before we conclude, we comment on executing our protocols over ring $\mathbb{Z}_{2^k}$ instead of a finite field. All protocols described so far except RotBS work unmodified when instantiated with building blocks over ring $\mathbb{Z}_{2^k}$. The difference is that RotBS protects a value, opens it, and uses i least significant bits in further computation (i.e., lines 10–12 of RotBS). We note that this operation can become even easier over ring $\mathbb{Z}_{2^k}$ because all values are automatically reduced modulo a power of 2. In particular, instead of prepending a large random value to statistically hide the overflow from i bits, we can directly open the sum $[r^{(i)}] + [d]$ and use the result as s, as long as all shares are reduced modulo 2^i prior to the opening.

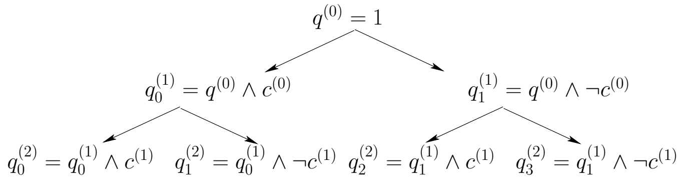
$q^{(0)} = 1$
$q_0^{(1)} = q^{(0)} \wedge c^{(0)}$
$q_1^{(1)} = q^{(0)} \wedge \neg c^{(0)}$
$q_0^{(2)} = q_0^{(1)} \wedge c^{(1)}$
$q_1^{(2)} = q_0^{(1)} \wedge \neg c^{(1)}$
$q_2^{(2)} = q_1^{(1)} \wedge c^{(1)}$
$q_3^{(2)} = q_1^{(1)} \wedge \neg c^{(1)}$

Figure 2: Illustration of the tag-based construction.

Protocol 5 $[z] \leftarrow \mathsf{TagBS}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [b])$

```
1: let $[\ell^{(i)}] = \langle [a_{\lfloor m/2^{i+1} \rfloor - 1}], [a_{\lfloor 3m/2^{i+1} \rfloor - 1}], \ldots, [a_{\lfloor (2^{i+1}-1)m/2^{i+1} \rfloor - 1}] \rangle$ for $i = 1, \ldots, \log(m) - 1$;
2: $[q^{(0)}] = \langle 1 \rangle$;
3: $[c] \leftarrow \mathsf{LE}([b], [a_{\lfloor m/2 \rfloor - 1}.key])$;
4: $[p] = [c] \cdot ([a_{\lfloor m/2 \rfloor - 1}] - [a_{m-1}]) + [a_{m-1}]$;
5: for $i = 1, \ldots, \log(m) - 1$ do
6:    for $j = 0, \ldots, 2^i - 1$ in parallel do
7:       if $j$ mod $2 = 0$ then
8:          $[q_j^{(i)}] = [q_{\lfloor j/2 \rfloor}^{(i-1)}] \cdot [c]$;
9:       else
10:         $[q_j^{(i)}] = [q_{\lfloor j/2 \rfloor}^{(i-1)}] \cdot (1 - [c])$;
11:      end if
12:   end for
13:   $[a] = \sum_{j=0}^{2^i-1} [q_j^{(i)}] \cdot [\ell_j^{(i)}]$;
14:   $[c] \leftarrow \mathsf{LE}([b], [a.key])$;
15:   $[p] = [c] \cdot ([a] - [p]) + [p]$;
16: end for
17: return $[z] = [p]$;
```

5.2 Tag-based Construction

Our second hierarchical solution utilizes a different mechanism for retrieving an element at a private location from each layer. While a generic protocol for reading an element at a private location could be used (e.g., multiplexor-based or as described in [10]), in the context of binary search we notice that the location read at layer $i + 1$ is highly correlated to the location previously read at layer i. This observation allows us to generate binary tags for each layer using tags of the layer before, where in each layer the tag of a single position is set to 1 and the tags at all other positions are set to 0. This representation consequently permits us to retrieve the element at the marked position using efficient dot product computation.

Our tag-based construction, TagBS, is given as Protocol 5. As before, let $[\ell^{(i)}]$ denote the elements at layer i and let us use similar notation $[q^{(i)}]$ to denote binary tags for layer i.

Initially, layer 0 has a single element and its tag is set to 1. The next layer has two elements, one of which will be set to 1 based on the result of the first comparison and the other will be set to 0. We continue computing tags for the current layer from the tags of the previous layer as follows: if the "parent" tag at position j is 0, both "children" tags at positions $2j$ and $2j + 1$ will be 0. If the "parent" tag is 1, one of the "children" tags will be 0 and one will be 1 based on the result of the current comparison c. This process is illustrated in Figure 2. Then to retrieve the marked element at the current layer, we compute the dot product of the elements and their tags in the current layer (line 13). Also, as before, we maintain another element in variable p, which will be used at the end if the result of the last comparison is false.

Protocol	Rounds		Interactive operations	
	Read	Write	Read	Write
Comparison-based CompBS	5	$+0$	$(4k-1)m-1$	$+(4k+m-3)$
ORAM-based OramBS	$(\log m)(3+\mathsf{ORAMRead})$	$+\mathsf{ORAMWrite}$	N/A	N/A
Rotation-based RotBS	$4\log m$	$+1$	$4k\log(m) + m\log(m) - 1$	$+(\log^2 m - \log m)/2$
Tag-based TagBS	$5\log m$	$+1$	$4k\log(m)+m-2$	$+2m-\log m$
Hybrid LayHBS, $\tilde{\imath} < \log(m) - 1$	$5(\log(m)-\tilde{\imath})+4$	$+1$	$(4k-3)2^{\tilde{\imath}}+m+2$	$+2m-\log(m)+\tilde{\imath}$
Hybrid SubHBS, $\alpha \cdot \beta = m$	$\mathsf{MBS}(\alpha)+\mathsf{BS}(\beta)+1$	varies	$\mathsf{MBS}(\alpha)+\mathsf{BS}(\beta)+\beta$	varies

Table 1: Performance of binary search using field-based building blocks in the semi-honest model with honest majority.

The overhead of this protocol is given in Table 1. Note that TagBS uses fewer interactive operations compared to rotation-based RotBS, but the latter has fewer rounds. Thus, we expect that TagBS will be a faster choice in typical circumstances, but RotBS can be beneficial for high-latency connections. In addition, comparison-based CompBS has constant round complexity, but is expected to be slower for larger datasets due to its communication volume.

6 Hybrid Constructions

In this section we discuss how combining multiple constructions in a single solution can be used to further improve performance of binary search. Section 6.1 discusses a solution in which portions of the binary search tree are processed using different algorithms, and section 6.2 presents a solution where previously developed constructions are applied only to a subset of the tree nodes resulting in sublinear communication cost. In particular, while all of our constructions including the one in section 6.1 require $O(m)$ communication, the solution of section 6.2 lowers communication to $O(\sqrt{m})$ by relying on an efficient dot product protocol.

6.1 Composition of Layers

The binary search constructions described so far have their own advantages. For example, comparison-based CompBS has constant round complexity, which is the lowest across all protocols. Its communication cost, however, is rather high and is linear in $m \cdot k$. These properties make it a good choice for datasets of small size, but performance is expected to deteriorate as the dataset size grows. In contrast, tag-based TagBS has the lowest communication complexity as the dataset size m increases, but the largest round complexity. Because it invokes only a logarithmic number of comparisons, it is expected to outperform other constructions for larger values of m, but perform relatively worse for very small values of m. In particular, the first three rounds of comparisons in that construction process only 7 elements, but use 15 rounds. This can be contrasted with the total of 5 rounds in CompBS. Coupled with the fact that communication latency is the major overhead when the dataset is small, TagBS is sub-optimal during processing of the top layers.

The hybrid construction we propose here combines flat and hierarchical structures to take advantage of the benefits of both of them. More precisely, we replace the top layers of the hierarchy with a flat structure and design a transition to feed the results of evaluating the flat structure to the next layer in the hierarchy. Although the idea is straightforward, its realization requires careful design because the constructions have different interfaces and rely on different intermediate results. We illustrate a transition mechanism on the example of combining CompBS and TagBS.

Recall that CompBS computes an array of bits $[c_0], \ldots, [c_{m-1}]$, where the bits in the beginning

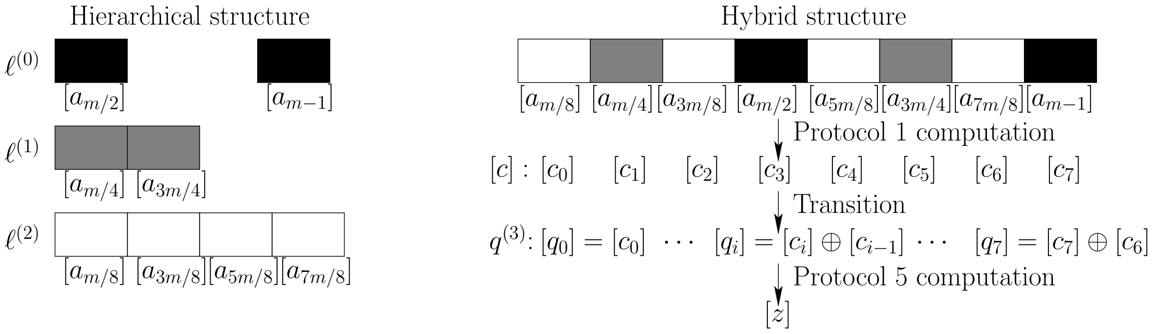
Hierarchical structure
$\ell^{(0)}$
$[a_{m/2}]$
$[a_{m-1}]$
$\ell^{(1)}$
$[a_{m/4}]$
$[a_{3m/4}]$
$\ell^{(2)}$
$[a_{m/8}]$ $[a_{3m/8}]$ $[a_{5m/8}]$ $[a_{7m/8}]$
Hybrid structure
$[a_{m/8}]$ $[a_{m/4}]$ $[a_{3m/8}]$ $[a_{m/2}]$ $[a_{5m/8}]$ $[a_{3m/4}]$ $[a_{7m/8}]$ $[a_{m-1}]$
Protocol 1 computation
$[c]$: $[c_0]$ $[c_1]$ $[c_2]$ $[c_3]$ $[c_4]$ $[c_5]$ $[c_6]$ $[c_7]$
Transition
$q^{(3)}$: $[q_0] = [c_0]$ $\cdots$ $[q_i] = [c_i] \oplus [c_{i-1}]$ $\cdots$ $[q_7] = [c_7] \oplus [c_6]$
Protocol 5 computation
$[z]$

Figure 3: Illustration of the layer-based hybrid construction.

of the array are 0s and switch to 1s at the location of the searched element $[b]$. The array is consequently used to compute another bit array $[d_0], \ldots, [d_{m-1}]$, in which all elements are 0 except for the location of the switch, and the non-zero bit is used to retrieve the desired element of the array $[a_0], \ldots, [a_{m-1}]$. TagBS, on the other hand, proceeds in a hierarchical manner and prior to moving to layer i expects state $[c]$, $[p]$ that indicates the position of the desired element in the already processed layers and computed tags $[q_j^{(i-1)}]$ for layer $i-1$. Our transition computes the value of $[p]$ and tags $[q_j^{(i)}]$ from the bit arrays computed by CompBS and no explicit $[c]$, as maintained by TagBS, is used in the process.

Suppose we would like to process the first $\tilde{\imath}$ layers, or equivalently $\tilde{m} = 2^{\tilde{\imath}}$ elements, using the flat structure. We execute the main portion of CompBS on elements associated with layers 0 through $\tilde{\imath}-1$, use transition to generate $[p]$ and the tags at layer $\tilde{\imath}$, and continue with the remaining computation as in TagBS. Figure 3 illustrates the process for $\tilde{\imath} = 3$. Note that the very last element of the original dataset, a_{m-1}, is appended to the elements that CompBS processes and is treated as an element of layer 0.

We determined that the relationship between $c_0, \ldots, c_{\tilde{m}-1}$ computed by CompBS and $q_0^{(\tilde{\imath})}, \ldots, q_{\tilde{m}-1}^{(\tilde{\imath})}$ needed in TagBS is rather simple. In particular, we have:

$$q_j^{(\tilde{\imath})} = \begin{cases} c_0 & \text{if } j = 0 \\ c_{j-1} \oplus c_j & \text{if } 0 < j \leq \tilde{m} - 1 \end{cases} \tag{2}$$

Somewhat surprisingly, each $q_j^{(\tilde{\imath})}$ is computed in the same way, while we expect differences in the computation of even and odd elements because only half of the values are associated with layer $\tilde{\imath}-1$. XOR of each $[c_{j-1}]$ and $[c_j]$ is easily computed as $[c_{j-1}] + [c_j] - 2[c_j] \cdot [c_{j-1}]$. This is equivalent to $2[d_j] - [c_j] + [c_{j-1}]$, where the $[d_j]$s are as computed by CompBS. Because availability of $[d_j]$s makes the computation of $[q_j^{(\tilde{\imath})}]$s local, a notable implication for us is that combining the two constructions reduces the overall cost below that of running CombBS and TagBS on the respective portions of the data.

It is also necessary to compute the appropriate value of $[p]$ during the transition. This computation comes first (i.e., to finish processing layer $\tilde{\imath} - 1$), and $[p]$ is computed in the same way as $[z]$ in CompBS, i.e., the first element which was determined to be $\leq b$ so far will be used as the next larger element if the remaining search returns that b is greater than all other elements. For completeness, we present this hybrid construction, called LayHBS, in Appendix A.

In practice, the best choice of $\tilde{\imath}$ depends on the setup. When the network latency between the computational parties is small, the round complexity may have less impact on performance and therefore a lower $\tilde{\imath}$ is preferred. In contrast, if the network latency is high, a larger $\tilde{\imath}$ reduces

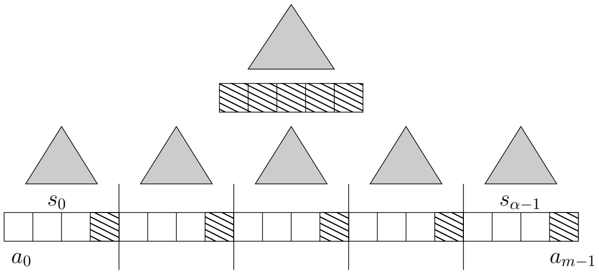

Figure 4: Illustration of the subtree-based hybrid construction. Shaded array elements are used to form the top tree of size α. Subtrees s_0 through $s_{\alpha-1}$ have size β.

Protocol 6 $[z] \leftarrow \mathsf{SubHBS}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [b], \alpha)$

```
let $\beta = m/\alpha$;
let $[s^{(i)}] = \langle [a_{i \cdot \beta}], [a_{i \cdot \beta + 1}], \ldots, [a_{i \cdot \beta + \beta - 1}] \rangle$ for $i = 0, \ldots, \alpha - 1$;
$\langle [d_0], \ldots, [d_{\alpha-1}] \rangle \leftarrow \mathsf{MBS}(\langle [s^{(0)}_{\beta-1}], [s^{(1)}_{\beta-1}], \ldots, [s^{(\alpha-1)}_{\beta-1}] \rangle, [b])$;
for $i = 0, \ldots, \beta - 1$ in parallel do
    $[u_i] \leftarrow \sum_{j=0}^{\alpha-1} [s^{(j)}_i] \cdot [d_j]$;
end for
$[z] \leftarrow \mathsf{BS}(\langle [u_0], \ldots, [u_{\beta-1}] \rangle, [b])$;
return $[z]$;
```

the round complexity and therefore could be a better choice. We provide additional comments in section 8. The exact cost is listed in Table 1 and this time is a function of $\tilde{\imath}$.

6.2 Composition of Subtrees

All solutions presented so far, except OramBS, have communication linear in the size of the dataset m. In this section we show how performance of previously presented constructions can be further improved to $O(\sqrt{m})$ communication using only standard building blocks.

The high-level idea behind this solution is as follows. Recall that, on input m elements, CompBS works by generating a bit array $[d_0], \ldots, [d_{m-1}]$ with a single element set to 1 and that array is used to retrieve the searched element using a dot product (which costs 1 interactive operation). Now suppose that instead of retrieving a single element at the end of the computation, we use the computed bits to retrieve a desired subset of the elements, or a subtree if the elements are organized in a hierarchy. The high-level structure of this approach is illustrated in Figure 4.

We organize the dataset in a hierarchy and run a modified binary search (which produces a bit array) on the top portion of the tree of size α. We consequently use dot products to obliviously localize the search to the relevant subtree of size $\beta = m/\alpha$, call binary search on that tree, and use its output as the result of the search. In other words, the top-level search allows us to determine in what portion of the dataset the searched element falls and the second low-level search determines the exact position and returns the desired element.

The solution, SubHBS, is formalized as Protocol 6. We use a modified version of a binary search protocol that produces a unity bit array, denoted as MBS. The second call to binary search, on the other hand, uses the conventional interface. We note that communication savings are possible because of the use of dot product protocols, which on input of two vectors of arbitrary size require only one interactive operation. This means that when we set $\alpha = \beta = O(\sqrt{m})$, communication complexity of this solution reduces from $O(m)$ to $O(\sqrt{m})$. We note that local computation remains $O(m)$, but this component of interactive protocols is much easier to speed up than communication,

Protocol 7 $\langle [d_0], \ldots, [d_{m-1}]\rangle \leftarrow \mathsf{TagMBS}(\langle [a_0], \ldots, [a_{m-1}]\rangle, [b])$

```
let $[\ell^{(i)}] = \langle [a_{\lfloor m/2^{i+1}\rfloor}], [a_{\lfloor 3m/2^{i+1}\rfloor}], \ldots, [a_{\lfloor (2^{i+1}-1)m/2^{i+1}\rfloor}]\rangle$ for $i = 0, \ldots, \log(m) - 1$;
$[q_0^{(0)}] = 1$;
for $i = 0, \ldots, \log(m) - 1$ do
    $[q^{(i)}] = \langle [q_0^{(i)}], \ldots, [q_{2^i-1}^{(i)}]\rangle$;
    $[a] = \sum_{j=0}^{2^i-1} [q_j^{(i)}] \cdot [\ell_j^{(i)}]$;
    $[c] \leftarrow \mathsf{LE}([b], [a.key])$;
    for $j = 0, \ldots, 2^i - 1$ in parallel do
        if $j \bmod 2 = 0$ then
            $[q_j^{(i+1)}] = [q_{\lfloor j/2\rfloor}^{(i)}] \cdot [c]$;
        else
            $[q_j^{(i+1)}] = [q_{\lfloor j/2\rfloor}^{(i)}] \cdot (1 - [c])$;
        end if
    end for
end for
return $\langle [d_0], \ldots, [d_{m-1}]\rangle = \langle [q_0^{(\log m)}], \ldots, [q_{m-1}^{(\log m)}]\rangle$;
```

e.g., by employing more powerful hardware and/or using multi-threading.

It should be clear that the call to binary search BS on line 7 of SubHBS can be instantiated with any binary search construction described so far. This includes hybrid LayHBS and even SubHBS itself, which in that case would be called recursively. Similarly, the first call to binary search on line 3 can be instantiated with any binary search protocol modified to produce a bit array instead of a single element. So far we only showed that CompBS can be naturally used for this purpose, but below we also show how to modify TagBS. This would imply that LayHBS could additionally be used for that purpose. Lastly, if we replace the call to BS on line 7 of SubHBS with a call to MBS, SubHBS itself can be invoked on line 3. Combined with the ability to choose parameter α (and consequently β), these options provide a rather significant performance optimization space in practice.

What remains is to discuss our solution for modifying TagBS to implement the interface for MBS. Recall that TagBS generates tags $[q^{(i)}]$ for each layer i in the hierarchy and maintains variable $[p]$. The tags have a useful structure, which each tag being a bit and having only a single tag set to 1 per layer. However, to generate a desired bit array $[d_0], \ldots, [d_{m-1}]$, we need to have a single bit set to 1 out of the entire dataset and not per layer. Instead of trying to combine different layers into a single array, out solution is to continue with the structure of TagBS and generate one more layer $[q^{(\log m)}]$, which this time will have a tag for each element of the original dataset and only a single element will be set to 1. We also remove the code for creating and maintaining the value of $[p]$ because it is no longer used in the protocol. The final solution TagMBS is formalized as Protocol 7. As noted before, this variant will allow for a variety of solutions to be used with SubHBS.

Performance of hierarchical SubHBS is shown in Table 1. Because the algorithm makes calls to binary search algorithms on datasets of smaller sizes, we use notation $\mathsf{BS}(x)$ and $\mathsf{MBS}(x)$ to denote the cost of binary search and modified binary search, respectively, invoked on input of size x. The cost of the write variant will depend on the binary search algorithms that SubHBS calls.

7 Update Operations

We next explore solutions for a binary search followed by an update operation. The first variant updates non-key fields of the element located by the search and does not require high-level changes to the logic of read constructions. The second variant is for updating the key itself, which requires changes to the logic due to the need to re-position the affected element within the dataset to maintain the sorted order.

7.1 Updating Non-Key Fields

In this write or update variant, we determine the target location by a binary search and then update the corresponding non-key fields. That is, the write operation updates one element at a private location, while the order of the elements remain unchanged. The high-level logic behind the modification is that we produce an m-bit array with zeros in all positions except the one which was located by the binary search, the bit of which is set to 1. Given this array, we can easily update all elements of the data set in such a way that only one element is updated with the desired value, while all other elements remains unchanged. Note that it is necessary to touch (re-randomize) all elements of the original dataset to ensure that the location of the selected element is not revealed.

To this extent, we revisit our binary search constructions and describe what changes are necessary to support this operation. We use v to denote the value with which the located element is to be updated with the understanding that the key field is not affected, i.e., we somewhat abuse the notation to update an element with v, but only non-key fields are updated.

Recall CompBS, where we perform a linear number of comparisons, update the resulting bit array to ensure only one bit is set to 1, and retrieve the desired element using the dot-product operation. The structure of that algorithm makes it easy to support write operations, where we only need to replace the retrieval step with an update step. In particular, the d_i values form the unity array that we need. Then after the computation of the d_is, we update the non-key fields of each a_i as $[d_i][v] + (1 - [d_i])[a_i] = [a_i] - ([a_i] - [v])[d_i]$. A complete specification of this variant, CompBSW, is given in Appendix A as Protocol 9 due to its close resemblance to CompBS. The extra work associated with this and other protocols compared to the read versions is given in Table 1.

Consider OramBS next. Recall that it makes $\log m$ ORAM (read) accesses and the element of interest is guaranteed to be on the selected path. The original OramBS avoids an extra ORAM access at the end by storing additional element p from the path. However, to perform the update, we still need to protect the location of the update and make an additional ORAM (write) access. Detailed OramBSW is given as Protocol 10 in Appendix A.

RotBS was a hierarchical approach with layer rotations by one-time random offsets to protect information about accessed elements. In that construction, we retrieved one element from each rotated layer, computed the element corresponding to the result of the search, and returned to the original representation of the data set. Now, as we would like to update an element of the set, performing the update on rotated layers and returning to the original layers afterwards would nullify the update. For that reason, we modify the solution to return the rotated layers, which become the starting point for the next invocation of binary search. This means that we (privately) maintain the current amount of rotation for each layer and add a new offset to it during each additional rotation. That is, after each binary search, the amount of rotation associated with each layer corresponds to the cumulative amount of rotations over all prior searches.

An issue arises in connection to storing cumulative offsets due to the need to keep each cumulative sum below the size of the layer after each addition (which corresponds to reduction modulo 2^i). Recall that in RotBS we avoided reducing the sum $d + r^{(i)}$ modulo 2^i and instead protected the carry bit prior to the opening the sum. This time, however, the cumulative value of $r^{(i)}$ increases after each binary search and thus we would need to protect more and more bits during the opening. Because the bitlength of the value being protected affects the size of the field over which we compute, it is not practical to let that value go unbounded. For that reason, we periodically reduce the cumulative sum modulo 2^i and denote the period by δ. RotBSW and the updated rotation that reflect these changes are listed as Protocols 11 and 12 in Appendix A. We also detail the mechanism for periodic reduction of the sum.

What remains to discuss in the context of rotation-based algorithm is how to update the result of the search. Recall that the protocol discloses $\log m$ masked locations, one of which must be our target. This means that it is sufficient for us to update those $\log m$ locations. Thus, if we find a way to write the new data to the target location without changing the data written to the remaining $\log(m) - 1$ locations, data-obliviousness will be maintained. Fortunately, the comparison results leave a hint that the last retrieved element $\leq b$ must be the one we want. In other words, we only need to find the last $[c^{(i)}]$ set to 1 and reset all other comparison results to 0. We do this iteratively, which does not increase the number of rounds compared to RotBS; see Appendix A for details.

The last construction we discuss is TagBS modified for writing. Given that TagMBS already produces a bit array, it becomes straightforward to realize its write variant. In particular, because TagMBS produces unity bit array $[d_0], \ldots, [d_{m-1}]$, all that is necessary is to append lines 9–11 of CompBSW to the end of TagMBS to obtain TagBSW. The cost difference compared to the original TagBS is given in Table 1.

7.2 Updating the Key Field

Now, updating the key of an element returned by the search impacts the structure of the solution. This is because a change in an element's key can lead to relocation of the element within the sorted array. This means that we need to locate both the target element's current location and the new location and obliviously reposition all elements between these two locations. We sketch a possible solution in this section.

Given two keys, i.e., the original key and the new key, we need to execute two instances of binary search with these two keys to determine the original and new locations of the target element. Because we represent the dataset as a sorted array, obliviously moving the in-between elements requires $O(m)$ work. We describe a simple solution which expects that the two binary search instances produce bit arrays.

Suppose that the two searches determine (private) locations x and y, respectively. Also suppose that the results of each search are encoded as two bit arrays $[c_0], \ldots, [c_{m-1}]$ and $[d_0], \ldots, [d_{m-1}]$, where the latter contains 0s with the exception of the target location and the former is filled with 0s prior to the target location and with 1s starting from the target location. These bit arrays are identical to what CompBS generates during the computation, but the latter array is also produced by TagMBS and hybrid constructions. Note that given one of the arrays, it is straightforward to produce the other. In what follows, we use superscript (1) and (2) to denote the results of the first and second searches, respectively.

The logic behind our solution is that any element with index smaller than $\min(x, y)$ and greater than $\max(x, y)$ remains unchanged. If $x > y$, each element in the range $[y + 1, x]$ should be overwritten with its left neighbor and new data $[v]$ should be written in location y. Otherwise, if $x < y$, each element in the range $[x, y - 1]$ should be overwritten with its right neighbor and the new data $[v]$ is written in location y. (When $x = y$, either case applies and there are no elements to shift.) Thus, for any given element, there are four possibilities: it remains unchanged, it moves left, right, or it gets replaced with the new value.

Because the computation is different depending on whether x or y is greater, as the first step of the computation, we determine the result of the comparison $[w] \leftarrow \mathsf{LE}([x],[y])$. Note that x and y are not provided to us in the above form, but given the information we possess, it is easy to compute them as $[x] = \sum_{i=0}^{m-1} i \cdot [d_i^{(1)}]$ and $[y] = \sum_{i=0}^{m-1} i \cdot [d_i^{(2)}]$, which is local computation.

Having computed the value of $[w]$, we can proceed with updating the elements themselves. We start by computing a new array $[c_0], \ldots, [c_{m-1}]$ with the elements computed as $[c_i] = [c_i^{(1)}] \oplus [c_i^{(2)}]$. Namely, we execute $[c_i] = [c_i^{(1)}] + [c_i^{(2)}] - 2[c_i^{(1)}] \cdot [c_i^{(2)}]$ for each $i \in [0, m-1]$ in parallel. This results in the elements between x and y being set to 1 while erasing the remaining elements, and the in-between elements are what we would like to move.

The next point to notice is that when $x > y$, the positions of the non-zero c_i values correspond exactly to the positions of the values we want to move, i.e., they range from indices y through $x-1$. However, when $x \leq y$, the positions of non-zero c_i elements range from x to $y-1$, while we need to move elements with indices $x+1$ through y. For that reason, as the next step, we use the value of $[w]$ to update the $[c_i]$s in parallel as $[c_i] = [c_i] + [w] \cdot ([d_i^{(1)}] - [d_i^{(2)}])$. Now we are ready to proceed with updating the dataset and update each element in parallel as

$$[\hat{a}_i] = [c_i] \cdot ([w] \cdot [a_{i+1}] + (1 - [w]) \cdot [a_{i-1}]) + (1 - [c_i]) \cdot ([d_i^{(2)}] \cdot [v] + (1 - [d_i^{(2)}]) \cdot [a_i]).$$

That is, the first of the two terms, conditioned on $c_i = 1$, corresponds to moving the in-between elements either left or right depending on w. The second term, conditioned on $c_i = 0$, corresponds to either updating the target element with the new value v or keeping the element unchanged depending on $d_i^{(2)}$, i.e., whether it is the target location. The formula is written for readability and can be expressed differently to lower the number of interactive operations (multiplications). This concludes our description.

8 Performance Evaluation

8.1 Cost in Different Settings

Table 1 shows costs of our constructions when instantiated with Shamir secret sharing in the semi-honest setting. If one would like to utilize a different underlying framework, for example, 3-party replicated secret sharing over ring $\mathbb{Z}_{2^k}$, the costs of the building blocks can change which will have an impact on the total cost. Using a different realization of LE would likely most significantly impact CompBS, but we expect that relative performance of different construction will remain similar.

If we realize the constructions in the malicious adversarial setting, many frameworks do not support dot product of communication independent of the input size. Note that all of CompBS, RotBS, TagBS, and the hybrid constructions utilize dot product operations and thus their communication will be impacted. The amount of impact varies based on the relationship of m to the original communication cost. For example, this is a relatively small increase for CompBS, close to doubling for TagBS, and significant increase for SubHBS. The order of the protocols in terms of their communication volumes, however, does not change.

8.2 Evaluation Setup

We implemented several of our algorithms and carried out experiments on both LAN and WAN. For LAN experiments, we used three machines with 2.1GHz processors connected via 1Gbps Ethernet (934Mbps throughput) with a one-way latency of 0.13ms. Our WAN experiments used local machines and one remote machine with a 2.4GHz processor. The link between the remote and local

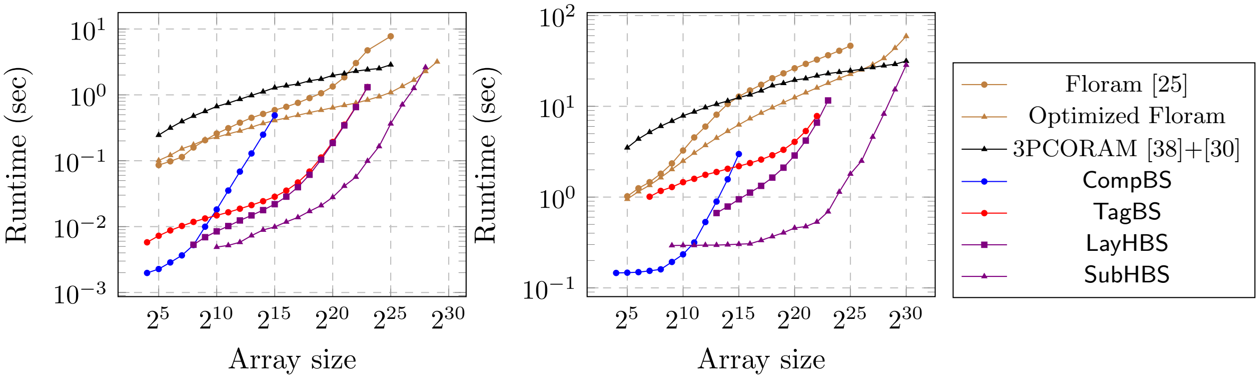

Figure 5: Performance of ring-based binary search in the semi-honest model on LAN (left) and WAN (right).

machines had throughput of 76–85Mbps and a one-way latency of 21ms. While the machine configurations are slightly different, we examine the times to ensure that the differences do not introduce inconsistencies in the experiments. That is, the computation time is determined by the slower machines, and the introduced slowdown on WAN is due to the higher latency and lower bandwidth in the WAN experiments. All experiments used a single thread and the dataset elements had a single field (the key) represented as a 32-bit integer.

8.3 Performance in the Semi-Honest Model

Our implementation in the semi-honest model is in the honest majority setting (i.e., $t < n/2$). Because ring-based computation is faster than computation over a field, we use replicated secret sharing over $\mathbb{Z}_{2^{32}}$ with three parties. Implementation of comparisons is adapted from field-based LT [18, 17] as described in [22, 9]. The difference with field-based performance is shown in Appendix B.

We compare performance of our constructions to the state-of-the-art Floram [25] and 3PC ORAM [38] after optimizing them. In particular, we run Floram's binary search implementation from [2] and also execute an optimized version using the results of section 4.3. We also apply Gentry et al.'s optimization [30] to 3PC ORAM which makes the cost of binary search asymptotically equal to a single ORAM access and run it using the implementation from [1]. Note that 3PC ORAM uses custom techniques not compatible with standard secure computation building blocks. For that reason we approximate performance of optimized 3PC ORAM binary search using comparisons and multiplications as implemented in this work. The exact times might be higher due to the need to convert between different representations. Furthermore, all ORAM schemes come with significant initialization costs not captured in our experiments.

The results of our experiments on both LAN and WAN are given in Figure 5. First note that optimized Floram binary search is on par with the original Floram binary search for small datasets and the difference starts to show when m becomes larger than 2^{10}. Also, the optimization applied to 3PC ORAM removes recursion and makes the construction faster.

Our CompBS outperforms all other protocols when the dataset size is small because the round complexity is the bottleneck with small m. With larger m, communication is dominant and CompBS is not competitive, as expected, but maintains its advantage longer on WAN because of the latency. TagBS has lower communication and shows a better runtime than optimized Floram for sizes up to 2^{22} on LAN and larger on WAN, but its linear communication becomes the bottleneck for large sizes.

Hybrid LayHBS can improve performance of TagBS by only a constant. The maximum improvement is achieved when $\tilde{\imath} = 6$ on LAN and $\tilde{\imath} = 12$ on WAN, where the runtime gap between CompBS and TagBS is the largest. Nevertheless, the difference in performance of TagBS and LayHBS diminishes as the size increases.

Our SubHBS shows significant advantage over all other protocols. It outperforms all other options for sizes is up to 2^{27} on LAN and 2^{11}–2^{30} on WAN due to its low communication. As the size increases, its curve becomes steep indicating that local computation is the bottleneck. In particular, the dot-product computation of $O(m)$ local work consumes 94% (45%) of the total time with $m = 2^{25}$ on LAN (resp., WAN). This means that the performance could be significantly improved for large sizes via multi-threading. We also would like to note that while it appears that SubHBS's curve is significantly steeper than that of (optimized) Floram binary search, this will not be the case if we keep increasing m. Both SubHBS and optimized Floram search require $O(m)$ local computation, but computation becomes the bottleneck for SubHBS at smaller sizes because of its superior performance for medium values of m. This is consistent with findings in [25] which show that Floram's curve becomes steep close to 2^{30} in a setting with 500Mbps bandwidth.

Recall that implementing SubHBS involves several optimization choices including the algorithms for the sub-searches and the values for parameter α. While using $\alpha = O(\sqrt{m})$ gives us theoretically the lowest communication, the network environment and the speed of sub-protocols also affect this decision. For each experiment, we vary the value of α and select the best performing protocols for the sub-searches to determine the best configuration. Specifically, for LAN, we used $\alpha = 2^6$ and modified CompBS for the first sub-search for m from 2^{10} to 2^{14} and $\alpha = 2^7$ and TagMBS for larger m; the second sub-search used the fastest protocol for the corresponding β. On WAN, we set $\alpha \approx \sqrt{m}$ for $m < 2^{16}$ and use $\alpha = 2^8$ for m between 2^{16} to 2^{19}. For larger m, we let $\beta = 2^{13}$ or 2^{14} and use SubHBS with $\alpha = 2^8$ recursively for the second sub-search. TagBS and TagMBS were not helpful in WAN experiments because CompBS is faster for small to medium sizes and SubHBS that combines CompBS variants becomes a better choice. We expect the best configurations to vary based on computation and communication resources.

The above choices give us that on LAN each party sends 22.9KB when $m = 2^{15}$, 65KB when $m = 2^{20}$, and 2.12MB when $m = 2^{25}$ per SubHBS. On WAN, communication becomes 107.5KB, 116.5KB, and 554KB for the same sizes. The differences are because we opt for CompBS with fewer rounds on WAN and also use α closer to $\sqrt{m}$ when m is large. As is clear from the above, our protocol configurations were optimized for runtime and do not always use minimally achievable communication. Compared to ORAM solutions, SubHBS uses less communication than Floram binary search, both asymptotically and in practice. For example, for $m = 2^{20}$, Floram search communicates 10MB per party. Communication of optimized 3PCORAM (of polylogarithmic cost) is 543KB for $m = 2^{20}$, but predictably will get closer to that of SubHBS and outperform it when the size grows to very large. Memory footprint of our constructions is proportional to the computation size (i.e., linear in m for most protocols).

Figure 6 illustrates communication costs of our and ORAM-based binary search protocols in the semi-honest setting. The communication costs of the ORAM-based solutions and our CompBS and TagBS are the same on both LAN and WAN, while communication of hybrid LayHBS and SubHBS vary since we adjust the settings in different environments to achieve the best runtime results. CompBS has the steepest curve among all solutions as it has the worst communication complexity, which is dominated by m instances of LE operations. TagBS uses only $\log m$ LE comparisons, but also invokes m multiplications, which make its curve relatively flat at small sizes and steep at larger sizes. LayHBS generates significantly more communication on WAN than on LAN at small sizes. This is because we used a larger $\tilde{\imath}$ on WAN, which means that a larger portion is processed by CompBS, inevitably increasing the communication cost. However, larger $\tilde{\imath}$ reduces the total number

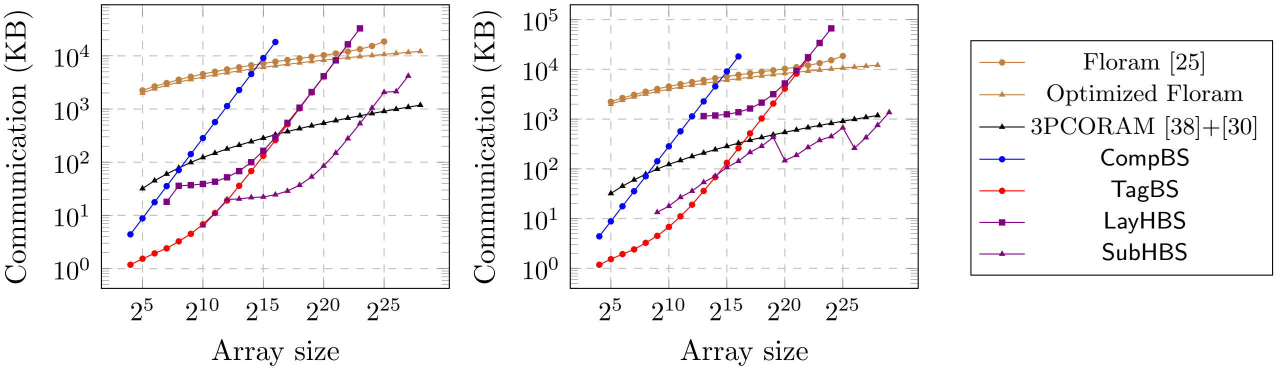

Figure 6: Communication costs (per party) of binary search in the semi-honest model on LAN (left) and WAN (right).

of communication rounds since CompBS has a constant round complexity, and round complexity is the bottleneck at small sizes on WAN. Thus, a larger $\tilde{\imath}$ can improve the overall runtime of LayHBS on WAN.

The curves of our SubHBS on LAN and WAN are significantly different because different settings are used, especially when the dataset size is large. As mentioned in Section 8.3, TagBS or LayHBS are of little use as sub-protocols of SubHBS on WAN because CompBS is faster for small to medium sizes. As a result, we mainly use CompBS in SubHBS on WAN, but TagBS and LayHBS on LAN. This makes the communication costs of SubHBS on WAN larger than that on LAN when the data set size is smaller than 2^{22}. For larger sizes, local computation becomes the dominating component of LAN experiments. The setting we use, i.e., $\alpha = 7$, is not the best choice for optimizing the communication cost, but it results in faster runtime as we can recursively use SubHBS for the second sub-search.

One can notice from Figure 6 that communication of SubHBS fluctuates as the size changes. This is due to the changes in the settings and the choices of sub-search protocols. More specifically, we used CompBS to do the second sub-search on LAN at small sizes and switched to TagBS and SubHBS at $m = 2^{12}$ and 2^{26}, respectively, which also reduced communication. Similarly, we replaced CompBS with a recursive invocation of SubHBS in the second and first sub-search on WAN at $m = 2^{20}$ and 2^{26} because CompBS's performance degrades at large sizes due to its high communication.

To summarize, our best protocols outperform alternative solutions for sizes up to $m = 2^{27}$ on LAN and $m = 2^{30}$ on WAN. SubHBS is up to 49× faster than binary search based on best performing ORAM (optimized Floram) on LAN and up to 27× on WAN.

8.4 Performance in the Malicious Model

To demonstrate that our constructions can be used in the malicious setting, we also evaluate our constructions in the malicious model on both LAN and WAN. Because relying on dot product protocols of constant communication is important for our constructions, we start with the techniques of Dalskov et al. [21] which have this property. The solution uses replicated secret sharing with three or four parties one of which is corrupt and we use the three party version. To evaluate performance when dot products involve linear communication, we use SPD$\mathbb{Z}_{2^k}$ [20, 22]. It is in the dishonest majority setting and our experiments use two parties, which often gives the best performance. Lastly, we also run binary search experiments using maliciously secure ORAM [40],

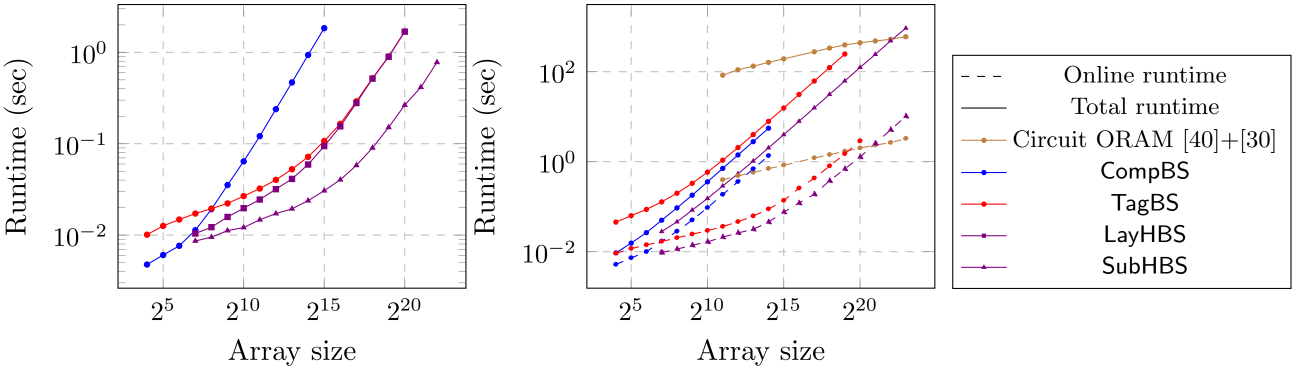

Figure 7: Performance of binary search in the malicious model with (left) and without (right) honest majority on LAN.

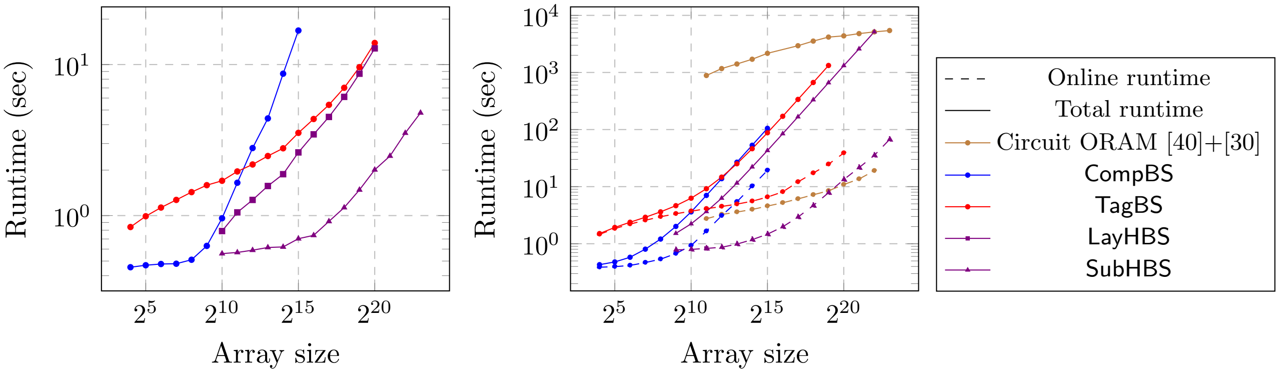

Figure 8: Performance of binary search in the malicious model with (left) and without (right) honest majority on WAN.

which is the only ORAM in the malicious model we are aware of, and use its BMR+Circuit ORAM variant with two parties. Because it is tree-based recursive ORAM, Gentry et al.'s optimization is applied in the experiments. All implementations are from MP-SPDZ [4] and ORAM is available only for $m \geq 2^{11}$.

Figure 7 and 8 show the results, with additional numbers available in Appendix B. The left plots with honest majority depict the total time, just like Figure 5. The right plots with dishonest majority show online and total runtime separately. The total runtime includes both online and estimated offline times, where the offline cost is computed based on the amount of precomputation and offline computation speed for our solutions in MP-SPDZ and the same triple generation speed is used for our protocols and Circuit ORAM. LayHBS is not shown in the right plot for clarity and has almost the same performance as TagBS for larger sizes.

Overall, our constructions show similar trends to those in Figure 5. Because the dot product has linear communication in the dishonest majority setting, both TagBS and SubHBS have linear communication, which narrows the gap between them. Due to the high latency, the gap is larger on WAN because SubHBS has a lower round complexity than TagBS. TagBS outperforms CompBS for $m > 2^8$ on LAN and $m > 2^{13}$ on WAN, with the biggest gap at $m = 2^6$ on LAN and $m = 2^9$ on WAN. Thus, we use $\tilde{\imath} = 6$ on LAN and $\tilde{\imath} = 9$ on WAN for our LayHBS in both malicious settings. Also interesting to note that the total time in the three-party honest majority setting is similar to the online time only with dishonest majority (with the exception of SubHBS when m is large,

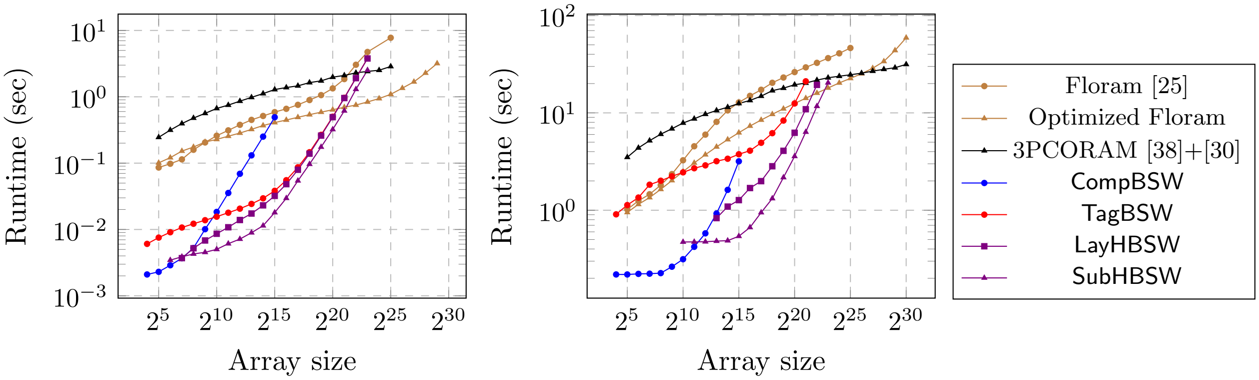

Figure 9: Performance of ring-based binary search (non-key) write in the semi-honest model on LAN (left) and WAN (right).

which is expected); Table 4 makes this clear.

Online time of our dishonest majority SubHBS is up to 18 times smaller than that of Circuit ORAM on LAN and up to 2 times on WAN; Circuit ORAM becomes faster on LAN when m reaches 2^{21} (resp., 2^{20} on WAN). Circuit ORAM shows its advantage earlier on WAN because of its low round complexity, which is consistent with how it is reported in [40]. The total runtime of Circuit ORAM is up to 140 and 240 times slower than that of SubHBS on LAN and WAN, respectively, and performance of our construction and Circuit ORAM becomes similar at 2^{22}. One may also observe that the total runtimes on WAN are an order of magnitude slower than on LAN. This is because not only does the low bandwidth affect the online computation, but it also slows down precomputation needed to generate multiplication triples.

8.5 Performance of Binary Search Followed by a Non-Key Write Operation

We also evaluate performance of our binary search constructions followed by an update operation in the semi-honest setting on both LAN and WAN. Recall that our binary search update solutions first perform binary search that produces an m-bit array with zeros in all positions except the target location, after which we obliviously update the entire array where only the target location is updated with the desired value. Because the this array update operation is the same across all constructions, the runtime differences between our binary search update solutions are primarily due to the (modified) search itself. Because oblivious array update requires linear communication, we expect to see similar trends to those when binary search is implemented in the malicious setting using linear-communication dot product. That is, performance of SubHBS is most severely affected when switching to linear communication, but in the case of malicious security with linear-communication dot product both SubHBS and its update version SubHBSW have the same asymptotic complexity.

Figure 9 shows performance of our and ORAM-based binary search solutions followed by a non-key write, with additional numbers available in Appendix B. The ORAM-based runtimes are the same as those in Figure 5 for conventional binary search because ORAM read and write operations have similar performance. As expected, our constructions show similar trends to those in Figure 5, while the curves of the write variant become steep at smaller sizes. This is because array update (of linear computation and communication) starts to dominate performance of all our solutions for dataset sizes larger than 2^{15} on both LAN and WAN. The gap between our solutions is also narrower.

We varied the value of α to get the best configuration for SubHBSW, but this has little impact on performance since array update is the dominating component for large sizes. For instance, array update consumes 88% (78%) of the total runtime of SubHBSW on LAN (resp., WAN) when $m = 2^{20}$. Overall, our best binary search construction is still faster than best-performing ORAM-based solution (optimized Floram) for dataset sizes up to 2^{21} on LAN and 2^{22} on WAN.

9 Conclusions

In this work we initiate the study of binary search protocols in secure multi-party computation, where on input a private sorted dataset and a private search key, one retrieves or updates the closest element that matches the search. We develop a suite of protocols with different properties and structure and further combine them to into hybrid solutions to improve performance and asymptotic complexity. Our fastest binary search protocol uses $O(\sqrt{m})$ communication for a dataset of size m. We further proceed by modifying these protocols to support the write operation, where the write affects either non-key fields or the key itself.

Our performance evaluation demonstrates that our solutions outperform existing ORAM constructions for dataset sizes up to a billion, even after optimizations to improve performance of ORAM schemes specifically in the context of binary search. We hope that this work will inspire others to work on this topic and make further progress in binary search protocols and in particular designing sublinear-cost constructions.

Acknowledgments

The authors acknowledge support from the Emulab project [55] for utilizing an Emulab machine for WAN experiments. This work was supported in part by a Google Faculty Research Award and Buffalo Blue Sky Initiative. Any options, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding sources.

References

[1] 3PC ORAM implementation. `https://github.com/Boyoung-/circuit-oram-3pc`.

[2] Floram implementation. `https://gitlab.com/neucrypt/floram/tree/floram-release`.

[3] GMP – The GNU multiple precision arithmetic library. `http://www.gmplib.org`.

[4] MP-SPDZ implementation. `https://github.com/data61/MP-SPDZ`.

[5] Sharemind. `https://sharemind-sdk.github.io/`.

[6] Abdelrahaman Aly, Edouard Cuvelier, Sophie Mawet, Olivier Pereira, and Mathieu Van Vyve. Securely solving simple combinatorial graph problems. In *International Conference on Financial Cryptography and Data Security (FC)*, pages 239–257, 2013.

[7] David W. Archer, Dan Bogdanov, Y. Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. From Keys to Databases – Real-World Applications of Secure Multi-Party Computation. *The Computer Journal*, 61(12):1749–1771, 2018.

[8] G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi. Optorama: Optimal oblivious RAM. In *EUROCRYPT*, pages 403–432, 2020.

[9] A. Baccarini, M. Blanton, and C. Yuan. Multi-party replicated secret sharing over a ring with applications to privacy-preserving machine learning. IACR Cryptology ePrint Archive Report 2020/1577, 2020.

[10] M. Blanton, A. Kang, and C. Yuan. Improved building blocks for secure multi-party computation based on secret sharing with honest majority. In *International Conference on Applied Cryptography and Network Security (ACNS)*, pages 377–397, 2020.

[11] Marina Blanton and Siddharth Saraph. Oblivious maximum bipartite matching size algorithm with applications to secure fingerprint identification. In *European Symposium on Research in Computer Security (ESORICS)*, pages 384–406, 2015.

[12] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 207–218, 2013.

[13] 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.

[14] Dan Bogdanov, Liina Kamm, Sven Laur, Pille Pruulmann-Vengerfeldt, Riivo Talviste, and Jan Willemson. Privacy-preserving statistical data analysis on federated databases. In *Annual Privacy Forum (APF)*, pages 30–55, 2014.

[15] P. Bunn, J. Katz, E. Kushilevitz, and R. Ostrovsky. Efficient 3-party distributed ORAM. In *Security and Cryptography for Networks (SCN)*, pages 215–232, 2020.

[16] D. Cash, A. Drucker, and A. Hoover. A lower bound for one-round oblivious RAM. In *TCC*, pages 457–485, 2020.

[17] O. Catrina. Round-efficient protocols for secure multiparty fixed-point arithmetic. In *International Conference on Communications (COMM)*, pages 431–436, 2018.

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

[19] H. Chen, I. Chillotti, and L. Ren. Onion ring ORAM: Efficient constant bandwidth oblivious RAM from (leveled) TFHE. In *ACM Conference on Computer and Communications Security (CCS)*, pages 345–360, 2019.

[20] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. SPD$\mathbb{Z}_2^k$: Efficient MPC mod 2^k for dishonest majority. In *CRYPTO*, pages 769–798, 2018.

[21] A. Dalskov, D. Escudero, and M. Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX Security Symposium*, 2021.

[22] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1102–1120, 2019.

[23] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC*, pages 285–304, 2006.

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

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

[26] Jack Doerner, David Evans, and Abhi Shelat. Secure stable matching at scale. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1602–1613, 2016.

[27] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *CRYPTO*, pages 823–852, 2020.

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

[29] R. Gennaro, M. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *ACM symposium on Principles of Distributed Computing*, pages 101–111, 1998.

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

[31] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *Advances in Cryptology – EUROCRYPT*, pages 640–658, 2014.

[32] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *ACM STOC*, pages 182–194, 1987.

[33] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications.* Cambridge University Press, 2004.

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

[35] S. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *ACM Conference on Computer and Communications Security (CCS)*, pages 513–524, 2012.

[36] T. Hoang, C. D. Ozkaptan, A. A. Yavuz, J. Guajardo, and T. Nguyen. S^{3}ORAM: 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.

[37] Z. Jafargholi, K. Larsen, and M. Simkin. Optimal oblivious priority queues. In *ACM-SIAM Symposiym on Discrete Algorithms (SODA)*, pages 2366–2383, 2021.

[38] S. Jarecki and B. Wei. 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. In *International Conference on Applied Cryptography and Network Security (ACNS)*, pages 360–378, 2018.

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

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

[41] Peeter Laud. Parallel oblivious array access for secure multiparty computation and privacy-preserving minimum spanning trees. *Proceedings on Privacy Enhancing Technologies (PETS)*, 2015(2):188–205, 2015.

[42] Sven Laur, Riivo Talviste, and Jan Willemson. From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In *Applied Cryptography and Network Security (ACNS)*, pages 84–101. 2013.

[43] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks. Automating efficient RAM-model secure computation. In *IEEE Symposium on Security and Privacy*, pages 623–638, 2014.

[44] S. Patel, G. Persiano, M. Raykova, and K. Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In *Annual Symposium on Foundations of Computer Science (FOCS)*, pages 871–882, 2018.

[45] C. Rao, K. Singh, and A. Kumar. Oblivious stable sorting protocol and oblivious binary search protocol for secure multi-party computation. *Journal of High Speed Networks*, 27(1):67–82, 2021.

[46] Risto Saar. A new database layer for the Sharemind privacy preserving computer. Bachelor's thesis. University of Tartu, 2011.

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

[48] E. Shi. Path oblivious heap: Optimal and practical oblivious priority queue. In *IEEE Symposium on Security and Privacy (S&P)*, pages 449–465, 2020.

[49] 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.

[50] 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.

[51] T. Toft. Secure data structures based on multi-party computation. In *ACM Principles of Distributed Computing (PODC)*, pages 291–292, 2011.

[52] 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.

[53] X. S. Wang, Y. Huang, T. 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.

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

[55] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 255–270, 2002.

[56] A. Yao. Protocols for secure computations. In *Symposium on Foundations of Computer Science*, pages 160–164, 1982.

[57] 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.

[58] Y. Zhang, M. Blanton, and G. Almashaqbeh. Implementing support for pointers to private data in a general-purpose secure multi-party compiler. *ACM Transactions on Privacy and Security (TOPS)*, 21(2), 2018.

[59] 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.

A Additional Protocols

This section lists complete specifications of several protocols, which were described in prior sections. In particular, Protocol 8 lists our hybrid LayHBS from section 6.1 on the example of combining CompBS and TagBS. Protocol 9 describes CompBSW, i.e., the write (to non-key fields) variant of comparison-based CompBS. Protocol 10 specifies our ORAM-based binary search protocol OramBSW which updates non-key fields based on the result of the search.

Protocols 11 and 12 describe rotation-based binary search followed by a write to non-key fields RotBSW and its building block RotateW, respectively. Their general description is offered in section 7.1 and here we specify certain algorithm details such as periodically reducing the accumulated amount of shift modulo 2^i for each layer and updating the target element with the new value. In particular, we keep a global variable *count* which is assumed to be accessible to both protocols and corresponds to the number of invocation of the binary search operations. Once the count reaches δ, we reduce the cumulative amount of rotation, $\hat{r}$, modulo 2^u, where 2^u is the size the corresponding layer (lines 10–12 in RotateW). We use protocol Mod2m from [18] for this purpose, which takes the total bitlength $(u + \log \delta)$ of $\hat{r}$ and the number of bits u to retain as its arguments. It is realized in a constant number of rounds and communication linear in u. Avoiding calling modulo reduction during each rotation reduces both the number of rounds and the number of interactive operations associated with the binary search, but may result in an increased field size, which affects communication associated with all interactive operations in the protocol. In more detail, when the bitlength of the search keys $k > \log m$, using δ such that $\log(m) - 1 + \log \delta = k$ does not lead to a larger field size and is always beneficial. Increasing δ above that value can still result in performance savings, but the cost associated with the increased field size needs to be balanced with the savings due to not executing Mod2m protocol. This also means that savings will be greater for datasets of smaller sizes.

Prior to the first invocation of RotBSW, we set *count* to 0 and execute the protocol with the value of r set to 0. For convenience, the dataset is represented in the form of layers.

To be able to update the target element, we first need to privately determine which of the disclosed $\log m$ locations is the target location using values $[c^{(i)}]$s. The last non-zero value should

Protocol 8 $[z] \leftarrow \mathsf{LayHBS}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [b], \tilde{\imath})$

```
for $i = 0, \ldots, 2^{\tilde{\imath}} - 1$ in parallel do
    $[c_i] \leftarrow \mathsf{LE}([b], [a_{(i+1)m/2^{\tilde{\imath}}-1}.key])$;
end for
$[d_0] = [c_0]$;
for $i = 1, \ldots, 2^{\tilde{\imath}} - 2$ in parallel do
    $[d_i] \leftarrow [c_i] \cdot (1 - [c_{i-1}])$;
end for
$[d_{2^{\tilde{\imath}}-1}] = 1 - [c_{2^{\tilde{\imath}}-2}]$;
$[p] \leftarrow \sum_{i=0}^{2^{\tilde{\imath}}-1} [a_i] \cdot [d_i]$;                    // BS1 ends
let $[\ell^{(i)}] = \langle [a_{\lfloor m/2^{i+1} \rfloor - 1}], [a_{\lfloor 3m/2^{i+1} \rfloor - 1}], \ldots, [a_{\lfloor (2^{i+1}-1)m/2^{i+1} \rfloor - 1}] \rangle$ for $i = \tilde{\imath}, \ldots, \log(m) - 1$;
$[q_0^{(\tilde{\imath})}] = [c_0]$;                    // BS4 starts at layer $\ell^{(\tilde{\imath})}$
for $i = 1, \ldots, 2^{\tilde{\imath}} - 1$ locally do
    $[q_i^{(\tilde{\imath})}] = 2[d_i] - [c_i] + [c_{i-1}]$;
end for
$[a] = \sum_{i=0}^{2^{\tilde{\imath}}-1} [q_i^{(\tilde{\imath})}] \cdot [\ell_i^{(\tilde{\imath})}]$;
$[c] \leftarrow \mathsf{LE}([b], [a.key])$;
$[p] = [c] \cdot ([a] - [p]) + [p]$;
for $i = \tilde{\imath} + 1, \ldots, \log(m) - 1$ do
    for $j = 0, \ldots, 2^i - 1$ in parallel do
        if $j \bmod 2 = 0$ then
            $[q_j^{(i)}] = [q_{\lfloor j/2 \rfloor}^{(i-1)}] \cdot [c]$;
        else
            $[q_j^{(i)}] = [q_{\lfloor j/2 \rfloor}^{(i-1)}] \cdot (1 - [c])$;
        end if
    end for
    $[a] = \sum_{j=0}^{2^i-1} [q_j^{(i)}] \cdot [\ell_j^{(i)}]$;
    $[c] \leftarrow \mathsf{LE}([b], [a.key])$;
    $[p] = [c] \cdot ([a] - [p]) + [p]$;
end for
return $[z] = [p]$;
```

be kept and others should be reset to 0. We perform this operation iteratively: during each iteration we update all previous $[c^{(i)}]$s on line 14 of RotBSW. After the end of the iterations, we obtains an array of $[c^{(i)}]$s, in which all elements are 0 except one, which is set to 1. We consequently write the new data $[v]$ to its desired location using the array of $c^{(i)}$s on lines 18–20.

There can be different ways of updating the $[c^{(i)}]$s, e.g., by using a prefix operation after all algorithm's iterations finish. We choose the above mechanism due to its round complexity. That is, as can be seen from Table 1, our rotation-based binary search offers the lowest round complexity among the hierarchical constructions and the above mechanism does not increase the number of rounds. In particular, the multiplications on line 14 use one round, but they can be combined with other interactive operations. This modification for supporting the write operation introduces only one additional round to update all layers at once and increases communication by $(\log^2(m) - \log m)/2$ invocations.

Protocol 9 $\langle[\hat{a}_0], \ldots, [\hat{a}_{m-1}]\rangle \leftarrow \mathsf{CompBSW}(\langle[a_0], \ldots, [a_{m-1}]\rangle, [b], [v])$

```
for $i = 0, \ldots, m-1$ in parallel do
    $[c_i] \leftarrow \mathsf{LE}([b], [a_i.key])$;
end for
$[d_0] = [c_0]$;
for $i = 1, \ldots, m-2$ in parallel do
    $[d_i] = [c_i] \cdot (1 - [c_{i-1}])$;
end for
$[d_{m-1}] = 1 - [c_{m-2}]$;
for $i = 0, \ldots, m-1$ in parallel do
    $[a'_i] = [a_i] - ([a_i] - [v]) \cdot [d_i]$;
end for
return $\langle[a'_0], \ldots, [a'_{m-1}]\rangle$;
```

Protocol 10 $\mathsf{ORAM}([\hat{a}_0], \ldots, [\hat{a}_{m-1}]) \leftarrow \mathsf{OramBSW}(\mathsf{ORAM}([a_0], \ldots, [a_{m-1}]), [b], [v])$

```
$[c] \leftarrow \mathsf{LE}([b], [a_{\lfloor m/2 \rfloor}.key])$;
$[d] = \lfloor m/2 \rfloor$;
$[p] = [c] \cdot (\lfloor m/2 \rfloor - (m-1)) + (m-1)$;
for $i = 0, \ldots, \log(m) - 1$ do
    $[d] = [d] + (1 - 2[c])m/2^{i+2}$;
    $[a] = \mathsf{ORAMRead}([d])$;
    $[c] \leftarrow \mathsf{LE}([b], [a.key])$;
    $[p] = [c] \cdot ([d] - [p]) + [p]$;
end for
$\mathsf{ORAMW}([p], [v])$;
return $\mathsf{ORAM}([\hat{a}_0], \ldots, [\hat{a}_{m-1}])$;
```

B Additional Performance Results

In this section we list additional performance results. Tables 2 and 3 show runtimes of our constructions in the semi-honest model with honest majority and three computational parties. They show performance using replicated secret sharing over ring $\mathbb{Z}_{2^k}$ and performance using Shamir secret sharing over field $\mathbb{F}_p$ with prime p. Performance difference is also visualized in Figure 10.

The field-based implementation used the GNU Multiple Precision Arithmetic Library (GMP) [3] for field arithmetic and executed our protocols within the PICCO compiler framework [59] with some recent optimizations such as [10]. As shown in Figure 10, field-based protocols have similar curves to ring-based curves, but performance is about an order of magnitude slower. Compared to the ring setting, computation becomes the bottleneck for LayHBS and SubHBS sooner, which is expected because local work is slower. The building blocks are also a few times slower with a field.

Setting $\tilde{\imath} = 4$ on LAN and $\tilde{\imath} = 9$ on WAN gives us the maximal improvement for field-based LayHBS. For SubHBS, the chosen α and sub-search algorithms are also different. For example, in our LAN experiments with $m = 2^{12}$, we set $\alpha = 2^4$. This is because modified CompBS for the first sub-search and LayHBS for the second sub-search give us the best overall performance with that choice of α. As dataset size increases, using SubHBS recursively for the second sub-search could achieve better results. In contrast, in the environment with high latency, setting α to be larger would be a better choice that provides the best round complexity.

Tables 4 and 5 demonstrate the runtimes of our protocols in the malicious model with and

Protocol 11 $(\langle[\hat{\ell}^{(0)}],\ldots,[\hat{\ell}^{(\log(m)-1)}]\rangle, \langle[\hat{r}^{(1)}],\ldots,[\hat{r}^{(\log(m)-1)}]\rangle) \leftarrow \mathsf{RotBSW}(\langle[\ell^{(0)}],\ldots,[\ell^{(\log(m)-1)}]\rangle, \langle[r^{(1)}],\ldots,[r^{(\log(m)-1)}]\rangle, [b], [v])$

```
$count = count + 1$;
for $i = 1, \ldots, \log(m) - 1$ in parallel do
    $\langle[\hat{\ell}^{(i)}], [\hat{r}^{(i)}]\rangle \leftarrow \mathsf{RotateW}([\ell^{(i)}], [r^{(i)}], i)$;
end for
$[c^{(0)}] \leftarrow \mathsf{LE}([b], [a_{\lfloor m/2 \rfloor}.key])$;
$[d] = 0$;
for $i = 1, \ldots, \log(m) - 1$ do
    $[d] = 2[d] + [c^{(i-1)}]$;
    $[w] \leftarrow \mathsf{RandInt}(\kappa + \log\delta)$;
    $s'^{(i)} = \mathsf{Open}([d + \hat{r}^{(i)} + 2^i[w])$;
    $s^{(i)} = s'^{(i)} \bmod 2^i$;
    $[a] = [\hat{\ell}^{(i)}_{s^{(i)}}]$;
    $[c^{(i)}] \leftarrow \mathsf{LE}([b], [a.key])$;
    for $j = 0, \ldots, i - 1$ in parallel do
        $[c^{(j)}] = (1 - [c^{(i)}]) \cdot [c^{(j)}]$;
    end for
end for
for $i = 0, \ldots, \log(m) - 1$ in parallel do
    $[\hat{\ell}^{(i)}_{s^{(i)}}] = [\hat{\ell}^{(i)}_{s^{(i)}}] + [c^{(i)}] \cdot ([v] - [\hat{\ell}^{(i)}_{s^{(i)}}])$ ;
end for
return $(\langle[\hat{\ell}^{(0)}], \ldots, [\hat{\ell}^{(\log(m)-1)}]\rangle, \langle[\hat{r}^{(1)}], \ldots, [\hat{r}^{(\log(m)-1)}]\rangle)$;
```

without honest majority on LAN and WAN. We show performance using three-party replicated secret sharing [21] for the honest majority setting and online performance using SPD$\mathbb{Z}_{2^k}$ [20, 22] with two parties for the dishonest majority setting. Setting $\alpha = O(\sqrt{m})$ using SubHBS achieved the best results for all the tested sizes. The runtime of SubHBS with honest majority starts to increase rapidly around size 2^{20}, which indicates that the computation cost becomes the dominating part of the overall cost. In contrast, we expect communication to be the dominating portion of SubHBS in the dishonest majority setting on WAN, where we observe a steep rise at size 2^{20}. While it may appear that the runtime of CompBS in the honest majority setting is worse than that in the dishonest majority setting, we recall that the implementation of Rep3 provides the total time including preprocessing, while the preprocessing cost in the dishonest majority setting is estimated separately and is not included in the tables.

C Security Analysis

In this work we use the standard definition of security in the presence of semi-honest participants. The definition requires that an adversary corrupting at most t participants is unable to learn any information about private inputs. This is modeled by showing that a simulator without access to the private inputs is able to simulate the adversarial view, which the adversary is not able to distinguish from the actual view of a protocol execution.

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 pre-*

Protocol 12 $([\hat{\ell}], [\hat{r}]) \leftarrow \mathsf{RotateW}(\ell = \langle [a_0], \ldots, [a_{2^u-1}] \rangle, [r], u)$

```
for $i = 0, \ldots, u-1$ in parallel do
    $[r_i] \leftarrow \mathsf{RandBit}();$
end for
for $i = 0, \ldots, u-1$ do
    for $j = 0, \ldots, 2^u - 1$ in parallel do
        $[a_j^{(i+1)}] \leftarrow [a_j^{(i)}] - ([a_j^{(i)}] - [a_{j-2^i \bmod 2^u}^{(i)}]) \cdot [r_i];$
    end for
end for
$[\hat{r}] \leftarrow \sum_{i=0}^{u-1} 2^i [r_i] + [r];$
if $((count \bmod \delta) = 0)$ then
    $[\hat{r}] = \mathsf{Mod2m}([\hat{r}], u + \log\delta, u);$
end if
let $[\hat{a}_i] = [a_i^{(u)}]$ for $i \in [0, 2^u - 1]$;
return $([\hat{\ell}], [\hat{r}]) = (\langle [\hat{a}_0], \ldots, [\hat{a}_{2^u-1}] \rangle, [\hat{r}]);$
```

Setup	Protocol	Dataset size																					
		2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}
Ring $\mathbb{Z}_{2^k}$	CompBS	1.98	2.27	2.86	3.66	5.25	9.97	18.2	35.3	68.4	129	249	488	–	–	–	–	–	–	–	–	–	–
	TagBS	5.78	7.26	8.75	10.24	11.8	13.3	14.8	16.5	18.7	21.2	24.3	28.4	35.2	46.8	68.5	112	193	353	663	1315	–	–
	LayHBS	–	–	–	–	5.31	6.91	8.47	10.2	12.4	14.8	17.8	21.9	28.5	39.7	61.3	104	186	344	653	1307	–	–
	SubHBS	–	–	–	–	–	–	4.89	5.19	5.79	7.3	8.9	9.9	11.8	13.8	17.1	20.9	27.9	41.3	56.7	99.1	217	454
Field $\mathbb{F}_p$	CompBS	5.9	10	18.3	32.7	64.2	129	261	523	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	TagBS	9.8	12	14.8	17.6	21	24	28.3	36.2	44.7	52.4	61.7	85.1	161	285	529	1023	–	–	–	–	–	–
	LayHBS	–	9.9	10	12	13.2	16.7	19.2	22.9	28.3	37.9	49.1	80.2	151	276	521	1016	–	–	–	–	–	–
	SubHBS	–	–	–	–	–	–	19.0	21.8	23.8	25.8	30.6	37.2	46.4	61.3	97.8	177	266	488	896	–	–	–

Table 2: Runtime of binary search protocols in the semi-honest setting with honest majority and three computational parties on LAN in milliseconds.

cisely, 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_1th, ..., i_tth 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 $\{S_I(\mathsf{in}_I, f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n))\} \equiv \{\mathrm{VIEW}_\Pi(I)\}$, where $\mathsf{in}_I = \bigcup_{P_i \in I} \{\mathsf{in}_i\}$ and $\equiv$ denotes computational or statistical indistinguishability.*

Note that showing a protocol secure under this definition will also guarantee that the computation is data-oblivious, i.e., does not depend on any private data and can be simulated without access to private data.

As mentioned earlier, for concreteness, we instantiate our constructions using classical Shamir secret sharing and corresponding building blocks for operations such as multiplication, comparisons, etc. This setting requires honest majority, i.e., $t < n/2$. We next proceed with showing our protocols secure according to definition 1.

Setup	Protocol	Dataset size																					
		2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}
Ring $\mathbb{Z}_{2^k}$	CompBS	0.146	0.147	0.149	0.154	0.160	0.193	0.234	0.316	0.530	0.893	1.57	2.98	–	–	–	–	–	–	–	–	–	–
	TagBS	–	–	–	1.01	1.17	1.29	1.46	1.59	1.76	1.89	2.05	2.19	2.37	2.59	2.89	3.32	4.06	5.34	7.81	–	–	–
	LayHBS	–	–	–	–	–	–	–	–	–	0.667	0.787	0.945	1.12	1.33	1.64	2.11	2.87	4.19	6.61	11.6	–	–
	SubHBS	–	–	–	–	–	0.293	0.294	0.294	0.296	0.296	0.299	0.303	0.306	0.334	0.368	0.405	0.457	0.474	0.533	0.691	1.14	1.8
Field $\mathbb{F}_p$	CompBS	0.216	0.224	0.242	0.272	0.319	0.516	0.788	1.39	2.48	4.83	–	–	–	–	–	–	–	–	–	–	–	–
	TagBS	0.731	0.915	1.09	1.29	1.47	1.65	1.84	2.03	2.21	2.43	2.62	2.87	3.19	3.82	4.61	6.2	9.02	14.8	23.7	–	–	–
	LayHBS	–	–	–	–	–	–	0.73	0.891	1.10	1.30	1.64	1.77	2.26	2.76	3.64	5.19	7.89	13.8	22.6	–	–	–
	SubHBS	–	–	–	–	–	0.412	0.431	0.448	0.467	0.493	0.527	0.578	0.642	0.703	0.794	0.850	1.01	1.39	1.99	3.19	4.90	9.0

Table 3: Runtime of binary search protocols in the semi-honest model with honest majority and three computational parties on WAN in seconds.

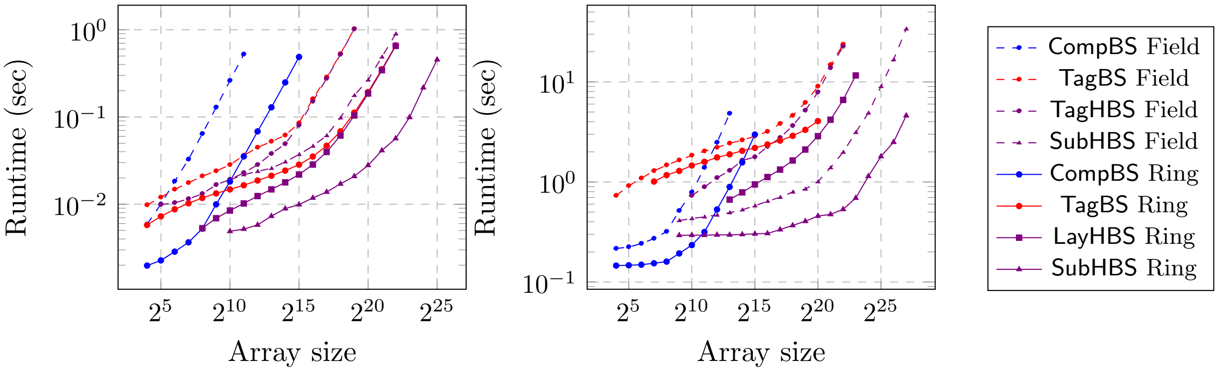

Figure 10: Performances of binary search constructions in the semi-honest model with honest majority and three computational parties on LAN (left) and WAN (right).

Theorem 1 *Let λ be a computational security parameter and κ be a statistical security parameter. Assuming the existence of secure protocols for multiplication, dot product and comparison* LE, *denoted by Π_{mult}, Π_{dot}, Π_{le}, respectively, Protocol 1 is secure according to Definition 1.*

Proof. For this protocol, the parties hold no private inputs and obtain no output. The implication is that it is required that no information about private a_is and b is revealed to corrupt parties I during the protocol execution. This input formulation also means that, upon protocol initiation, the input shares available to the corrupt parties I information-theoretically reveal no information about the underlying input, i.e., they can be reconstructed to any possible values with the same probability.

We build our simulator S_I as follows:

- In step 1, S_I invokes the simulator for Π_{le} m times and simulates the view of the parties in I.
- In step 6, S_I invokes the simulator for Π_{mult} $m-2$ times and simulates the view of the parties in I.
- In step 9, S_I invokes the simulator for Π_{dot} of two m-element vectors and simulates the view of the parties in I.

Next, we need to show that the real and simulated view are indistinguishable. The simulated view consists of the input shares available to the corrupt parties and the messages received during simulation of Π_{le}, simulation of Π_{mult}, and simulation of Π_{dot}. Because the input shares of t or fewer participants are distributed uniformly at random within the field, that component of the view has identical distributions in the real and simulated views and are therefore indistinguishable. Also,

Setup	Protocol	Dataset size																			
		2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
Rep3 [21]	CompBS	4.76	6.05	7.63	11.3	19.2	35.3	64.1	121	239	469	937	1840	–	–	–	–	–	–	–	–
	TagBS	10.1	12.6	14.8	17.2	19.5	22.2	26.7	32.3	40.2	52.6	71.8	107	165	291	522	904	1697	–	–	–
	LayHBS	–	–	–	10.4	12.2	15.8	19.7	24.5	31.8	41.2	59.2	94.1	155	279	516	894	1685	–	–	–
	SubHBS	–	–	–	8.62	9.5	11.2	12.1	14.7	17.2	19.4	23.8	30.6	40.3	58	90.1	151	265	415	778	–
$\mathrm{SPD}\mathbb{Z}_{2^k}$ [20, 22]	CompBS	5.18	7.25	9.95	17.1	28.2	50.3	94.6	186	358	688	1350	–	–	–	–	–	–	–	–	–
	TagBS	9.21	11.7	14.1	16.8	20.3	24.4	29.3	36.1	46.1	61.9	88.4	137	256	429	798	1490	2880	–	–	–
	LayHBS	–	–	–	11	15.1	19.4	24.2	31.1	41.5	56.6	80.3	131	252	421	791	1480	2871	–	–	–
	SubHBS	–	–	–	9.6	11.4	13.9	16.5	21.1	26.1	31.5	45.5	75.8	120	189	376	689	1270	2540	5110	10200

Table 4: Runtime of binary search protocols in the malicious model with honest majority (replicated secret sharing, 3 parties, total time) and dishonest majority ($\mathrm{SPD}\mathbb{Z}_{2^k}$, 2 parties, online only) on LAN in milliseconds.

Setup	Protocol	Dataset size																			
		2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
Rep3 [21]	CompBS	0.455	0.469	0.48	0.51	0.63	0.96	1.65	2.8	4.4	8.7	16.8	–	–	–	–	–	–	–	–	–
	TagBS	0.84	0.99	1.13	1.27	1.43	1.59	1.7	1.96	2.18	2.48	2.79	3.53	4.36	5.41	7.02	9.61	13.9	–	–	–
	LayHBS	–	–	–	–	–	–	0.79	1.05	1.57	1.88	2.62	3.45	4.5	6.11	8.7	8.7	–	–	–	–
	SubHBS	–	–	–	–	–	–	0.56	0.57	0.59	0.615	0.62	0.703	0.74	0.915	1.13	1.48	2.01	2.48	3.52	4.79
$\mathrm{SPD}\mathbb{Z}_{2^k}$ [20, 22]	CompBS	0.388	0.397	0.417	0.468	0.537	0.672	0.93	1.66	3.1	5.44	10.2	19.3	–	–	–	–	–	–	–	–
	TagBS	1.46	1.84	2.21	2.58	2.97	3.35	3.72	4.1	4.49	4.94	5.56	6.57	8.1	12.1	17.3	24.7	38.8	–	–	–
	LayHBS	–	–	–	–	–	–	0.91	1.29	1.71	2.12	2.69	3.64	5.37	8.01	12.9	20.3	34.7	–	–	–
	SubHBS	–	–	–	–	–	0.78	0.796	0.829	0.867	0.985	1.18	1.47	1.98	2.93	4.66	7.83	13.3	21.5	35.3	67

Table 5: Runtime of binary search protocols in the malicious model with honest majority (replicated secret sharing, 3 parties, total time) and dishonest majority ($\mathrm{SPD}\mathbb{Z}_{2^k}$, 2 parties, online only) on WAN in seconds.

because protocols Π_{le}, Π_{mult}, and Π_{dot} are secure (meaning that their simulators are guaranteed to produce indistinguishable views), we get that the remaining components of the view are also indistinguishable, resulting in the overall indistinguishability of the simulation.

The type of indistinguishability that we obtain depends on the indistinguishability guarantees of protocols Π_{le}, Π_{mult}, and Π_{dot}. That is, if at least one of them provides computational guarantees, our simulation will also be computationally secure. Similarly, if at least one of them uses statistical security, our simulation will inherit that requirement as well. Note that many interactive protocols (including Π_{le}, Π_{mult}, Π_{dot}) assume secure pair-wise communication channels, which are normally implemented using computationally secure techniques.

Note that because this protocol produces no private output to any of the corrupt parties, output correctness of the output in the simulated view does not need to be shown. However, if this protocol is followed by another operation that discloses a value to one of more corrupt participants (e.g., open operation), it will always be possible for the simulator to generate the view that will lead to the corrupt parties reconstructing the right output. This is due to the properties of (n, t) secret sharing that will allow any combination of shares that t or fewer participants holds to reconstruct to any possible value in the field. □

Theorem 2 *Let λ and κ be computational and statistical security parameters, respectively. Assuming the existence of secure protocols for multiplication, comparison* LE, *and ORAM read access* ORAMR, *denoted by Π_{mult}, Π_{le}, Π_{oramr}, respectively, Protocol 2 is secure according to Definition 1.*

Setup	Protocol	Dataset size																			
		2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
LAN	CompBSW	2.13	2.3	2.89	3.68	5.26	10.1	18.4	35.5	69.1	131	251	492	–	–	–	–	–	–	–	–
	TagBSW	6.07	7.55	9.09	10.7	12.2	13.8	15.6	17.9	20.6	24.3	29.5	38.3	55.3	86.6	146	267	501	966	1921	3790
	LayHBSW	–	–	–	3.7	5.21	6.83	8.62	10.8	13.9	17.4	23.1	32.1	48.2	79.5	139	260	493	957	1911	3780
	SubHBSW	–	–	3.45	3.88	4.28	4.53	5.02	6.11	7.17	8.94	11.4	18.0	29.6	54.3	95.6	175	322	615	1298	2518
WAN	CompBSW	219	219	222	223	226	263	314	421	579	925	1620	3180	–	–	–	–	–	–	–	–
	TagBSW	908	1130	1350	1570	1830	2010	2230	2450	2700	2890	3190	3380	3760	4090	4930	6190	8340	12500	21100	37900
	LayHBSW	–	–	–	–	–	–	–	–	–	830	1090	1270	1690	1990	2830	4080	6250	10900	19400	35600
	SubHBSW	–	–	–	–	–	–	472	473	475	482	487	541	666	947	1320	2170	3580	6360	11700	20400

Table 6: Runtime of binary search non-key write protocols in the semi-honest setting with honest majority and three computational parties on LAN and WAN in milliseconds.

Proof. As before, we prove security of Protocol 2 in the presence of semi-honest participants according to Definition 1. Recall that the parties hold no private inputs and obtain no output. Our simulator S_I works as follows:

- In step 1, S_I invokes the simulator for Π_{le} and simulates the view of the parties in I.
- In step 3, S_I invokes the simulator for Π_{mult} and simulates the view of the parties in I.
- During evaluation of the loop, S_I iteratively invokes simulators for Π_{oramr} (step 6), Π_{le} (step 7), and Π_{mult} (step 8) $\log m$ times to simulate the view of the parties in I.

The simulated view now consists of the input shares available to the corrupt parties and the messages received during the simulation of Π_{le}, Π_{mult}, and Π_{oramr}. As before, we simulate the input shares by distributing random field elements to the corrupt parties, which have the same distribution as during the real execution and are therefore indistinguishable. In addition, because of security of protocols Π_{le}, Π_{mult}, and Π_{oramr}, their simulators are guaranteed to produce indistinguishable views. Thus, all components of the view are indistinguishable and, as a result, the overall simulation is indistinguishable as well. □

Theorem 3 *Let λ and κ be computational and statistical security parameters, respectively. Assuming the existence of secure protocols for multiplication and random bit generation* RandBit*, denoted by Π_{mult} and $\Pi_{randbit}$, respectively, Protocol 4 is secure according to Definition 1.*

Proof. As before, it is straightforward to prove security of Protocol 4 in the presence of semi-honest participants based on Definition 1. Our simulator S_I needs to invoke the simulator for $\Pi_{randbit}$ u times in step 2 and the simulator for Π_{mult} u times in batches of size 2^u in step 7 to simulates the view of the parties in I.

As before, indistinguishability follows from information-theoretic security of the secret sharing scheme in the presence of at most t corrupt participants and security of protocols Pi_{mult} and $\Pi_{randbit}$ which must come with simulators that produce indistinguishable views. □

So far security of Protocols 1, 2, and 4 was straightforward to show because they only combine secure building blocks. We next discuss Protocol 3, which involves more interesting analysis because it opens values during the computation.

Theorem 4 *Let λ and κ be computational and statistical security parameters, respectively. Assuming the existence of secure protocols for multiplication, random integer generation* RandInt*, comparison* LE*, and rotation* Rotate*, denoted by Π_{mult}, $\Pi_{randint}$, Π_{le}, Π_{rotate}, respectively, Protocol 3 is secure according to Definition 1.*

Proof. As before, we prove security of Protocol 3 in presence of a semi-honest adversaries according to Definition 1, by building a simulator and shoring indistinguishability of the real and simulated views. Recall that the parties hold no private inputs and obtain no output.

Our simulator S_I can be built as follows:

1. In step 3, S_I invokes the simulator for Π_{rotate} on the layers of varying sizes $\log m - 1$ times and simulates the view of parties in I.
2. In step 5, S_I invokes the simulator for Π_{le} and simulates the view of parties in I.
3. In step 6, S_I invokes the simulator for Π_{mult} to simulate the multiplication view.
4. Simulation of the for loop on lines 8–16 proceeds as follows:
   (a) Assuming non-interactive implementation of $\Pi_{randint}$, there is nothing for S_I to simulate for step 10.
   (b) To simulate opening of s' shares, S_I generates a random integer $s' \in \mathbb{Z}_{2^{\kappa+i+1}}$, uses its knowledge of the corrupt parties' shares prior to the opening[4] to compute the remaining shares which would reconstruct to the chosen s', and communicates the computed shares to the parties in I according to the specification of Π_{open}.
   (c) Consequently, S_I invokes the simulators for Π_{le} and Π_{mult} to simulate the view of the parties in I for steps 14 and 15, respectively.

   This portion is repeated and performed $\log(m) - 1$ times to simulate the view of the parties in I.

Next, we demonstrate that the real and simulated view are indistinguishable. Besides the simulations of interactive building blocks such as Π_{mult}, Π_{le}, and Π_{rotate}, which are required to come with simulators that guarantee indistinguishability, this simulation includes an important component which makes it different from prior proofs. In particular, in the simulated view the parties receive a random $(\kappa+i+1)$-bit integer s', while in the real view it was computed according to the protocol specification. That is, in the protocol s' is computed as $d+r^{(i)}+2^i w$, where $r^{(i)}$ is a secret one-time i-bit random value and w is a secret one-time $(\kappa+1)$-bit random value. The purpose of $r^{(i)}$ was to perfectly protect d when disclosing $d + r \bmod 2^i$ and the purpose of w was to statistically protect the carry bit after addition of i-bit d and $r^{(i)}$.

We next argue that the distribution of s' in real execution is statically close to the uniform distribution over $(\kappa + i + 1)$-bit integers (as used in the simulation) and thus we can simulate the view of corrupt parties during protocol execution by drawing a random element from that space and obtain statistical indistinguishability. In more detail, because we protect the $(i+1)$st carry bit of $d + r^{(i)}$ with a $(\kappa + 1)$-bit random integer, information about the carry bit can only be revealed with at most negligible probability in the security parameter κ. The remaining i bits of $d + r^{(i)}$ and s' are distributed uniformly at random over the range $[0, 2^i - 1]$ and reveal no information information about d, i.e., perfect secrecy is achieved.

Combining this argument with the fact that the simulators of the interactive building blocks produce indistinguishable views, we obtain that the real and simulated views are indistinguishable in the presence of at most t semi-honest adversaries. □

Security of other protocols is not difficult to show using the same logic as that in the proofs of Protocols 1, 2, and 4. Namely, because they only invoke secure building blocks, a composition of secure building blocks will result in security of the overall protocols and indistinguishable simulation

[4] Maintaining shares of the corrupted parties for the variables used in the computation of s' would require the simulator to remember and maintain additional information, but this can be accomplished in our setting. In particular, this is because we know the corrupt parties' inputs and messages they receive, because the corrupt parties follow the computation, and because we are working with a redundant secret sharing scheme that permits extraction of corrupt minority's shares from the shares they distribute to the simulated parties during the execution of interactive building blocks.

can be built by invoking simulators associated with the building blocks. Note that this also holds for protocols that realize binary search followed by a write operation, in which building blocks would need to be called in a different way, but there is no conceptual difference for the purposes of security proofs. For that reason, we omit security arguments for the remaining protocols.