Efficient Oblivious Parallel Array Reads and Writes for Secure Multiparty Computation

Peeter Laud
Cybernetica AS
peeter.laud@cyber.ee

August 16, 2014

Abstract

In this note we describe efficient protocols to perform in parallel many reads and writes in private arrays according to private indices. The protocol is implemented on top of the Arithmetic Black Box (ABB) and can be freely composed to build larger privacy-preserving applications. For a large class of secure multiparty computation (SMC) protocols, we believe our technique to have better practical performance than any previous ORAM technique that has been adapted for use in SMC. We also argue that for a significant class of SMC protocols, our technique has better asymptotic performance than previous approaches.

1 Problem statement

In Secure Multiparty Computation (SMC), k parties compute $(y_1, \ldots, y_k) = f(x_1, \ldots, x_k)$, with the party P_i providing the input x_i and learning no more than the output y_i. For any functionality f, there exists a SMC protocol for it [Yao82, GMW87]. A universally composable [Can01] abstraction for SMC is the arithmetic black box (ABB) [DN03]. The ideal functionality $\mathcal{F}_{\mathsf{ABB}}$ allows the parties to store private data in it, perform computations with data inside the ABB, and reveal the results of computations. According to this description, it does not leak anything about the results of the intermediate computations, but only those values whose declassification is explicitly requested by the parties. Hence, any secure implementation of ABB also protects the secrecy of inputs and intermediate computations. There exist a number of practical implementations of the ABB [BDNP08, DGKN09, BLW08, BSMD10, HKS+10, MK10], differing in the underlying protocol sets they use and in the set of operations with private values that they make available for higher-level protocols.

These ABB implementations may be quite efficient for realizing applications working with private data, if the control flow and the data access patterns of the application do not depend on private values. For hiding data access patterns, oblivious RAM (ORAM) techniques [GO96] may be used. These techniques have their overhead, which is increased when they are combined with SMC. Existing combinations of ORAM with SMC report at least $O(\log^3 n)$ overhead for accessing an element of an n-element array [KS14].

In this work, we propose a different method for reading and writing data in SMC according to private addresses. We note that SMC applications are often highly parallelized, because the protocols provided by ABB implementations often have significant latency. Hence it makes sense to try to bundle several data accesses together. In the following, we assume that we have private vector $\vec{v}$ of m elements. We provide two protocols on top of ABB: for reading its elements n times, and for writing its elements n times. The asymptotic complexity of both

protocols is $O((m+n)\log(m+n))$, while the constants hidden in the O-notation should be reasonable. These protocols could be interleaved with the rest of the SMC application in order to provide oblivious data access capability to it.

In the following, we let $[\![x]\!]$ denote that some value has been stored in the ABB and is accessible under handle x. We require the ABB to provide operations for doing arithmetic, and comparing shared values. We let the notation $[\![x]\!] \otimes [\![y]\!]$ denote that the operation $\otimes$ is performed with values stored under handles x and y. In ABB implementations this involves the invocation of the protocol for $\otimes$.

We also require the ABB to provide *oblivious shuffles*. For certain ABB implementations, these can be added as described in [LWZ11]. A more general approach is to use Waksman networks [Wak68]. Given a shuffle $[\![\sigma]\!]$ for m elements, and a private vector of length m, it is possible to apply this shuffle to this vector, permuting its elements and producing a new private vector. It is also possible to *unapply* the shuffle to this vector, performing the inverse permutation of its elements. The complexity of the protocols implementing these ABB operations is either $O(m)$ or $O(m \log m)$ (for constant number of parties).

With oblivious shuffles and comparison operations, vectors of private values (of length n) can be sorted in $O(n \log n)$ time and with reasonable constants hidden in the O-notation [HKI+12]. In our protocols, we let $[\![\sigma]\!] \leftarrow \mathsf{sort}([\![\vec{v}^{(1)}]\!], \ldots, [\![\vec{v}^{(r)}]\!])$ denote that the private vectors $[\![\vec{v}^{(1)}]\!], \ldots, [\![\vec{v}^{(r)}]\!]$, all of the same length, have been lexicographically sorted in non-decreasing order, with the order between elements of $\vec{v}^{(1)}$ being the most significant and the order of elements at certain positions of $\vec{v}^{(r)}$ being consulted only if the $\vec{v}^{(1)}, \ldots, \vec{v}^{(r-1)}$ all have the same elements at these positions. The sort-operation does not actually reorder these vectors, but produces an oblivious shuffle $[\![\sigma]\!]$, the application of which to each $\vec{v}^{(i)}$ would bring them to sorted order. See [LW14] for a precise specification of sort. Also note that we require the sorting to be stable.

2 Protocol for reading

In Alg. 1, we present our protocol for obliviously reading several elements of an array. Given an array $\vec{v}$ of length m, we let $\mathsf{prefixsum}(\vec{v})$ denote a vector $\vec{w}$, also of length m, where $w_i = \sum_{j=1}^{i} v_j$ for all $j \in \{1, \ldots, m\}$. Computing $\mathsf{prefixsum}([\![\vec{v}]\!])$ is a free operation in existing ABB implementations, because addition of elements, not requiring any communication between the parties, is counted as having negligible complexity. We can also define the inverse operation $\mathsf{prefixsum}^{-1}$: if $\vec{w} = \mathsf{prefixsum}(\vec{v})$ then $\vec{v} = \mathsf{prefixsum}^{-1}(\vec{w})$. The inverse operation is even easier to compute: $v_1 = w_1$ and $v_i = w_i - w_{i-1}$ for all $i \in \{2, \ldots, m\}$.

We see that in Alg. 1, the permutation σ orders the indices which we want to read, as well as the indices $1, \ldots, n$ of the "original array" $\vec{v}$. Due to the stability of the sort, each index of the "original array" ends up before the reading indices equal to it. In $\mathsf{apply}(\sigma, \vec{u})$, each element v'_i of $\vec{v}'$, located in the same position as the index i of the "original array" in sorted $\vec{t}$, is followed by zero or more 0-s. The prefix summing restores the elements of $\vec{v}$, with the 0-s also replaced with the element that precedes them. Unapplying σ restores the original order of $\vec{u}$ and we can read out the elements of $\vec{v}$ from the latter half of $\vec{u}'$.

The protocol presented in Alg. 1 clearly preserves the security guarantees of the implementation of the underlying ABB, as it applies only ABB operations, classifies only public constants and declassifies nothing. Its complexity is dominated by the complexity of the sorting operation, which is $O((m+n)\log(m+n))$. We also note that the round complexity of Alg. 1 is $O(\log(m+n))$.

Note that instead of reading elements from an array, the elements of which are indexed with $1, \ldots, m$, the presented protocol could also be used to read the private values from a dictionary, the elements of which are indexed with (private) $[\![j_1]\!], \ldots, [\![j_m]\!]$. In this case, t_i

Algorithm 1: Reading n values from the private array

Data: A private vector $[\![\vec{v}]\!]$ of length m
Data: A private vector $[\![\vec{z}]\!]$ of length n, with $1 \le z_i \le m$ for all i
Result: A private vector $[\![\vec{w}]\!]$ of length n, with $w_i = v_{z_i}$ for all i
$[\![\vec{v}']\!] \leftarrow \mathsf{prefixsum}^{-1}([\![\vec{v}]\!])$
foreach $i \in \{1, \ldots, m\}$ **do**
  $[\![t_i]\!] \leftarrow i$
  $[\![u_i]\!] \leftarrow [\![v'_i]\!]$
foreach $i \in \{1, \ldots, n\}$ **do**
  $[\![t_{m+i}]\!] \leftarrow [\![z_i]\!]$
  $[\![u_{m+i}]\!] \leftarrow 0$
$[\![\sigma]\!] \leftarrow \mathsf{sort}([\![\vec{t}]\!])$
$[\![\vec{u}']\!] \leftarrow \mathsf{unapply}([\![\sigma]\!]; \mathsf{prefixsum}(\mathsf{apply}([\![\sigma]\!]; [\![\vec{u}]\!])))$
foreach $i \in \{1, \ldots, n\}$ **do** $[\![w_i]\!] \leftarrow [\![u'_{m+i}]\!]$
return $[\![\vec{w}]\!]$

(where $1 \le i \le m$) is not initialized with i, but with j_i. Note that in this case, the algorithm cannot detect if all indices that we attempt to read are present in the dictionary.

3 Protocol for writing

For specifying the parallel writing protocol, we have to fix how multiple attempts to write to the same field are resolved. We thus require that each writing request comes with a numeric *priority*; the request with highest priority goes through (if it is not unique, then one is selected arbitrarily). We can also give priorities to the existing elements of the array. Normally they should have the lowest priority (if any attempt to write them actually means that they must be overwritten). However, in case where the array element collects the maximum value during some process (e.g. finding the best path from one vertex of some graph to another), with the writes to this element representing candidate values, the priority of the existing element could be equal to this element. This is useful in e.g. the Bellman-Ford algorithm.

We thus assume that there exists an algorithm $\mathsf{compute\_priority}$ which, when applied to an element $[\![w_i]\!]$ of the vector $[\![\vec{w}]\!]$, as well as to its index i, returns the priority of keeping the current value of w_i. The parallel writing protocol is given in Alg. 2, with the bulk of the work done in Alg. 3. The writing algorithm receives a vector of values $[\![\vec{v}]\!]$ to be written, together with the indices $[\![\vec{j}]\!]$ showing where they have to be written, and the writing priorities $[\![\vec{p}]\!]$. Alg. 2 transforms the current vector $[\![\vec{w}]\!]$ to the same form, and they both are given to Alg. 3 for processing. The data are then sorted according to indices and priorities (with higher-priority elements coming first). The vector $[\![\vec{b}]\!]$ is used to indicate the highest-priority position for each index: $b_i = 0$ iff the i-th element in the vector $\vec{j}'$ is the first (hence the highest-priority) value equal to j'_i. Performing the second sort in Alg. 3 moves the highest-priority values to the first m positions. The sorting is stable, hence the values correspond to the indices $1, \ldots, m$ in this order. We thus have to apply the shuffles induced by both sorts to the vector of values $\vec{v}$, and take the first m elements of the result.

The writing protocol is secure for the same reasons as the reading protocol. Its complexity is dominated by the two sorting operations, it is $O((m+n)\log(m+n))$, with the round complexity being $O(\log(m+n))$. Similarly to the reading protocol, the writing protocol can be adapted to write into a dictionary instead.

Algorithm 2: Obliviously writing n values to a private array

Data: Private vectors $[\![\vec{j}]\!]$, $[\![\vec{v}]\!]$, $[\![\vec{p}]\!]$ of length n, where $1 \le j_i \le m$ for all i
Data: Private array $[\![\vec{w}]\!]$ of length m
Result: $\vec{w}$ updated: values in $\vec{v}$ written to indices in $\vec{j}$, if priorities in $\vec{p}$ are high enough (see main text for complete specification)

foreach $i \in \{1, \ldots, n\}$ **do**
  $[\![j'_i]\!] \leftarrow [\![j_i]\!]$
  $[\![v'_i]\!] \leftarrow [\![v_i]\!]$
  $[\![p'_i]\!] \leftarrow [\![p_i]\!]$

foreach $i \in \{1, \ldots, m\}$ **do**
  $[\![j'_{n+i}]\!] \leftarrow i$
  $[\![v'_{n+i}]\!] \leftarrow [\![w_i]\!]$
  $[\![p'_{n+i}]\!] \leftarrow \mathsf{compute\_priority}(i, [\![w_i]\!])$

$[\![w]\!] := \mathsf{select\_highest\_priority\_values}([\![\vec{j'}]\!], [\![\vec{v'}]\!], [\![\vec{p'}]\!])$; `// Alg. 3`

Algorithm 3: Selecting highest-priority values into a private array of length m

Data: Private vectors $[\![\vec{j}]\!]$, $[\![\vec{v}]\!]$, $[\![\vec{p}]\!]$ of length N, where (i) $j_i \in \{1, \ldots, m\}$ for all i, and (ii) for each $k \in \{1, \ldots, m\}$ exists i, such that $j_i = k$.
Result: Private vector $[\![\vec{w}]\!]$ of length m, where $w_k = x$ iff there exists i, such that $j_i = k$, $v_i = x$, and $p_i = \max_{i'}\{p_{i'} \mid j_{i'} = k\}$

foreach $i \in \{1, \ldots, N\}$ **do** $[\![q_i]\!] \leftarrow -[\![p_i]\!]$
$[\![\sigma]\!] \leftarrow \mathsf{sort}([\![\vec{j}]\!], [\![\vec{q}]\!])$
$[\![\vec{j'}]\!] \leftarrow \mathsf{apply}([\![\sigma]\!]; [\![\vec{j}]\!])$
$[\![b_1]\!] \leftarrow 0$
foreach $i \in \{2, \ldots, N\}$ **do** $[\![b_i]\!] \leftarrow [\![j'_i]\!] \stackrel{?}{=} [\![j'_{i-1}]\!]$
$[\![\tau]\!] \leftarrow \mathsf{sort}([\![\vec{b}]\!])$
$[\![\vec{w'}]\!] \leftarrow \mathsf{apply}([\![\tau]\!]; \mathsf{apply}([\![\sigma]\!]; [\![\vec{v}]\!]))$
foreach $i \in \{1, \ldots, m\}$ **do** $[\![w_i]\!] \leftarrow [\![w'_i]\!]$
return $[\![\vec{w}]\!]$

A possible optimization. Note that the most complex operations of Alg. 1 and Alg. 3 — the sortings — do not depend on all inputs to these algorithms. Hence, we can reuse the shuffles computed by those sortings, as long as we read from or write to the same places, with the same priorities.

4 Discussion of performance and applicability

Using our algorithms, the cost of n parallel data accesses is $O((m+n)\log(m+n))$, where m is the size of the vector from which we're reading values. Dividing by n, we get that the cost of one access is $O((1+\frac{m}{n})\log(m+n))$. In practice, the cost will depend a lot on our ability to perform many data accesses in parallel. Fortunately, this goal to parallelize coincides with one of the design goals for privacy-preserving applications in general, at least for those where the used ABB implementation is based on secret sharing and requires ongoing communication between the parties. Parallelization allows to reduce the number of communication rounds necessary for the application, reducing the performance penalty caused by network latency.

Suppose that our application is such that on average, we can access in parallel a fraction of $1/f(m)$ of the memory it uses (where $1 \leq f(m) \leq m$). Hence, we are performing $m/f(m)$ data accesses in parallel, requiring $O(m \log m)$ work in total, or $O(f(m) \log m)$ for one access. Recall that for ORAM implementations over SMC, the reported overheads are at least $O(\log^3 m)$. Hence our approach has better asymptotic complexity for applications where we can keep $f(m)$ small.

Parallel random access machines (PRAM) are a theoretical model for parallel computations, for which a sizable body of efficient algorithms exists. Using our parallel reading and writing protocols, any algorithm for priority-CRCW PRAM (PRAM, where many processors can read or write the same memory cell in parallel, with priorities determining which write goes through) can be implemented on an ABB, as long as the control flow of the algorithm does not depend on private data. A goal in designing PRAM algorithms is to make their running time polylogarithmic in the size of the input, while using a polynomial number of processors. There is even a large class of tasks, for which there exist PRAM algorithms with logarithmic running time.

An algorithm with running time t must on each step access on average at least $1/t$ fraction of the memory it uses. A PRAM algorithm that runs in $O(\log m)$ time must access on average at least $\Omega(1/\log m)$ fraction of its memory at each step, i.e. $f(m)$ is $O(\log m)$. When implementing such algorithm on top of SMC using the reading and writing protocols presented in this note, we can say that the overhead of these protocols is $O(\log^2 m)$. For algorithms that access a larger fraction of their memory at each step (e.g. the Bellman-Ford algorithm for finding shortest paths in graphs; for which also the optimization described above applies), the overhead is even smaller.

References

[BDNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266, New York, NY, USA, 2008. ACM.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

[BSMD10] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, pages 223–239, Washington, DC, USA, 2010.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.

[DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.

[DN03] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*, pages 218–229. ACM, 1987.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[HKI+12] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon, editors, *ICISC*, volume 7839 of *Lecture Notes in Computer Science*, pages 202–216. Springer, 2012.

[HKS+10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security*, pages 451–462, New York, NY, USA, 2010. ACM.

[KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for mpc. Cryptology ePrint Archive, Report 2014/137, 2014. `http://eprint.iacr.org/`.

[LW14] Peeter Laud and Jan Willemson. Composable Oblivious Extended Permutations. Cryptology ePrint Archive, Report 2014/400, 2014. `http://eprint.iacr.org/`.

[LWZ11] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-Efficient Oblivious Database Manipulation. In *Proceedings of the 14th International Conference on Information Security. ISC'11*, pages 262–277, 2011.

[MK10] Lior Malka and Jonathan Katz. Vmcrypt - modular software architecture for scalable secure computation. Cryptology ePrint Archive, Report 2010/584, 2010. `http://eprint.iacr.org/`.

[Wak68] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, 1968.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE, 1982.