PrivTech Seminar on Secure Computation & Oblivious RAM

Aakash Sharma Gizem Gülesir
TU Darmstadt
{aakash.sharma,gizem.guelesir}@stud.tu-darmstadt.de

Abstract

All oblivious RAMs (ORAMs) are traditionally rated based on two factors: bandwidth overhead and client storage. On the contrary, recent studies on ORAM constructions show that the size of the ORAM circuit is the more effective factor for the overall performance of ORAM for practical use in secure computation protocols. Therefore, in this report we will focus on circuit complexity of several recent ORAM designs and discuss an optimized ORAM design for secure computation protocols: SCORAM. We will also discuss a secure computation framework called ObliVM that has a specific language which uses circuits to hide the details of the computation from the parties. The ObliVM compiler insulates the program into a cryptographic protocol while providing security, efficiency and automation.

1 Introduction

Secure two-party computation allows two parties to compute on their private inputs using a common function without revealing anything else except the output of the computation. All cryptographic protocol designs should ensure input obliviousness by having no correlation between the program flow and the inputs of the program. For such operations, many works have been proposed that represent the function as a Boolean circuit. However, a more simple way to represent a function is a RAM (Random Access Memory Model) circuit, where the memory accesses are kept hidden. Using traditional techniques, this requires replacing all memory indexes into a scan of the whole memory which is not efficient. In order to overcome this difficulty, Goldreich [Gol87] proposed an ORAM (Oblivious RAM) design which compiles a RAM program into a secure computation protocol and is also adopted by Gordon et al. [GKK+12]. ORAM hides away a simple memory access to an index i into a sequence of multiple memory accesses independent of the value of i. This sequence of memory accesses is completely different, even if the same index i, is accessed twice in a row.

Previous studies about ORAM have been largely focused on reducing bandwidth overhead between two parties and reducing memory utilized at both ends, in terms of (1) the client's storage requirements and (2) the server's memory footprint. Various approaches have been successful in limiting all three overheads in various combinations of $O(\log^c(n))$ where $c \in \{0, 1, 2, 3\}$.

Due to large overheads, secure computation has not been successful in large dataset deployments which use the circuit-model. The RAM model approach has more potential to scale in large dataset scenarios. Practical ORAM techniques [PR10] have shown how ORAM leads to an efficient secure computation for large datasets. Secure repetitive binary searches were demonstrated in sub-linear time. Even for run-once tasks, the RAM-model performed well in experiments on large datasets. Some secure computation problems showed dramatic increase in efficiency with the Path-ORAM technique.

1.1 Background on ORAM schemes

A binary-tree based framework for constructing ORAM schemes was first proposed by Shi. et. al. Many efficient schemes extended this approach with changes in the *eviction* strategy. In such schemes, N blocks of data are organized in a binary tree of height $L = \log N$; while each node acts as a bucket containing Z blocks of the form:

$$\{idx \| label \| data\}$$

where *idx* is index, *label* is leaf identifier and *data* is data/payload.

A *position map* that transforms memory addresses into *leaf labels* is available at the client's end. Leaf labels are randomly assigned and reassigned while accessing the blocks. Among the three operations (1) *ReadAndRemove*, (2) *Add* and (3) *Eviction* provided by ORAM schemes, *Eviction* is the key difference between different schemes.

1.2 Oblivious Programming techniques and frameworks

In oblivious programming, every memory location is set either to public or secret. Every user can execute their programs securely without seeing each others values using a cryptographic protocol. Even though the protocol secures each instruction, memory accesses and messages transferred between the parties, the program counter (memory trace), memory access addresses and the value of public variables are still vulnerable to the other parties' observation. When a program is memory trace oblivious, transforming it into a circuit representation is straightforward. Circuits can take memory accesses as an input. Therefore, as long as the program is trace oblivious, privacy can be ensured. The majority of the previous approaches about oblivious programming except SCVM [LHS+14] use the simple way which does not take into account this issue, and simply translates the accesses of the dynamic memory into a linear scan of memory on the circuit. This is too costly for huge amounts of data.

Another procedure to ensure instruction trace obliviousness is executing both branches and activating the result of only one of the branches when there is a branch operation in the program.

1.3 Contributions of [WHC+14] and [LWN+15]

In [WHC+14], theoretical analyses and experiments were conducted on optimized versions of four existing ORAM schemes. The new heuristic design called SCORAM, performed better than the other existing schemes during the experiments. As ORAM in secure computation settings differs from previous studies in storage outsourcing and secure processor execution, the traditional evaluations measure *bandwidth overhead*, which is no longer a true indicator of performance. In secure computation scenarios, the overhead due to the traditional ORAM designs for outsourcing storage can overshadow the other measures such as *bandwidth overhead*. Therefore, [WHC+14] supports the argument that the circuit complexity of ORAM is an important factor for evaluating efficiency of ORAM schemes.

The new ORAM scheme called SCORAM is verified by experiments and proved to be almost 10x smaller in circuit size than any previous ORAM scheme. Not only the performance is better than the other ORAM schemes in the secure computation setting, but it also allows to build oblivious data structures [WNL+14] for secure computation.

The ObliVM programming framework [LWN+15] has been studied for building cryptographic protocols. Various algorithms were transformed into efficient oblivious representations with minimum effort. For expert programmers, the ObliVM framework provides rich libraries even at the low-level circuit design and expressive language features to build custom cryptographic protocols.

1.4 Outline

A popular ORAM scheme called Path ORAM is discussed and issues with using Path ORAM in a secure computing scenario are presented in section 2. After pointing out the issues, we present SCORAM in section 3. We also present performance evaluation measures for ORAM in secure computation. After introducing SCORAM, we discuss the ObliVM framework which presents an opportunity for programmers to easily design cryptographic protocols in section 4. We also introduce some of the features of the ObliVM-lang in section 4. Lastly, we show the experiments done on various algorithms using the ObliVM framework and showcase how easy the ObliVM framework is to use for a novice programmer in section 5.

2 Path ORAM

In this section, we will briefly discuss the inner workings of one of the popular ORAM schemes called Path ORAM [SvDS+13]. We focus on Path ORAM because of its smaller bandwidth overhead in comparison to all tree-based ORAMs. Algorithm 1 defines the three simple steps to access an *element* along a Path P and the eviction strategy used in Path ORAM.

Algorithm 1 Path ORAM access for block x

```
1: Lookup for path $P := pos[x]$
2: Read path $P$ into $stash$
3: Try to write $stash$ back to path $P$, and pack as close
   to leaf node as possible.
```

Unlike the binary-tree ORAM scheme where one block is evicted from two random buckets at each level and put in child node, Path ORAM adopts a greedy eviction strategy that works with a *stash* maintained in client storage. Blocks on the path P are first unioned with the *stash*. Each block in the updated *stash* is then placed as close as possible to its destination leaf in path P subject to the path invariant. Finally, all blocks that were written on the path, are removed from the *stash*.

Remember that not every block from *stash* can be written to a leaf node. In case the access path P is not on the current path of an item in stash, Path-ORAM attempts to push a block to the Least Common Ancestor (LCA) of them. If the LCA level is occupied, it will attempt to write at the next higher level. Based on work done by Stefanov et al. [SvDS+13], it was determined that a bucket size of 4 is optimal for this eviction strategy.

2.1 Shortcomings of Path ORAM

Based on the experiments done by Wang et. al. [WHC+14] on ORAM designs in practical parameter settings, Path ORAM did not perform well. Unexpectedly, it performed even worse than a Binary-tree ORAM. This is caused by the eviction algorithm which requires scanning over the path. For every element in stash, the algorithm searches for a perfect match to push the bucket along the path.

With D as the number of bits in each block, i.e. payload, and N as the number of blocks in ORAM, a naive implementation of Path ORAM's eviction strategy in the circuit model results in a circuit of size $O(D \log^2 N)$, which is not efficient.

3 SCORAM

In the previous section, we discussed how Path ORAM despite all the simplicity could not perform well for secure computation. We now present, a new scheme called SCORAM by Wang et. al [WHC+14] which is an optimized version of the tree-based ORAM with an innovative eviction algorithm. We will also discuss further optimizations and the new measures which are more suitable for performance evaluation of ORAM schemes in a secure computing scenario. We will also compare SCORAM with other popular ORAM schemes based on the results obtained from experiments.

SCORAM is designed to perform better than previously available ORAM implementations. Eviction is handled by performing the *flush()* operation (Algorithm 2) α times. A greedy push pass is implemented to avoid overflows in SCORAM. In the *bucket overflow* handling strategy, after selecting a random path with every access, the deepest block along the path from *root* to *leaf-1* level is evicted. A block's eviction depends on free space availability at its child nodes.

Algorithm 2 flush()

```
1: path := UniformRandom(0,..., $N-1$)
2: bucket[0, ..., $L-1$] := array of buckets from leaf to root
3: $B_1$ := the block in the stash with smallest LCA($path$, $B_1.label$)
4: for $i$ from 0 to $L-1$ do (from leaf to root)
5:     if bucket[$i$] is not full and LCA($path$, $B.label$) $\leq i$ and $B_1$ has not been added already then
6:         Add $B_1$ to bucket[$i$]
7: for $i$ from $L-1$ to 1 do (from root to leaf)
8:     $B_2$ := the block in bucket[$i$] with smallest LCA($path$, $B_2$.label)
9:     if bucket[$i-1$] is not full and LCA($path$, $B_2.label$)$\leq i$ then
10:        Move $B_2$ from bucket[$i$] to bucket[$i-1$]
```

3.1 Other optimizations

Optimizations suggested by Wang et. al. [WHC+14] were basically on two levels (1) parameter optimization and (2) circuit level optimizations. Experiments resulted in methodologies similar to the ones suggested by Stefanov et. al. [SvDS+13]. Based on simulation results, α was determined to be optimal at value of 4. The stash size was found to be super-linear in log N. A single-bit field *isDummy* to indicate whether a block is a dummy or not was added to the design. This enabled an efficient oblivious removal of blocks from bucket.

3.2 Performance Evaluation

Wang et. al. [WHC+14] proposed new metrics for evaluating the performance of ORAM schemes in secure computing scenarios. These metrics can be categorized as *Cryptographic backend independent* metrics such as "AND gate count", *Cryptographic backened dependent* metrics such as "number of encryptions" or "bandwidth", and *Implementation and machine dependent* metrics such as "runtime".

3.2.1 SCORAM v other ORAM techniques

During the performance evaluations done by Wang et. al. [WHC+14] on existing ORAM schemes and SCORAM, SCORAM was found to perform much better than the other ORAM schemes on the new performance

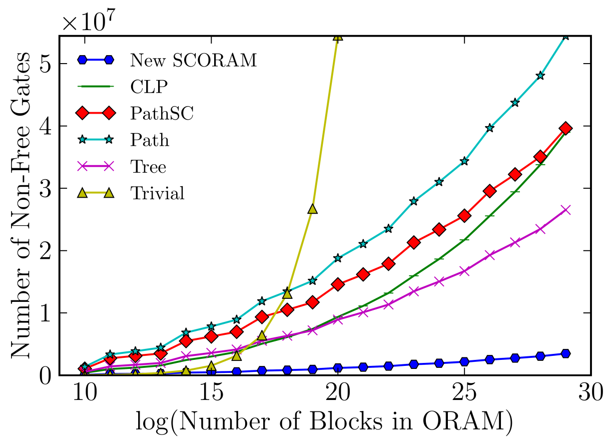

Figure 1: Comparison of various ORAMs in terms of "Number of AND gates", adopted from WHCS14

metrics. In comparison to Binary Tree ORAM which performed second best, SCORAM was 1/7th of its size and required 27 times less inputs for the same operation. The evaluations were done with the security parameter set to 80 with payload bit-length of 32 bits.

As clearly depicted in Figure 1, SCORAM outperformed existing ORAM schemes. Figure 1 shows "*Number of AND gates*" required for building various ORAM schemes which is in correlation with the growth in bandwidth. SCORAM also performed similarly better in the other measures such as size of the circuit and number of encryptions.

4 ObliVM

While the previously introduced SCORAM scheme allows the design of more efficient ORAM schemes for secure computation, implementing such a cryptographic protocol is still a tedious task best left to experts. In this chapter, we will summarize the ObliVM framework [LWN+15] which enables even non-experts to design cryptographic protocols based on ORAM and secure computation.

One of the important concerns with the design of ObliVM is memory trace obliviousness. The straightforward way to enable the Oblivious RAM (ORAM) abstraction is to make sure that all store and access operations on arrays whose access patterns depend on secret inputs are done on ORAM. Since this approach is generic (taken by SCVM [LHS+14]), it may not perform as well as the most efficient implementation for every oblivious program.

On the other hand, there are recent studies about task specific oblivious algorithms. Even though task specific algorithms outperform the generic ORAM constructions, it would be outweighed in terms of the time consumed for the development and expertise effort.

ObliVM provides a bit of both user and compiler friendly environment. While non-expert programmers are able to understand and implement in the high level programming abstraction, on the other hand, expert programmers are capable of extending the ObliVM framework by adding new features. ObliVM transforms the program abstractions into an efficient circuit implementation. New abstraction features can be added as a library in ObliVM-lang. Expert programmers can add new libraries to ObliVM-lang, which requires implementing all the primitives including basic instructions such as multiplication and addition to be implemented by the programmer. In addition to this, expert programmers can implement custom cryptographic protocols with flexibility of a backend choice (requiring not necessarily a Garbled circuit backend). Other programmers can easily reuse the custom protocols from the source language by using native types and function calls.

4.1 Development framework and efficiency

Since the ObliVM language offers ease and flexibility, it allows programmers to extend the language for custom algorithms such as streaming, machine learning and graph algorithms. As a result programming effort is reduced and efficiency is increased compared to other similar studies about oblivious programming. It is shown that ObliVM circuits are magnitudes of order smaller than circuits generated with similar applications or hand crafted designs. Besides, ObliVM generated circuits are only 0.5% to 2% bigger in size than the other oblivious algorithms hand-crafted by experts [LWN+15]. Based on the case studies, ObliVM minimizes the development effort and knowledge required in order to build secure applications.

4.2 Compilation and ORAM support

There is another concern about compile time efficiency. There is a huge compilation time difference between fully materialized circuits and on the fly generated circuits. Some first generation compilers form the code containing fully materialized circuits, which results in long compile times. On top of the long compile time, usually compilation needs to be repeated for every input data size which causes even longer compilation duration.

Algorithms		**Complexity**		
		Our Complexity	Generic ORAM	Best Known
Sparse Graphs	Dijkstra's Algorithm	$O((E+V)log^2V)$	$O((E+V)log^3V)$	$O((E+V)log^3V)$ (Generic ORAM baseline)
	Prim's Algorithm	$O((E+V)log^2V)$	$O((E+V)log^3V)$	$O(E\frac{log^3V}{loglogV})$ for $E=O(Vlog^{\gamma}), \gamma \geq 0$ $O(E\frac{log^3V}{log^{\delta}V})$ for $E=O(V2^{log^{\delta}V})$, $\delta \in (0,1)$ $O(Elog^2V)$ for $E=\Omega(V^1+\varepsilon)$, $\varepsilon \in (0,1]$
Dense Graph	Depth First Search	$O(V^2logV)$	$O(V^2log^2V)$	$O(V^2log^2V)$

Table 1: Summary of graph algorithms' cost analysis with various techniques. Results are calculated in terms of circuit size.

On the contrary, ObliVM uses on the fly circuit generation which adds insignificant overhead compared to the time required for the cryptographic protocol computation. The pipelining technique is used in order to keep the circuit, never completely materialized. This process only affects less than 1% of the total runtime.

4.3 ObliVM Language Features

The ObliVM language, which extends the SCVM language, introduces new features such as phantom functions, generic constants and random types. The security type system of the ObliVM language has to make sure that execution traces such as memory trace, instruction trace, function trace or a declassification trace are not leaking information. We introduce some of the ObliVM-lang built-in functions, types and constants.

4.3.1 Phantom functions

The key idea to provide memory and instruction trace obliviousness is avoiding leakage of information for each function body in the program. Phantom functions are used to ensure that execution flow of the program does not depend on any variable to preserve security. Each function has two modes, a real mode and a phantom mode. In the real mode, all executions are done normal. In the phantom mode, all instructions will be executed, regardless of the conditional variable. For instance, in the following code the phantom function *prefixSum* will always be executed in all cases of the if statement.

$$\text{if } (s) \text{ then } x = prefixSum\ (n);$$

When s is $false$, the $prefixSum$ function will be called but the statements inside the function will not execute. And the function generates traces with the same probability as if the condition were $true$. The code above generates following target code:

$$prefixSum(idx,\ indicator)$$

where indicator specifies whether to operate the function in the real or phantom mode. In any case (indicator is $true$ or $false$), all instructions will be executed and the traces will be identical.

4.3.2 Generic constants

ObliVM-lang has the feature of generic constants which increases code reuse. The constant variable $@m$ is used to represent a variable with m bits. For example, $int@m$ means an integer type with m bits. Since ObliVM-lang does not force to define the parameter m as hard coded constant, this results in more reusability.

4.3.3 Random types

Since obliviousness guarantees that the joint distribution of memory traces is the same independent of the secret inputs, randomness is a crucial feature of oblivious programs. A random number is always secretly shared between two parties. For example, ObliVM-lang provides a random type $rnd32$, which is a 32-bit random integer. Following, the built-in function signature RND takes 32-bit integer m as an input and returns m random bits.

$$rnd@m\ RND(public\ int32\ m)$$

When a random number is assigned to a public variable more than once, a secret random number becomes observable via correlation through the public variables it was assigned to. Therefore, for security reasons, each random number should not be declassified more than once.

5 Experiments

In this section, we will demonstrate the complexity of several graph algorithms, implemented efficiently in the ObliVM framework, and compare their complexity to Generic ORAM techniques and the previously best known implementations. We give an outline of the results in the Table 1. Additionally, for graph algorithms in ObliVM, a loop coalescing technique is used, which reduces the algorithmic complexity. In loop coalescing, the compiler splits the code of one nested bounded while loop into one normal loop with multiple blocks with branching statements. This transformation is done in two steps, and results in O(n + m) cost instead of O(nm). In the evaluation, (1) the loop coalescing technique is used on oblivious Dijkstra for shortest path compuation on sparse graphs and (2) the weight update operation is avoided, unlike for the priority-queue-based Dijkstra.

With these two variations, the oblivious single source shortest path (SSSP) algorithm has better algorithmic results compared to the other techniques. In comparison with the priority-queue-based Dijkstra's algorithm, the weights are not updated (when a shorter edge u is found between the two vertex) inside the priority queue. The new pair (newDis, u) is inserted into the priority queue (PQ) which causes multiple entries for the same vertex in the PQ. These problems bring up two issues: (1) the size of the priority queue cannot be limited to the number of the vertices; and (2) the same vertex would be popped and processed more than once. Each *insert* and *deleteMin* operation of the PQ will be bounded to O($\log^2 V$), since the size of the priority queue would be bounded by the number of edges which is O(V^2) (for sparse graphs E = O(V^2)). The lines 6-7 of the Algorithm 3 solve the second issue which makes every vertex to be processed at most once, by use of setting the distance to the vertex to a negative number once the vertex v is processed.

The outermost *for-loop* (the lines 3-18 of Algorithm 3) has constant number of ORAM accesses with priority queue operations *insert* and *deleteMin* which results in O($\log^2 V$) time. Therefore, the total runtime of the algorithm is O((V+E) $\log^2 V$). Moreover, the comparison of algorithmic results with various techniques is summarized in Table 1.

Algorithm 3 Oblivious Dijkstra's Algorithm

Secret Input: e: concatenation of adjacency lists stored in a single ORAM array. Each vertex's neighbors are stored adjacent to each other.
Secret Input: s: the source node
Secret Input: $s[u]$: sum of out-degree over vertices from 1 to u
Output: dis: the shortest distance from s to each node

```
1:  $dis := [\infty, \infty, \dots, \infty]$; $dis[source] = 0$
2:  $PQ.push(0, s)$; $innerLoop := false$
3:  for $i := 0 \to 2V + E$ do
4:      if not $innerLoop$ then
5:          $(dist, u) := PQ.deleteMin()$
6:          if $dis[u] == dist$ then
7:              $dis[u] := -dis[u]$; $i := s[u]$
8:              $innerLoop := true$
9:      else
10:         if $i < s[u+1]$ then
11:             $(u,v,w) := e[i]$
12:             $newDist := dist + w$
13:             if $newDist < dis[u]$ then
14:                 $dis[u] := newDist$
15:                 $PQ.insert(newDist, v')$
16:             $i = i + 1$
17:         else
18:             $innerLoop = false$
```

6 Conclusion

The development of SCORAM and the ObliVM framework is an important step towards achieving secure two party computation protocol in practical scenarios which involve large datasets. Also, the expressiveness and ease of use of the ObliVM programming framework would speedup adaptation of secure two party computation protocols in more scenarios. The ObliVM framework and many sample implementations are available at http://www.oblivm.com.

References

[CLP13] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with olog2n overhead. *CoRR*, abs/1307.3699, 2013.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *Proceedings of the 2012 ACM conference on Computer and communications security*, ACM, pages 513–524. ACM, 2012.

[Gol87] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Annual Symposium of Theory of Computing – STOC'87*, pages 182–194. ACM, 1987.

[LHS+14] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating efficient ram-model secure computation. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 623–638, Washington, DC, USA, 2014. IEEE Computer Society.

[LWN^{+}15] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. Oblivm: A programming framework for secure computation. In *Security and Privacy*, pages 359–376. IEEE, 2015.

[PR10] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Proceedings of the 30th Annual Conference on Advances in Cryptology*, CRYPTO'10, pages 502–519, Berlin, Heidelberg, 2010. Springer-Verlag.

[RHH14] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *Security and Privacy – SP'14*, pages 655–670. IEEE, 2014.

[SCSL11] Elaine Shi, T.-H.Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o((logn)3) worst-case cost. In *Advances in Cryptology ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 197–214. Springer Berlin Heidelberg, 2011.

[SvDS^{+}13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

[WHC^{+}14] Xiao Shaun Wang, Yan Huang, T-H. Hubert Chan, Abhi Shelat, and Elaine Shi. Scoram: Oblivious ram for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 191–202, New York, NY, USA, 2014. ACM.

[WNL^{+}14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 215–226, New York, NY, USA, 2014. ACM.