Optimizing TLB for Access Pattern Privacy Protection in Data Outsourcing

Yao Liu[1,2(✉)], Qingkai Zeng[1,2], and Pinghai Yuan[1,2]

[1] Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China
yaoliu1985@hotmail.com, zqk@nju.edu.cn
[2] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

Abstract. Oblivious RAM (ORAM) is a protocol to hide access pattern to an untrusted storage. ORAM prevents a curious adversary identifying what data address the user is accessing through observing the bits flows between the user and the untrusted storage system. Basically, ORAM protocols store user's data in shuffled form on the untrusted storage and substitute the original access with multiple access to random addresses to cover the real target. Such redundancy introduce significant performance overhead.

Traditional Translation Lookaside Buffer (TLB) exploits temporal locality hide memory latency in DRAM systems. However, the ORAM locality is totally different and thus traditional TLB eviction strategy have a poor performance. In this paper, we propose O-TLB which exploits ORAM temporal locality and optimized TLB eviction strategy to reduce server-side memory I/O operations. Intuitively, exploiting locality for performance may expose this locality which breaks obliviousness. We challenge this intuition by exploiting locality based on server-side ORAM data structures. Unlike previous works, our approach do not sacrifice any provable security. Specifically, previous optimization works leaks access pattern through timing channel and do no fit with adaptive asynchronous obliviousness (AAOB) in a multiple users scenario. While in our method, the timing do not vary with locality of program and O-TLB optimization can be adopted directly keeping AAOB. Our simulation result show that with O-TLB scheme, the underlying ORAM server-side I/O performance is improved by 11%.

Keywords: Data outsource · Access pattern privacy
Oblivious RAM · TLB · Temporal locality

1 Introduction

Outsourcing data to cloud storage become popular for its reliability, low cost and ease of management. Although the service provider can prove that the user's data is encrypted and guarantees integrity, users's access pattern is still exposed. In another word, how users access their data may lead to sensitive information leakage.

X. Lin et al. (Eds.): SecureComm 2017, LNICST 238, pp. 571–584, 2018.
https://doi.org/10.1007/978-3-319-78813-5_29

Islam et al. [9] proves an attack on searchable encryption scheme by using only access pattern is feasible. For instance, a user stores a sets of encrypted text on the cloud and make queries with various encrypted key words. With continuous observing the addresses touched, the attacker eventually reveals the linkability between different key words and trapdoors. With enough sample queries and a few known queries as prior knowledge, the attacker is capable of recovering a large number of the key words. As a real world example, by observing only access pattern to an encrypted email repository, an attacker can infer up to 80% of the queries.

Access pattern privacy leakage is also found in trusted processor and untrusted memory settings. Shinde et al. [20] show that Intel SGX is vulnerable to page fault side channel. SGX establishes an "enclaved" environment to protect user space process from potentially compromised operating system. Although the underlying OS is not able to hijack control flow of a process inside an enclaved space or extract plain text directly, the OS still manage page fault exceptions. Their experiment indicates, 27% on average and up to 100%, encryption key bits from cryptographic routines in OpenSSL and Libgcrypt can be recovered by only watching the traffic between enclaved process and memory management units.

Oblivious RAM (ORAM) is a cryptographic primitive was proposed by Goldreich and Ostrovsky in their ground breaking work [14] in the aspect of software protection. They claim that access pattern to an external storage may lead to software theft. Their square-root ORAM protocol is the first non-trivial approach to make accesses oblivious. The approach requires only $O(log(1))$ client storage size to keep the protocol flowing but incurs $O((logN)^3)$ bandwidth blowup. Follow-up works [4,6,15,21–24] make efforts to decreasing bandwidth overhead. Part of the works modify on the ORAM protocol itself, as Path ORAM [21] substitutes Goldreich's hierarchical structure with a binary search tree which shrinks the bandwidth blowup to $O(log(N))$ and amortize the reshuffling cost to each ORAM access. SSS-ORAM occupies large amount of server-side storage but has the best performance and [11] is the best solution for limited resources and small block size.

Other works refine the protocol in the aspect of implementation. Treetop caching proposed in [24] moves the hottest part of server-side storage to client for better performance. Fletcher et al. introduced PLB [4] caching most recent accessed blocks of recursive position map which dramatically reduces total number of ORAM accesses with a cost of small extra client storage. As Wang et al. [23] consider a scenario when oblivious program with intense memory accesses reside with a non-oblivious program require relatively low bandwidth. The non-oblivious program cause lots of unnecessary waiting cycles to the oblivious one. They address this problem by filling these waiting cycles with next ORAM cycles. Our work also refines the implementation of ORAM to gain practical improvement.

In this paper, we propose "Oblivious Tree-based Locality" which is completely different from original underlying principle of TLB techniques. We make

effort to optimizing TLB eviction strategy by exploiting Oblivious Tree-based Locality from random nature of underling ORAM protocols. However, exploiting data locality for performance and keep this locality hiding from curious server seem contradictory. Intuitively, programs with good locality exhibit better performance than those with poor locality which reveals program's locality to server. We challenge this intuition by speeding up every programs with equal magnitude. In this way, our approach do not sacrifice any provable security of underly ORAM. And we stress that our approach do NOT leak any information through timing channel and do not need any kind of timing channel protection. It is very important since previous optimization leaks programs' locality through timing channel. Enabling ORAM timing channel protection lead to heavy response delay which makes their approaches unpractical. Furthermore, in multiple users setting their approaches break "adaptive asynchronous obliviousness" while ours can be applied without modification.

Our Contributions, in a Nutshell:

1. Traditional TLB eviction strategy is studied in the context of ORAM. We made an observation that directly applies traditional temporal locality to server works poorly.
2. A new concept called Oblivious Tree-based Locality is proposed and we use it to built O-TLB scheme. The implementation of O-TLB is discussed in detail Path ORAM.
3. Security of O-TLB scheme is carefully examined including timing channel and adaptive asynchronous obliviousness. We prove that our scheme achieves the same level of security as underling ORAM protocols.

The rest of the paper is organized as follows: Sect. 2 gives the threaten model and settings. Section 3 provides the background knowledge of general ORAM and Path ORAM in particular. Section 4 studies traditional TLB techniques for ORAM protocols. In Sect. 5, we propose Oblivious Tree-based Locality and O-TLB for Path ORAM and discuss how underlying ORAM can benefit from our scheme. Section 6 security of O-TLB is discussed and compared to related works. Section 7 presents our evaluation methodology. The simulation result is exhibited which proves effectiveness and security of our optimization. Section 8 presents our conclusion (Table 1).

2 Threat Model

In this section, we briefly introduce the two settings and threat model for general ORAM.

2.1 Settings

Client-server setting, which is adopted in this paper, describes a scenario that a trusted client runs a private or public program on encrypted private data stored

Table 1. Notations

N	Total data block number
B	Block size in bit
id	Data block logical identifier
pos	Data block actual position in server
L	Path ORAM tree hight
Z	Path ORAM bucket capacity
l	Level of a specific node

in remote cloud server. Following other ORAM works, we assume the server is honest but curious, which means it correctly evaluate the functions and make no temper with the cipher-text. However, the adversary may continuously observe and take records on the client access opcode (read/write) and target addresses combined with cipher-text to deduct client's sensitive information.

The other setting is trusted processor with an untrusted RAM which is a Trust Computing Base (usually contains processer alone) operates in an untrusted environment for a remote user. The program runs on the TCB can be both private or public but operates on private data. When last-level cache misses, the processor interact with untrusted external memory (e.g. DRAM). The private data is encrypted when going out of TCB boundary but adversary may still tap pins on the memory bus to constantly observe the traffic between TCB and memory.

2.2 Security Definition

Informally, the security definition demands that the server learns nothing about the access pattern. Typical access patterns include:

1. whether an access is a read or write
2. which data unit is accessed and timestamp of last accessed
3. Relative between access like, whether two access refer to the same data
4. overall pattern (sequential, random etc.)

Definition 1 *(ORAM Definition). Let* $\overleftarrow{y} = ((op_1, addr_1, data_1), \ldots, (op_M, addr_M, data_M))$ *denote a data query sequence of length* M *(*$|\overleftarrow{y}| = M$*), where* op_i *denotes the opcode of the i-th operation (read or write). And* $addr_i$ *denotes the target address for that operation while* $data_i$ *denotes the data if the* op_i *is a write. Let* $ORAM(\overleftarrow{y})$ *denote the final operation sequence between client and server under an ORAM protocol which is exposed to the adversary. The ORAM protocol guarantees that for any* $\overleftarrow{y}$ *and* $\overleftarrow{y}'$, $ORAM(\overleftarrow{y})$ *and* $ORAM(\overleftarrow{y}')$ *are computationally indistinguishable when* $|\overleftarrow{y}| = |\overleftarrow{y}'|$. *Also, for any* $\overleftarrow{y}$ *the data returned to the client from ORAM is consistent with* $\overleftarrow{y}$ *(functionally equal) with overwhelming probability.*

2.3 Threats Outside of Scope

Timing Channel: Generally speaking, timing channel is not considered in ORAM studies. This is reasonable when timing change, which related to access pattern, is negligible. For example, CPU in client may take few more cycles to access a specific data block locally. This timing change will be overwhelmed by the unstable network delays. However, obvious timing changes closely couple with data locality of program is unacceptable. We will discuss this further in Sect. 6.

Active Adversary: An active adversary can temper with encrypted content breaking integrity of user data or return incorrect evaluation result to client. Many extraordinary works have already addressed this problem which is orthogonal to our work, like MAC (message authentication code) can ensure integrity of user data and correctness of function evaluation. In this paper, we only consider a passive adversary with capability of observing the traffic between server and client.

Total Number of ORAM Accesses: Total number of ORAM accesses normally is not part of access pattern. For AAOB security, it is not entirely true. Further discussion can be found in Sect. 6.

Total Server Side Storage Occupation: The server will definitely know the capacity of ORAM and thus the total ORAM data blocks is not part of access pattern protection.

3 Background

In section, we present necessary background knowledge of Path ORAM since it is representative work. Many related works use Path ORAM as underlying ORAM.

3.1 General ORAM Protocol Introduction

Basically, ORAM substitute an original access with redundant read and write operations. The ORAM is functionally equal to the original access, as user can access their data in ORAM transparently just like normal RAMs. The data is stored in encrypted and shuffled form. Once a data block is accessed by client, it must be re-encrypted with new randomness and relocate to other positions in the server and invalidates the old copy. The operation sequence exposed to server do not varies with client's input. For example, after a ORAM access is completed, whatever client's original access is, the adversary sees only ,say, ten reads and five writes to pure random addresses.

Sequential scan and reshuffling are the most fundamental technique for ORAM. Sequential scan is also referred as trivial ORAM or naive ORAM. To hide real opcode and operand, the access to a specific element from N incurs a complete sequential read and write operation to all N elements, thus, the client have to access 2N (both write and read) data block instead of one. Trivial ORAM are commonly used by ORAM protocol as component when the scale is relatively small. Reshuffling means that once a element is accessed, its real address is exposed to the adversary. It must be relocated somewhere else immediately to prevent the adversary tracing the element. So all ORAM protocols, except trivial ORAM, maintain a lookup table that mapping data block's *id*, which can be seen by client only, to actual position *pos* on the server.

Base of ORAM Randomness: Pseudo-Random Functions (PRFs) are used as the base of ORAM randomness. They provide random numbers for relocating data block and keys for one time encryption. The most important point is that the output of these function do NOT vary with user's input (include write or read, target address, write data) or else the whole protection falls apart. The actual outcomes of our O-TLB optimization for one specific access sequence is wholly decided by output of PRFs and thus have no relationship with user's input.

Evaluation Metric: Usually, using ORAM incurs relatively large overhead. To evaluate the performance of a ORAM protocol, there are two different method.

1. Theoretical Evaluation: Bandwidth blowup and Client Storage are introduced. Basically, bandwidth blowup is how much extra data blocks that the ORAM have to access to hide access pattern. In trivial ORAM, we say the protocol incurs a $O(N)$ asymptotic bandwidth blowup. While the client storage overhead refers to the minimum client space to keep the ORAM running.
2. Practical Evaluation: Some works refining the implementation of ORAM usually do not have an asymptotic improvement. Their optimization is exhibited in a probabilistic manner [4]. Server responding time are used to evaluate these improvement as mention in Sect. 1. Our O-TLB is among this type of work.

3.2 Path ORAM

Path ORAM was introduce by Shi et al. which is adopted by lots of works as underlying ORAM protocol for its extreme simpleness and low bandwidth blowup. Path ORAM departures from Goldreich's square-root ORAM, substitute hierarchical structure with a binary tree for more natural search process. Another highlight is that Path ORAM couples write back with path fetches which amortizes the overhead to every ORAM access. We will presents Path ORAM in two aspect.

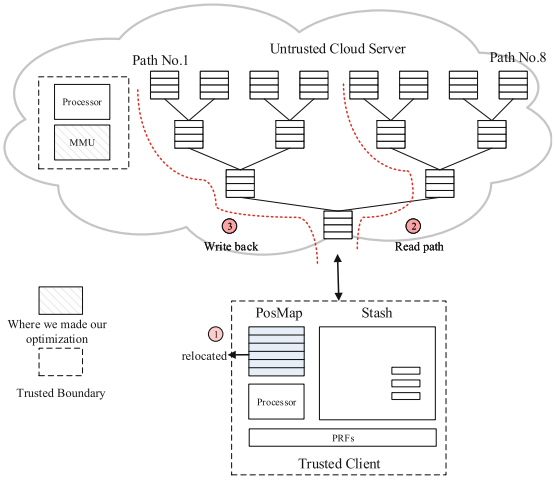

Fig. 1. Path ORAM protocol

Data Organization Protocol: Data Organization Protocol define that how the algorithm organized the data blocks and stored. Path ORAM organize the data blocks as a binary search tree. As figure shows in Fig. 1, Path protocol organizes N blocks to form a complete binary tree with level $L = LogN$. Each node of the tree have Z slots and also called buckets. One bucket is capable to contains up to Z data blocks with a triple of form:

$$\{id||pos||data\}$$

Where id is a private index of a data block revealed only to client, while pos is a leaf identifier specify the "path" on which the block is located, as $data$ is the payload. The term "path" refers to all buckets from leaf node to the root, marked with dash lines in Fig. 1. Finally, Path ORAM maintains a core invariant: If a data block is stored on the server, it must be found on one bucket from the path specified by pos.

Data Block Access Protocol: One complete Path ORAM access consists of tree operations, as indicated in Fig. 1.

1. Relocating: Once the ORAM protocol accepts an access request, the initial step is to get the mapping from id to pos and immediately substitute it with a new random pos'.
2. Read Path: With the mapping, the ORAM load all buckets along the path to stash and decrypt them to find the data block with identifier id. The core invariant guarantees the designated data block can be found.
3. Write Back: The client re-encrypts all blocks with new randomness key and evicts all blocks with pos to current path, greedily fill the buckets from leaf to

root. Remember that the target block is relocated to path tagged with pos', there is a good chance that $pos \neq pos'$. Such that the block stays in stash and waits for another access to pos' to get itself evicted.

Each path have $logN$ buckets and each of them have Z slots such that the protocol roughly have to access $2Zlog(N)$ data blocks to ensure obliviousness. Thus, the asymptotic overhead is $O(logN)$.

4 Traditional TLB on ORAM

In this section, we first review the key points for traditional TLB technique and discuss the problems if applies it to ORAM directly.

Modern Memory Management Unit (MMU) built an abstract layer called virtual memory on top of physical memory for easier allocation, management and security purpose. Before access to a desired physical memory location, MMU need to look up the page table, which is also a regular chunk of the RAM, to translate virtual memory to physical memory. This page table walk is recursive and introduces several extra accesses to the RAM. Since memory access is very fundamental and frequent, the overhead of page translation is significant.

MMU adopt Translation lookaside buffer (TLB) to reduce the delay taken to access a user memory location. The TLB caches a fixed number of the most recent translation result. If a memory location is invoked, the first step is to go though TLB to find whether there is a corresponding entry. If TLB hits, then the expensive page walk operation is skipped. Since the size of TLB is small and fixed, the number of cache entry is very limited. The principle behind the cache is temporal locality. In another word, reasonable TLB hit rate is based on the observation that if a location is invoked then there is a good chance it will be accessed again in short future. This is no longer true if DRAM is substituted by ORAM.

Recall that reshuffle technique, it ensures even several client queries are point to the same block with id, the server will see accesses to random position in the server memory. For traditional TLB in server processor, this is totally against its anticipation. More generally, higher cache hit rate means the actual access pattern is very close to the anticipated access pattern which defines the cache behavior. As such, we make the observation that traditional TLB works poorly for ORAM.

5 Oblivious TLB

In this section, we propose our ORAM locality definitions and use them to make optimizations.

5.1 Oblivious Tree-Based Locality

Based on the observations above we propose **Oblivious Tree-based Locality**:

Target location of tree-base ORAM is purely random and independent. The nodes close to the root have good locality while nodes close to the leaves have poor locality. Equal level nodes have equal locality.

Ongoing Locality is also proposed:

Inside an ORAM access, target eviction path have good locality and will be accessed in short term.

5.2 O-TLB for Path ORAM

As Fig. 1 shows, since each Path ORAM access loads an entire path from root to the leaf, the access possibility for touching each node can be denoted by 2^{-l}. Instead of evict the oldest translation entry, we use a tree level bits to keep track of which level every page belongs to which tree level.

Please recall Path ORAM protocol steps. It loads a whole path, where the corresponding page translation entry is cache with TLB, and the path will be evicted. That means the pages inside the path will be accessed again shortly. We exploit this locality for performance. We maintain a ongoing bit indicate whether the page belong to a unfinished path. When a LoadPath operation is detected, ongoing bit of all TLB entries for this path is set.

When the O-TLB is full and one entry should be evict. O-TLB picks one entry with lowest value (most close to leaf) and check its ongoing bit. If the ongoing bit is cleared, then the entry is evicted. Otherwise, O-TLB picks the second lowest one. Entries with same level bit is chosen randomly, as according to Oblivious Tree-based Locality, this do not hurt performance.

6 Related Works and O-TLB Security Analysis

In this section, related work is presented and explains how O-TLB achieves same security of underlying ORAM.

6.1 Dynamic Super Block Technique

Some works [24,26] exploit ORAM spatial locality for performance. They borrowed the memory prefetch technique from model system. Originally, client data blocks is independent and distribute randomly in the path of ORAM tree storage structure on the server. The key point of their method is that client data blocks exhibiting spatial locality are united to form super block and stored to one path as a whole. As such, once a block, which belongs to a super block, is loaded to the client, other blocks reside in the same super block is loaded as well. If the program do have spatial locality, then it is very likely that next desired data block is already loaded by last ORAM access. A reduction of total ORAM access number surely buys performance gain.

6.2 Access Pattern Leakage from Timing

Although their improvement is significant, we think their optimization is unrealistic. They made a optimization on *client side*, specifically on client data blocks which is extremely tightly coupled with access pattern. The timing should severely varies with locality of program. For example, if the client silent for a period of time, then the adversary may make a good guess that the client is running a program with good locality.

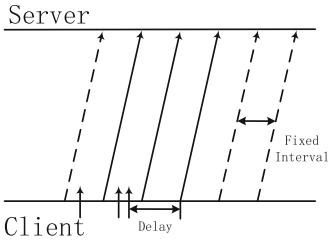

Fig. 2. Periotic timing protection

Although they claim periotic ORAM issue can be mounted to protect timing, we think this incurs too heavy delay as shown in Fig. 2. The idea of periotic ORAM timing protect is simple that one ORAM access is issue in regular interval. If no request is in the queue then a dummy access is issued. The requests for client are no long served in a on-demand way, but they have to wait for next access slot. At present, the research community has turned single client model to multiple. TaoStore [17] describe a multi-client model with proxy as shown in Fig. 3. Concurrency leads to a intense ORAM access requests, where periotic timing protection will incur severe delay. If shorten the interval, lots of resource is wasted when the clients are idle because of dummy ORAM access. If vary interval dynamically, this too leaks locality through timing. On the contrary, our work is a *server side* optimization. This nature ensures O-TLB improves all client program equally. Although, the actual performance gain is decide random number generated by client's PRFs. However underly ORAM must guarantees the output of PRFs do not varies with query parameters. This ensures our optimization have no relationship with access pattern.

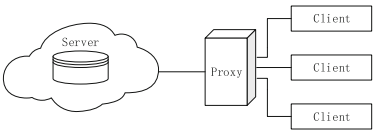

Fig. 3. TaoStore model

6.3 Against Adaptive Asynchronous Obliviousness Security

TaoStore propose Adaptive Asynchronous Obliviousness which is first complete adversary model for multiple-client ORAM model. The relevant key point is once client issue a request or accept query result the adversary is notified. As such, Dynamic Super block optimization will cause the number of request of client differs from the number of proxy request, which immediately leaks the locality. Because proxy may get some desire data blocks from super block to answer clients' request.

While O-TLB only shorten the cost of server PathLoad and Write back process and do not vary the total number of ORAM access. So our optimization do not break AAOB and can be incorporated by multi-client with proxy ORAM model like TaoStore.

7 Evaluation

In this section evaluation settings is given and simulation result is exhibited which proves both efficiency and security.

7.1 Methodology

Graphite [10] is used as the simulator in our experiments. The hardware configurations and ORAM settings are listed in Table 2. We choose a ORAM block size of 1k for simplification, because four 1k blocks form a 4k bucket ($Z = 4$) which equal to regular page size. We use Splash-2 benchmark because programs in the benchmark have different locality to test if our optimization gain varies with programs locality.

Table 2. Processor core configuration

Core	1GHz in order core
L1 Cache	32 KB 4-way
L2 Cache	512 KB 8-way
Cacheline size	64 bytes
TLB	64 entries 4-way

7.2 Metrics

We stress that we do not test overall completion time of programs. We only test how many store and load operations the server memory is saving due to O-TLB hit.

7.3 Result

As Fig. 4 exhibited, the performance gain is around 11% and do not vary with locality of programs.

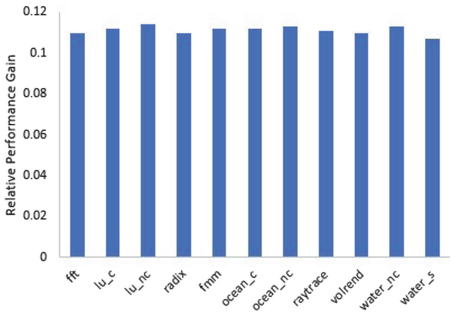

Fig. 4. O-TLB relative performance gain

8 Conclusion

We study and make the observation that traditional TLB works poorly on ORAM. We propose new locality definitions for ORAM and use it to build O-TLB. Related works are presented and compared with O-TLB. We prove O-TLB optimization do not hurt security of underling ORAM in any form, especially timing channel. Furthermore, O-TLB can be easily adopted with multi-client settings and varies of tree-like ORAM protocols. Simulation is made and the result supports our claims.

Acknowledgment. We would like to thank the anonymous reviewers for their constructive and helpful comments.

This work has been partly supported by National NSF of China under Grant No. 61772266, 61572248, 61431008.

References

1. Blass, E.O., Mayberry, T., Noubir, G., Onarlioglu, K.: Toward robust hidden volumes using write-only oblivious RAM. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS 2014, pp. 203–214. ACM (2014). https://doi.org/10.1145/2660267.2660313
2. Dautrich, J., Stefanov, E., Shi, E.: Burst ORAM: minimizing ORAM response times for bursty access patterns. In: 23rd USENIX Security Symposium (USENIX Security 2014), pp. 749–764. USENIX Association (2014). https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/dautrich
3. Devadas, S., van Dijk, M., Fletcher, C.W., Ren, L., Shi, E., Wichs, D.: Onion ORAM: a constant bandwidth blowup oblivious RAM. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9563, pp. 145–174. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49099-0_6
4. Fletcher, C.W., Ren, L., Kwon, A., van Dijk, M., Devadas, S.: Freecursive ORAM: [nearly] free recursion and integrity verification for position-based oblivious RAM. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, pp. 103–116. ACM (2015)

5. Fletchery, C.W., Ren, L., Yu, X., Dijk, M.V.: Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In: IEEE International Symposium on High PERFORMANCE Computer Architecture, pp. 213–224 (2014)
6. Gentry, C., Goldman, K.A., Halevi, S., Julta, C., Raykova, M., Wichs, D.: Optimizing ORAM and using it efficiently for secure computation. In: De Cristofaro, E., Wright, M. (eds.) PETS 2013. LNCS, vol. 7981, pp. 1–18. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39077-7_1
7. Goodrich, M.T., Ohrimenko, O., Tamassia, R.: Data-oblivious graph drawing model and algorithms. Computer Science (2012)
8. Gordon, S.D., Liu, F.-H., Shi, E.: Constant-round MPC with fairness and guarantee of output delivery. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 63–82. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48000-7_4
9. Islam, M., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: ramification. attack and mitigation. In: Proceedings of NDSS (2012)
10. Kasture, H.: Graphite: a parallel distributed simulator for multicores. In: IEEE International Symposium on High PERFORMANCE Computer Architecture, pp. 1–12 (2010)
11. Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in)security of hash-based oblivious ram and a new balancing scheme. In: SODA (2012)
12. Liu, C., Hicks, M., Shi, E.: Memory trace oblivious program execution. In: Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium, CSF 2013, pp. 51–65. IEEE Computer Society. https://doi.org/10.1109/CSF.2013.11
13. Maas, M., Love, E., Stefanov, E., Tiwari, M., Shi, E., Asanovic, K., Kubiatowicz, J., Song, D.: PHANTOM: practical oblivious computation in a secure processor. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, pp. 311–324. ACM (2013). https://doi.org/10.1145/2508859.2516692
14. Ostrovsky, R.M.: Software protection and simulation on oblivious rams
15. Ren, L., Fletcher, C., Kwon, A., Stefanov, E., Shi, E., Dijk, M.V., Devadas, S.: Constants count: practical improvements to oblivious RAM. pp. 415–430
16. Ren, L., Yu, X., Fletcher, C.W., Van Dijk, M., Devadas, S.: Design space exploration and optimization of path oblivious ram in secure processors. ACM SIGARCH Comput. Archit. News **41**(3), 571–582 (2013)
17. Sahin, C., Zakhary, V., Abbadi, A.E., Lin, H., Tessaro, S.: TaoStore: Overcoming asynchronicity in oblivious data storage, pp. 198–217
18. Shi, E., Chan, T.-H.H., Stefanov, E., Li, M.: Oblivious RAM with $O((\log N)^3)$ worst-case cost. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 197–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25385-0_11
19. Shi, E., Stefanov, E., Papamanthou, C.: Practical dynamic proofs of retrievability. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, pp. 325–336. ACM. (2013). https://doi.org/10.1145/2508859.2516669
20. Shinde, S., Chua, Z.L., Narayanan, V., Saxena, P.: Preventing page faults from telling your secrets. ACM (2016)
21. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious RAM protocol. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, pp. 299–310. ACM (2013)

22. Stefanov, E., Shi, E.: ObliviStore: high performance oblivious cloud storage. In: Security and Privacy, pp. 253–267
23. Wang, R., Zhang, Y., Yang, J.: Cooperative path-ORAM for effective memory bandwidth sharing in server setting
24. Wang, X.S., Huang, Y., Chan, T.H.H., Shelat, A., Shi, E.: SCORAM: oblivious RAM for secure computation. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS 2014, pp. 191–202. ACM (2014)
25. Wang, X.S., Nayak, K., Liu, C., Chan, T.H.H., Shi, E., Stefanov, E., Huang, Y.: Oblivious data structures. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS 2014, pp. 215–226. ACM (2014)
26. Yu, X., Haider, S.K., Ren, L., Fletcher, C.: PrORAM: dynamic prefetcher for oblivious RAM. In: ACM/IEEE International Symposium on Computer Architecture, pp. 616–628
27. Zahur, S., Wang, X., Raykova, M., Gascon, A., Doerner, J., Evans, D., Katz, J.: Revisiting square-root ORAM: efficient random access in multi-party computation, pp. 218–234