A Comparative Study of Privacy-Preserving Computation Techniques

Contrasting ORAM, MPC, TEEs, Structured Encryption, and Homomorphic Encryption

Sergiu-Nicolae Stancu

Supervisor: Lilika Markatou

[1]**EEMCS, Delft University of Technology, The Netherlands**

Abstract

Outsourcing data to the cloud can pose serious security threats due to an attacker observing the data access patterns, even though data is encrypted. Ideally, confidentiality should not depend on the server being a trusted party. Oblivious Random Access Machines are tackling this problem by obfuscating data access patterns and ensuring obliviousness of the server towards the data.
Thus, this paper studies the evolution of Oblivious Random Access Machines and highlights the steps taken so far to discover a more practical algorithm for hiding data access patterns on an untrusted server. In addition, other approaches for privacy-preserving computation, such as Homomorphic Encryption, Structured Encryption, Multi-Party Computation and Trusted Execution Environments are discussed and contrasted to ORAM to assess the costs and benefits they come with, but also the trade-offs in security and usability.

1 Introduction

As the world increasingly depends on data, and outsourcing storage becomes cheaper, security becomes a greater concern. This happens because the storage server can observe the interactions that the client has with its encrypted data. Although encryption is normally enough to ensure confidentiality of data at rest, when data is accessed multiple times, the server can infer details like loop structure or branching of the running program. Islam et al. [33] showed that data access patterns alone can reveal up to 80% of queries on data. Another example, used by Pinkas and Reinman [56] is that a malicious server can tell which information is more important based on the frequency of accesses and then concentrate its resources towards decrypting only the data that are repeatedly accessed. This becomes increasingly problematic when the underlying data contains sensitive information, such as private health or financial details. Thus, the need for techniques that allow working with encrypted outsourced data but also maintain confidentiality arises.
Thus, any client who chooses to store sensitive data in the cloud needs to be aware of its risks and find a way to hide the data access patterns for the server. One proposed solution to this problem of data access patterns leaking information is Oblivious Random Access Machines. ORAM aims to make all program data access patterns look the same from the perspective of an untrusted server observing the memory locations that are accessed. This idea was first introduced by Goldreich and Ostrovsky [23], who came up with two solutions that lay out a foundational concept for future research. Then multiple constructions improved the initial efficiency with the aim of making ORAM a practical and usable solution to hide data access patterns and to ensure zero leakage of information.
This paper presents the main contributions of different ORAM techniques proposed thus far, following the continuous improvements that the field has benefited from to reach the current state-of-the-art constructions. Then, Path ORAM [66] will be chosen for a comparison with other Privacy-Preserving Computation techniques, given it is a general purpose ORAM that has already been adopted by some secure processors [17, 43, 18], for its low worst-case bandwidth blowup, but also its simplicity. Then, the survey will discuss differences in security, usability, efficiency, and functionality, but also way in which they can complement each other. Thus, in Section 2, the background and methodology used to carry out the investigation will be discussed. Then, Section 3 will present in detail the innovations and technical aspects raised in the relevant papers. Then, in Section 4, characteristics such as functionality, efficiency, security, and usability will be isolated and highlighted based on the facts presented in Section 3. After that, Section 5 introduces Homomorphic Encryption, Structured Encryption, Multi-Party Computation and Trusted Execution Environments and compares them according to the criteria defined in Subsection 5.2. Then, ways in which they can complement each other or fit specific use cases are discussed. In the end, ethical considerations and responsible research aspects are presented, ending with the conclusion and acknowledgment of the work that remains to be done in the field.

2 Motivation

As technology advances quickly and becomes more complex, there is more infrastructure that we need to protect and more possible ways for attackers to exploit weaknesses in our systems. That is why security cannot be just an afterthought, but has to be a priority while designing a system. Oblivious Random Access Machines tackle part of this problem by ensuring zero leakage of data in an outsourced storage setting.
As pointed out in previous research by Chow et al. [11], there are many potential cloud users that have yet to join the cloud, but refrain from doing so because of the perceived risks. Major corporations are mostly putting only their less sensitive data in the cloud. Their main concern that is brought up is the lack of control over their data and this is what prevents cloud from becoming even more popular. Thus, this paper studies the evolution of ORAM and highlights the steps taken so far to discover a more practical algorithm to hide data access patterns on an untrusted server. In addition, other approaches for privacy-preserving computation, such as Homomorphic Encryption, Structured Encryption, Multi-Party Computation and Trusted Execution Environments are discussed and contrasted to ORAM to assess the costs and benefits they come with, and the trade-offs in terms of security and usability. The purpose of this is not only to serve as an objective overview of some Privacy-Preserving Computation Techniques available so far, but also to act as an inspiration for future research on ORAM or suitable use-cases for different techniques. Thus, the research questions are "how did ORAM evolve to its current state and what are the state-of-the-art constructions", but also "how does it compare to FHE, StE, MPC and TEEs in terms of functionality, efficiency, security and usability".

2.1 Background

ORAM was first introduced by Goldreich in 1987 in the paper *Towards a Theory of Software Protection and Simulation* *by Oblivious RAMs* [22] which proposed a theoretically interesting algorithm that makes data access patterns oblivious to a malicious server. After that, Ostrovsky [52] improved the theoretical bounds of Goldreich, coming up with a novel solution that used a hierarchy of hash tables. Initially, the focus was on minimizing the amortized cost [23, 68, 27, 56, 8] and ignored the worst-case cost, making them impractical, but provided an interesting result and yet another stepping stone in the field. A huge change occurred with the first techniques that achieved sublinear worst-case cost, most notably Stefanov et al.[65] who proposed a new partitioning scheme, Goodrich et al. [29] who investigated a way to de-amortize the initial ORAM constructions [23], but also Kushilevitz et al. [39] who improved on the Cuckoo Hashing [54] idea introduced by Goodrich and Mitzenmacher [27]. Then, Shi et al. [61] used a novel binary tree-based technique that was later used by Stefanov et al. in the famous Path ORAM [66] paper . Path ORAM will be the main technique used in the comparison with other Privacy-Preserving Computation methods for its simplicity and ease of understanding, more techniques and recent work will be analyzed. The main contribution of each paper will be presented and explained in order to provide a starting point for someone researching the field, but also give an overview of the techniques proposed thus far which can be used as an inspiration for future constructions.

2.2 Methodology

The study was conducted as a literature survey, having Path ORAM [66] as a reference work for ORAM. Following the references used in the work by Stefanov et al. [66], the evolution path of ORAM could be traced back and analyzed in detail. Then, to gauge the more recent direction that research in the field has taken, queries on Scopus were used, from which a few relevant papers were selected and analyzed. From this point on, references were again followed back to fill the gap between the formulation of Path ORAM and the present work.
For the comparison part, meetings were held in which a person for each of the five techniques studied in this paper was present, presenting the costs and trade-offs of the technique, but also sharing papers where different techniques were combined.

3 ORAM Constructions

3.1 Problem Definition

A trusted client with small storage is assumed to execute a program on an untrusted server as can be seen in Figure 1. The server stores data in encrypted blocks, and the communication between the two parties happens in tuples of the form (operation, block id, data) that are sent over the network.

The **security definitions** of [65] are adopted. Therefore, no information should be leaked about which data is being accessed, when was the last time a block was accessed, whether the same data are being accessed, whether the access is a read or write, or if the access pattern is sequential or random. N will be the number of blocks outsourced to the server and the blocks will be of size B bytes each. More formally, if

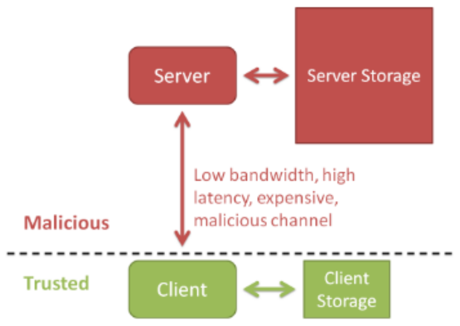

Figure 1: Overview of ORAM System Architecture [65]

the access pattern $A(x)$ is the sequence of accesses to the remote untrusted server that stores the data blocks, then an oblivious RAM is considered secure if for any two inputs x and x' having the same length, the access patterns $A(x)$ and $A(x')$ are computationally indistinguishable for anyone but the client as mentioned in Definition 1 from Pinkas and Reinman [56].
Overhead is defined as the number of blocks that must be accessed on the server for a single read or write to take place. Usually, overhead is an upper bound function of the number of blocks outsourced to the server. Overhead may be referred to as cost, and it is worth noting that it may be analyzed from two perspectives: the amortized cost and the worst-case cost. In addition, client storage is considered to take into account different capabilities of clients and be able to suit different needs. Server storage is also a metric that is highlighted in different ORAM constructions, since it may be an important part to consider when looking at the practical considerations of a scheme. An overview of some foundational ORAM schemes that will be discussed in more detail can be observed in Table 1.

3.2 Square Root Solution

The Square Root Algorithm proposed by Goldreich in 1987 [22] and was the first non-trivial ORAM that managed to achieve an amortized overhead of $O(\sqrt{N}\log^2 N)$ with a worst-case cost of $O(N\log^2 N)$, which is also the worst-case cost of construction. To make the simulation oblivious, the author first proposed splitting the server storage into $N + 2\sqrt{N}$ blocks. The first $N + \sqrt{N}$ blocks were reserved for the N outsourced blocks and $\sqrt{N}$ dummy blocks. The remaining $\sqrt{N}$ blocks are used as a "shelter" that will temporarily store the accessed data.
The algorithm is executed in epochs of size $\sqrt{N}$. As a setup for an epoch, the client needs to shuffle the $N + \sqrt{N}$ real blocks and dummy blocks. To do that, the algorithm uses an oblivious Sorting Algorithm [5] that incurs a $O(Nlog^2(N))$ overhead. It is worth mentioning that an asymptotic upper bound of $O(Nlog(N))$ can be achieved using the AKS Sorting Network [1], but it is less practical due to higher constants.

To access an item during an epoch, the client scans the entire shelter. In case it is in the shelter, then it looks for the next dummy item, in order to hide the actual block that was read, otherwise it accesses the item corresponding to the permutation that was last used. After the item is read, the CPU writes to the shelter, which can be real or fake depending on the previous read. Thus, from the server's perspective, it looks like the client scans the whole shelter every time, then does a random lookup for one of the $N + \sqrt{N}$ blocks. To prevent the server from linking a block that was read with one that was written at a different location, probabilistic encryption is used [25]. Also, the client makes sure that it does not access the same dummy block twice. Thus, the amortized overhead is $O(\sqrt{N}log^2(N))$, since the reshuffling happens once every $\sqrt{N}$ steps.

To reduce overhead, the client can store the stash in a smaller ORAM and amortize the sorting costs over more steps, managing to achieve an amortized overhead of $O\left(2^{\sqrt{2\log_2 N \cdot \log_2(\log_2 N)}}\right)$ for constant client storage. Another important theorem from the paper is the $O(logN)$ theoretic lower bound for client-server communication.

3.3 The Hierarchical Solution

In 1990, Ostrovsky came up with a novel technique [52] that improved the bounds of the initial solution proposed by Goldreich [22]. Here, outsourced storage is split into $\log N$ hash tables, where each hash table $i \in \{0, 1, 2, 3, \ldots, \log_2 N\}$ contains 2^i buckets. Each hash table uses its own hash function H_i and the invariant of the algorithm is that a block x will always reside in a level i, between 0 and $\log_2 N$ in the bucket $H_i(x)$. To access a block, the algorithm fully scans the first level, then it reads a bucket in each level. If the block is already found, the algorithm still does an access at every level to ensure obliviousness through a dummy read. Otherwise, it looks in the $H_i(x)$ bucket. After reading an item, it always writes it back to the first level.

At a certain point, the first level might fill up. That causes the need for a reshuffling, in which items from the first level are written obliviously to the second level. Still, the simulation cannot reveal any information about the actual load of a level, therefore, this reshuffling happens every 2^i accesses on the server. Thus, after 2^i steps, the algorithm has to merge levels *i* and *i+1* and and use a new hashing function for level *i+1*. This is done with the AKS Sorting Network [1] which obliviously sorts N elements in $O(NlogN)$ steps. However, the constants of the AKS Sorting Network make it less practical than the Batcher's Sorting Network [5]. Thus, the scheme reaches an amortized cost of $O(log^3N)$ for constant client storage, but has a worst-case cost of $O(Nlog(N))$. Also, server storage is $O(NlogN)$ due to the space complexity of the sorting algorith.

3.4 SSS Construction

The construction by Stefanov, Shi and Song from 2012 [65], which was later referred to as the SSS construction, is another considerable improvement since it was the first one to achieve a logarithmic overhead for the worst-case cost. They proposed a novel partitioning scheme, where the storage would be split into partitions which were functioning as smaller ORAMs. The benefit of this construction is that it improved the best ORAM construction available at that moment by a factor of 63. However, the client storage was actually $O(N)$ with a very small constant that, in practice, resembled a $O(\sqrt{N})$ construction. The main idea is that, after every read or write, a block gets reassigned to a new partition, and there is a background eviction service that makes sure that blocks end up in the correct partition after they were read. Each partition worked as a hierarchical ORAM, similar the one proposed by Goodrich and Mitzenmacher [27]. In addition, an important improvement in overhead came from the larger client storage, which can now shuffle a partition locally, but also the idea of spreading the reshuffling over multiple operations to improve online overhead, which was first introduced by Ostrovky and Shoup [53]. Stefanov et al. [65] also proposed a theoretical construction that actually achieves client storage of $O(\sqrt{N})$ by storing the position map on the server and accessing it recursively, but comes with another factor of $log(N)$ for amortized and worst-case overhead.

3.5 Binary Tree ORAMs

An important addition to the ORAM field was the *Oblivious RAM with $O(log^3N)$ access overhead* solution proposed by Shi et al. [61] in 2011. This paper used a binary tree construction that later inspired Stefanov et al. [66] in their seminal work about Path ORAM. However, this was not the first time a binary tree was used in an ORAM construction, since Damgård et al. [13] used a binary search tree in a theoretically interesting solution that did not rely on a random oracle. The idea is that the server is laid out as a binary tree, where each node in the binary tree is a fully functional ORAM of capacity $O(logN)$. Then, each block is randomly assigned a leaf node in the binary tree and will reside somewhere along the path from the root to that leaf. Every time a block is accessed, the client reads the entire path from the root to the leaf, randomly assigns a new leaf to the block, and finishes with writing the accessed block to the root node.

The client stores a data structure that assigns each block id to one of the leaves, but this information can also be outsourced to the server, incurring an additional overhead of $O(logN)$ in bandwidth, but lowering the client storage to a constant size. To ensure that the buckets do not overflow, an eviction scheme is running in the background, where for each level from 1 to $log_2(N)$, a certain number of nodes are selected for eviction, according to an eviction parameter. Then, for the selected nodes, a write happens to both of its children to make sure that it does not reveal any information to the server. It should be noted that due to the randomness in eviction and assigning of new leaves, an overflow can occur with low probability, depending on the bucket size and eviction parameter. Thus, the worst-case cost of this construction becomes $\tilde{O}(\log^3 N)$, assuming the bucket ORAMs use the Hierarchical Construction proposed by Ostrovsky [52] and that the position map is stored on the server.

As mentioned above, this construction was used as a stepping stone for Stefanov et al. in the Path ORAM paper [66].

This paper was influential not only for its reduced worst-case overhead compared to earlier constructions, but also for its simplicity and applicability. For example, it was also used in secure processor architecture, with Ascend [17] being the first processor to implement ORAM to protect against memory access pattern leakage. Then Ren et al. [58] proposed some optimizations that could be made to achieve a realistic overhead and to move a step closer to practicality.

Path ORAM used a stash and stored the entire path from the root to the corresponding leaf, when reading a block from the server. Then contents of the stash are evicted as other accesses happen, making sure that every block stays on the path from the root to its corresponding leaf or in the stash. Another change happened to the nodes of the binary tree, as they are now of constant size Z, which is usually 4 or 5. Again, the position map has to be stored on the server to keep the client storage low, so the final overhead depends on the size of the blocks, having a $O(logN)$ worst-case overhead for blocks $\Omega(log^2N)$ bits and a $O(log^2N)$ worst-case cost for blocks of size $\Omega(logN)$ bits. Both constructions use a relatively small amount of client storage, $O(\log N)\omega(1)$ and $O(ZN)$, where Z is constant, but it is worth noting for practical implications.

3.6 Other notable ORAM contributions

In 2010, Pinkas and Reinman [56] proposed an ORAM construction that first used a hierarchy of Cuckoo hash tables [54] and the Randomized Shellsort algorithm proposed by Goodrich [26]. Their idea combined the initial Hierarchical Solution [52] proposed by Ostrovsky with the newer tools of Cuckoo Hashing and Randomized Shellsort. This influenced future authors, but also provided a solution that improved the amortized overhead to $O(log^2N)$ with smaller constants thanks to the new sorting algorithm, but also faster look-ups in the hash tables, compared to linear search in a bucket of logarithmic size in the case of the Hierarchical Solution. That is because in the case of Cuckoo hashing, only two hash tables are stored with a hash function for each and an element will reside in exactly one of the two tables at either $h_1(x)$ in the first, or $h_2(x)$ in the second table, where h_1 and h_2 are the corresponding hash functions for the tables.

The idea of using Cuckoo hashing in this setting brought a new problem, as pointed out by Kushilevitz et al. [39] in 2012. The authors proved that information can be leaked when a hash table overflows, because the access patterns that happen in a non-overflowing hash table may not be consistent with the random ones that take place in the case of a random hash function. However, Pinkas and Reinman still inspired Goodrich et al.[28] who used their Cuckoo Hash Table idea to which they added a stash, following the work of Kirsch et al. [37] from 2009 who suggested the addition of a shared stash of size $O(logN)$ to make the Cuckoo hashing more robust by decreasing the probability of rehashing. In their construction, the client did not have to maintain state, so making it suitable for a multiuser setting and also improved the amortized cost, achieving the lower bound introduced by Goldreich and Ostrovsky [23]. However, the worst-case cost was still linear and made the scheme unpractical, while the client storage was also larger, being $O(n^v), v > 0$. Thus, the scheme by Goodrich et al. proved to be yet another construction of theoretical interest that lacked practicality.

ObliviStore [64] is another construction that focuses on improving the practicality of ORAM and was proposed in 2013. To this extent, ObliviStore makes use of parallelism through asynchronous operations, an oblivious load balancer on the client-side and a scheduler that makes sure the shuffling done in a communication round does not exceed $O(logN)$. The authors used the partitioning scheme proposed by Stefanov et al. [65] an year earlier and implemented the ORAM scheme in C#, achieving an I / O overhead of 40-50 blocks to access a single block, while also adapting easily to a distributed setting. However, asynchronous additions pose a new security threat, as an attacker can now infer knowledge of stored data based on the timing of accesses. This is mitigated by using semaphores whose behavior depends only on information that is observable by the server. In addition, the scheme uses a deterministic eviction schedule to ensure that the server does not infer any knowledge based on that.

Following the ideas from ObliviStore, Burst ORAM [14] in 2014 focused on the online performance of ORAM which was first mentioned in the work of Boneh et al. [8] in 2011. Thus, Dautrich et al. [14] use three queues for scheduling, one for read jobs, one for write jobs, and one for new jobs. The idea is that they prioritize reads and use a novel bandwidth reduction technique to achieve constant time responses even in the case of bursty traffic. Here, instead of returning all blocks that are accessed for a read, they apply an xor operation to the real and all the dummy blocks that are read from the server, so that the client can easily obtain the real block once retrieved. This XOR technique manages to reduce the overall bandwidth to 23-26 blocks returned per access, improving the earlier results of ObliviStore [64]. A disadvantage of this construction is the larger client storage which is $O(\sqrt{N})$ compared to logarithmic client-side storage in the case of other constructions such as Path ORAM [66].

After Burst ORAM, Ring ORAM [57] was proposed in 2014, which aimed to provide a better overall bandwidth for both small and large client storage sizes. It used the tree-based approach of Path ORAM [66], while adding the XOR technique mentioned by Dautrich et al. [14] and made use of de-amortization techniques for evictions. That made Ring ORAM faster than Path ORAM for the cost of higher overall complexity of the scheme.

4 Characteristics

4.1 Functionality

ORAM is generally used for scenarios when one or multiple clients outsource encrypted data to one or multiple remote repositories. Normally, it is assumed that the clients are trusted and they need to outsource the data to untrusted servers, for cheaper storage. Then, the communication between the client and the server happens through read and write requests where blocks of encrypted data are sent. In order to perform some computation on the data, the server has to retrieve a block, decrypt it, compute, and then re-encrypt the block and send it back to the server. The problem is that usually, the storage provider cannot be fully trusted to store and observe the interactions that the client has with the data, since Islam et al. [33] prove that a substantial amount of information can be inferred by an honest but curious attacker, just by looking at the data access patterns. Thus, ORAM hides this information through probabilistic encryption and by constantly moving of the data blocks. It usually assumed that the remote server is not used for other computations, making the communication one-way, with the client being the one initiating the requests. Thus, initial ORAM schemes [22, 52] followed this idea for a single client and a single server, but newer constructions deviated from this setting, for example Goodrich et al. [28] proposed a solution for multiple clients to access a server without maintaining state from one access to another. Other ORAM schemes such as ObliviStore [64] focused on the case of distributed storage, which is more common in practice. Then, authors extended ORAM schemes for malicious servers as well, with Apon et al. [2] who formalized the definition of VOS, *Verifiable Oblivious Storage*, providing integrity with the use of Fully Homomorphic Encryption. However, this technique failed to be a viable solution due to the bootstrapping which incurs a huge overhead on the server-side. More recently, Hoang et al. [31] proposed a technique that uses authenticated secret sharing to ensure integrity of the data, extending ORAM functionalities to active adversaries as well which works in a distributed server setting.

4.2 Efficiency

Efficiency of ORAM schemes can be approached from multiple angles. Most importantly, an ORAM scheme needs to consider the amortized bandwidth blow-up, the worst-case bandwidth blow-up, client and server storage, and also block size. Initially, ORAM techniques aimed to minimize the amortized bandwidth used for the communication between the server and the client. This ignored the worst-case cost which can make the techniques unpractical most of the time. Later, the focus shifted towards minimizing worst-case cost, as that would mean a technique could be used all the time without incurring a significant overhead on some accesses. Also, Goldreich [22] proved that ORAM's bandwidth has a lower bound of $O(logN)$ blocks. As bandwidth is usually limited, most research tried to close this gap between the worst-case cost and the theoretical lower bound proposed by Goldreich [22]. This meant that client storage, server storage, or block size, were not always top priorities when designing a new ORAM scheme. Thus, there are multiple ORAM techniques that manage to achieve the lower bound of $O(logN)$ introduced by Goldreich [22], but with different assumptions or caveats. For example, Path ORAM [66] has a worst-case cost of $O(logN)$ blocks only when the blocks are of size $\Omega(log^2N)$ and client storage is logarithmic in terms of the number of blocks. Ring ORAM [57] and Burst ORAM [14] achieved the same worst-case cost, but for blocks that are poly-logarithmic in size and a logarithmic client storage. Later, attempts towards closing the efficiency gap have been made with PanORAMa [55] that achieves an amortized cost of $logN * loglogN$, for any block size $B = \Omega(logN)$ bits and a constant number of blocks stored on the client side. The $loglogN$ gap was erased by OptORAMa [4] which proposed the first optimal oblivious RAM scheme for blocks of size $B = \Omega(logN)$ and constant client storage using a hierarchy of hash tables and efficient multi-array shuffling. However, OptORAMa [4] hides a huge constant due to the use of certain expander graphs in its construction, making it impractical at the moment.
Thus, different techniques suit different use cases and a user may choose a different implementation based on its needs. For example, Burst ORAM can achieve constant bandwidth blowup in scenarios where a user focuses more on online bandwidth, rather than overall bandwidth. Also, there are constructions that aim to improve the efficiency of other aspects. For example, Octopus ORAM [42] focused on minimizing server storage trading it for slightly higher client-side storage. Also rORAM [9] uses data locality to make an efficient scheme for performing range queries.

4.3 Security

The purpose of ORAM is to hide data access patterns and provide a software-based solution to accessing data stored in an untrusted server. Being a software-based solution means that ORAM can be vulnerable to different side-channel attacks. For example, at attacker may observe the frequency of data access patterns, or may measure how long each access take and based on that, tell if it was a real or a fake access. Besides, a malicious server may be able to infer knowledge based on power consumption or electromagnetic emissions. Even though ORAM can achieve zero leakage in theory, when it is deployed on hardware, other issues can arise, which are generally not considered in the ORAM papers. Another potential problem is that ORAMs usually use pseudo-random generation functions, meaning that the algorithm's behavior is not deterministic and there is a probability that the stash, in case of Path ORAM [66] can overflow which would result in a malfunction of the technique. However, the probability of the stash overflowing is inversely polynomial in terms of the number of blocks, making it negligible. As for the server, it was usually assumed that servers follow the honest-but-curious adversary model, but Hoang et al. [31] showed how authenticated secret sharing can be used to achieve integrity in the malicious adversary scenario.

4.4 Usability

As mentioned earlier, ORAM hides the data access patterns considering an untrusted server. This can have multiple applications and can contribute towards minimizing leakage in different settings. Path ORAM [66] was already implemented in various secure processors, thanks to its simplicity and relatively low overhead. For example, Fletcher et al. [17, 18] proposed a secure processor architecture that hides which instruction is being run at any point in time. This can be particularly useful in cases where an untrusted server runs a proprietary encrypted program which should not disclose its structure. Also, Maas et al. [43] made a step towards practical oblivious processors with the proposal of Phantom that made use of parallelism and memory controllers to improve performance. Besides, it can also be used to build secure query processors for selections, joins, grouping and aggregations in databases as shown by Arasu and Kaushil [3]. Later Eskandarian and Zaharia [16] proposed ObliDB, another oblivious database engine. In addition, ORAM can also be used in distributed file systems to maintain and protect against malicious adversaries, as shown by Hoang et al. [31]. Subsections 5.3, 5.4, 5.5 and 5.6 discuss more use cases where ORAM can complement other privacy-preserving computation techniques to achieve different goals.

5 Comparison

The comparison will be made with 4 other techniques, namely Homomorphic Encryption, Structured Encryption, Secure Multiparty Computation and Trusted Execution Environments. They all act as important tools in the landscape of computation and managing of encrypted data, so the comparison aims to better highlight the nuances of each and help a reader understand which one is best for different scenarios, but also show how they can complement each other to achieve a certain goal. First, each technique is briefly introduced in Subsection 5.1, followed by a pairwise comparison. The main points of comparison are described in Subsection 5.2 and two tables, providing a high-level comparison are present in the Appendix. First table 2 displays the functionality and usability and the second one 3 shows some important aspects of security and performance.

5.1 Overview of Techniques Compared to ORAM

Homomorphic encryption generally refers to application of functions directly on encrypted data, without the need of decryption. Initially, the concept of *privacy homomorphism* was proposed, which allowed some operations on encrypted data. [59]. Later, other *Partial Homomorphic* techniques were proposed which allowed to perform either additions or multiplications for a limited number of times. After that, *Somewhat Homomorphic* techniques were discovered, which managed to do both additions and multiplications, but for a limited amount of times. In 2009, Craig Gentry [20] discovered the first *Fully Homomorphic* scheme that allowed an unlimited amount of computations, thanks to a technique called bootstrapping. Thus, the comparison will focus on FHE, for being the most relevant and capable technique of the aforementioned.

Structured Encryption is a cryptographic technique that allows efficient querying over encrypted data. It was inspired by *Secure Searchable Encryption* [63] which allowed static key-word searches on encrypted documents. Later, more advanced techniques were proposed, which allowed querying on matrices, labeled data and graphs, but were still static as they did not allow insertions nor deletions of data. Later, Kamara et al. [35] introduced the first dynamic schemes, allowing secure insertions and deletions to the outsourced data, while also achieving optimal efficiency. More recently, Kamara and Moataz [34] proposed a solution for encrypting multi-maps without leaking information related to the size of the responses.

Secure multi-party computation is a technique that is used when multiple parties try to compute a function based on everyone's input, without revealing this input to any of the other parties. An initial solution to this problem involving two parties was introduced by Yao [69] with garbled circuits. Then, Goldreich et al. [45] solved the problem for more than two parties. Then, Ben-Or et al. [67] proposed another algorithm that uses secret sharing and worked even when some of the parties were malicious. Later, Yao's garbled circuits idea was generalized for multiple-parties by Beaver et al. [6].

Trusted Execution Environments provide a different solution to secure computation, leveraging hardware-based isolation, rather than relying solely on a cryptographic algorithm. A trusted execution environment aims to provide confidentiality and integrity for both the data and the code loaded inside it even in the presence of a malicious operating system. However, a caveat of this approach is that the manufacturer has to be trusted to implement secure hardware and provide an attestation mechanism for the user. *Intel SGX* [12] is a popular implementation of TEEs based on secure enclaves. Later, Keystone [40] provided an open source solution focused on transparency and customizability. Other TEEs, such as Intel TDX [47] and AMD SEV [36] , relied on Confidential Virtual Machines [47], making TEEs easier to adopt, thanks to their ability to provide hardware-enforced isolation, without the need to modify existing applications or operating systems.

5.2 Criteria definition

The comparison with FHE, StE, MPC and TEEs will be made based on the 4 subsections of Section 4, namely Functionality, Efficiency, Security and Usability.

Functionality will contrast the different capabilities of different techniques, by looking at how they work and what exactly do they offer for the user. Here, several advantages and disadvantages will be highlighted to better help the reader understand which is more suitable for a specific scenario.

Efficiency is the second criterion and refers to the overhead or additional computations that are required to adopt a specific technique. It will also talk about the additional storage that is required for different parties in order for an algorithm or solution to work correctly. This aims to provide a clear overview of the costs of adopting a certain solution in order to help decide on its feasibility in specific scenarios.

Security considers the amount of information that is leaked by different techniques, what are their adversary models, but also mentions discovered vulnerabilities and probability of failure if that exists. This is an important part of the comparison, since it displays the existing threats and limitations.

Usability is the last aspect that is considered for comparison and aims to highlight different use cases and settings in which a solution was already adopted or otherwise mention what were the limitations or drawbacks that prevented the method from being adopted. This focuses more on the applicability of the techniques which were discussed more from a theoretical perspective until this point.

5.3 ORAM and Fully Homomorphic Encryption

ORAM and FHE perform slightly different tasks, given FHE tackles the problem of performing computations directly on encrypted data, without the need to decrypt it first. In this sense, FHE allows a client to perform any computation on the data stored on the server. On the other hand, ORAM just hides the access patterns of the client-server interaction. In order to perform a computation, a client needs to retrieve a specific block, decrypt it, do the computation, then re-encrypt the resulting block and send it back to the server, but also make sure that the whole process is oblivious to the server. This surely impacts the performance of an ORAM scheme, mainly because of the network latencies that it would have to deal with to transport all blocks. To avoid this, FHE tries to compute without decrypting first. The drawback is that performing these computations directly on encrypted data take a lot of time due to the polynomial operations involved, but also the costly key generation algorithm. This makes it an interesting theoretical concept, but fails to find its practicality for most use cases at the moment. However, it may still be used in scenarios where time constraints are more relaxed and having zero leakage is important, or when it is important to compute a function directly without the server knowing any of the outcomes.

However, an interesting theoretical application stands in outsourcing computation to the server in the case of ORAM, thus reducing communication overhead to a constant. This was proposed as an improvement to ORAM techniques that managed to achieve the lower bound of $\Omega(\log(N))$ blocks proposed by Goldreich [22], such as Ring ORAM [57] for blocks of size $\Omega(log^2 N)$. Although some of the ORAM constructions implicitly used server-side computation, most traditional ORAM techniques assumed that the client is the one doing the computations, while the server only acts as storage that retrieves blocks based on the requests of the client. This idea was challenged by Gentry et al. [21], who initially proposed using Fully Homomorphic encryption to improve bandwidth cost. After that, multiple ORAM techniques tried to make use of server-side computation to improve previous ORAM bounds. For example, Mayberry et al. [44] combined ORAM with Private Information Retrieval [38] protocols to mitigate the weaknesses of each and improve overhead for large block sizes. Later, the first ORAM to achieve constant bandwidth blowup was the Onion ORAM [15], which used either Additively Homomorphic Encryption or Somewhat Homomorphic Encryption for selections and evictions. Of course, this reduced the bandwidth to constant, but computations on the server became the bottleneck of the process, while the blocks had to be of size $\Omega(log^6 N)$, which is infeasible in practice. This was still an influential piece of work for its novel approach to ORAM, complementing it with HE techniques, which paved the path towards practical sublogarithmic bandwidth blow-up schemes. It inspired multiple authors such as Moataz et al. [48] or Chen et al. [10] to improve block size or server computation through different HE techniques.

5.4 ORAM and Structured Encryption

ORAM and Structured Encryption are generally used for outsourcing data of a client to an untrusted server. The difference in functionality is that StE usually stores data that have a specific structure, for example, a multimap or a matrix, compared to ORAM which stores it in encrypted blocks. Another difference is related to the security of each scheme. In case of Structured Encryption, the server can see the access patterns of the data and may also be able to infer keyword frequencies or link different keywords based on the access patterns. ORAM, on the other hand, does not leak any information in its protocol, but does all of it with a cost. Thus, a classic ORAM scheme would incur a communication overhead of at least $log(N)$ blocks, which are needed for the obfuscation of data access patterns, while StE only needs to send the query which may have different sizes, depending on its nature. This makes Structured Encryption usable in scenarios where a certain amount of leakage is tolerated. For example, MongoDB [49] uses StE for its Queryable Encryption [50] feature, which allows a user to run expressive queries over encrypted data. Another advantage of StE is its ability to query different data structures efficiently, which might suit various scenarios. However, it is worth noting that it only works against a honest-but-curious adversary.

As mentioned earlier, searching on encrypted data leaks data access patterns, and that can be a security issue for some applications. Garg et al. [19] showed how Garbled RAMs [41] can be used to achieve a constant-round asymptotically efficient Secure Searchable Encryption scheme that also hides data access patterns. This had an overhead λ times higher than Path ORAM [66], where λ is the security parameter. Boldyreva and Tang also showed how Path ORAM [66] can complement StE to hide query, access, and volume patterns when outsourcing an encrypted multi-map.

5.5 ORAM and Secure Multi-party Computation

ORAM and Secure Multi-Party Computation perform different tasks in the context of privacy-preserving computations. Mainly because MPC allows parties to compute a function between multiple parties without any of them revealing their input, while ORAM usually works in a client-server setting, where the client sends read and write requests to the server. Because of its functionality, ORAM assumes that the client is a trusted party and only deals with a semi-honest or malicious server. On the other hand, MPC protocols do not make the assumption that a certain party is trusted but ensures the output is correct given certain assumptions, such as the number of malicious parties. In terms of security, none of the techniques should leak information through their functionalities, but an attacker may be able to infer information through physical attacks on the local machines where the protocols are deployed. For efficiency, MPC is highly dependent on the complexity of the function being computed, while ORAM has a clear overhead depending on the ORAM scheme used. Based on its capabilities, MPC comes with different use cases than ORAM, as it can be used for auctions [51] or financial data analysis [7], which is an important step forward for research while maintaining the confidentiality of sensitive information.

ORAM and MPC can also complement each other. For example, Gordon et al. [30] showed how a two-party computation protocol can extend the Shi et al.[61] binary tree ORAM to make the server find the answer to its query and nothing else about the server's storage. This was a step forward in the privacy-preserving computation landscape because the client does not need to be a trusted party in order for the computation to occur correctly. Another interesting application was shown by Lu and Ostrovsky [41], who used Yao's garbled circuits idea [69] to efficiently obtain garbled RAM programs. This allowed a client to send the garbled RAM to the server and run it securely without further interaction. In addition, MPC protocols can be used to improve Path ORAM in certain scenarios. For example, Hoang et al. [32] used Shamir Secret Sharing [60] and a MPC protocol to obtain a constant bandwidth blowup ORAM in a distributed setting where there are at least 3 servers.

5.6 ORAM and Trusted Execution Environments

ORAM and TEEs both tackle the problem of maintaining privacy while a computation happens, but they approach it from different perspectives. ORAM provides a software-based cryptographic technique to hide data access patterns in a client-server setting, while TEEs offer a hardware-based solution that relies on vendors to create and implement an isolated execution environment. The purpose of TEEs is to provide a security guarantee even when a computation happens on a server that cannot be trusted. TEEs allow any computation to happen, as they provide a secure enclave where code can be run without getting tampered with by a malicious operating system. However, a TEE will have small memory and will have to work with the memory of the untrusted server. This becomes a potential source of leakages as the server can still observe the access patterns on data, even though it is encrypted. Thus, it allows the server to perform computations securely and provide integrity through attestation mechanisms. Thus, TEEs can ensure protection even against malicious servers without a trade-off in efficiency. Thus, they do not incur significant overhead, compared to a software-based solution such as ORAM, but the manufacturer has to be a trusted party so conflicts of interest can arise. As, with Structured Encryption, ORAM can complement TEEs to hide the data access patterns. For example, Eskandarian and Zaharia [16] built an oblivious query processing engine that supports aggregation, joins, insertion, deletion, and point queries by combining ORAM with Intel SGX [12]. This improvement was to extend the functionality previously proposed by Mishra et al. [46], which explored oblivious indexes using ORAM and acted as a stepping stone for the secure contact discovery feature of Signal [62]. Signal used Path ORAM and secure enclaves, such as Intel SGX [12] to ensure users can discover which of their contacts use Signal, without the app being able to construct the social graph with identities.

6 Responsible Research

It should be noted that the information presented in this paper can have a significant impact if a reader decides on a particular technique or algorithm to solve its problem. This can not only incur financial losses, but also reputational damage for a company in case the decision is based on untruthful claims. Thus, great care was taken in collecting and objectively evaluating previous work in the field. For that, only peer-reviewed papers were considered for the survey and a close collaboration was maintained between the student and the supervisor to mitigate the risk of false information being emitted in the survey.

In addition, it should be mentioned that there are no conflicts of interest with the authors mentioned in the report, but also with the authors not mentioned in the report. The survey started with the 10 most cited papers related to Oblivious RAM according to Scopus queries and the seminal work of Stefanov et al.[66]. From there, a snowballing approach was followed to ensure the knowledge gap to current papers is filled, but also the continuous progress is presented. Security, usability, functionality and efficiency were the most important keywords that helped in choosing the papers to investigate. Then, other Scopus queries were used to gather papers where multiple techniques were combined.

7 Conclusions and Future Work

This survey investigated the evolution of Oblivious Random Access Machines and positioned them in the landscape of Privacy-Preserving Computation Techniques, namely Fully Homomorphic Encryption, Structured Encryption, Secure Multi-party Computation, and Trusted Execution Environments. The main research question was how did ORAM reach the current state and it was shown how different ORAM schemes optimize for different tasks, but most of them were inspired by either the initial Hierarchical Solution by Ostrovsky [52] or the binary tree construction of Shi et al. [61]. ORAM can provide strong protection against data access pattern leakage which is overlooked by multiple other cryptographic techniques. Among ORAM constructions, Path ORAM [66] stands out due to its simplicity and adoption in real-world systems such as secure processors or oblivious database engines.

The comparative analyses showed that FHE excels in scenarios requiring direct encrypted computation, but suffers from significant computation complexity. On the other hand, it could complement ORAM and improve brind down bandwidth costs to constant. StE allows for efficient queries on encrypted data structures, but leaks data access patterns and can use ORAM to hide that. MPC provides strong security guarantees and does not require a trusted party, but can require a lot of communication for computations to happen. TEEs offer an efficient solution to preserve privacy, but require a trusted manufacturer.

Notably, the paper highlights how hybrid solutions combining multiple techniques can build systems with stronger security guarantees. For example, ORAM and TEEs can make an efficient and oblivious query engine, as shown by ObliDB[16]. There are several promising paths for future research in the field. For example, an improvement in FHE techniques can make ORAM widely adopted and ensure it can efficiently work against malicious servers. In addition, more attention can be brought towards side-channel resilience in case of ORAM and software-based techniques. Besides that, more recent work can be analyzed, as there are papers that are continuously published in the field and this survey stopped at papers published in 2024.

Table 1: Comparison of ORAM Constructions

Scheme	**Block Size**	**Amortized Cost**	**Worst-case Cost**	**Client Storage**	**Server Storage**
Ostrovsky [52]	$\Omega(1)$	$O(\log^3 N)$	$O(Nlog(N))$	$O(1)$	$O(N\log(N))$
SSS [65]	$\Omega(\log N)$	$O(\log^2 N)$	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(N)$
Binary Tree ORAM [61]	$\Omega(\log N)$	$\tilde{O}(\log^2 N)$	$\tilde{O}(\log^3 N)$	$O(1)$	$O(Nlog(N))$
Goodrich-Mitzenmacher [27]	$\boldsymbol{\Theta(1)}$	$O(\log^2 N)$	$O(N\log N)$	$O(N^\alpha), \alpha < 1$	$\boldsymbol{O(N)}$
Path ORAM[66] (B = $\boldsymbol{\Omega(\log^2 N)}$)	$\Omega(\log^2 N)$	$O(\log N)$	$O(\log N)$	$O(\log N)\omega(1)$	$O(ZN)$
Path ORAM[66] (B = $\boldsymbol{O(\log N)}$)	$\Omega(\log N)$	$O(\log^2 N)$	$O(\log^2 N)$	$O(\log N)\omega(1)$	$O(ZN)$
Ring ORAM[24, 57]	$\boldsymbol{\Omega(\log^2 N)}$	$\boldsymbol{O(\log N)}$	$\boldsymbol{O(\log N)}$	$O(\log N)$	$O(ZN)$
Burst ORAM[14]	$\Omega(\log^2 N)$	$\tilde{O}(\log N)$	$\tilde{O}(\log N)$	$\tilde{O}(\log N)$	$O(ZN)$

$\tilde{O}(f(N))$ is used to hide the polynomials of log(log(N)), so $O(f(N)polylog(log(N))$

Z is a parameter denoting the number of blocks in a node of the binary tree

A Appendix A: Comparison Table of ORAM techniques

B Appendix B: Other Privacy Enhancing Techniques

Table 2: Functionality and Usability Comparison of Privacy-Enhancing Techniques

Technique	Computation Type	Parties Communication	Applicability	Use Cases
FHE	Any computation	Non-interactive Client–server	Available in open-source libraries	Medical data analysis Recommender systems Confidential ML
MPC	General computation (excluding specialized protocols)	Multiple clients or distributed parties	Used in practice but with limitations	Secure auctions DNA comparison Collaborative research
ORAM	Data access	Non-interactive Client(s)–server(s)	Used in secure processors and oblivious DBs	SGX integration ObliDB, Signal protocol
StE	Specific data access on encrypted structures	Non-interactive Client–server	Practical protocols for specific structures	Encrypted DBMS (e.g. MongoDB)
TEE	Any computation	Interactive Client–server with attestation service	Optional in real world cloud deployment	Data analytics Trusted AI workloads Medical Federated Learning

Table 3: Security and Performance Comparison of Privacy-Enhancing Techniques

Technique	Threat Model	Information Leakage	Performance Overhead
FHE	IND-CCA2 Adaptive attack	None by itself	High: Key Generation & Polynomial Operations
MPC	Semi-honest or malicious	Nothing beyond function output	Constant or Linear
ORAM	Semi-honest or malicious	Leakage through side-channel attacks	Logarithmic
StE	Semi-honest	Access pattern sometimes response volume	Sublinear
TEE	Malicious actor controlling server	Access patterns, plaintext in CPU	Generally near-native, bottleneck in I/O heavy

References

[1] Miklós Ajtai, Janos Komlos, and Endre Szemerédi. An o(n log n) sorting network. volume 3, pages 1–9, 01 1983.

[2] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable Oblivious Storage. In Hugo Krawczyk, editor, *Public-Key Cryptography – PKC 2014*, pages 131–148, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[3] Arvind Arasu and Raghav Kaushik. Oblivious Query Processing, 2014.

[4] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal Oblivious RAM. *J. ACM*, 70(1):4:1–4:70, December 2022.

[5] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring)*, page 307, Atlantic City, New Jersey, 1968. ACM Press.

[6] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing - STOC '90*, pages 503–513, Baltimore, Maryland, United States, 1990. ACM Press.

[7] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying Secure Multi-Party Computation for Financial Data Analysis. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security*, pages 57–64, Berlin, Heidelberg, 2012. Springer.

[8] Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote Oblivious Storage: Making Oblivious RAM Practical.

[9] Anrin Chakraborti, Adam J. Aviv, Seung Geol Choi, Travis Mayberry, Daniel S. Roche, and Radu Sion. rORAM: Efficient Range ORAM with O(log2 N) Locality. In *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA, 2019. Internet Society.

[10] Hao Chen, Ilaria Chillotti, and Ling Ren. Onion Ring ORAM: Efficient Constant Bandwidth Oblivious RAM from (Leveled) TFHE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 345–360, London United Kingdom, November 2019. ACM.

[11] Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. Controlling data in the cloud: outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, page 85–90, New York, NY, USA, 2009. Association for Computing Machinery.

[12] Victor Costan and Srinivas Devadas. Intel SGX Explained, 2016. Publication info: Preprint.

[13] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly Secure Oblivious RAM without Random Oracles. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, and Yuval Ishai, editors, *Theory of Cryptography*, volume 6597, pages 144–163. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. Series Title: Lecture Notes in Computer Science.

[14] Jonathan Dautrich and Elaine Shi. Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns.

[15] Srinivas Devadas, Marten Van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM. In Eyal Kushilevitz and Tal Malkin, editors, *Theory of Cryptography*, volume 9563, pages 145–174. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. Series Title: Lecture Notes in Computer Science.

[16] Saba Eskandarian and Matei Zaharia. ObliDB: oblivious query processing for secure databases. *Proceedings of the VLDB Endowment*, 13(2):169–183, October 2019.

[17] Christopher W Fletcher. Ascend: An Architecture for Performing Secure Computation on Encrypted Data.

[18] Christopher W. Fletcher, Marten Van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 3–8, Raleigh North Carolina USA, October 2012. ACM.

[19] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: Efficient Oblivious RAM in Two Rounds with Applications to Searchable Encryption. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 563–592, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[20] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, Bethesda MD USA, May 2009. ACM.

[21] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and Using It Efficiently for Secure Computation. In Emiliano De Cristofaro and Matthew Wright, editors, *Privacy Enhancing Technologies*, pages 1–18, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[22] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM conference on Theory of computing - STOC '87*, pages 182–194, New York, New York, United States, 1987. ACM Press.

[23] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[24] Oded Goldreich, M. Sahai, and D. Ishai. Ring oram: A new approach to oblivious ram. In *Proceedings of the 31st Annual International Cryptology Conference (CRYPTO 2014)*, volume 8617 of *Lecture Notes in Computer Science*, pages 288–307. Springer, 2014.

[25] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, April 1984.

[26] Michael T. Goodrich. Randomized Shellsort: A Simple Data-Oblivious Sorting Algorithm. *Journal of the ACM*, 58(6):1–26, December 2011.

[27] Michael T. Goodrich and Michael Mitzenmacher. Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In Luca Aceto, Monika Henzinger, and Jiří Sgall, editors, *Automata, Languages and Programming*, volume 6756, pages 576–587. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. Series Title: Lecture Notes in Computer Science.

[28] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-Preserving Group Data Access via Stateless Oblivious RAM Simulation. In *Proceedings of the 2012 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 157–167. _eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611973099.14.

[29] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 95–100, Chicago Illinois USA, October 2011. ACM.

[30] 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*, pages 513–524, Raleigh North Carolina USA, October 2012. ACM.

[31] Thang Hoang, Jorge Guajardo, and Attila Yavuz. MACAO: A Maliciously-Secure and Client-Efficient Active ORAM Framework. In *Proceedings 2020 Network and Distributed System Security Symposium*, San Diego, CA, 2020. Internet Society.

[32] Thang Hoang, Ceyhun D Ozkaptan, Attila A Yavuz, Jorge Guajardo, and Tam Nguyen. S3oram: A computation-efficient and constant client bandwidth blowup oram with shamir secret sharing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 491–505, 2017.

[33] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2012.

[34] Seny Kamara and Tarik Moataz. Encrypted multi-maps with computationally-secure leakage. *IACR Cryptol. ePrint Arch.*, page 978, 2018.

[35] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 965–976, Raleigh North Carolina USA, October 2012. ACM.

[36] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 13:12, 2016.

[37] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More Robust Hashing: Cuckoo Hashing with a Stash*. *SIAM Journal on Computing*, 39(4):1543–1561, 2009. Num Pages: 19 Place: Philadelphia, United States Publisher: Society for Industrial and Applied Mathematics.

[38] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 364–373, October 1997. ISSN: 0272-5428.

[39] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (In)security of Hash-based Oblivious RAM and a New Balancing Scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, Philadelphia, PA, January 2012. Society for Industrial and Applied Mathematics.

[40] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Steve Lu and Rafail Ostrovsky. How to Garble RAM Programs? In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 719–734, Berlin, Heidelberg, 2013. Springer.

[42] Qiumao Ma and Wensheng Zhang. Octopus oram: An oblivious ram with communication and server storage efficiency. *ICST Transactions on Security and Safety*, 6(20), 2019.

[43] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*, pages 311–324, Berlin, Germany, 2013. ACM Press.

[44] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Efficient Private File Retrieval by Combining ORAM and PIR, 2013. Publication info: Published elsewhere. Unknown status.

[45] Silvio Micali, Oded Goldreich, and Avi Wigderson. How to play any mental game. In *Proceedings of the Nineteenth ACM Symp. on Theory of Computing, STOC*, pages 218–229. ACM New York, 1987.

[46] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An Efficient Oblivious Search Index. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 279–296, San Francisco, CA, May 2018. IEEE.

[47] Masanori Misono, Dimitrios Stavrakakis, Nuno Santos, and Pramod Bhatotia. Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 8(3):1–42, December 2024.

[48] Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. Constant Communication ORAM with Small Blocksize. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 862–873, Denver Colorado USA, October 2015. ACM.

[49] MongoDB, Inc. Queryable encryption features, 2024. Accessed: 2025-06-16.

[50] MongoDB, Inc. Queryable encryption features, 2024. Accessed: 2025-06-16.

[51] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 129–139, Denver Colorado USA, November 1999. ACM.

[52] Rafail Ostrovsky. Efficient Computation on Oblivious RAMs (Extended Abstract).

[53] Rafail Ostrovsky and Victor Shoup. Private information storage. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 294–303, 1997.

[54] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[55] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMa: Oblivious RAM with Logarithmic Overhead. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 871–882, October 2018. ISSN: 2575-8454.

[56] Benny Pinkas and Tzachy Reinman. Oblivious RAM Revisited. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, and Tal Rabin, editors, *Advances in Cryptology – CRYPTO 2010*, volume 6223, pages 502–519. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. Series Title: Lecture Notes in Computer Science.

[57] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Ring ORAM: Closing the Gap Between Small and Large Client Storage Oblivious RAM.

[58] Ling Ren, Xiangyao Yu, Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors.

[59] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

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

[61] Elaine Shi. Oblivious RAM with O((log N)3) Worst-Case Cost.

[62] Signal. Signal contact discovery, 2024. Accessed: 2025-06-17.

[63] Dawn Xiaoding Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 44–55, May 2000. ISSN: 1081-6011.

[64] E. Stefanov and E. Shi. ObliviStore: High Performance Oblivious Cloud Storage. In *2013 IEEE Symposium on Security and Privacy*, pages 253–267, Berkeley, CA, May 2013. IEEE.

[65] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. 2012. Cited by: 141.

[66] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. *Journal of the ACM (JACM)*, 65(4):1–26, 2018.

[67] Avi Wigderson, MB Or, and S Goldwasser. Completeness theorems for noncryptographic fault-tolerant distributed computations. In *Proceedings of the 20th Annual Symposium on the Theory of Computing (STOC'88)*, pages 1–10, 1988.

[68] Peter Williams and Radu Sion. Usable pir. In *NDSS*, pages 139–152, 2008.

[69] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, October 1986. ISSN: 0272-5428.