Efficient Oblivious Substring Search via Architectural Support

Nicholas Mainardi
Politecnico di Milano – DEIB
Milano, Italy
nicholas.mainardi@polimi.it

Davide Sampietro
Politecnico di Milano – DEIB
Milano, Italy
davide.sampietro@mail.polimi.it

Alessandro Barenghi
Politecnico di Milano – DEIB
Milano, Italy
alessandro.barenghi@polimi.it

Gerardo Pelosi
Politecnico di Milano – DEIB
Milano, Italy
gerardo.pelosi@polimi.it

ABSTRACT

Performing private and efficient searches over encrypted outsourced data enables a flourishing growth of cloud based services managing sensitive data as the genomic, medical and financial ones. We tackle the problem of building an efficient indexing data structure, enabling the secure and private execution of substring search queries over an outsourced document collection. Our solution combines the efficiency of an index-based substring search algorithm with the secure-execution features provided by the SGX technology and the access pattern indistinguishability guarantees provided by an Oblivious RAM. To prevent the information leakage from the access pattern side-channel vulnerabilities affecting SGX based applications, we redesign three ORAM algorithms, and perform a comparative evaluation to find the best engineering trade-offs for a privacy-preserving index-based substring search protocol. The practicality of our solution is supported by a response time of about 1 second to retrieve all the positions of a protein in the 3 GB string of the human genome.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**; **Management and querying of encrypted data**; *Security protocols.*

KEYWORDS

Substring search, ORAM, Intel SGX, Privacy-preserving protocol

ACM Reference Format:
Nicholas Mainardi, Davide Sampietro, Alessandro Barenghi, and Gerardo Pelosi. 2020. Efficient Oblivious Substring Search via Architectural Support. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3427228.3427296

ACSAC 2020, December 7–11, 2020, Austin, USA

ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427296

1 INTRODUCTION

The advent of cloud computing has pushed many players in the information technology area to offload both data and computation onto remote servers hosted by a cloud service provider. Indeed, outsourcing data storage and computation to the cloud enables significant cost savings and allows remote access to data and applications with high availability. The main challenge to be faced to enable cloud based solutions without concerns is the loss of privacy of outsourced data. Indeed, in case these data are sensitive or valuable, e.g., biomedical or financial ones, the risk of privacy loss will likely prevent a tenant from offloading the computation onto the service provider, with financial drawbacks for both parties.

Encrypting outsourced data is sufficient to qualify a computation offloaded onto an untrusted party as *privacy-preserving* only if the selected cryptographic strategy (such as Fully Homomorphic Encryption (FHE) [16]) allows computation over encrypted data; nonetheless, such solutions generally incur in prohibitive performance penalties. Therefore, technical alternatives, relying on security guarantees provided by the computing hardware, have been developed. In particular, the Software Guard Extensions (SGX) [13] technology, introduced by Intel in CPUs since the *Skylake* microarchitecture, provides trusted execution environments, known as *secure enclaves*, that guarantee confidentiality and integrity for both the code and the data of an application running within them, even against the OS or the hypervisor of the machine hosting the enclave. SGX relies on performing whole-memory encryption for all the data of the program residing within the enclave, and performing on-the-fly decryption/encryption whenever data are moved to/from the CPU. Access control on enclave memory is enforced at hardware level. The entire process is transparent to the program being run and has minimal performance overheads [13]. Whilst SGX is solid from a cryptographic standpoint, it has been shown that microarchitectural side channel attacks [5, 6, 34] are able to retrieve the per-CPU cryptographic keys employed to secure the computations inside the enclaves. While Intel is committed to mitigate microarchitectural side channel attacks in general, and indeed has provided mitigations for the aforementioned attacks [19–21], it has also stated that the SGX threat model [12] does not provide protection against adversaries reconstructing the memory access patterns of applications running inside an enclave. This information is inferred by the adversary either from the sequence of page faults [51] experienced by the application running within the enclave or by measuring the latency experienced by an application

controlled by the attacker which shares cache lines with the victim application running within the enclave [4, 7]. Although several countermeasures have been proposed to prevent or detect these attacks against SGX technology [18, 33, 41, 43], none of them is currently effective against all mentioned attacks.

An approach to make the information leaked to the adversary in all such attacks useless is to ensure that the memory access patterns of an application running within an enclave are independent from the data being processed, a property of the application referred to as *obliviousness*. This property has been achieved in existing works [1, 29, 40] by employing a building block known as Oblivious RAM (ORAM) [44], which allows a client with limited storage capabilities to outsource data onto an untrusted server, guaranteeing that the server, which observes only physical accesses to the ORAM data structure, cannot infer which elements are accessed by the client. The ORAM client and server reside in two distinct machines and communicate via a network channel. Nonetheless, if SGX is available on the untrusted server, the ORAM client can be moved inside an enclave to significantly reduce the communication latency between the ORAM client and the ORAM server. Since leaking the access patterns of the ORAM client through SGX side channels is sufficient to invalidate the privacy guarantees of ORAM, it is necessary to make the ORAM client oblivious too. Coherently with the terminology introduced in [29], we refer to an ORAM with an oblivious client as a Doubly Oblivious RAM (DORAM).

In this work, we design a privacy-preserving computation based on SGX for substring search based on an inverted index (referred to as *full-text index* from now on). Given a document collection $\mathbf{D}=\{D_1, \ldots, D_z\}$ with $z \geq 1$ documents over an alphabet Σ, the data owner builds a full-text index enabling the look-up of the repetitions (or *occurrences*) of a substring q over Σ in $\mathbf{D}$. In a Privacy-Preserving Substring Search (PPSS) protocol, the data owner outsources both the full-text index and the collection $\mathbf{D}$ to an untrusted server; then, given a query for a substring q, chosen by the data owner, the remote server computes, for each document D_i, $i \in \{1, \ldots, z\}$, in $\mathbf{D}$, the set S_i of positions of the occurrences of q in D_i. Our design of a PPSS protocol provides efficient queries while leaking to the untrusted server no more information than the number z of documents in $\mathbf{D}$, the size n of the full-text index, the length m of the substring q, and the overall number o_q of occurrences of q in $\mathbf{D}$.

Our solution, called Oblivious Substring Queries on Remote Enclave (ObSQRE), runs a substring search algorithm within an SGX enclave hosted on an untrusted server. The main challenges tackled in our work reside in designing a substring search algorithm with a data-independent control flow, which allows to hide memory access patterns to its code pages, and efficiently combining such algorithm with a DORAM to obliviously retrieve entries from the full-text index. ObSQRE is composed of three procedures: Setup, Load and Query. The first one, executed by the data owner once as a pre-processing stage, computes the full-text index from $\mathbf{D}$. The index is encrypted with a semantically secure symmetric cipher and outsourced with the encrypted documents to the untrusted server. The integrity of the index is guaranteed by encrypting it with an Authenticated Encryption with Authenticated Data (AEAD) scheme, such as AES-Galois Counter Mode (AES-GCM), whose encryption procedure computes also a keyed digest that is verified upon decryption. Whenever the data owner is willing to perform queries, it asks the untrusted server to instantiate the ObSQRE SGX enclave, which contains the code of both the DORAM client and the oblivious substring search algorithm; the *remote attestation* procedure, provided by SGX technology, allows the data owner both to verify that the enclave has been correctly instantiated by the remote server and to establish a secure communication channel with the enclave. When enclave is running, the Load procedure instantiates the DORAM data structures (both inside and outside the enclave). Then, the enclave receives via secure channel the decryption key and the digest for the full-text index from the data owner, decrypts the index and stores it in the DORAM. Once Load is over, the data owner can submit queries to the enclave. In the Query procedure, the data owner sends through the secure channel the substring q to be searched. Then, the oblivious substring search algorithm runs inside the enclave, employing the full-text index to compute the positions of the occurrences of q over documents of $\mathbf{D}$, which are sent back to the data owner.

Contributions. We propose ObSQRE, the first PPSS protocol employing SGX enclaves, designing two oblivious substring search algorithms based on the backward search method [14]. Our proposal closes the information leakage gap in Intel's attacker model, making the leakage coming from the memory access patterns of the substring search algorithm running inside an enclave useless for the adversary. Our protocol is secure against malicious adversaries (i.e., they may induce arbitrary misbehaviors in the protocol), and achieves optimal bandwidth $O(m+o_q)$, as the data owner sends the substring q and receives the results in one communication round, requiring at server side a polylogarithmic computational cost of $O((m+o_q)\log^3(n))$.

We also proposed our own doubly oblivious version of Path [44] ORAM, which improves over state-of-the-art designs, and we present the first doubly oblivious versions of Circuit [49][1] and Ring [36] ORAMs.

We perform an exhaustive experimental campaign showing how ObSQRE is an effective PPSS solution for practically relevant use-cases employing genomic and financial datasets.

Related Work. The adoption of a DORAM to hide the access pattern to data structures employed by a generic application running inside an enclave was first proposed in ZeroTrace [40]. Nonetheless, differently from ObSQRE, ZeroTrace does not protect the memory access pattern to the code segment of the application. Obfuscuro [1] extended the approach in [40] to hide the access pattern also to the code pages of a generic application running within an enclave employing a DORAM. This approach, although applicable to a generic application, introduces an higher overhead than ObSQRE. Indeed, Obfuscuro [1] must perform two DORAM accesses every 3 to 5 executed assembly instructions, while ObSQRE performs $2m+o_q$ DORAM accesses in total for a single query.

The relevance of substring search queries for a variety of real-world applications fostered several research efforts to privately compute such queries. Some of these solutions [10, 26, 46] achieve bandwidth and computational costs depending only on m and o_q, and require only a few communication rounds between the data owner and the untrusted server. Nonetheless, these solutions leak

[1] A Circuit DORAM was employed in the experimental results of ZeroTrace [40], but its design was not reported in the paper

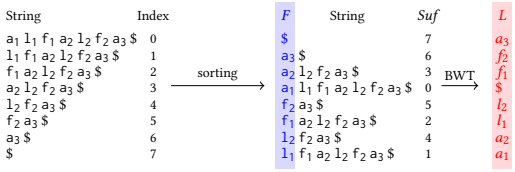


Figure 1: BWT L and SA Suf of the string *alfalfa*

the access patterns of the queries on the outsourced full-text index. When this information is combined with the background knowledge coming from the application domain, the confidentiality of the document collection is compromised [9, 38]. These attacks pushed for the availability of efficient PPSS protocols avoiding this information leakage. In many of the existing solutions [22, 27, 42, 48] the low leakage comes at the cost of making the computation cost on server side linear in the document collection size. The PPSS protocol described in [30] allows to reduce such cost to polylogarithmic; however, it requires $O(m+o_q)$ communication rounds and a polylogarithmic bandwidth cost.

Regarding SGX based solutions, privacy-preserving search indexes, such as Oblix [29], Bunker [2] or HardIDX [15], do not allow to search arbitrary substrings but only a given set of keywords.

2 PRELIMINARIES

In this section, we describe the building blocks of ObSQRE: *backward search* algorithm [14] and ORAM protocols.

2.1 Backward Search Algorithm

Given a text, or string, s with n characters over an alphabet Σ and a substring $q \in \Sigma^*$ with m characters, the backward search algorithm employs a full-text index built from the Burrows-Wheeler Transform (BWT) [8] and the Suffix Array (SA) [28] of s to retrieve the positions of all the o_q repetitions of q in s. To construct the full-text index, a symbol \$, preceding any character of Σ in any ordering relation (e.g., alphabetical) over $\Sigma \cup \{\$\}$, is used to mark the end of the string s. For the string s, the i-th, $i=0,\ldots,n$, *suffix* of s is the substring $s[i,\ldots,n]$; the integer i is referred to as the index of the suffix. The SA stores the indexes of all suffixes of the string sorted in lexicographical order and can be built in $O(n)$ time complexity [31]. The BWT L of s is a permutation of the original string that yields better compression ratio with run-length encoding techniques; it can be built in $O(n)$ time from the SA Suf as $L[i]=s[Suf[i]-1 \bmod n+1], i=0,\ldots,n$. Figure 1 shows the computation of the SA and the BWT for the string *alfalfa*.

The backward search algorithm employs three data structures: the SA of the string s, the dictionary C that binds a character $c \in \Sigma$ to the number of characters smaller than c in the string s (according to the order relation employed to sort suffixes in the SA), and the full-text index $\tilde{L}$ constructed from the BWT L of the string s. A fundamental building block in this algorithm is the computation of the function Rank, which, given a character $c \in \Sigma$ and an integer $i \in \{0,\ldots,n\}$, employs the full-text index $\tilde{L}$ to compute the number of occurrences of c in the prefix $L[0,\ldots,i]$ of L, i.e., $\text{Rank}(c,i) = |\{j \in \{0,\ldots,i\} \text{ s.t. } L[j]=c\}|$. We will show different

Algorithm 1: Backward search for a string s of length n

Input: q: a substring with length $1 \le m \le n$
Output: $\mathsf{R_q}$: set of positions in s with leading character of occurrences of q
Data: $\tilde{L}$: full-text index constructed from the BWT of s required by Rank
C: dictionary storing $\forall c \in \Sigma$ the number of chars in s smaller than c
Suf: the SA with length $n+1$ of the string s

1 $\alpha \leftarrow C(q[m-1]),\ \beta \leftarrow \alpha + \text{Rank}(q[m-1], n), \mathsf{R_q} \leftarrow \emptyset$
2 **for** $i \leftarrow m-2$ ***downto*** 0 **do**
3 $\mathsf{c} \leftarrow q[i],\ \mathsf{r} \leftarrow C(\mathsf{c})$
4 $\alpha \leftarrow \mathsf{r} + \text{Rank}(\mathsf{c}, \alpha - 1)$
5 $\beta \leftarrow \mathsf{r} + \text{Rank}(\mathsf{c}, \beta - 1)$
6 **for** $i \leftarrow \alpha$ ***to*** $\beta - 1$ **do**
7 $\mathsf{R_q} \leftarrow \mathsf{R_q} \cup \{Suf[i]\}$
8 **return** $\mathsf{R_q}$

methods to implement this procedure, each employing its own full-text index $\tilde{L}$, in our oblivious substring search algorithms.

We provide an operative description of the backward search algorithm in Alg. 1, pointing the reader interested in the detailed correctness analysis to [14, 27]. Given a string q with m characters, Alg. 1 first computes the number o_q of occurrences of q in the string s (lines 2-5) processing the characters of the substring backwards; at the end of this loop, the number of occurrences $o_q=\beta-\alpha$. Then, it retrieves the positions of all these occurrences as the o_q consecutive entries $\{Suf[\alpha],\ldots,Suf[\beta-1]\}$ of the SA (lines 6-7). The two loops in Alg. 1 perform $m-1$ and o_q iterations, respectively; as each iteration costs $O(T_{rank})$, where T_{rank} denotes the computational cost of the Rank procedure, and $O(1)$, respectively, Alg. 1 has $O(m \cdot T_{rank}+o_q)$ cost.

Backward Search in a Document Collection. Given a document collection $\mathbf{D}$ with $z \ge 1$ documents $D_1,\ldots,D_z$, we bind to the i-th character of the j-th document the pair (doc, off), with $doc=j$ and $off=i-1$; then, we build a single string s by appending the delimiter \$ to each document and concatenating all the documents, i.e. $s=D_1\$D_2\$\ldots D_z\$$. We replace the index of each suffix of s with the pair (doc, off) bound to the first character of the suffix at hand. Given a substring q with m characters, Alg. 1 employs the full-text index $\tilde{L}$, the dictionary C and the SA for the string s to compute the set $\mathsf{R_q}$ with o_q pairs (doc, off); by grouping all the pairs with the same document id doc, we obtain, for each document in $\mathbf{D}$, the set of positions of all the occurrences of q in the document at hand. This algorithm finds all and only the occurrences of q over $\mathbf{D}$: indeed, any occurrence of q in a document of $\mathbf{D}$ is found in s too; conversely, each occurrence of q in s identifies m characters with no delimiter \$, which correspond to an occurrence of q in a document of $\mathbf{D}$.

2.2 ORAM Protocols

We now describe in detail Path ORAM [44] and then we sketch the differences introduced in Ring [36] and Circuit [49] ORAMs.

Path ORAM. Path ORAM splits a dataset of L bits in blocks of B bits and assigns to each one of them a unique identifier, referred to as block id (*bid*). Although $l=\lceil \frac{L}{B} \rceil$ blocks are sufficient to store the dataset, Path ORAM increases the number of blocks to $N \cdot Z$, where $N=2^{\lfloor \log_2(l) \rfloor+1}-1$ and $Z \ge 1$; these additional blocks, called *dummy*, allow to hide how the l real blocks are scrambled inside the ORAM. The id of dummy blocks is set to a special value $\perp$ to distinguish them from real ones. All the $N \cdot Z$ blocks are partitioned

in N *buckets*, each one containing Z blocks; then, the buckets are arranged as a balanced complete binary tree with N nodes, each storing one bucket. Each bucket is encrypted with a semantically secure scheme; a bucket is full if it contains Z real blocks. The $\frac{N+1}{2}$ leaves of the tree are labeled with a leaf id *lid*, a $\log_2(N+1)-1$ bits-wide integer that identifies the path of the tree to reach the leaf at hand; specifically, the i-th bit of *lid* (i=0 is the least significant bit) is 0 (resp. 1) if the leaf belongs to the left (resp. right) subtree of the i-th node in the path from the root to the leaf at hand.

To retrieve real blocks from the ORAM, each of them is mapped to a *lid*, which identifies the path of the tree where the block must reside; this mapping is stored in a data structure called *position map*. Any modification of the tree must preserve this mapping, otherwise blocks cannot be retrieved any longer. All real blocks store their corresponding *lid* in order to be placed in the proper path. Another data structure, called *stash*, stores the accessed real blocks that have not been pushed back to the ORAM tree yet. The stash analysis of Path ORAM [44] proves that for $Z \geq 4$ the number of blocks in the stash, denoted with S, is $O(1)$ with overwhelming probability; thus, the stash can be stored at client side to conceal it from the server.

The Access procedure retrieves the content of a specific block from the ORAM. Given a block id *bid*, the procedure obtains the leaf id *lid* corresponding to block *bid* from the position map, and updates the corresponding entry with a randomly sampled leaf id lid'. Then, it invokes two other procedures: FindBlock(bid, lid, lid') and Eviction(lid). The former starts by retrieving from the server the whole path containing the leaf with id *lid*. The client decrypts the fetched path, appends all the real blocks to the stash and looks for the *bid* block in it. If the block is found, its leaf id is replaced with lid'. FindBlock returns the content of the block, if found, $\perp$ otherwise. The Eviction procedure writes back the fetched path, with id *lid*, to the ORAM tree, filling the buckets with as many blocks as possible from the stash. The client computes, for each block in the stash, the deepest bucket of the evicted path that can store the block at hand and, if found, it moves the block from the stash to this bucket. A bucket can store a block with leaf id lid' if it is not full and it belongs to both the evicted path, with id *lid*, and the path with id lid' (to preserve the property that a block is found on the path corresponding to its leaf id). The eviction stops when no more blocks in the stash can be moved to the evicted path; thus, the client re-encrypts the path and writes it back to the ORAM. Both procedures cost $O(\log(N) \cdot Z \cdot B)$ on server side and $O(B(S + \log(N) \cdot Z))$ on client side, while their bandwidth is $O(\log(N) \cdot Z \cdot B)$, as the client and the server exchange a whole path.

Path ORAM allows to hide the accessed block only if the secret mapping between block ids and leaf ids, stored in the position map, is concealed from the server; nonetheless, as the position map has l entries, each of $\log(N)$ bits, it cannot be stored by a client with limited storage capabilities. To overcome this issue, another ORAM, denoted as ORAM_1, is employed to store the position map: indeed, if each block of ORAM_1 contains up to C entries of the position map, the position map of ORAM_1 has $\lceil \frac{l}{C} \rceil$ entries, thus reducing the size of the position map by a factor of C. By recursively applying this strategy to store the position maps of smaller ORAMs, eventually the position map becomes compact enough to be stored at client side. Indeed, by employing $\Theta(\log_C(l))$ recursive ORAMs, the size of the position map of the smallest ORAM becomes $O(1)$. This recursive strategy introduces a logarithmic factor in both the bandwidth and the computational cost, which become $O(C \cdot B \cdot \log^2(N) \cdot Z)$.

Ring ORAM. Ring ORAM improves over Path ORAM in two ways: the FindBlock procedure achieves a bandwidth of $O(B \cdot \log(N))$ by fetching from the server a single block per bucket instead of the entire bucket; the Eviction procedure is performed once every $A \geq 1$ accesses to the ORAM instead of being performed for each access. To reduce the bandwidth of FindBlock procedure, each bucket is enriched with some metadata; instead of fetching entire buckets along the path *lid*, the FindBlock procedure retrieves only their metadata. Then, for each bucket, FindBlock invokes the SelectOffset procedure which selects the offset of the block *bid*, if found in the bucket, or the offset of a dummy block otherwise. To ensure that there are enough dummy blocks in each bucket to be chosen by the SelectOffset procedure, buckets in the Ring ORAM have $Z+D$ blocks, where the additional D slots always store dummy blocks. To prevent the adversary from learning if SelectOffset chooses a real or dummy block, all of them are randomly shuffled. The offsets computed by SelectOffset are sent to the server, which retrieves the corresponding blocks from the ORAM tree.

The only real block fetched from the ORAM in the FindBlock procedure is appended to the stash, thus making an Eviction after each FindBlock unnecessary. Indeed, an eviction happens every $A \geq 1$ accesses, a parameter of Ring ORAM that depends on Z. In order to maximize the average number of blocks evicted from the stash, the paths to be evicted are chosen according to a deterministic schedule, following the ids of the paths in increasing order. This guarantees that the overlap between two consecutive evicted paths is limited to the bucket stored in the root node of the ORAM tree, as a bucket at level i of the tree belongs to the evicted path every 2^i consecutive evictions.

Circuit ORAM. Circuit ORAM is a refinement of Path ORAM tailored for hardware implemented clients, where the server is a large memory on the same machine (or even on the same die). Therefore, this ORAM trades off a low bandwidth for the compactness of the circuit implementing the ORAM client. This is achieved with a simplified Eviction procedure that evicts at most one block from the stash. The stash growth is limited as FindBlock appends at most the block with id *bid* to the stash, if found in the fetched path. This path, with the block *bid* replaced by a dummy one, is re-encrypted and written back to the ORAM tree. The path to be evicted is chosen with the same deterministic schedule of Ring ORAM to minimize the probability that no block can be evicted from the stash. To avoid a monotonic growth of the stash in case no blocks from the stash can be evicted, 2 evictions are performed for each access. The additional eviction, although forcing the ORAM to fetch and write back 3 paths per access, allows to keep the stash about 1 order of magnitude smaller than Path and Ring ORAMs.

3 DOUBLY OBLIVIOUS RAMS

In this section, we describe the design of oblivious clients for the three ORAMs described in Section 2.2, obtaining three corresponding DORAMs. In doing this, we employ two operations: oblivious write OblWrite and oblivious swap OblSwap. The former (resp. the latter), given three input parameters *cond*, a and b, writes the

Algorithm 2: FindBlock in Path/Circuit DORAMs

Input: *bid*: id of the block to be retrieved from the DORAM
lid: id of the path where block *bid* may be located
lid′: id of the path where block *bid* will be evicted
Output: The block with id *bid*
Data: Stash: *S* blocks not evicted to the DORAM yet

```
1  Blks ← ReadPath(lid) , dest.bid ← ⊥
2  foreach blk ∈ Blks do
3      OblSwap(blk.bid = bid, dest, blk)
4  write ← dest.bid = bid
5  foreach blk ∈ Stash do
6      OblWrite(bid = blk.bid, dest, blk)
7      OblWrite(blk.bid = ⊥ ∧ write, blk, dest)
8      OblWrite(bid = blk.bid, blk.lid, lid′)
9      write ← write ∧ blk.bid ≠ bid
10 WritePath(Blks, lid)
11 return dest
```

content of b to a (resp. swaps the content of b and a) if and only the boolean expression *cond* is true. To implement OblWrite, we employ the `x86_64` assembly instruction `CMOVNZ`, which moves the content of the source operand to the destination one if the zero flag is not set. `CMOVNZ` is oblivious as its operands are always loaded in the CPU and written back regardless of the status of the flag. The OblSwap operation, given the input parameters *cond*, a, b, first computes OblWrite(*cond*, *tmp*, $a \oplus b$), where *tmp* is initially set to 0; then, it updates a and b with $a \oplus tmp$ and $b \oplus tmp$, respectively.

In all our DORAMs, the stash cannot have a dynamic size, lest the number of blocks moved between the DORAM and the stash is leaked. Thus, in all our DORAMs the stash has a fixed size S, and an *overflow* error occurs if the number of blocks in the stash is higher than S. Empty entries in the stash are filled with dummy blocks. The stash analysis of Path, Ring and Circuit ORAMs provides upper bounds for S making the probability of overflows negligible.

If a recursive position map is employed, the ORAMs that store the position map must be doubly oblivious too. Once a block from each of these DORAMs is fetched, the client obliviously swaps each one of the C entries in the block with a memory location `dest`, initialized with the new leaf id *lid′*, actually performing the swap only for the entry corresponding to the block to be retrieved from the next DORAM in the recursion. Eventually, `dest` stores the id of the path to be fetched from the next DORAM, while the corresponding entry in the block stores the updated id *lid′*.

3.1 Path DORAM

We start with a description of the oblivious Eviction procedure proposed in ZeroTrace [40] and employed with minor modifications in all existing works. This procedure, for each block of the stash (even dummy ones), sweeps over the evicted path, which is initialized with dummy blocks, from the leaf to the root bucket, obliviously swapping each block with the entry of the stash at hand; the block of the stash is actually swapped with a dummy block found in the deepest non-full bucket that can store the block at hand. The computational cost of ZeroTrace Eviction is thus $O((S+\log(N)\cdot Z)\log(N)\cdot Z\cdot B)$, since both the blocks of the stash and the blocks of the path fetched by the FindBlock procedure must be evicted.

Table 1: Format of the bucket metadata in Ring ORAM.

Field	Bit width	Size	Description
Bids	$\log(N+1)$	$Z+D$	Block ids of all blocks
Lids	$\log(N+1)$	Z	Leaf ids of real blocks
IV	λ	1	IV for bucket decryption
Invalid	1	$Z+D$	Flags keeping track of invalid blocks
Cnt	$\log D$	1	Count accesses to bucket

In our Path DORAM, we modify this Eviction procedure by introducing an optimization, called *in-place eviction*, that allows to approximately halve its cost. Indeed, instead of appending all the blocks of the evicted path to the stash and then evict them as all other blocks in the stash, the client tries to push these blocks as down as possible in the path before performing stash eviction. This optimization allows to swap a block in the path only with deeper buckets instead of swapping it with all the buckets in the path. Even with this optimization, our oblivious Eviction still exhibits an $O(\log(N)\cdot Z))$ computational overhead with respect to the non oblivious Eviction of Path ORAM. To reduce this overhead, which severely affects the performance of DORAM accesses, in our Path DORAM we aim at making evictions less frequent. To make evictions unnecessary after each FindBlock, as in Ring ORAM, we need to ensure that only one block is appended to the stash for each access. We achieve this employing the FindBlock procedure of Circuit ORAM, which actually moves to the stash only the block *bid*, if found in the fetched path, and writes back the fetched path to the tree, replacing the block *bid* with a dummy one. We note that, although we add a write back operation for each access, this strategy allows our Path DORAM to perform evictions every $A \geq 1$ accesses; since a write back costs $O(\log(N)\cdot Z\cdot B)$, it is asymptotically faster than an eviction, thus improving the performance of our Path DORAM. To choose which path to evict, Path DORAM employs the same deterministic schedule of Ring ORAM; thus the same values of Z, S and A suggested by authors of Ring ORAM [36] can be employed in our Path DORAM.

Our oblivious FindBlock procedure (see Alg. 2) starts by fetching the path with id *lid* from the DORAM tree (line 1) and it obliviously looks for the block with id *bid* over the fetched path (lines 2- 3). If the block is found, it is moved to `dest` and replaced by a dummy block in the fetched path (line 3), otherwise neither `dest` nor the fetched path are modified. Then, the FindBlock procedure obliviously sweeps (lines 4-9) over the stash to either write to the stash the block *bid*, if found in the fetched path, or to search the block in the stash. In the former case, the proper update of `write` flag (line 9) ensures that the block *bid* is written to the first empty entry in the stash (line 7); in the latter case, the block *bid* found in the stash is written to `dest` (line 6). In both cases, the leaf id of the block *bid* in the stash is updated to *lid′* (line 8). Finally, the FindBlock procedure writes back the fetched path to the DORAM tree (line 10).

3.2 Ring DORAM

We recall that Ring ORAM enriches each bucket with metadata that are employed by the SelectOffset procedure to choose, for each bucket, one block to be retrieved from the ORAM tree. The structure of the bucket metadata employed in our Ring DORAM is outlined in Tab. 1. The first field of the metadata stores the ids of the

Algorithm 3: SelectOffset in Ring DORAM

Input: *bid*: id of the block to be fetched from the ORAM
Meta: metadata of a bucket of the path where block *bid* may reside
Output: off: position in the bucket of the block with id *bid*, if found, otherwise the position of a randomly chosen valid dummy

```
1  found ← false, max ← −1
2  for i ∈ {0, ..., Z + D − 1} do
3      if ¬Meta.invalid[i] then
4          rnd ←R {0, ..., 255}
5          max_dummy ← Meta.bids[i]=⊥ ∧ rnd > max
6          sel ← Meta.bids[i]=bid ∨ (max_dummy ∧ ¬found)
7          OblWrite(sel, off, i), OblWrite(max_dummy, max, rnd)
8          found ← found ∨ Meta.bids[i] = bid
9  return off
```

Algorithm 4: FindBlock in Ring DORAM

Input: *bid*: id of the block to be retrieved from the DORAM
lid: id of the path where block *bid* may be located
lid′: id of the path where block *bid* will be evicted
Output: the block with id *bid*
Data: Stash: S real/dummy blocks not evicted to the DORAM yet

```
1  Metadata ← FetchBucketsMetadata(lid), Offsets ← ∅
2  foreach Meta ∈ Metadata do
3      off ← SelectOffset(bid, Meta)
4      Offsets ← Offsets ∪ {off}
5      Meta.invalid[off] ← true, Meta.cnt + +
6  Blks ← FetchBlocks(lid, Offsets), dest.bid ← ⊥
7  foreach blk ∈ Blks do
8      OblSwap(blk.bid = bid, dest, blk)
9  write ← dest.bid = bid
10 foreach blk ∈ Stash do
11     OblWrite(bid = blk.bid, dest, blk)
12     OblWrite(blk.bid = ⊥ ∧ write, blk, dest)
13     OblWrite(bid = blk.bid, blk.lid, lid′)
14     write ← write ∧ blk.bid ≠ bid
15 foreach i ← 0 to log((N+1)/2)−1 do
16     if Metadata[i].cnt ≥ D then
17         Blks ← FetchValidBlocksInBucket(lid, i)
18         Bucket ← EarlyReshuffle(Blks, Metadata[i])
19         WriteBucket(lid, i, bucket)
20 WriteMetadata(lid, Metadata)
21 return dest
```

blocks found in the Z+D slots of the bucket. We store also the ids of dummy blocks to allow the proper computation of the offset of the block chosen in the SelectOffset procedure. The last two fields of the metadata are needed to ensure two fundamental properties about Ring DORAM: i) each block is fetched from its bucket at most once since the last time the bucket was written back to the DORAM tree; ii) each bucket must be accessed at most D times since the last time it was written to the DORAM tree. The first property is needed to prevent the adversary from distinguishing dummy blocks from real ones from their access frequencies. Indeed, a real block is chosen by SelectOffset procedure, and fetched from a bucket, only when it corresponds to the block *bid* that must be retrieved by FindBlock, while a dummy block may be chosen in all other cases. To ensure this property, the SelectOffset procedure must always choose a valid block, with a block being marked as invalid in the bucket metadata as soon as it is chosen by SelectOffset. The second property ensures that there are always enough dummy blocks to be chosen in a bucket by SelectOffset: indeed, after D accesses to the bucket, no valid dummy blocks may have left in the bucket. To this extent, the bucket metadata keeps track of the number of accesses to the bucket with a counter cnt; when D accesses are reached, a maintenance task called EarlyReshuffle must be invoked. This procedure, upon receiving the Z valid blocks of the bucket, randomly shuffles them with D dummy blocks; then, the bucket is encrypted and written back to the DORAM tree. As the blocks are re-shuffled, they can all be marked as valid, and cnt is reset as the bucket has at least D valid dummy blocks available.

We now describe the oblivious procedures of our Ring DORAM. The SelectOffset procedure (Alg. 3) iterates over all the blocks in the bucket, skipping invalid ones (line 3) as they cannot be chosen. Note that there is no need to hide which blocks are skipped, as the adversary can easily know which blocks are invalid by logging blocks chosen in previous accesses to the bucket at hand. For each valid block, a number rnd is uniformly sampled from a fixed domain (e.g., $\{0, \ldots, 255\}$ in line 4) and the offset of the valid block is obliviously written to the variable off (line 7). The update of max_dummy (line 5), sel (line 6) and found (line 8) flags ensures that eventually the variable off stores the position in the bucket of the block with id *bid*, if found in the bucket; otherwise, off stores the position of the valid dummy block with the highest random number among the ones sampled for all the dummy valid blocks. As all these numbers are sampled from the same distribution, each one of them has the same probability of being the highest, thus this method chooses uniformly at random a block among the dummy valid ones without revealing which blocks are dummies.

In the oblivious EarlyReshuffle procedure, Z blocks have to be randomly placed over Z+D slots of the bucket. To this extent, the i-th block, i=1, ..., Z, is obliviously written in the *off*-th free slot of the bucket, where *off* is uniformly sampled from $\{1, \ldots, Z+D-i+1\}$; the block and leaf ids in the bucket metadata are updated accordingly. Since the bucket is initialized with all dummy blocks, after Z sweeps, each writing one block, D slots of the bucket certainly contain a dummy block. As this strategy requires Z oblivious writes over a bucket with Z+D slots, EarlyReshuffle costs $O(Z \cdot (Z+D) \cdot B)$ per bucket. We show in Appendix A.3 that this strategy guarantees that each block is placed with uniform probability over all the Z+D slots of the bucket.

In the oblivious FindBlock procedure, reported in Alg. 4, first the metadata for all the buckets along the path with id *lid* are fetched (line 1). Then, the procedure iterates over the metadata to choose one block per bucket to be retrieved from the server (lines 2-5). The offset of the chosen block in the bucket, computed by the SelectOffset procedure (line 3), is appended to the set Offsets (line 4). Furthermore, the bucket metadata are updated by marking the chosen block as invalid and by increasing the number of accesses to the bucket (line 5). Subsequently, the DORAM server, upon receiving from the client the id of the path *lid* and, for each bucket in this path, the offset of the block to be retrieved, sends back the $O(\log(N))$ blocks requested by the client (line 6), among which the client obliviously searches the block with id *bid* through a linear sweep (lines 7-8). Then, the DORAM client iterates over the stash (lines 9-14) to either insert the block retrieved from the server (line 12) or locate the block with id *bid* (line 11). Afterwards, the FindBlock procedure, for each bucket, invokes, if necessary (line 16), the EarlyReshuffle procedure (line 18), fetching the Z valid blocks left in the bucket from the DORAM tree (line 17) and

Algorithm 5: EVICTION in Circuit DORAM

Input: Path: path to be evicted
lid: id of Path
Data: Stash: S real/dummy blocks not evicted to the DORAM yet

```
1  dest ← ComputeDestinations(Path, lid)
2  max_depth ← −1, target ← dest[0], hold.bid = ⊥
3  foreach blk ∈ Stash do
4      depth ← MaxDepth(blk.lid, lid)
5      OblSwap(dest[0]≠⊥∧depth>max_depth, hold, blk)
6      OblWrite(depth > max_depth, max_depth, depth)
7  for i ← 0 to log(N+1)−2 do
8      max_depth ← i, deeper_bucket ← (target≠i ∧ target≠⊥)
9      foreach blk ∈ Path[i] do
10         depth ← MaxDepth(blk.lid, lid)
11         swap ← (dest[i+1]≠⊥ ∧ depth>max_depth) ∨
             (dest[i+1]=⊥∧blk.bid=⊥)
12         OblSwap(swap ∧ ¬deeper_bucket, hold, blk)
13         OblWrite(depth>max_depth, max_depth, depth)
14     OblWrite(¬deeper_bucket, target, dest[i+1])
```

writing the whole bucket back after the reshuffle (line 19). Lastly, the FINDBLOCK procedure writes back the updated metadata to the DORAM tree (line 20).

The EVICTION procedure follows the blueprint of the oblivious one described in Path DORAM. Indeed, although buckets in Ring DORAM has $Z+D$ slots, at most Z of them may be filled with real blocks; hence, buckets with Z slots can be employed during evictions, as in Path DORAM. At the end of the EVICTION, the EARLYRESHUFFLE procedure is invoked on each of these buckets with Z blocks to construct a bucket with $Z+D$ blocks, which is written back to the DORAM tree after re-encryption. As in Path DORAM, EVICTION is split in two phases: in-place and stash eviction. While the latter works exactly as in Path DORAM, in the former, for each bucket, only the $R \leq Z$ valid real blocks must be evicted, while all the other $Z+D-R$ blocks can be discarded, as they are either dummies or invalid real blocks. To avoid leaking R to the adversary, during in-place eviction the client has to always choose Z blocks for each bucket. In particular, for each bucket, the client must choose Z blocks out of the $V \geq Z$ valid blocks: R real valid blocks and $Z-R$ blocks among the $V-R$ dummy valid blocks. To this extent, we employ the Knuth's algorithm reported in [25, pag. 142], which chooses uniformly at random k elements out of h ones, with $k \leq h$; this algorithm can be trivially made oblivious by relying on oblivious write/swap primitives while retaining $O(h)$ computational complexity. Once the offsets of the Z blocks are computed, they are obliviously fetched from the bucket with Z linear sweeps and evicted in deeper buckets. The overall computational cost of EVICTION procedure is $O(\log(N) \cdot B \cdot Z(\log(N) \cdot Z + S + Z + D))$, which is the sum of the costs of in-place eviction, stash eviction and EARLYRESHUFFLE for all the buckets along the evicted path, respectively; this cost is amortized over $A \geq 1$ DORAM accesses.

3.3 Circuit DORAM

The simplicity of the client in Circuit ORAM makes its oblivious design the easiest one among our three DORAMs. The FINDBLOCK procedure in our Circuit DORAM is equivalent to our Path DORAM, reported in Alg. 2. Conversely, the EVICTION procedure of Circuit ORAM significantly differs from the one of Path and Ring ORAMs. Specifically, the non-oblivious eviction involves two sweeps over the metadata of the evicted path (which correspond to the block ids and the corresponding leaf ids), and a single sweep over the evicted path. In the oblivious EVICTION procedure, reported in Alg. 5, the two sweeps over the metadata are performed by the COMPUTEDESTINATIONS procedure (line 1), which compute, for the stash and for each bucket in the evicted path, the additional metadata dest. To avoid leaking these metadata while computing them, we employ oblivious writes to remove conditional dependent updates to these metadata. In the subsequent sweep over the evicted path (lines 2-14), dest specify how the blocks must be moved among buckets: indeed, for each $i \in \{0, \ldots, \log(N+1)-1\}$ (dest[0] refers to the stash), dest[i] stores the bucket where the block of the i-th bucket that can go deepest in the path must be moved, while dest[i]=⊥ if no block from the the i-th bucket must be moved down in the path. During the sweep over the evicted path, at most one block, stored in hold, is simultaneously moved down along this path; the variable target stores the destination bucket of such block. Throughout the sweep over the evicted path, a procedure MAXDEPTH allows to compute the deepest bucket of the path that can store a given block by hinging upon the leaf id of the block at hand and the id of the evicted path. First, the block in the stash that can go deepest in the path is obliviously moved to hold through a linear sweep of the stash (lines 3-6). Then, this block is moved to its destination bucket, where it is swapped (line 12) with either a dummy block, in case no block in the destination bucket must be moved down (i.e., if dest[i+1]=⊥ in line 11), or with the block in the destination bucket than can go deepest in the path (lines 11, 13). The computational cost of oblivious EVICTION is $O(B \cdot (S + \log(N) \cdot Z))$, given by the linear sweeps over the stash and the evicted path, respectively.

Table 2: Client-side asymptotic computational costs

	FINDBLOCK		EVICTION	
	ORAM	our DORAM	ORAM	our DORAM
Path	$O(\log(N) \cdot Z \cdot B)$	$O(\log(N) \cdot Z \cdot B)$	$O(B \cdot \log(N) \cdot Z)$	$O(\frac{\log^2(N) \cdot Z^2 \cdot B}{A})$
Ring	$O(\log(N) \cdot B)$	$O(\log(N) \cdot B)$	$O(\frac{B \cdot \log(N) \cdot Z}{A})$	$O(\frac{\log^2(N) \cdot Z^2 \cdot B}{A})$
Circuit	$O(\log(N) \cdot Z \cdot B)$	$O(\log(N) \cdot Z \cdot B)$	$O(\log(N) \cdot Z \cdot B)$	$O(\log(N) \cdot Z \cdot B)$

To conclude, we summarize the computational costs of clients of our DORAMs in Tab. 2. We observe that Circuit DORAM is asymptotically faster than Ring and Path DORAMs; nonetheless, as 3 paths have to be fetched and written back for each DORAM access, a performance gain may be observed only for DORAMs with a significant number of blocks. Instead, Ring DORAM saves a factor of Z in the computational cost of the FINDBLOCK procedure; nonetheless, its oblivious algorithms involve cumbersome operations, which may increase actual response time of DORAM accesses.

3.4 Security Against Malicious Adversaries

In all our DORAMs, we add a mechanism to efficiently detect any tampering (including replacement with old blocks) on any path fetched from the DORAM, while storing in the enclave a single digest for the whole DORAM. Specifically, we combine the DORAM

Algorithm 6: Oblivious RANK procedure with ABWT strategy for a string $s \in \Sigma^n$ with BWT L

```
Input: c: character of the alphabet Σ
       i: integer in {0, ..., n+1}
Output: ctr: number of occurrences of c in L[0, ..., i]
Data: DORAM: DORAM storing the ABWT A_P with sample period P
1  Entry ← DORAM.Access(⌊i/P⌋)
2  foreach char ∈ Σ do
3      OblWrite(c = char, ctr, Entry.rank[char])
4  for j ← 0 to P − 1 do
5      OblWrite(Entry.l[j] = c ∧ j ≤ i mod P, ctr, ctr + 1)
6  return ctr
```

tree with a Merkle tree, as proposed in [37] for Path ORAM; however, we encrypt the buckets with an AEAD scheme to avoid an unkeyed hash computation for the digest of each bucket.

LEMMA 1. *When the client algorithms of our DORAMs are run inside an SGX enclave, any malicious adversary, with full control over the untrusted machine hosting the enclave and able to observe through SGX side channels the pattern on code and data memory accesses of algorithms run inside the enclave, learns no more than the public parameters of the DORAMs (e.g., B, Z) and the size of the dataset stored in the DORAMs. Furthermore, the clients of our DORAMs can detect any tampering on code and data performed by a malicious adversary.*

These security guarantees stem from the obliviousness of the client algorithms of our DORAMs: indeed, as formally proven in Appendix A.1, both their control flow and their memory access pattern are independent from the block accessed by the DORAM.

4 OBLIVIOUS SUBSTRING SEARCH

We first present two algorithms to obliviously compute the RANK procedure required by Alg. 1, and then we employ them to build two oblivious backwards search algorithms that derive the positions of occurrences o_q of a substring $q \in \Sigma^m$ over a string $s \in \Sigma^n$.

Augmented BWT. This algorithm obliviously computes the RANK procedure by employing the Augmented BWT (ABWT) as the full-text index $\tilde{L}$ constructed from the BWT L of s. Given an integer parameter P, called *sample period*, the ABWT A_P is an array with $\lceil \frac{n+1}{P} \rceil$ entries, each containing a pair of elements $(rank, l)$; for the i-th entry of A_P, $A_P[i].rank$ is a dictionary of $|\Sigma|+1$ entries that binds to a character $c \in \Sigma$ the value $\text{RANK}(c, i \cdot P - 1)$, while $A_P[i].l$ is a string of P characters, namely the substring $L[i \cdot P, \ldots, (i+1) \cdot P - 1]$ of the BWT L. The value $\text{RANK}(c, i)$, $c \in \Sigma$, $i \in \{0, \ldots, n\}$, is computed from the $j = \lfloor \frac{i}{P} \rfloor$-th entry of A_P as the sum of $A_P[j].rank[c]$ and the number of occurrences of character c in $A_P[j].l[0, \ldots, i \bmod P]$.

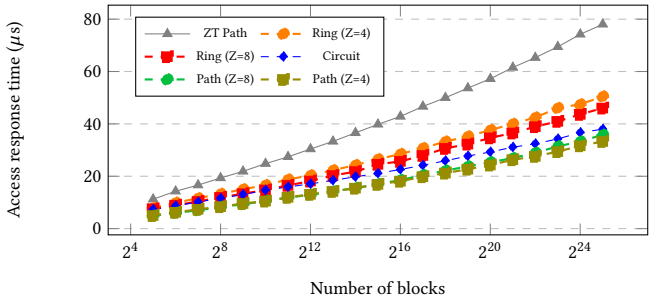
In the oblivious implementation of RANK procedure, reported in Alg. 6, the ABWT is stored inside a DORAM; the algorithm first fetches the block storing the $h = \lfloor \frac{i}{P} \rfloor$-th entry of A_P (line 1); then, `ctr` is set to $A_P[h].rank[c]$ through a linear sweep over the entries of $A_P[h].rank$ (lines 2-3); lastly, the algorithm sweeps over the string $A_P[h].l$, obliviously increasing by 1 `ctr` whenever a character among the first $i \bmod P + 1$ ones equals c (lines 4-5). Each access to an entry of A_P costs $O(C \cdot \log^2(n) \cdot Z \cdot B)$ if the ABWT is stored in Circuit DORAM, while the cost becomes $O(C \cdot \log^3(n) \cdot Z^2 \cdot B)$ in case of Path or Ring DORAMs, as each EVICTION costs $O(\log^2(n) \cdot Z^2 \cdot B)$ instead of $O(\log(n) \cdot Z \cdot B)$. Given that $B = O(\log(n) + |\Sigma| + P)$ and Z, C, $|\Sigma|$ and P are small constants, our ABWT based oblivious RANK procedure has $O(\log^3(n))$ computational cost.

Algorithm 7: Non-oblivious RANK procedure with balanced BST for a string $s \in \Sigma^n$ with BWT L

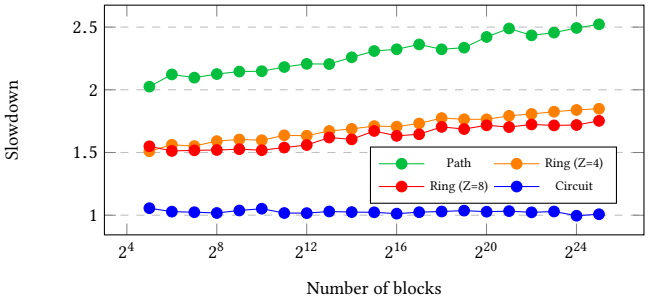
```
Input: c: character of the alphabet Σ
       i: integer in {0, ..., n+1}
Output: Rank(c, i): number of occurrences of c in L[0, ..., i]
Data: BST: balanced BST constructed from the string s
      Enum: enumeration of characters in Σ
      Occ: dictionary binding a char c∈Σ to Rank_L(c, n)
1  node ← BST.root, k ← Enum(c)·(n + 1)+i
2  while node ≠ ⊥ do
3      if node.key = k then
4          return node.value
5      go_left ← 0, parent ← node, node ← node.right
6      if node.key < k then
7          go_left ← 1, node ← node.left
8  if parent.key<Enum(c)·(n+1)∨parent.key≥(Enum(c)+1)·(n+1) then
9      return Occ(c)·go_left
10 return parent.value − go_left
```

Oblivious Data Structure BWT. In the Oblivious Data Structure BWT (ODSBWT) method, instead of an array, we employ a balanced Binary Search Tree (BST) as a full-text index constructed from the BWT L. To build this index, we employ an enumeration `Enum` of characters $c \in \Sigma$. For each $j \in \{0, \ldots, n\}$, we create a node in the BST as a key-value pair $(\text{Enum}(s[j]) \cdot (n+1) + pos_L(s[j]), \text{RANK}(s[j], pos_L(s[j])))$, where $pos_L(s[j])$ denotes the position in the BWT L of the character $s[j]$. Once the BST is built, $\text{RANK}(c, i)$, $c \in \Sigma$, $i \in \{0, \ldots, n\}$ can be computed by looking-up the node with key $\texttt{k} = \texttt{Enum}(c) \cdot (n+1) + i$ in the balanced BST, as outlined in Alg. 7. If the node is found (line 4), then $\text{RANK}(c, i)$ equals the value stored in this node by construction of the BST. Otherwise, the last node explored is either the predecessor (if `go_left=0`) or the successor (if `go_left=1`) of the node with key `k`. Since $\texttt{Enum}(\texttt{c}) \cdot (n+1)$ is added to the key of each node, then all the nodes referring to occurrences of the same character c have consecutive keys. As a consequence, the predecessor node corresponds to the last occurrence of c in $L[0, \ldots, i]$, while the successor node corresponds to first occurrence of c in $L[i+1, \ldots, n]$. In the former case, $\text{RANK}(c, i)$ equals the value of the predecessor node, while in the latter case the value of the successor node must be decremented by 1 (line 10). In case there is no occurrence of c in $L[0, \ldots, i]$ (resp. $L[i+1, \ldots, n]$), the predecessor (resp. successor) node refers to an occurrence of a character $c' \neq c$, as checked in line 8; thus, $\text{RANK}(c, i)$ equals 0 (resp. the number of occurrences of c in L), as returned in line 9.

To make this algorithm oblivious, each of the $O(\log(n))$ nodes visited in the search path of the tree should be accessed with a DORAM. In particular, we rely on the Oblivious Data Structure (ODS) [50] framework to obliviously access nodes in the BST. Indeed, ODS relies on the fact that each node of a BST can be reached only from another node, i.e., its parent in the tree, to store the position map entries inside ORAM blocks. Specifically, each node of the BST stores the ids of the paths of the ORAM tree containing the blocks that store the children of the node at hand. Therefore, to access any node of the BST, the client only needs to store the root node: from this one, the client chooses to fetch one of its two children, employing the corresponding leaf id stored in the root node. This



(a) Response time of accesses for our DORAMs and the Path DORAM of ZeroTrace [40]



(b) Ratio between response times of DORAMs and SORAMs accesses

Figure 2: DORAM Benchmarks

procedure is repeated to visit the entire path of the BST. In this way, the look-up of a node in a BST with n nodes stored inside an ODS requires $\log(n)$ *direct* (i.e., with no recursive position map) accesses, one for each level of the BST, to the DORAM. In our ODS, we roughly halve the look-up cost by applying a trick proposed by Gentry [17]: instead of storing all the nodes of the BST in the same DORAM, a distinct DORAM is employed for each level of the BST.

Employing distinct Circuit DORAMs (resp. Path or Ring DORAMs) to store each level of the BST allows to obliviously compute the Rank procedure with $O(\log^2(n)\cdot Z\cdot B)$ (resp. $O(\log^3(n)\cdot Z^2\cdot B)$) cost. Since $B{=}O(\log(n))$ and $Z{=}O(1)$, the ODSBWT based oblivious Rank procedure has the same $O(\log^3(n))$ computational cost of the ABWT one; nonetheless, the ODSBWT method accesses $\log_2(n)$ DORAMs with small blocks instead of $\log_C(n)$ DORAMs with large blocks, in turn allowing different implementation tradeoffs.

Oblivious Backward Search and ObSQRE Security Analysys. To make Alg. 1 oblivious, both the dictionary C and the SA *Suf* must be obliviously accessed: indeed, the entries fetched from the dictionary C would leak the characters of q, while the entries retrieved from the SA would leak the values α and β, which are related to both the string s and the substring q. To prevent such information leakages, *Suf* is stored inside a DORAM, while C (with its $|\Sigma|$ entries) is stored inside the enclave and each search over it is (obliviously) performed through a linear sweep. These implementation choices together with each of the proposed oblivious Rank procedures yield two substring search algorithms with computational cost $O((m{+}o_q)\log^3(n))$ and the following security guarantees.

Lemma 2. *Consider a document collection* $\mathbf{D}$ *with* $z{\geq}1$ *documents and* $d{\geq}1$ *substrings* $q_1,\ldots,q_d$ *with* $m_1,\ldots,m_d$ *characters, respectively. Consider a malicious adversary with full control of the machine hosting the SGX enclave, which learns the access pattern of algorithms run inside the enclave through side channel attacks. ObSQRE, when endowed with a DORAM secure as per Lemma 1, exhibits a leakage* $\mathcal{L}{=}\{n, z, m_1, o_{q_1}, \ldots, m_d, o_{q_d}\}$ *to such adversary and allows the data owner to detect any computation or data tampering.*

We formally define and prove in Appendix A.2 the security guarantees stated in Lemma 2. These security guarantees derive from three factors: the confidentiality and integrity guarantees of SGX; the control flow of our substring search algorithms being independent from any other information than the leakage $\mathcal{L}$; access pattern privacy guarantees on data structures given by our DORAMs.

5 EXPERIMENTAL RESULTS

We realized a publicly available C++ implementation [39] of ObSQRE employing the Intel SGX SDK 2.5 [12]. To encrypt blocks in the DORAM, we employed the AES implementation of WolfSSL [45]. We performed all our tests on an Ubuntu 16.04 LTS server equipped with 64 GiB of RAM memory and an Intel Xeon E3-1220 v6 CPU clocked at 3 GHz, where SGX is available. To evaluate our substring search algorithms with different alphabets, we considered three datasets: the 21-st human chromosome [3] (`Chr` in short), which encodes DNA sequences employing 7 symbols of the FASTA format [11]; the `SwissProt` database [47], (`Prot` in short), which contains $550k$ human proteins, encoded with 25 symbols as a sequence of aminoacids; the Enron dataset [24], (`Enron` in short), which contains real emails of a financial firm over an alphabet of 96 ASCII characters.

DORAM Benchmarking. First of all, we compared the response time of the Access procedure for our three DORAMs, excluding accesses to the position map in order to make these tests meaningful also for ODS. We instantiated each of these DORAMs with parameters chosen according to the configurations provided by the authors of corresponding ORAM, except for Path DORAM where we employed the same configurations of Ring DORAM. For Path and Ring DORAMs, we considered two possible configurations to explore the trade-off between the size of a bucket and the eviction period A: indeed, while smaller buckets reduce the computational cost of DORAM procedures, evictions, which are the most expensive operations, are performed more frequently. The configurations employed for our tests are reported in Table 3; we empirically verified that no stash overflow occurs after 2^{30} round robin accesses with the chosen parameters. For all the configurations, we measured the response time to access one block, averaged over 1024 accesses, for DORAMs with 2^i, $i{\in}\{5,\ldots,25\}$, real blocks storing 8 bytes of data each. In all the tests, we fully initialized all the blocks in the DORAM before measuring the response time.

Figure 2a reports the results of our benchmark, showing that Path DORAM is the fastest one among our DORAMs. This outcome

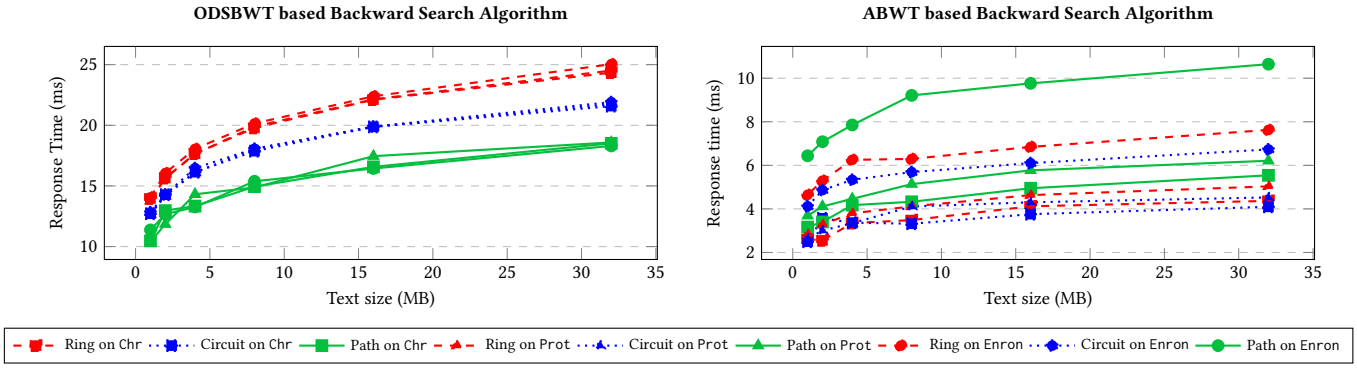


Figure 3: Comparison of ObSQRE oblivious substring search algorithms for `Chr`, `Prot` and `Enron` datasets.

Table 3: Parameters chosen for DORAMs. S is the stash size, A the eviction period, Z (resp. D) the max. (resp. min.) number of real (resp. dummy) blocks per bucket

DORAM	Z	S	D	A
Path [44]	4	32	0	3
	8	41	0	8
Circuit [49]	3	8	0	1
Ring [36]	4	32	6	3
	8	41	13	8

is motivated by the simplicity of the client operations w.r.t. Ring DORAM and by the higher eviction period than Circuit DORAM. Nonetheless, we observe that the response time of Circuit DORAM grows slower w.r.t. Path and Ring DORAMs, confirming that Circuit DORAM is asymptotically faster. It is worth noting the different impact of the eviction period and bucket size in Ring and Path DORAMs: indeed, in the former, evicting less frequently achieves better performance, even if each eviction is slower due to the larger buckets; conversely, since in Path DORAM both Eviction and FindBlock procedures become slower with larger buckets, a lower eviction period yields better performance. To compare our DORAMs with existing ones, we report in Fig. 2a the access response time of our own implementation of the Path DORAM proposed in ZeroTrace [40], referred to as ZT Path DORAM. The comparison shows that all our DORAMs are faster than ZT one. In particular, our Path DORAM is about 2× faster than ZT one, clearly showing the performance gain given by amortizing the cost of evictions over $A \geq 1$ accesses.

To assess the overhead introduced by our oblivious clients in DORAMs, in Fig. 2b we compare their access response time with the one of SORAMs, which correspond to the original versions of the ORAMs. For Path ORAM, we compare the configuration achieving best performance for the DORAM with a configuration for the Singly Oblivious RAM (SORAM) suggested by authors in [44], i.e., Z=4 and S=64. The results in Fig. 2b show that the computational overhead introduced by oblivious clients is acceptable, being at most 2.5×. As expected, the slowdown of Circuit DORAM is negligible, given the simple modifications required to make the client oblivious. Conversely, Path DORAM exhibits the highest slowdown: this overhead comes from making the Eviction procedure oblivious and from the additional path that is written back to the DORAM tree in the FindBlock one.

ObSQRE PPSS Protocol Evaluation. We now evaluate the two proposed oblivious substring search algorithms, combining each of them with each of our DORAMs, choosing, for Path and Ring ones, the most efficient configuration identified in our benchmark (Fig. 2a). We evaluate the response time to compute the number of occurrences of a substring with 24 characters for increasing sizes of the datasets, without considering the retrieval of the positions of the said occurrences as they depend neither on the specific substring search algorithm nor on the alphabet size. To achieve the maximum performance for the ABWT based algorithm, we perform an exhaustive parameter space exploration to find the optimal values for the sample period P and the factor C, which are employed to construct the ABWT and the recursive position map for the DORAM, respectively.

The results of the evaluation for both our oblivious substring search algorithms are reported in Fig. 3. We observe that, regardless of the DORAM being employed, the ABWT based algorithm is by far the fastest, as its response time is about 3 to 5 times smaller than ODSBWT one. This performance gap is due to the $\log_C(\lceil \frac{n+1}{P} \rceil)$ DORAMs accessed in the ABWT based Rank procedure instead of the $\log_2(n)$ ones in the ODSBWT based method. The comparison among different datasets reveals that the ABWT algorithm is more affected by the alphabet size $|\Sigma|$, as, regardless of the DORAM, the queries for the `Enron` dataset are slower than `Chr` and `Prot` ones. This is expected, as the size of the entries in the ABWT, and thus the block size of the DORAM storing it, depends linearly on $|\Sigma|$; conversely, ODSBWT algorithm is negligibly affected by $|\Sigma|$. Concerning the different DORAMs employed to store the full-text index, the ABWT based algorithm achieves the best performance when combined with Circuit DORAM (blue lines in the right pane in Fig. 3), while Path DORAM outperforms the other ones in the ODSBWT algorithm (green lines in the left pane in Fig. 3). However, while Path DORAM is the most efficient in our benchmarks in Fig. 2a, it exhibits the largest slowdown when employed for the ABWT algorithm. This outcome is due to the low value of the factor C (comparatively with the values derived for the other DORAMs) which is identified as optimal for Path DORAM in the previously mentioned exhaustive parameter space exploration for the ABWT

algorithm. Indeed, a low C implies a high number of deployed Path DORAMs to recursively store and access the position map. Since the only performance benefit given by a low value for C is that the blocks of all these DORAMs are smaller, our conjecture is that Path DORAM is more affected by the block size than other DORAMs; to validate our hypothesis, we evaluate the response time of our DORAMs with blocks of increasing size, hereby observing a much worse performance degradation in Path DORAM than in Circuit and Ring ones.

Once determined that ObSQRE achieves the best performance when the ABWT based backward search is paired with Circuit DORAM, we validated the practicality of this solution on two realistic use cases: the look-up of the occurrences of a DNA sequence corresponding to a protein in the entire human genome, whose size is approximately 3 GB, and the look-up of all the occurrences of three typical strings (i.e., *Fitch*, *Business Trip* and *Investment Portfolio*) in the financial domain over the whole `Enron` email corpus, whose size is about 1 GB. Furthermore, we evaluated the overhead of ObSQRE w.r.t. baseline solutions with weaker security guarantees: an application running the ABWT based substring search algorithm outside the enclave, which has no security guarantees; an application running the algorithm inside the enclave but employing Path SORAM (i.e., the fastest among our SORAMs) instead of Circuit DORAMs, which is secure only if the critical leakage of memory access patterns inside SGX is ignored (as in SGX threat model). Table 4 outlines the results of this evaluation. Although the overhead incurred by ObSQRE over a solution with no security guarantees in both the use cases amounts to about 3 order of magnitudes, ObSQRE is only 3× slower than a solution that ensures confidentiality of the data in the SGX threat model. Furthermore, the results show the practicality of ObSQRE in real-world scenarios, as the occurrences of a protein over the whole genome are found in only 1.019 seconds, while the occurrences of a string in the whole Enron corpus are found in just few milliseconds.

Finally, the benefits provided by running a DORAM client inside an enclave (on the server side) compared to the traditional setting where the ORAM client and server run on separate machines are a performance gain in overall response time which ranges from one to two orders of magnitude, in our genomic use-case. Indeed, considering the RTTs provided by the Akamai's content delivery network (CDN), from local connections to intercontinental ones [32, Tab. 1], the network latency of our genomic use-case in the traditional ORAM setting would range from 10 to 600 seconds, depending on the type of network connection, without even considering further network latencies due to recursive accesses to the position map.

Table 4: Performance of ObSQRE and less secure alternatives applied to the genomic and Enron datasets with different lengths of the queried subsring (m). Response time to compute the number of occurrences is denoted as S, while R denotes the time to retrieve all the corresponding positions

Dataset	m	#Occ.	ObSQRE (ms)	SGX+ SORAM (ms)	no SGX+ no ORAM (μs)
Genome	**3050**	1	**S:** 1019 **R:** 0.6	**S:** 347 **R:** 0.3	**S:** 167 **R:** 0.2
Enron	**5**	2657	**S:** 1.6 **R:** 2.1	**S:** 0.6 **R:** 0.8	**S:** 0.8 **R:** 10.4
Enron	**13**	290	**S:** 5.0 **R:** 0.5	**S:** 1.7 **R:** 0.2	**S:** 3.1 **R:** 1.1
Enron	**20**	154	**S:** 7.8 **R:** 0.6	**S:** 2.8 **R:** 0.2	**S:** 5.1 **R:** 0.6

6 CONCLUDING REMARKS

ObSQRE is the first solution enabling substring search queries over outsourced data combining the SGX technology and the design of a DORAM to provide private data access with no information leakage coming from memory access patterns. The experimental evaluation demonstrates the practical deployment of ObSQRE on off-the-shelf hardware with real-world genomic and financial datasets.

ACKNOWLEDGMENTS

This work was supported in part by the EU Commission grant: "WorkingAge" (H2020 RIA) Grant agreement no. 826232.

REFERENCES

[1] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. 2019. OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/obfuscuro-a-commodity-obfuscation-engine-on-intel-sgx/

[2] Ghous Amjad, Seny Kamara, and Tarik Moataz. 2019. Forward and Backward Private Searchable Encryption with SGX. In *Proceedings of the 12th European Workshop on Systems Security, EuroSec@EuroSys 2019, Dresden, Germany, March 25, 2019*. ACM, 4:1–4:6. https://doi.org/10.1145/3301417.3312496

[3] DA Benson, M Cavanaugh, K Clark, I Karsch-Mizrachi, DJ Lipman, J Ostell, and EW Sayers. 2013. GenBank. Nucleic Acids Res. (Jan 2013). https://doi.org/10.1093/nar/gks1195 The 9th International Symposium on String Processing and Information Retrieval.

[4] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*. USENIX Association. https://www.usenix.org/Conference/woot17/workshop-program/presentation/brasser

[5] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[6] Jo Van Bulck, Danile Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*.

[7] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution, See [23], 1041–1056.

[8] Michael Burrows and David Wheeler. 1994. *A block-sorting lossless data compression algorithm*. Technical Report. Digital Equipment Corporation. 18 pages. http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf

[9] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-Abuse Attacks Against Searchable Encryption, See [35], 668–679. http://dl.acm.org/citation.cfm?id=2810103

[10] Melissa Chase and Emily Shen. 2015. Substring-Searchable Symmetric Encryption. *PoPETs* 2015, 2 (2015), 263–281. http://www.degruyter.com/view/j/popets.2015.2015.issue-2/popets-2015-0014/popets-2015-0014.xml

[11] P.J. Cock, C.J. Fields, N. Goto, M.L. Heuer, and P.M. Rice. 2010. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research* 38, 6 (2010), 1767–1771. https://doi.org/10.1093/nar/gkp1137.PMC2847217.PMID20015970

[12] Intel Corporation. 2019. *Intel® Software Guard Extensions (Intel® SGX) SDK for Linux* OS* (v2.5 ed.). https://download.01.org/intel-sgx/linux-2.5/docs/

[13] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.

[14] Paolo Ferragina and Giovanni Manzini. 2005. Indexing compressed text. *J. ACM* 52, 4 (2005), 552–581. https://doi.org/10.1145/1082036.1082039

[15] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. 2018. HardIDX: Practical and secure index with SGX in a malicious environment. *Journal of Computer Security* 26, 5 (2018), 677–706. https://doi.org/10.3233/JCS-171103

[16] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*. ACM, 169–178. https://doi.org/10.1145/1536414.1536440

[17] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. 2013. Optimizing ORAM and Using It Efficiently for Secure Computation. In *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings (Lecture Notes in Computer Science)*, Vol. 7981. Springer, 1–18. https://doi.org/10.1007/978-3-642-39077-7_1

[18] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, István Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory, See [23], 217–233.

[19] Intel Corporation. 2018. *Description and mitigation overview for L1 Terminal Fault*. https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault

[20] Intel Corporation. 2020. Deep Dive: Load Value Injection. https://software.intel.com/security-software-guidance/insights/deep-dive-load-value-injection

[21] Intel Corporation. 2020. *Deep Dive: Special Register Buffer Data Sampling*. https://software.intel.com/security-software-guidance/insights/deep-dive-special-register-buffer-data-sampling

[22] Yu Ishimaki, Hiroki Imabayashi, and Hayato Yamana. 2017. Private Substring Search on Homomorphically Encrypted Data. In *2017 IEEE International Conference on Smart Computing, SMARTCOMP 2017, Hong Kong, China, May 29-31, 2017*. IEEE Computer Society, 1–6. https://doi.org/10.1109/SMARTCOMP.2017.7947038

[23] Engin Kirda and Thomas Ristenpart (Eds.). 2017. *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association.

[24] Bryan Klimt and Yiming Yang. 2004. The Enron Corpus: A New Dataset for Email Classification Research. In *Machine Learning: ECML 2004, 15th European Conference on Machine Learning, Pisa, Italy, September 20-24, 2004, Proceedings (Lecture Notes in Computer Science)*, Vol. 3201. Springer, 217–226. https://doi.org/10.1007/978-3-540-30115-8_22

[25] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (third ed.). Addison-Wesley, Boston.

[26] Iraklis Leontiadis and Ming Li. 2018. Storage Efficient Substring Searchable Symmetric Encryption. In *Proceedings of the 6th International Workshop on Security in Cloud Computing, SCC@AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*. ACM, 3–13. https://doi.org/10.1145/3201595.3201598

[27] Nicholas Mainardi, Alessandro Barenghi, and Gerardo Pelosi. 2019. Privacy preserving substring search protocol with polylogarithmic communication cost. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*. ACM, 297–312. https://doi.org/10.1145/3359789.3359842

[28] Udi Manber and Eugene W. Myers. 1993. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.* 22, 5 (1993), 935–948. https://doi.org/10.1137/0222058

[29] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 279–296. https://doi.org/10.1109/SP.2018.00045

[30] Tarik Moataz and Erik-Oliver Blass. 2015. Oblivious Substring Search with Updates. *IACR Cryptology ePrint Archive* 2015 (2015), 722. http://eprint.iacr.org/2015/722

[31] Ge Nong, Sen Zhang, and Wai Hong Chan. 2009. Linear Suffix Array Construction by Almost Pure Induced-Sorting. In *2009 Data Compression Conference (DCC 2009), 16-18 March 2009, Snowbird, UT, USA*. IEEE Computer Society, 193–202. https://doi.org/10.1109/DCC.2009.42

[32] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. 2010. The Akamai network: a platform for high-performance internet applications. *Operating Systems Review* 44, 3 (2010), 2–19. https://doi.org/10.1145/1842733.1842736

[33] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 227–240. https://www.usenix.org/conference/atc18/presentation/oleksenko

[34] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *42th IEEE Symposium on Security and Privacy (S&P'21)*. https://download.vusec.net/papers/crosstalk_sp21.pdf Intel Bounty Reward.

[35] Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). 2015. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. ACM. http://dl.acm.org/citation.cfm?id=2810103

[36] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2015. Constants Count: Practical Improvements to Oblivious RAM. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. USENIX Association, 415–430. https://www.usenix.org/Conference/usenixsecurity15/technical-sessions/presentation/ren-ling

[37] Ling Ren, Christopher W. Fletcher, Xiangyao Yu, Marten van Dijk, and Srinivas Devadas. 2013. Integrity verification for path Oblivious-RAM. In *IEEE High Performance Extreme Computing Conference, HPEC 2013, Waltham, MA, USA, September 10-12, 2013*. IEEE, 1–6. https://doi.org/10.1109/HPEC.2013.6670339

[38] Cédric Van Rompay, Refik Molva, and Melek Önen. 2017. A Leakage-Abuse Attack Against Multi-User Searchable Encryption. *PoPETs* 2017, 3 (2017), 168. https://doi.org/10.1515/popets-2017-0034

[39] Davide Sampietro and Nicholas Mainardi. 2020. ObSQRE: Oblivious Substring Queries on Remote Enclave. https://github.com/DavideSampietro/ObSQRE

[40] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_02B-4_Sasy_paper.pdf

[41] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/t-sgx-eradicating-controlled-channel-attacks-against-enclave-programs/

[42] Kana Shimizu, Koji Nuida, and Gunnar Rätsch. 2016. Efficient privacy-preserving string search and an application in genomics. *Bioinformatics* 32, 11 (2016), 1652–1661. https://doi.org/10.1093/bioinformatics/btw050

[43] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*. ACM, 317–328. https://doi.org/10.1145/2897845.2897885

[44] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. ACM, 299–310. https://doi.org/10.1145/2508859.2516660

[45] Larry Stefonic and Todd Ouska. 2019. The wolfSSL Embedded TLS Library. https://www.wolfssl.com/

[46] Mikhail Strizhov, Zachary Osman, and Indrajit Ray. 2016. Substring Position Search over Encrypted Cloud Data Supporting Efficient Multi-User Setup. *Future Internet* 8, 3 (2016), 28. https://doi.org/10.3390/fi8030028

[47] The UniProt Consortium. 2018. UniProt: the universal protein knowledgebase. *Nucleic Acids Research* 46, 5 (02 2018), 2699–2699. https://doi.org/10.1093/nar/gky092 arXiv:http://oup.prod.sis.lan/nar/article-pdf/46/5/2699/24388239/gky092.pdf

[48] Bing Wang, Wei Song, Wenjing Lou, and Y. Thomas Hou. 2017. Privacy-preserving pattern matching over encrypted genetic data in cloud computing. In *2017 IEEE Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*. IEEE, 1–9. https://doi.org/10.1109/INFOCOM.2017.8057178

[49] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound, See [35], 850–861. https://doi.org/10.1145/2810103.2813634

[50] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. ACM, 215–226. https://doi.org/10.1145/2660267.2660314

[51] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 640–656. https://doi.org/10.1109/SP.2015.45

A FORMAL SECURITY ANALYSIS

We provide here the formal proofs of the security guarantees of ObSQRE, hereby formalizing the security notion reported in Lemma 2. In our threat model, we assume a powerful malicious adversary that has total control of the machine where the SGX enclave is running. Thus, thanks to the security guarantees of SGX, the attacker has no direct access to the data stored inside the enclave and it cannot interfere with the computation performed within the enclave; nonetheless, it has access to every data outside the enclave and may tamper with the computation performed outside the enclave.

Experiment $out \leftarrow \mathbf{Real}_{\rho,\mathcal{A}}(\lambda)$:
$(D, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_{Init}(1^{\lambda})$
$res_0 \leftarrow \rho.\textsc{Init}([D_i]_{i=1}^{n})$
$\forall i \in \{1,\ldots,d\}$:
$(bid_i, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_{Acc,i}(\mathtt{st}_{\mathcal{A}}, D, [bid_j]_{j=1}^{i-1}, \mathcal{L}_{Init}, [\mathcal{L}_{Acc,j}]_{j=1}^{i-1}, \mathcal{T}_{Init}, [\mathcal{T}_{Acc,j}]_{j=1}^{i-1})$
$res_i \leftarrow \rho.\textsc{Access}(bid_i)$
$out \leftarrow \{[res_i]_{i=0}^{d}, [\mathcal{T}_{Acc,i}]_{i=1}^{d}, \mathcal{T}_{Init}, \mathtt{st}_{\mathcal{A}}\}$

Experiment $out \leftarrow \mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}(\lambda)$:
$(D, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_{Init}(1^{\lambda})$
$(D^{\mathcal{S}}, \mathtt{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_{Init}(\mathcal{L}_{Init})$
$res_0 \leftarrow \rho.\textsc{Init}([D_i^{\mathcal{S}}]_{i=1}^{n})$
$\forall i \in \{1,\ldots,d\}$:
$(bid_i, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_{Acc,i}(\mathtt{st}_{\mathcal{A}}, D, [bid_j]_{j=1}^{i-1}, \mathcal{L}_{Init}, [\mathcal{L}_{Acc,j}]_{j=1}^{i-1}, \mathcal{T}_{Init}, [\mathcal{T}_{Acc,j}]_{j=1}^{i-1})$
$(bid_i^{\mathcal{S}}, \mathtt{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_{Acc,i}(\mathtt{st}_{\mathcal{S}}, \mathcal{L}_{Init}, [\mathcal{L}_{Acc,j}]_{j=1}^{i})$
$res_i \leftarrow \rho.\textsc{Access}(bid_i^{\mathcal{S}})$
if $res_i \neq \mathtt{abort}$: $res_i \leftarrow D_{bid_i}$
$out \leftarrow \{[res_i]_{i=0}^{d}, [\mathcal{T}_{Acc,i}]_{i=1}^{d}, \mathcal{T}_{Init}, \mathtt{st}_{\mathcal{A}}\}$

Figure 4: Security Experiments for DORAM protocol ρ

Our security analysis is carried out in two main steps: first, we define and prove the security guarantees of our DORAMs; then, we define and prove the security guarantees of ObSQRE assuming that a secure DORAM is employed. For the sake of conciseness, we prove the security only for the components of our best performing solution, composed by Circuit DORAM and ABWT based oblivious substring search algorithm; similar proofs can be constructed for Path and Ring DORAMs and for the ODSBWT algorithm.

As customary in the context of privacy-preserving protocols, our security definitions are based on the simulation paradigm, which mandates to prove the security guarantees of the protocol by showing that its actual execution can be simulated by a simulator $\mathcal{S}$ which knows only the information $\mathcal{L}$ leaked to the adversary in the actual execution of the protocol. The leakage $\mathcal{L}$ must be defined beforehand, and it is specific for each protocol; the existence of the simulator $\mathcal{S}$ allows to prove that the adversary learns no more information than $\mathcal{L}$ throughout the execution of the protocol.

A.1 DORAM Security

We assume that the DORAM stores a dataset D split in n blocks, each of size B bits. In our security definition, besides the Access procedure already described throughout the manuscript, we consider also an Init procedure, which is employed to build the DORAM tree and insert the n blocks of D in the DORAM: specifically, for each block, the algorithm obliviously adds to the stash the block at hand and then evicts the stash to the DORAM tree following the eviction strategy of the DORAM. In order to properly construct the DORAM tree, the Init procedure employs several parameters specified by the user: the recursive factor C to build the recursive position map, the maximum number of real blocks per bucket Z, the stash size S, the number of dummy blocks per buckets D and the eviction period A. The security guarantees of DORAM are not weakened if the adversary knows these values, since they depend only on the number of blocks of the DORAM; thus, to simplify our security analysis we assume that the Init procedure always employs the same values for these parameters. Since our DORAMs employs an integrity-check mechanism based on Merkle-trees, both the Init and Access procedures may return the special value `abort` in case they detect data tampering on the DORAM.

To outline the security definition for the DORAM primitive, we need to introduce two further concepts: the trace $\mathcal{T}$ and the leakage $\mathcal{L}$ for the adversary. The former represents the information directly observed by the adversary while interacting with the DORAM:

Definition 1 (Trace of DORAM). *Given a DORAM whose client runs inside an SGX enclave and the DORAM tree is stored in the unprotected memory, the trace of the DORAM is $\mathcal{T} = \{Code_{AP}, Data_{AP}, Data_{Srv}\}$, with $Code_{AP}$ and $Data_{AP}$ denoting the code and data access patterns of the DORAM client, respectively, while $Data_{Srv}$ denotes the information sent by the DORAM client outside the enclave.*

In our security definition, we split the trace in two components $\mathcal{T}_{Init}$ and $\mathcal{T}_{Acc}$, which refer to the trace of the Init and the Access procedures, respectively. The leakage $\mathcal{L}$ denotes the information about the dataset and the accessed blocks which is inferred by the adversary from the trace $\mathcal{T}$. We split the leakage $\mathcal{L}$ in two components $\mathcal{L}_{Init}$ and $\mathcal{L}_{Acc}$ that represent the information inferred by the adversary from $\mathcal{T}_{Init}$ and $\mathcal{T}_{Acc}$, respectively.

Definition 2 (DORAM Security). *Given a security parameter λ, a DORAM ρ with trace $\mathcal{T}$ as in Def. 1, leakage $\mathcal{L} = \{\mathcal{L}_{Init}, \mathcal{L}_{Acc}\}$ and an integer $d \geq 1$, consider the two interactive experiments $\mathbf{Real}_{\rho,\mathcal{A}}$ and $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$, outlined in Fig. 4, between a challenger and an adversary $\mathcal{A}$ consisting of $d+1$ probabilistic polynomial time algorithms, i.e., $\mathcal{A}=\{\mathcal{A}_{Init}, \mathcal{A}_{Acc,1}, \ldots, \mathcal{A}_{Acc,d}\}$. Throughout the experiments, the challenger may invoke the DORAM ρ and a simulator $\mathcal{S}$ consisting of $d+1$ probabilistic polynomial time algorithms, i.e., $\mathcal{S}=\{\mathcal{S}_{Init}, \mathcal{S}_{Acc,1}, \ldots, \mathcal{S}_{Acc,d}\}$; the adversary $\mathcal{A}$ can tamper with data and computation of the DORAM as described in our threat model. Denoting as $\mathcal{D}$ a probabilistic polynomial time algorithm that, given the output o of an experiment determines if o refers to $\mathbf{Real}_{\rho,\mathcal{A}}$ ($\mathcal{D}(o) = 0$) or $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ ($\mathcal{D}(o) = 1$) experiment, the DORAM ρ, with leakage $\mathcal{L}$ and trace $\mathcal{T}$, is secure against malicious probabilistic polynomial time adversaries $\mathcal{A}$ if, for every possible $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for every $\mathcal{D}$:*

$$\mathtt{Pr}(\mathcal{D}(o)=1 | o \leftarrow \mathbf{Real}_{\rho,\mathcal{A}}) - \mathtt{Pr}(\mathcal{D}(o)=1 | o \leftarrow \mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}) \leq \epsilon(\lambda)$$

where $\epsilon(\cdot)$ is a negligible function.

In the experiments outlined in Fig. 4, both the adversary and the simulator are stateful, that is they have a state $\mathtt{st}_{\mathcal{A}}$ (resp. $\mathtt{st}_{\mathcal{S}}$) employed to store any information learned throughout the experiment. In short, our security definition is satisfied if the outputs of the two experiments outlined in Fig. 4 are computationally indistinguishable. This property is sufficient to guarantee that the

DORAM protocol leaks to any malicious adversary no more information than the leakage $\mathcal{L}$. Indeed, the information available to the adversary at the end of the $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiment, namely the trace of the DORAM $\mathcal{T}$ and the state $\mathtt{st}_{\mathcal{A}}$ of the adversary, depends on the fake input data $\{D^{\mathcal{S}}, [bid_i^{\mathcal{S}}]_{i=1}^{d}\}$ constructed by the simulator $\mathcal{S}$. Since $\mathcal{S}$constructs the fake input data by knowing only the leakage $\mathcal{L}$ provided by the challenger, no more information than $\mathcal{L}$ about the actual data can be inferred by the adversary from the operations observed over fake data. Since this experiment is computationally indistinguishable from the $\mathbf{Real}_{\rho,\mathcal{A}}$ one, then the information available to the adversary in the $\mathbf{Real}_{\rho,\mathcal{A}}$ experiment cannot reveal more information than $\mathcal{L}$ about the actual data.

Furthermore, since the results of the d+1 operations must be indistinguishable between the two experiments, our definition guarantees also that a malicious adversary cannot affect the result of the computation without being detected by the DORAM client. Indeed, in the $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiment, the challenger ensures that the result of the computation is always the correct one, unless the DORAM ρ has detected a misbehavior of the adversary. Thus, in case there exists a misbehavior of the adversary affecting the correctness of the result that is not detected by the DORAM, the result would be correct in the $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiment but wrong in the $\mathbf{Real}_{\rho,\mathcal{A}}$ one, hence making them distinguishable.

Theorem 1. *Our DORAMs meet the security guarantees of Def. 2 against a malicious adversary with leakage* $\mathcal{L} = \{\mathcal{L}_{Init}, \mathcal{L}_{Acc,1}, \ldots, \mathcal{L}_{Acc,d}\}$, *where* $\mathcal{L}_{Init} = \{n, B\}$ *and* $\mathcal{L}_{Acc,i} = \emptyset$, $i = 1, \ldots, d$.

Proof. To prove this statement, we show the construction of a simulator $\mathcal{S}$ that makes the transcript of the $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiment computationally indistinguishable from the $\mathbf{Real}_{\rho,\mathcal{A}}$ one.

Simulator $\mathcal{S}_{Init}$. This algorithm, given $\mathcal{L}_{Init}$ as input, randomly samples n blocks $D_i^{\mathcal{S}}$ of B bits, and then it invokes the Init procedure of the DORAM to instantiate the DORAM with these n blocks. The traces $\mathcal{T}_{Init}$ observed by the adversary in the two experiments are indistinguishable: indeed, the DORAM is constructed with similar parameters, and the Init procedure accesses the same blocks in both the experiments (as path for evictions are chosen according to a deterministic schedule). Since the blocks inserted in the DORAM are encrypted with a semantically secure cipher, it is not possible to distinguish the blocks with random data employed in the $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiment from the blocks with actual data of the $\mathbf{Real}_{\rho,\mathcal{A}}$ one. The result res_0 of the Init procedure is the same in both experiments, as the tampering of the tree is detected independently from the data stored inside the DORAM. Furthermore, except for such tampering, there are no other adversarial behaviors that may alter the result of the computation, since it entirely involves code and data stored inside the SGX enclave.

Simulator $\mathcal{S}_{\mathbf{Acc,i}}, \mathbf{i} \in \{\mathbf{1}, \ldots, \mathbf{d}\}$. This simulator simply chooses at random the block id $bid_i^{\mathcal{S}}$ to be accessed by the DORAM. We now show that the traces $\mathcal{T}_{Acc,i}$ of our Circuit DORAM observed by the adversary in both the experiments are computationally indistinguishable. We start by proving the following claim about the code and data access patterns of the client algorithm in Circuit DORAM:

Theorem 2. *The code and data access patterns (*$Code_{AP}$ *and* $Data_{AP}$*) of our Circuit DORAM client in the Access procedure are independent from the block id bid given as input to the procedure*

Proof. The Access procedure of the DORAM has two main phases: the former retrieves from the position map the leaf id lid corresponding to block bid, replacing lid with a new random leaf id lid' in the corresponding entry of the position map; the latter employs the FindBlock and Eviction procedures to retrieve the block with id bid from the DORAM. We first prove our claim for these two procedures; then, we prove it also for the first phase in the case the position map is recursively stored in several DORAMs.

FindBlock. The DORAM client executes the FindBlock procedure reported in Alg. 2. Both the control flow and the memory locations accessed by this procedure are clearly independent from the block id bid: indeed, the former depends only on parameters of the DORAM known to the adversary, while all conditionally dependent memory accesses are performed through oblivious operations.

Eviction. This procedure, reported in Alg. 5, is executed over two paths, chosen by the client with a deterministic schedule known to the adversary and independent from bid; furthermore, as the control flow of this procedure depends only on parameters of the DORAM known to the adversary and all the conditionally dependent memory accesses are performed through oblivious operations, both code and data access patterns of this procedure are independent from the block id bid.

Recursive Position Map. We recall that the position map of the DORAM is stored in $O(\log_C(n))$ DORAMs of increasing size, and the client stores only the position map of the smallest among these DORAMs. The algorithm to retrieve the leaf id corresponding to block bid has $O(log_C(n))$ iterations; in each of them, the algorithm accesses one of the DORAMs storing the position map, hinging upon FindBlock and Eviction procedures: as we have just shown, their operations are independent from bid. The algorithm, after retrieving a block from these DORAMs, performs a linear sweep over such block; this block is a small array with $O(C)$ entries, and each of them is obliviously swapped with a target memory location through OblSwap operation. Therefore, the sweep over the block does not depend on bid. No other operations are performed in each iteration of the algorithm, thus making the retrieval of the leaf id lid corresponding to the block with id bid independent from bid. □

The claim in Thm. 2 implies that there is no difference on the access patterns observed by the adversary in the $\mathbf{Real}_{\rho,\mathcal{A}}$ and $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiments. Regarding $Data_{srv}$, that is the information sent outside the enclave from the DORAM client, we observe that this is limited to the leaf ids of the paths being fetched or evicted, and the blocks of these paths written back to the DORAM tree. The paths to be fetched in our DORAMs are chosen in the same way as in the corresponding ORAM; thus, the leaf ids of these paths are distributed as in the corresponding ORAM. Since leaf ids of fetched paths in Circuit ORAM are uniformly distributed, independently from the accessed blocks, then the distribution of the leaf ids fetched by our DORAM is uniform in both the $\mathbf{Real}_{\rho,\mathcal{A}}$ and $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiments. Regarding the paths being evicted, in all our DORAMs they are chosen according to a deterministic schedule that depends only on the eviction period A, which is the same in both experiments; thus, the ids of the evicted paths cannot be employed to distinguish the traces between the two experiments. Finally, since the blocks are encrypted with a semantically secure

Experiment $out \leftarrow \mathbf{Real}_{\mathcal{P},\mathcal{A}}(\lambda)$:

$(\mathbf{D}, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_0(1^\lambda)$
$\mathcal{I} \leftarrow \mathcal{P}.\textsc{Setup}(\mathbf{D})$
$res_0 \leftarrow \mathcal{P}.\textsc{Load}(\mathcal{I})$
$\forall i \in \{1,\ldots,d\}$:
  $(q_i, occ_i, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_i(\mathtt{st}_{\mathcal{A}}, \mathbf{D}, [q_j]_{j=1}^{i-1}, [occ_j]_{j=1}^{i-1}, \mathcal{L}_{Setup}, \mathcal{L}_{Load}, [\mathcal{L}_{Query,j}]_{j=1}^{i-1}, \mathcal{T}_{Setup}, \mathcal{T}_{Load}, [\mathcal{T}_{Query,j}]_{j=1}^{i-1})$
  $res_i \leftarrow \mathcal{P}.\textsc{Query}(q_i, occ_i)$
$out \leftarrow \{[res_i]_{i=0}^{d}, [\mathcal{T}_{Query,i}]_{i=1}^{d}, \mathcal{T}_{Setup}, \mathcal{T}_{Load}, \mathtt{st}_{\mathcal{A}}\}$

Experiment $out \leftarrow \mathbf{Ideal}_{\mathcal{P},\mathcal{A},\mathcal{S}}(\lambda)$:

$(\mathbf{D}, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_0(1^\lambda)$
$(\mathcal{I}^{\mathcal{S}}, \mathtt{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_0(\mathcal{L}_{Setup}, \mathcal{L}_{Load})$
$res_0 \leftarrow \mathcal{P}.\textsc{Load}(\mathcal{I}^{\mathcal{S}})$
$\forall i \in \{1,\ldots,d\}$:
  $(q_i, occ_i, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_i(\mathtt{st}_{\mathcal{A}}, \mathbf{D}, [q_j]_{j=1}^{i-1}, [occ_j]_{j=1}^{i-1}, \mathcal{L}_{Setup}, \mathcal{L}_{Load}, [\mathcal{L}_{Query,j}]_{j=1}^{i-1}, \mathcal{T}_{Setup}, \mathcal{T}_{Load}, [\mathcal{T}_{Query,j}]_{j=1}^{i-1})$
  $(q_i^{\mathcal{S}}, occ_i^{\mathcal{S}}, \mathtt{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_i(\mathtt{st}_{\mathcal{S}}, \mathcal{L}_{Setup}, \mathcal{L}_{Load}, [\mathcal{L}_{Query,j}]_{j=1}^{i})$
  $res_i \leftarrow \mathcal{P}.\textsc{Query}(q_i^{\mathcal{S}}, occ_i^{\mathcal{S}})$
  if $res_i \neq \mathtt{abort}$: $res_i \leftarrow R_{q_i}$
$out \leftarrow \{[res_i]_{i=0}^{d}, [\mathcal{T}_{Query,i}]_{i=1}^{d}, \mathcal{T}_{Setup}, \mathcal{T}_{Load}, \mathtt{st}_{\mathcal{A}}\}$

Figure 5: Security Experiments for PPSS protocol $\mathcal{P}$

scheme, the paths being written back after fetch or eviction appear as indistinguishable random data in both the experiments. In conclusion, the traces $\mathcal{T}_{Acc,i}$, $i = 1,\ldots,d$ are computationally indistinguishable between the $\mathbf{Real}_{\rho,\mathcal{A}}$ and $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiments.

Regarding the results res_i of the Access procedure, the integrity check mechanism ensures that any tampering on the path fetched from the DORAM tree is detected in both experiments. Conversely, in case the adversary decides to tamper with a randomly chosen path before knowing which path will be fetched, the results between the two experiments may differ; nonetheless, as the adversary cannot guess with other than uniform probability the path being fetched in the Access procedure in both experiments, the statistical distribution of tampering detection is equivalent to the distribution of correctly guessing the path being fetched, which is uniform in both experiments. The adversary has no other ways to tamper with data and computation, thus proving that the results res_i are computationally indistinguishable between the experiments. □

A.2 ObSQRE Security Analysis

We now prove the security guarantees of ObSQRE, assuming that a DORAM fulfilling the security requirements of Thm. 1 is employed in our oblivious substring search algorithms. Some of these algorithms may employ several parameters, such as the sample period employed in the construction of the ABWT; since these parameters are not sensitive and can be derived by the adversary itself, for simplicity in our security analysis we assume that the same value is always employed. Similarly, we assume that the alphabet of the documents in the document collection $\mathbf{D}$ is publicly known.

As in the DORAM security analysis, we define the traces of the three procedures as the information observed by the adversary throughout the execution of these procedures. For the ones running inside the enclave, namely Load and Query, the traces $\mathcal{T}_{Load}$ and $\mathcal{T}_{Query}$ are defined as in Def. 1, with the DORAM client being trivially replaced by the corresponding procedure; instead, since the Setup procedure is executed at data-owner's side, its trace $\mathcal{T}_{Setup}$ is limited to the encrypted full-text index sent to the untrusted server, denoted as $\mathcal{I}$. We also define the leakage $\mathcal{L}$ as the information inferred by the adversary about the document collection $\mathbf{D}$, the substring queried q and the occurrences of q in $\mathbf{D}$; we split $\mathcal{L}$ in three components $\mathcal{L}_{Setup}$, $\mathcal{L}_{Load}$ and $\mathcal{L}_{Query}$, denoting the leakage in the corresponding ObSQRE procedure.

In our security definition, we consider a modified Query procedure that, instead of retrieving all the o_q occurrences of a string q in $\mathbf{D}$, allows to specify the number occ of occurrences to be retrieved. This procedure can be implemented by fetching occ entries instead of o_q ones from the suffix array SA in the second phase of backwards search algorithm (lines 6-7 of Alg. 1). This modification allows to prove that ObSQRE is secure against an adversary that can choose the query to be performed after observing the traces of previous queries. In this way, the protocol remains secure even if the adversary can somehow control ObSQRE operations (e.g., forcing to always querying the same string).

Definition 3 (PPSS Security). *Given a security parameter λ, a PPSS protocol $\mathcal{P}$ with trace $\mathcal{T} = \{\mathcal{T}_{Setup}, \mathcal{T}_{Load}, \mathcal{T}_{Query}\}$, leakage $\mathcal{L} = \{\mathcal{L}_{Init}, \mathcal{L}_{Load}, \mathcal{L}_{Query}\}$ and an integer $d \geq 1$, consider the two interactive experiments $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ and $\mathbf{Ideal}_{\mathcal{P},\mathcal{A},\mathcal{S}}$, outlined in Fig. 5, between a challenger and an adversary $\mathcal{A}$ consisting of d+1 probabilistic polynomial time algorithms $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_d$. Throughout the experiments, the challenger may invoke the protocol $\mathcal{P}$ and a probabilistic polynomial time simulator $\mathcal{S}$ consisting of d+1 probabilistic polynomial time algorithms $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_d$; the adversary $\mathcal{A}$ can tamper with data and computation of the PPSS protocol as described in our threat model. Denoting as $\mathcal{D}$ a probabilistic polynomial time algorithm that, given the output o of an experiment determines if o refers to $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ ($\mathcal{D}(o) = 0$) or $\mathbf{Ideal}_{\mathcal{P},\mathcal{A},\mathcal{S}}$ ($\mathcal{D}(o) = 1$) experiment, the PPSS protocol $\mathcal{P}$, with leakage $\mathcal{L}$ and trace $\mathcal{T}$, is secure against malicious probabilistic polynomial time adversaries $\mathcal{A}$ if, for every possible $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for every $\mathcal{D}$:*

$$\mathtt{Pr}(\mathcal{D}(o)=1 | o \leftarrow \mathbf{Real}_{\mathcal{P},\mathcal{A}}) - \mathtt{Pr}(\mathcal{D}(o)=1 | o \leftarrow \mathbf{Ideal}_{\mathcal{P},\mathcal{A},\mathcal{S}}) \leq \epsilon(\lambda)$$

where $\epsilon(\cdot)$ is a negligible function.

Theorem 3. *For a document collection $\mathbf{D}=\{D_1,\ldots,D_z\}$ with $z \geq 1$ documents and $d \geq 1$ strings $q_1,\ldots,q_d$, assuming that a DORAM with the security guarantees outlined in Thm. 1 is employed, ObSQRE is secure according to Def. 3 with a leakage $\mathcal{L} = \{\mathcal{L}_{Setup}, \mathcal{L}_{Load}, \mathcal{L}_{Query,1},\ldots,\mathcal{L}_{Query,d}\}$ defined as:*

- $\mathcal{L}_{Setup} = \{z, n = \sum_{i=j}^{z} |D_j|\}$
- $\mathcal{L}_{Load} = \{n\}$
- $\mathcal{L}_{Query,i} = \{m_i, occ_i\}, i \in \{1,\ldots,d\}$, *where $m_i = |q_i|$ and occ_i is the number of occurrences of q_i in $\mathbf{D}$ retrieved by the user*

Proof. To prove the theorem, we describe the simulator $\mathcal{S}$ and we show that the output of the $\mathbf{Ideal}_{\mathcal{P},\mathcal{A},\mathcal{S}}$ experiment is computationally indistinguishable from the output of the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ one.

Simulator $\mathcal{S}_0$. This simulator, upon receiving $\mathcal{L}_{Setup}$ and $\mathcal{L}_{Load}$, constructs a document collection $\mathbf{D}^{\mathcal{S}}$ over the publicly known alphabet Σ by randomly sampling z strings whose lengths sum up to n. Then, the simulator computes the ABWT-based full-text index from $\mathbf{D}^{\mathcal{S}}$ and encrypts it with an AEAD scheme, obtaining the encrypted index $\mathcal{I}^{\mathcal{S}}$. This index has the same size of the index $\mathcal{I}$ computed from the document collection $\mathbf{D}$ chosen by the adversary; furthermore, each of its entries are encrypted with a semantically secure scheme, in turn making the indexes $\mathcal{I}, \mathcal{I}^{\mathcal{S}}$ (and thus the traces $\mathcal{T}_{Setup}$ in the experiments) computationally indistinguishable. The traces $\mathcal{T}_{Load}$ are also computationally indistinguishable in both the experiments. Indeed, in the Load procedure, $\mathcal{I}^{\mathcal{S}}$ is decrypted, and then the ABWT and the SA are inserted into the DORAM through the Init operation. The security guarantees of the DORAM ensures that this operation leaks only the number and the size of the DORAM blocks: their number is proportional, for both the ABWT and the SA, to n, which is the same in both experiments; the size of each block corresponds to the size of each entry of the ABWT and the SA, respectively, which are already known to the adversary. Finally, the security guarantees of the DORAM ensures that Init procedure is secure against any tampering to the DORAM tree, while the AEAD scheme guarantees that any tampering on the encrypted indexes $\mathcal{I}, \mathcal{I}^{\mathcal{S}}$ is detected in the Load procedure, hence making the results res_0 equivalent in both experiments.

Simulator $\mathcal{S}_{\mathbf{i}}, \mathbf{i} \in \{\mathbf{1}, \ldots, \mathbf{d}\}$. This simulator, upon receiving the leakage $\mathcal{L}_{Query,i}$, chooses a random string $q_i^{\mathcal{S}}$ of length m_i and sets $occ_i^{\mathcal{S}} = occ_i$. The Query procedure employs the oblivious backward search algorithm with ABWT based oblivious Rank procedure. The number of iterations of backwards search depends only on m_i and occ_i in both the experiments. The linear sweeps over the dictionary C adds to the trace $\mathcal{T}_{Query}$ only its size $|\Sigma|$, as each entry is involved in an oblivious write. The oblivious Rank procedure, outlined in Alg. 6, retrieves a block from the DORAM, whose security guarantees ensures that no information is leaked during the Access operation. After retrieving such block, the Rank procedure obliviously sweeps over this block, an operation that reveals only the block size, which is already known from $\mathcal{T}_{Load}$. Concerning the result of the query, all the operations, except for the DORAM Access, are performed inside the SGX enclave, where any code and data tampering is prevented. As the security guarantees of DORAM ensures that accesses are secure against any tampering strategy, then the results of the queries are computationally indistinguishable in both experiments. □

A.3 Oblivious EarlyReshuffle Analysis

We prove that the strategy employed by EarlyReshuffle procedure of our Ring DORAM places Z blocks out of the $Z+D$ slots available in the bucket uniformly at random. To this extent, we define the event $\mathcal{E}_{i,j}$, $i \in \{1, \ldots, Z\}$, $j \in \{0, \ldots, Z+D-1\}$, which is verified if the slot j of the bucket is full in the i-th iteration. Similarly, we define the event $\mathcal{B}_{i,j}$, $i \in \{1, \ldots, Z\}$, $j \in \{0, \ldots, Z+D-1\}$, which is verified if the i-th block placed by EarlyReshuffle is assigned to the slot j of the bucket. Clearly, the i-th block is assigned to the slot j if and only if this slot is chosen in the i-th iteration and it is never chosen in all previous iterations, i.e, $\mathcal{B}_{i,j} = \bigwedge_{h=1}^{i-1} \neg\mathcal{E}_{h,j} \wedge \mathcal{E}_{i,j}$. The probability of the event $\mathcal{B}_{ij}$ can be thus computed as:

$$\begin{aligned}
\Pr(\mathcal{B}_{i,j}) &= \Pr(\bigwedge_{h=1}^{i-1} \neg\mathcal{E}_{h,j} \wedge \mathcal{E}_{ij}) = \Pr(E_{i,j} | \bigwedge_{h=1}^{i-1} \neg\mathcal{E}_{hj}) \Pr(\bigwedge_{h=1}^{i-1} \neg\mathcal{E}_{h,j}) \\
&= \Pr(E_{i,j} | \bigwedge_{h=1}^{i-1} \neg\mathcal{E}_{h,j}) \Pr(\neg\mathcal{E}_{i-1,j} | \Pr(\bigwedge_{h=1}^{i-2} \neg\mathcal{E}_{h,j})) \Pr(\bigwedge_{h=1}^{i-2} \neg\mathcal{E}_{h,j}) \quad (1) \\
&= \Pr(E_{i,j} | \bigwedge_{h=1}^{i-1} \neg\mathcal{E}_{h,j}) \Pr(\neg\mathcal{E}_{i-1,j} | \Pr(\bigwedge_{h=1}^{i-2} \neg\mathcal{E}_{h,j}) \cdots \Pr(\neg\mathcal{E}_{1,j})
\end{aligned}$$

We now compute each of these probabilities. $\Pr(\neg\mathcal{E}_{1,j})$ is the probability that the slot j is not chosen in the first iteration; since each of the $Z+D$ slots may be chosen with uniform probability, $\Pr(\neg\mathcal{E}_{1,j}) = \frac{Z+D-1}{Z+D}$. $\Pr(\neg E_{h,j} | \bigwedge_{k=1}^{h-1} \neg\mathcal{E}_{k,j})$ is the probability that the slot j is not chosen among the $Z+D-h+1$ ones still available in the h-th iteration; since each of them may be chosen with uniform probability, then $\Pr(\neg E_{h,j} | \bigwedge_{k=1}^{h-1} \neg\mathcal{E}_{k,j}) = \frac{Z+D-h}{Z+D-h+1}$. Finally, $\Pr(E_{i,j} | \bigwedge_{z=1}^{i-1} \neg\mathcal{E}_{z,j})$ is the probability that the slot j is chosen in the i-th iteration; as the slot is chosen uniformly at random among $Z+D-i+1$ ones, then $\Pr(E_{i,j} | \bigwedge_{z=1}^{i-1} \neg\mathcal{E}_{z,j}) = \frac{1}{Z+D-i+1}$. Substituting these probabilities in Equation 1, we obtain:

$$\Pr(\mathcal{B}_{i,j}) = \frac{1}{Z+D-i+1} \prod_{h=1}^{i-1} \frac{Z+D-h}{Z+D-h+1} = \frac{1}{Z+D}$$

Since the analysis may be repeated for each slot j and for each of the z blocks, we conclude that each block is placed with uniform probability over all the $Z+D$ slots of the bucket.