Memory Access Pattern Protection for Resource-constrained Devices

Yuto Nakano[1], Carlos Cid[2], Shinsaku Kiyomoto[1], and Yutaka Miyake[1]

[1] KDDI R&D Laboratories Inc.
2-1-15 Ohara, Fujimino, Saitama 356-8502, Japan
{yuto,kiyomoto,miyake}@kddilabs.jp
[2] Information Security Group, Royal Holloway, University of London
Egham, TW20 0EX, UK
Carlos.Cid@rhul.ac.uk

Abstract. We propose a practice-oriented scheme for protecting RAM access pattern. We first consider an instance which relies on the use of a secure (trusted) hardware buffer; it achieves both security and performance levels acceptable in practice by adapting ideas from oblivious RAM mechanisms, yet without the expensive (re-)shuffling of buffers. Another instance requires no special hardware, but as a result leads to a higher, yet practical overhead. One of the main features of the proposal is to maintain the *history* of memory access to help hiding the access pattern. We claim that under reasonable assumptions, the first scheme with trusted memory is secure with overhead of only $6\times$, as is the second scheme with overhead of $(2m+2\ell_h+2)\times$ where m and ℓ_h are respectively the size of the buffer and history. We note that although the proposal is particularly focused on the software execution protection environment, its security may well be appropriate for most uses in the remote storage environment, to prevent access pattern leakage of cloud storage with much lower performance overhead than existing solutions.

Key words: Access Pattern Protection, Oblivious RAM, Shuffle Buffer

1 Introduction

The problem of preventing leakage of information arising from both running software on untrusted systems, as well as storing data in remote untrusted servers has attracted much attention. While encryption can be used to protect data confidentiality, the problem of protecting access pattern with manageable and practical overhead is harder to address.

In the software execution environment, one can identify two main motivations for protecting memory access pattern privacy: the traditional application is to protect Intellectual Property, and prevent software piracy. More recently, it has been shown how access pattern leakage can be used to attack certain implementations of cryptographic algorithms (in the so-called *cache attacks* [15]). In the remote storage environment, one would like to protect access pattern from

a curious but not malicious server, which may benefit from gaining information about the client's pattern of access of stored data.

The traditional solution for memory access pattern protection is known as *Oblivious RAM*. It was first proposed by Goldreich [5], and later extended by Goldreich and Ostrovsky [6]. The main construction is based on the *hierarchical solution*, in which the data structure is organised in levels consisting of hash-tables (using secret hash functions known by the client only), and requires periodic expensive *oblivious* re-shuffling of data.

In the past few years, many improvements have been proposed, for example [1, 16, 4, 9, 17, 12]. Several schemes consider improvements for the application to cloud computing [3, 19, 7, 8, 10, 11, 13, 20]. Improvements typically arise from the use of different data structures and hash function schemes, more efficient sorting algorithms (for the oblivious shuffling step), and the use of secure local (client) memory. Besides the hierarchical solution, Goldreich and Ostrovsky [6] also proposed the *square root* construction, which uses secret random permutations when storing data. Boneh *et al.* [2] have recently presented a hybrid algorithm between the square-root and hierarchical algorithms, and proposed a new notion of *oblivious storage*.

Despite much recent progress, where both the asymptotic efficiency as well as the constant terms of oblivious RAM solutions have been improved (making it particularly attractive for remote storage access pattern protection), current solutions remain inefficient for *software execution protection*, i.e. to prevent leakage of relatively limited-in-size memory access pattern. In these cases, the constant terms involved in the computational complexity make the overhead unacceptably high. Yet an efficient and secure mechanism for access pattern protection is a particularly desirable feature in this environment. For instance, an emerging issue is the rapid increase of malicious software targeting smartphones. Most existing protection schemes, originally designed for PCs, are not suitable for smartphones due to several limitations, such as the computational power and available storage size. In these environments, solutions range from use of obfuscation to hardware-based access pattern protection mechanisms. For instance, Zhuang *et al.* [21] proposed a practical, hardware-assisted scheme for embedded processors, with low computational overhead. Their *control flow obfuscation* scheme for embedded processors employs a small secure hardware obfuscator to hide program recurrence. Their proposal however trades security for low overhead (and the cost of the trusted hardware buffer), and in some situations an adversary with access to the device can retrieve information about memory access.

In this paper, we propose a practice-oriented scheme for protecting RAM access pattern. We first consider an instance which, similar to the proposal by Zhuang *et al.*, also relies on the use of a secure (trusted) hardware buffer. However it achieves higher security by adapting ideas from Goldreich and Ostrovsky's square root solution, yet without the expensive (re-)shuffling of buffers. By applying this scheme, we can construct a secure platform that is suitable for executing software that deals with user private information. A potential application is to secure program execution in smartphones: these devices typically

contain sensitive user information and are increasingly under severe threat from malware. Most smartphones have a SIM card, which is generally considered as a secure area, and deployment and security of our scheme could rely on the SIM card. Another instance requires no special hardware, but as a result leads to a higher, yet practical overhead. This scheme can offer the same level of security without any special hardware. Many applications have their security relying on IC cards or other trusted hardware. One of the roles of trusted hardware is offering a secure computation, which can be realised with our proposal. Another role is a secure storage for a secret information, which is yet to be realised.

The main feature of our proposal is to maintain the *history* of memory access, which together with the access of dummy data, helps one to hide data access pattern. The security of the schemes depends on the size of the buffer (as cache or in RAM) and how the history is used. We claim that under reasonable assumptions and by selecting appropriate parameters, the schemes achieve both security and performance levels acceptable in practice. We note that although the proposal is particularly focused on the software execution protection environment (i.e. to prevent RAM access pattern leakage), its security may well be appropriate for most uses in the remote storage environment, to prevent access pattern leakage of cloud storage with much lower performance overhead than existing solutions.

2 Access Pattern Protection Problem

We can model the problem of access pattern protection as follows. We consider a *client*, with potentially small secure memory, and a *server* providing large insecure storage. This storage consists of several data blocks (for simplicity, all of the same size). In the software execution environment, we can think of the CPU as the client, and RAM as the server, where a malicious entity (e.g. malware) has access to RAM and the memory bus used in the communication between the CPU and memory.

The client accesses data by making requests to the server to either retrieve the contents of a particular data block at location i or by writing x into a data block at location j. We denote these operations by `read`(i) and `write`(j, x), respectively. We consider the security goals as: to protect the confidentiality[3] of data, as well as hide the client's access pattern to the stored data blocks from a computationally bounded adversary, which can access the data storage and the communication channel between the client and the server.

We can address the first goal by using a semantically secure probabilistic encryption scheme, such that two encryptions of the same data block will look indistinguishable to a computationally bounded adversary. The use of encryption adds a constant overhead to the system. For the remaining of this work, we will

[3] In environments where there is a requirement, we will also want to protect data integrity; this can be done by adding a MAC or by using an authenticated encryption algorithm.

assume that the data is stored encrypted; it is decrypted whenever it is read from memory, and re-encrypted whenever it is written into memory.

For the second goal, we would like that a particular sequence of operations in the stored data does not substantially leak any information, other than how many data blocks were accessed. To formalise it, we use the definition in [16] for a *secure oblivious RAM* system.

Definition 1 ([16]). *The input y of the client is a sequence of data blocks, denoted by $((v_1, x_1), \ldots, (v_n, x_n))$ and a corresponding sequence of operations, denoted by $(op_1, \ldots, op_m)$, where each operation is either a read operation, denoted* `read`*(v), which retrieves the data of the block indexed by v, or a write operation, denoted* `write`*(v, x), which sets the value of block v to be equal to x.*

The access pattern $A(y)$ is the sequence of accesses to the remote storage system. It contains both the indices accessed in the system and the data blocks read or written. An oblivious RAM system is considered secure if for any two inputs y and y' of the client, of equal length, the access patterns $A(y)$ and $A(y')$ are computationally indistinguishable for anyone but the client.

The typical, straightforward method for preventing an adversary from distinguishing between the `read` and `write` operations is to always perform both operations in every access. As a result of using this method, an access pattern $A(y)$ (for the purpose of indistinguishability) can be thought as simply as a sequence of indices i (corresponding to the data blocks accessed). The trivial solution to the access pattern protection problem consists of accessing all data blocks on the memory for each query. Another trivial solution is to use a secure client hardware. Both schemes are however too costly in practice.

In addition, data blocks are typically organised in memory based on a secret permutation or hash function in an oblivious way. This is the most expensive component of the schemes, and is the main responsible for the (amortised) computational overhead. Our proposal does not employ oblivious re-shuffling of memory; while this will affect the security provided by the scheme, we claim that under reasonable assumptions, the proposals achieve both security and performance levels acceptable in practice.

3 Related Work

We briefly introduce below some of the main research results related to our work.

3.1 Oblivious RAM

Oblivious RAM is the traditional solution for memory access pattern protection, having been first proposed by Goldreich [5], and later extended by Goldreich and Ostrovsky [6]. The main construction is based on the *hierarchical solution*, in which the data structure is organised in levels consisting of hash-tables (using secret hash functions known by the client only), and requires frequent *oblivious* re-shuffling of data. Data is scanned by visiting each level, after which the item

found is written in the top level. Levels eventually overflow with data, leading to their move downwards. This process requires the re-shuffling of data, which is done *obliviously*. This is the most complex component of the construction, and is the main factor in its (amortised) complexity overhead. The original scheme requires $O(n \log n)$ in storage, and has computation overhead of $O((\log n)^3)$ per query (using a particularly impractical sorting algorithm, or $O((\log n)^4)$ with more reasonable constant in practice).

Since the original proposal of hierarchical solution, several improved schemes have been proposed. Pinkas and Reinman [16] improved the performance of oblivious RAM using Cuckoo hashing and a new oblivious sorting algorithm. However, Kushilevitz *et al.* [12] pointed out a security flaw of the scheme presented in [16]. They also proposed a new scheme with $O((\log n)^3)$ worst-case overhead. Goodrich and Mitzenmacher [8] achieved $O((\log n)^2)$ amortised cost with $O(1)$ client-side storage. Their scheme could achieve higher performance of $O(\log n)$ by using $O(n^\alpha)$ client storage where $0 \leq \alpha \leq 1$. Stefanov *et al.* [18] proposed a scheme with an amortised overhead of (20–35)$\times$ by partitioning a bigger oblivious RAM into smaller oblivious RAMs. An efficient worst-case overhead scheme was proposed by Goodrich *et al.* [9]. This scheme also requires $O(n^\tau)$ client storage, where τ is a small constant, and achieves $O(\log n)$ access overhead and $O(n)$ storage overhead.

Goldreich [5, 6] also proposed the *square root* construction, which uses secret random permutations when storing data. The solution requires $O(n + \sqrt{n})$ in storage, and has computation overhead of $O(\sqrt{n} \log n)$ per query. See appendix of the extended version of [16] for an overview, or refer to the original paper for a detailed description. We note that our proposal borrows ideas from the square-root solution.

3.2 Hardware-assisted Control Flow Obfuscation

Oblivious RAM constructions remain too expensive to be implemented on embedded processors. In [21], Zhuang *et al.* proposed a practical, hardware-assisted scheme for embedded processors, with low computational overhead. Their *control flow obfuscation* scheme for embedded processors employs a small secure hardware obfuscator (called *shuffle buffer*) to hide program recurrence. The shuffle buffer is within the CPU trusted boundary, and an adversary is not able to observe access pattern in the shuffle buffer. The first m blocks in memory are moved to the shuffle buffer, which can hold m data blocks. When making a request for a data block, if the block is found in the shuffle buffer, access the block. Otherwise fetch the block from memory and a random block in the shuffle buffer is written back to memory. For more details, refer to [21].

This scheme however leaks information about access of data blocks. The scheme accesses memory only when it is necessary (i.e. when the block is not in the shuffle buffer), hence, one knows the exact block being accessed. Furthermore, an access in memory to a data block which was previously swapped out from the buffer indicates with high probability the existence of repeated access to a particular data.

Despite the limitations of the proposal, it adds a very low overhead to the program execution (besides the read/write and encryption/decryption overhead, only an extra read/write operation due to cache misses). We will surmount this limitation in our proposal.

4 Our Scheme for Memory Access Pattern Protection

While oblivious RAM constructions can completely hide memory access pattern, they are too expensive to be implemented on resource-constrained devices. In this section, we describe our proposal for a practice-oriented memory access pattern protection scheme. One of its main features is the inclusion of an extra buffer used to maintain the *history* of memory access, to help hiding the access pattern. We present below two instances: the first one uses a secure (trusted) hardware buffer, as in [21]; however it achieves higher security by adapting ideas from oblivious RAM mechanisms, yet without the expensive (re-)shuffling of buffers. The second instance requires no special hardware, but as a result leads to a higher, yet practical overhead.

4.1 Assumptions

We consider the problem of hiding the access pattern of memory with n data blocks. Our scheme will assume that the *client* has access to an efficient pseudo-random number generator (to make random choices of addresses), and a semantically secure probabilistic encryption scheme. Data is always stored encrypted: it is decrypted whenever it is read from memory, and re-encrypted whenever it is written into memory. Furthermore, either a `read` or `write` operation will always perform the two operations in every access, the difference is the value being written in the `write` step. As a result of using this method, an access pattern (for the purpose of indistinguishability) can be thought as simply as a sequence of indices i (corresponding to the data blocks accessed).

Our scheme will also require a way to generate a pseudo-random permutation, to map memory addresses. This can be achieved by using a deterministic encryption algorithm $\mathcal{E}$. This mapping will be described either by the function $\mathcal{E}$, with the output computed in each call, or explicitly described as a table look-up (with input-output pairs). We note that the latter requires $O(n)$ memory within the client's trusted boundary (as in [21]), which in some scenarios may be impractical.

Perhaps the main challenge of access pattern protection schemes is to hide the repeated access of data blocks. In general, when not deploying expensive ORAM solutions, access pattern can typically be distinguishable by observing a long series of accesses. Therefore, we define a relaxed, still practical, security definition of access pattern protection, which we call *δ-length security*. Assume that during a certain sequence of data access, a particular block 'a' is accessed twice by a program at t-th access and $(t+\delta)$-th access; we call this a *δ-distance access* of 'a'. Informally, an access pattern protection scheme is δ-length secure if

the probability that an adversary identifies repeated accesses in $A(y)$ at distance at most δ is small.

Definition 2. *We say that an access pattern protection scheme is δ-length ϵ-secure if the probability that an adversary identifies any d-distance access in $A(y)$ is at most ϵ for every $d \leq \delta$.*

4.2 Set-up

On loading n data blocks to memory, our scheme will use $\mathcal{E}$ to permute the corresponding addresses. Dummy data will be typically added to the original data, so that at initialisation we have kn data blocks being loaded to memory, with $k \geq 1$ a small constant. The constant k will be selected based on the typical *epoch* length of the program and the availability of memory within the client's trusted boundary to describe the random permutation $\mathcal{E}$. We discuss the use of dummy data in more detail in Section 5.3.

Our scheme will partition memory into two regions: a (secure) buffer $\mathcal{M}$ and an unsecured memory $\mathcal{L}$, of sizes m and ℓ, respectively, with $m \ll \ell$. It follows that $m+\ell = kn$. Furthermore, we will require a secure table $\mathcal{H}$, called the *history* table. The table $\mathcal{H}$ stores addresses of data blocks that have been moved from the secure buffer $\mathcal{M}$ to the unsecured memory $\mathcal{L}$, and has size ℓ_h. We denote the address of a data block 'a' as i_a. Typically, we may have $\mathcal{H}$ implemented as part of $\mathcal{M}$. In the case where we are able to store the permutation mapping table explicitly, $\mathcal{H}$ can be implemented by adding a *set* bit into the table. Despite these choices, in our discussions below we consider $\mathcal{H}$ as a separate table.

4.3 Instance 1: Construction with Secure Memory

The first instance of our scheme considers the buffer $\mathcal{M}$ being implemented within the client's trusted boundary (as in [21]). The table $\mathcal{H}$ is also stored within this boundary. After loading data into memory, m data blocks are copied into $\mathcal{M}$; the table $\mathcal{H}$ starts empty[4]. On the first access to a data block 'a' (either a `read`(a) or `write`(a, x)), we search for 'a' in $\mathcal{M}$ and access $\mathcal{L}$ twice: if 'a' is in $\mathcal{M}$, we replace two random elements (not 'a') from $\mathcal{M}$ with two random *dummy* elements from $\mathcal{L}$ and we access 'a'; if 'a' is not in $\mathcal{M}$, two random elements from $\mathcal{M}$ are replaced by 'a' and one random *dummy* element from $\mathcal{L}$ (and 'a' is accessed). In both cases, the corresponding addresses of blocks being kicked out from $\mathcal{M}$ are written in $\mathcal{H}$. In subsequent calls, we proceed as follows:

1. if 'a' is in $\mathcal{M}$, we replace two random elements (not 'a') from $\mathcal{M}$ by a random element from $\mathcal{L}$ and a random element from $\mathcal{L}$ which had already been accessed before (as recorded in the *history* buffer), and we access 'a'.

[4] although, before the program starts to run, the scheme can operate by accessing a number of dummy data blocks to populate the history buffer.

Algorithm 1 Pseudocode of access pattern protection scheme

```
scan M for 'a'
if 'a' ∈ M then
    replace two random elements (not 'a') in M with two random blocks in L, one
    of them is chosen from the history H and the other is randomly chosen from L
else
    scan H for 'i_a'
    if 'i_a' ∈ H then
        replace two random blocks in M with a random block in L and 'a'
    else
        replace two random blocks in M with 'a' and a random block whose address
        is registered in H
    end if
end if
choose ℓ_h elements from ℓ_h + 2 to update history table H
access 'a'
```

2. if 'a' is not in $\mathcal{M}$, and its address is in the history table, we replace two random elements from $\mathcal{M}$ by a random element from $\mathcal{L}$ and 'a' (as recorded in the *history* buffer). Note that $\mathcal{H}$ holds only addresses and data itself is stored in $\mathcal{L}$.
3. if 'a' is not in $\mathcal{M}$, and its address is not in the history table either, we replace two random elements from $\mathcal{M}$ by 'a' and a random element from $\mathcal{L}$ which had already been accessed before (as recorded in the *history* buffer).

Every time the data blocks are kicked out from $\mathcal{M}$ to $\mathcal{L}$, data blocks are written in $\mathcal{L}$ taking their original position (as described by $\mathcal{E}$), and addresses of those blocks are registered in the history table $\mathcal{H}$. As the program continues to access data blocks, the table may eventually get full. When this is the case, at each access we select at random ℓ_h elements among the $\ell_h + 2$ elements (the current history elements and the two new ones). We have a pseudocode of our scheme in Algorithm 1 and show an example in the appendix.

The Objective of the History Buffer. The goal of our scheme is to efficiently protect the privacy of data access pattern. It is clear that if during a run of the program, data blocks are only accessed once, then the use of a random permutation alone will suffice to hide the access pattern. The case of relevance is thus when a data block is accessed more than once. When it is accessed for the first time, it is copied into $\mathcal{M}$. If accessed again, and it is still in $\mathcal{M}$, then access is oblivious from an adversary; still we would like to hide the fact that the data accessed was found in $\mathcal{M}$. If it is no longer in $\mathcal{M}$, then we would like to hide the fact we are accessing it again from an adversary. Thus, if we consider the case that a program keeps reading 'a' at different intervals, we have the following three types of access to consider:

1. 'a'$\in \mathcal{M}$;

2. 'a'$\notin \mathcal{M}$ and 'i_a' $\in \mathcal{H}$;
3. 'a'$\notin \mathcal{M}$ and 'i_a' $\notin \mathcal{H}$.

Note that case 3 is likely to occur when 'a' is accessed for the first time. As discussed, the cases that require most attention are 2 and 3 when 'a' is accessed again: since we do not perform an oblivious re-shuffling, the adversary would notice that 'a' is being accessed again.

To address this, in our scheme, we first search 'a' in $\mathcal{M}$ and then, depending on the cases described above, the access of memory is done as follows:

1'. access (r, p),
2'. access (r, a),
3'. access (a, p),

where r is a random location in $\mathcal{L}$ and p is a location recorded in the table $\mathcal{H}$.

Note that we need to keep the contents in the history table secret: although the adversary can record all blocks previously accessed, in practice we may not be able to keep the addresses of all accessed blocks in the history table (since ℓ_h may be small). If we had the addresses in cleartext, one may note that in case 3' above, although 'a' had being accessed before, its address was no longer in the history buffer (for lack of space).

4.4 Instance 2: Construction without Secure Memory

We consider a second instance of our proposal, which does not require a secure buffer. The buffer $\mathcal{M}$ and history table $\mathcal{H}$ are kept in the unsecured memory area, as with $\mathcal{L}$. In this case, access to $\mathcal{M}$ (and $\mathcal{H}$) is made by reading and writing every data block in the buffers (which requires decryption and encryption of data). Thus, to find out whether 'a' is in $\mathcal{M}$, we read/write all values; when replacing data blocks in $\mathcal{M}$, we again read/write all values. Except for this, the access is made as described in Section 4.3. The security provided is the same, but the computation overhead is obviously increased, and it is dependent of the sizes of buffers $\mathcal{M}$ and $\mathcal{H}$.

5 Security Analysis

We discuss the security of our scheme.

5.1 Access Pattern Hiding

Regarding recurrence, recall that in our scheme, we first search 'a' in $\mathcal{M}$ and then, depending on the cases described in Section 4, the access of memory is as follows:

1. 'a'$\in \mathcal{M}$, and 'a' is accessed: access (r, p) in $\mathcal{L}$;
2. 'a'$\notin \mathcal{M}$, 'a'$\in \mathcal{H}$, and 'a' is accessed: access (r, a) in $\mathcal{L}$;

3. 'a'$\notin \mathcal{M}$, 'a'$\notin \mathcal{H}$, and 'a' is accessed: access (a, p) in $\mathcal{L}$.

Assume that the program accesses 'a' at time t; we denote it by $X_t = a$. We have the following lemma.

Lemma 1. *Assume that $X_t = a$, and let m and ℓ_h denote the sizes of the $\mathcal{M}$ and $\mathcal{H}$, respectively. Then after δ steps we have $p_M = \Pr[a \in \mathcal{M}] \geq \left(\frac{m-2}{m}\right)^\delta$ and $p_H = \Pr[a \in \mathcal{H}] \geq \left(\frac{\ell_h}{\ell_h+2}\right)^\delta$.*

Proof. To compute $\Pr[a \in \mathcal{M}]$, note that the right-hand side of the expression corresponds to the probability that an element remains in a set of size m after δ replacements of 2 elements at time, which is how the scheme manages the buffer $\mathcal{M}$. The inequality comes from the fact that even if removed after $d < \delta$ steps, 'a' may be re-inserted during the normal operation of the scheme. Showing the second probability is similar (noting however that in the history buffer, the scheme draws 2 elements among $\ell_h + 2$ elements). □

For the sake of simplicity, we will in the remaining of this paper assume equality in the two expressions above. Furthermore, we will also assume that the two events are independent (obviously the probability of 'a' being in $\mathcal{H}$ after δ steps will be influenced by whether/when 'a' leaves the buffer $\mathcal{M}$; however we believe this assumption is reasonable for typical values of m, ℓ_h and δ – and substantially simplifies our computations). The simple lemma below then follows.

Lemma 2. *Consider the three cases for memory access discussed above, and assume that $X_t = X_{t+\delta} = a$. Then the probability that we have case 1 is p_M, the probability that we have case 2 is $p_H(1 - p_M)$, and the probability that we have case 3 is $(1 - p_H)(1 - p_M)$.*

Now assume that an adversary observes the scheme at time $t + \delta$ in case 1, i.e. we have $a \in \mathcal{M}$ and access to (r, p) from $\mathcal{L}$. Then following a conservative estimate, we have $\Pr[X_{t+\delta} = a \mid \text{ case 1 }] \leq \frac{1}{m-2}$. For case 2, we have a similar upper bound: $\Pr[X_{t+\delta} = a \mid \text{ case 2 }] \leq \frac{1}{m-2}$. Case 3 is perhaps the one in which an adversary can extract more information (since it is very likely that, unlike the other two cases, the pair of elements (a, p) drawn from $\mathcal{L}$ have already been observed by the adversary). We will again adopt a conservative approach, and have the upper-bound $\Pr[X_{t+\delta} = a \mid \text{ case 3 }] \leq 1/2$.

Theorem 1. *The proposed scheme is δ-length ϵ-secure access pattern protection scheme, where*

$$\epsilon \leq \frac{p_M}{(m-2)^2} + \frac{(1-p_M)p_H}{(m-2)^2} + \frac{(1-p_M)(1-p_H)}{2(m-2)}.$$

Proof. Let $\mathcal{A}$ be adversary who is able to observe an access sequence $X_i = a_i$, for $i = 1, \ldots, N$. Let us assume that $X_t = X_{t+\delta} = a$. We wish to compute the probability $\Pr[X_t = a, X_{t+\delta} = a]$. We assume that the first access to a is at time t (the proof and figures can be slightly modified when this is not the case), and

that the accesses at time t and $t+\delta$ are independent events (i.e. we assume no knowledge of statistics of the original program being protected). Then we have $\Pr[X_t = a] \leq 1/(m-2)$, and by lemmas and discussion above.

$$\begin{aligned}\Pr[X_t = a, X_{t+\delta} = a] &= \Pr[X_t = a] \cdot \Pr[X_{t+\delta} = a] \\ &\leq \frac{1}{m-2}(\Pr[X_{t+\delta} = a \mid \text{ case 1 }] \cdot \Pr[\text{ case 1 }] \\ &\quad + \Pr[X_{t+\delta} = a \mid \text{ case 2 }] \cdot \Pr[\text{ case 2 }] \\ &\quad + \Pr[X_{t+\delta} = a \mid \text{ case 3 }] \cdot \Pr[\text{ case 3 }]) \\ &\leq \frac{p_M}{(m-2)^2} + \frac{(1-p_M)p_H}{(m-2)^2} + \frac{(1-p_M)(1-p_H)}{2(m-2)}.\end{aligned}$$

□

5.2 Parameters: Size of Secure Memory and History Table

The choice for sizes of the secure memory $\mathcal{M}$ and history table $\mathcal{H}$ have an obvious influence in the security and efficiency/cost of the scheme: it follows from Theorem 1 that large values for m and ℓ_h significantly decreases the chances that an adversary can identify a repeated access to a particular memory block. However large $\mathcal{M}$ and $\mathcal{H}$ negatively affect the performance of the scheme (as well as increase its costs in the hardware-assisted version).

For instance, if we take the size of the shuffle buffer as 128 16-byte blocks (as in [21]), and the same size for $\mathcal{H}$ (that is, 512 32-bit addresses), then for $\delta = 20$, we have $p_M \approx 0.73$ and $p_H \approx 0.92$, and as a result $\epsilon \leq 1.4 \times 10^{-4}$. The value increases to 4.4×10^{-4} if $\delta = 50$, and to 1.06×10^{-3} if $\delta = 100$ In general, SIM cards have a capacity of 64KB, which means we can allocate much more space, say 1024 blocks and 4×1024 history addresses. In this case, we have $p_M \approx 0.82$ and $p_H \approx 0.95$, and as a result $\epsilon \leq 0.5 \times 10^{-5}$ for $\delta = 100$.

As discussed before, we note however that our scheme does not achieve strong indistinguishability: As we use a static permutation $\mathcal{E}$, addresses are fixed (after the permutation), and this implies some leakage of information (an adversary will, for instance, know when the sets of potentially accessed blocks are disjoint, implying no recurrence). Overall, we believe that, a choice of parameters can be made to achieve both security and performance levels acceptable in practice.

5.3 How much dummy data should we use?

Oblivious RAMs provide security by making sure that a data block is only accessed once while it remains at the same address. When it is accessed again, it will be allocated to a completely different address and the adversary will have no information about the contents and whether/when it was accessed before. Our scheme, on the other hand, tries to ensure the security in the circumstance that the data block is accessed multiple times while in the same address.

If there are several dummy data blocks, that can be used as random elements that the protection scheme can access. They will also prevent the adversary from

determining which blocks are actually accessed by the program. We can also make dummy accesses to the actual data blocks when it is not accessed by the program. When the program accesses a data block, the access pattern protection scheme will access 2 blocks as the scheme always swaps 2 blocks between $\mathcal{M}$ and $\mathcal{L}$. If we add n dummy data blocks, the number of accesses to the actual data and dummy data will be roughly the same. Guessing the access pattern correctly becomes harder as n increases, and less dummy blocks will be required. While we still need to confirm it experimentally, we expect that the required number of dummy blocks will be at most $2n$ and the constant k will be $1 \leq k \leq 3$.

6 Comparison

Our scheme requires secure memory for storing m data blocks. It also requires storage for the history table of size ℓ_h. When we read a data block, we first access the secure memory to check whether the data block sought is in $\mathcal{M}$. We then move two data blocks into $\mathcal{M}$ and read the data block. We also need to access and update the history table. Thus the total cost is 4 operations, plus the cost associated with reading and writing the history table (which can be only two extra accesses if $\mathcal{H}$ is within the secured boundary). Thus we have the overhead of 7 operations. We discuss how to improve the overhead later this section. The history can be potentially stored (encrypted) in the memory area $\mathcal{L}$. When $\mathcal{H}$ is not within the secured boundary, we have to read and update all blocks in the table. Thus we have the overhead of $4 + 2\ell_h$. Finally, if using *dummy* data, we also have data storage overhead in $\mathcal{L}$ depending on the constant k.

Our scheme can also be implemented without any special hardware. The difference in this case is that we need to scan the entire buffer $\mathcal{M}$ twice, plus the access to the history table and two swaps. We have therefore the cost of $2m + 2\ell_h + 2$ operations. As we assume m is much smaller than n, our scheme without a hardware is still more efficient than most of oblivious RAMs. According to [18], practical overhead of oblivious RAMs are, in general, on the range of thousands to hundred of thousands. When we choose $m = \ell_h = 128$, the overhead is 514, which is less than half of that of oblivious RAMs. The scheme proposed of [18] achieved the overhead of $(20 - 35)\times$, which is faster than our second scheme, still our scheme has the advantage in terms of storage size, that is, ours requires less than $3n$ storage for n original data while the scheme in [18] requires $4n + o(n)$.

Overall, the parameters m, ℓ_h and k can be set to suitable values, to offer the appropriate trade-off between security and implementation/computational costs. While oblivious RAMs are too expensive especially for resource constrained devices and Zhuang *et al.*'s scheme is not suitable scheme for this purpose, our scheme can offer reasonable security for the access pattern protection problem with constant computational overhead and practical storage.

Improvement of Performance. When we update the history table, addresses of two data blocks are registered into the history table independently in the

Table 1. Comparison with Other Schemes

	Pattern Leakage	Computational Overhead	Storage
Square root [6]	No	$O(\sqrt{n}\log n)$	$O(n+\sqrt{n})$
Pinkas-Reinman [16]	No	$O((\log n)^2)$	$8n$
Stefanov *et al.* [18]	No	$O(\log n)$	$4n + o(n)$
Goodrich-Mitzenmacher [8]	No	$O(\log n)$	$8n$
Lu-Ostrovsky [14]	No	$O(\log n)$	$O(n)$
Kushilevitz *et al.* [12]	No	$O((\log n)^2/\log\log n)$	$O(n)$
Zhuang *et al.* [21]	Yes	2	$2n$
Ours w/ Sec. Mem.	see Theorem 1	6	$\leq 3n$
Ours w/o Sec. Mem.	see Theorem 1	$2(m + \ell_h + 1)$	$\leq 3n$

original scheme. Assume that two addresses 'i_a' and 'i_b' need to be registered, then two random locations are selected and those locations are updated with the new blocks. Therefore, the update requires two operations in each cycle. The update can be done with only one operation by modifying the entry as a bigger one $i_a||i_b$, where $||$ is concatenation. When choosing one data block from the history table, one can first choose the concatenated block and then can choose higher half or lower half of the block. As a result, we can reduce the cost for updating the history table from 2 to 1 and the overall overhead is improved to 6. Table 1 gives the comparison of the performance with other schemes.

7 Conclusion

In this paper, we proposed two new schemes for protecting memory access patterns. The distinctive character of our scheme is that we do not re-shuffle the order of the data blocks in memory. To protect the access pattern without re-shuffling, we used a *history* of the accesses. We first considered an instance which is similar to the proposal in [21], and also relies on the use of a secure (trusted) hardware buffer; however it achieves higher security by adapting ideas from Oblivious RAM mechanisms, without the expensive (re-)shuffling of buffers. Another instance requires no special hardware, but as a result leads to a higher, yet practical overhead. We defined a new security notion called δ-length ϵ-security and proved that the proposed two schemes are secure in this notion. We also discussed the size of parameters, which are the size of secure memory, history table and dummy data and compared the performance with existing schemes. We claim that under reasonable assumptions, the schemes can achieve both security and performance levels acceptable in practice.

References

1. M. Ajtai. Oblivious RAMs without cryptographic assumptions. In L. J. Schulman, editor, *STOC*, pages 181–190. ACM, 2010.

2. D. Boneh, D. Mazieres, and R. A. Popa. Remote Oblivious Storage: Making Oblivious RAM Practical. Technical Report MIT-CSAIL-TR-2011-018, Massachusetts Institute of Technology, 2011.
3. B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private Information Retrieval. In *FOCS*, pages 41–50. IEEE Computer Society, 1995.
4. I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly Secure Oblivious RAM without Random Oracles. In Y. Ishai, editor, *TCC*, volume 6597 of *LNCS*, pages 144–163. Springer, 2011.
5. O. Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In A. V. Aho, editor, *STOC*, pages 182–194. ACM, 1987.
6. O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.
7. M. T. Goodrich. Data-Oblivious External-Memory Algorithms for the Compaction, Selection, and Sorting of Outsourced Data. In R. Rajaraman and F. Meyer auf der Heide, editors, *SPAA*, pages 379–388. ACM, 2011.
8. M. T. Goodrich and M. Mitzenmacher. Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In L. Aceto, M. Henzinger, and J. Sgall, editors, *ICALP (2)*, volume 6756 of *LNCS*, pages 576–587. Springer, 2011.
9. M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with Efficient Worst-Case Access Overhead. In C. Cachin and T. Ristenpart, editors, *CCSW*, pages 95–100. ACM, 2011.
10. M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical Oblivious Storage. In E. Bertino and R. S. Sandhu, editors, *CODASPY*, pages 13–24. ACM, 2012.
11. M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-Preserving Group Data Access via Stateless Oblivious RAM Simulation. In Y. Rabani, editor, *SODA*, pages 157–167. SIAM, 2012.
12. E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (In)security of Hash-based Oblivious RAM and a New Balancing Scheme. In D. Randall, editor, *SODA*, pages 143–156. SIAM, 2012.
13. J. R. Lorch, J. W. Mickens, B. Parno, M. Raykova, and J. Schiffman. Toward Practical Private Access to Data Centers via Parallel ORAM. *IACR Cryptology ePrint Archive*, 2012:133, 2012.
14. S. Lu and R. Ostrovsky. Distributed Oblivious RAM for Secure Two-Party Computation. *IACR Cryptology ePrint Archive*, 2011:384, 2011.
15. D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In D. Pointcheval, editor, *CT-RSA*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.
16. B. Pinkas and T. Reinman. Oblivious RAM Revisited. In T. Rabin, editor, *CRYPTO*, volume 6223 of *LNCS*, pages 502–519. Springer, 2010.
17. E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with O((logN)3) Worst-Case Cost. In D. H. Lee and X. Wang, editors, *ASIACRYPT*, volume 7073 of *LNCS*, pages 197–214. Springer, 2011.
18. E. Stefanov, E. Shi, and D. Song. Towards Practical Oblivious RAM. *CoRR*, abs/1106.3652, 2011.
19. P. Williams and R. Sion. Usable PIR. In *NDSS*. The Internet Society, 2008.
20. P. Williams and R. Sion. SR-ORAM: Single Round-trip Oblivious RAM. In *ACNS*, industrial track, pages 19–33, 2012.
21. X. Zhuang, T. Zhang, H.-H. S. Lee, and S. Pande. Hardware Assisted Control Flow Obfuscation for Embedded Processors. In M. J. Irwin, W. Zhao, L. Lavagno, and S. A. Mahlke, editors, *CASES*, pages 292–302. ACM, 2004.

Appendix

We present an example of the scheme in Figure 1. In this example, we have $m = 4$ and $\ell_h = 6$, then the program reads data in the order $5 \to D \to 8 \to 5$. For simplicity, we will not be using dummy data (i.e. $k = 1$) and we have a fixed permutation.

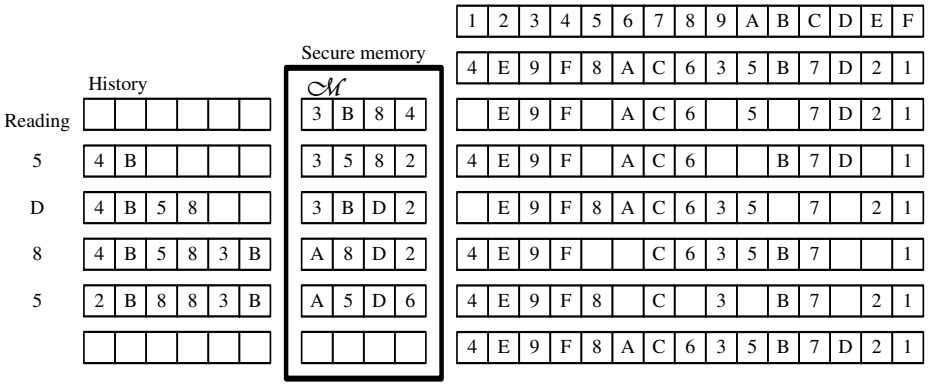

Fig. 1. Our Scheme

1. All data in the memory $\mathcal{L}$ is randomly permuted and $\mathcal{M}$ is filled with the 4 random blocks.
2. When the program tries to access 5, since it is not in $\mathcal{M}$, blocks 5 and 2 are brought into $\mathcal{M}$. Since we do not have any history yet, block 5 and a random block is chosen. Two blocks (4 and B) in $\mathcal{M}$ are written back to $\mathcal{L}$, and their addresses are entered in the history.
3. When the program accesses D, the block D is brought into $\mathcal{M}$. The block B is chosen from the history and also brought into $\mathcal{M}$. Blocks 5 and 8 are instead written back to $\mathcal{L}$, and their addresses are entered in the history.
4. When the program accesses 8, since 8 is in the history, block A is randomly chosen. Blocks 3 and B are written back to $\mathcal{L}$, and their addresses are entered in the history.
5. When the program accesses 5, since the address of 5 is in the history, the block 6 is randomly chosen to be brought into the $\mathcal{M}$. Blocks 2 and 8 are written back to $\mathcal{L}$. Now the history table is full, two addresses of random blocks (in this example, 4 and 5) are replaced with those of 2 and 8.
6. All blocks in $\mathcal{M}$ are written back into $\mathcal{L}$.