INVITED PAPER *Special Section on Frontiers of Internet of Things*

Comparison of Access Pattern Protection Schemes and Proposals for Efficient Implementation*

Yuto NAKANO[†a)], **Shinsaku KIYOMOTO**[†], **Yutaka MIYAKE**[†], *and* **Kouichi SAKURAI**[††], ***Members***

SUMMARY Oblivious RAM (ORAM) schemes, the concept introduced by Goldreich and Ostrovsky, are very useful technique for protecting users' privacy when storing data in remote untrusted servers and running software on untrusted systems. However they are usually considered impractical due to their huge overhead. In order to reduce overhead, many improvements have been presented. Thanks to these improvements, ORAM schemes can be considered practical on cloud environment where users can expect huge storage and high computational power. Especially for private information retrieval (PIR), some literatures demonstrated they are usable. Also dedicated PIRs have been proposed and shown that they are usable in practice. Yet, they are still impractical for protecting software running on untrusted systems. We first survey recent researches on ORAM and PIR. Then, we present a practical software-based memory protection scheme applicable to several environments. The main feature of our scheme is that it records the history of accesses and uses the history to hide the access pattern. We also address implementing issues of ORAM and propose practical solutions for these issues.

key words: *access pattern protection, oblivious RAM, private information retrieval*

1. Introduction

The emergence of Internet of Things (IoT) and cloud computing bring new problems in accessing to remote untrusted servers and running software on untrusted systems. The leakage of information and its protection arising from both running software on untrusted systems, as well as storing data in remote untrusted servers have attracted much attention. Encryption can ensure the confidentiality, however, sometimes confidentiality is not enough as the pattern of accesses to servers can leak important information. In the data storing environment, one can identify the main motivation for protecting access pattern to the storage: the access pattern may leak which data is important for the user, as the important data will typically be accessed more often. The traditional solution for memory access pattern protection is known as Oblivious RAM (ORAM) [3], [4]. The similar solution is called private information retrieval (PIR), which enables the client to hide his/her access pattern from the *honest-but-curious* servers [5]. Both schemes required huge overheads and were considered unpractical. After the first proposals, a lot of improvements have been proposed.

In this paper, we first introduce a series of related works on ORAM and PIR, and compare their overhead. The target of both schemes is the same: how to efficiently protect the pattern of accesses. ORAM supports both the read and write operations to RAM (or server) while PIR usually consider the read operation. We then introduce our lightweight scheme and consider practical implementation of the scheme. The main feature of our scheme is based on the history of accesses. We also propose three techniques for the higher performance.

2. Related Works

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

2.1 Oblivious RAM

Software piracy has been a major concern for all software developers as it is very easy to copy software and use copied software. Many software protection mechanisms have been proposed such as licence management mechanisms, obfuscation mechanisms and combination of secure hardware. Many of those mechanisms are, however, ad-hoc and not based on theoretical foundations. Goldreich [3], later extended by Goldreich and Ostrovsky [4], proposed software protection mechanism called Oblivious RAM. By using ORAM schemes, one can execute a program in a way that a polynomial-time adversary can only know how long it takes to finish the program even if the adversary can alter memory contents during execution. The application of ORAM to realise a private access to remote servers has also been considered, for example [6], [10]. Goldreich and Ostrovsky proposed two main constructions, called *hierarchical solution* and *square root solution*.

The square root solution attaches area called *shelter*, which can contain $\sqrt{N}$ blocks, to main memory and adds $\sqrt{N}$ dummy blocks. Then original N data blocks and $\sqrt{N}$ dummy blocks are permuted by the secret permutation π. When the client accesses to data block x, one first scans entire shelter to confirm if x is inside the shelter. One accesses to dummy location in memory if x is in the shelter, accesses to $\pi(x)$ to copy the block x to the shelter, otherwise. As one block is copied to the shelter at each access, the shelter gets full after $\sqrt{N}$ accesses. Once the shelter gets full, new permutation π' is chosen and all blocks are re-shuffled

Manuscript received January 15, 2014.
Manuscript revised July 9, 2014.

[†] The authors are with KDDI R&D Laboratories Inc., Fujimino-shi, 356–8502 Japan.

[††] The author is with Kyushu University, Fukuoka-shi, 819–0395 Japan.

[*] Part of this paper was presented at CARDIS 2012 [1] and ISA 2014 [2].

a) E-mail: yuto@kddilabs.jp

DOI: 10.1587/transinf.2013THP0007

according to π', which causes huge overhead. The security of this scheme is based on an one-way function. The one-way function is a function which can be easily computed in one-way, that is computing $y = f(x)$ for given x, however, computing the other way, that is $f^{-1}(y)$ for given y, is computationally infeasible.

In the hierarchical solution for RAM with N items, the data structure is organised in $n = O(\log N)$ levels consisting of hash-tables with 2^i buckets $(1 \le i \le n)$, each bucket containing $O(\log N)$ items. The storage requirement is therefore $O(N \log N)$. Data is mapped into the levels using different secret hash functions h_i, known by the client only.

An element r is read with the `read`(r) operation as follows:

1. Scan the entire first level.
2. If the element is not found, then for each level i $(2 \le i \le n)$, compute $j = h_i(r)$ and read the j-th bucket in the level i buffer, until the requested element is found.
3. Once the element is found (including during the scan of the first level), continue with the procedure in step 2. by reading a dummy location in each level i given by $h_i(\text{dummy} \circ t)$, where t is a counter.
4. The entire first level is scanned again, and the element r is written to the first level.

When we wish to update an element r with the `write`(r, x) operation, we perform the `read`(r) procedure as above, but insert the new value x into the first level at the step 4. It follows that the number of access operations for a data item request (either `read` or `write`) is $O(\log^2 N)$.

Because of the writing in step 4, buffer levels eventually overflow with data. Indeed, after 2^i requests, the buffer at level i will be full, and its full contents are moved down to level $i + 1$. Every time content is moved to a lower level, all data in both levels are permuted and a new random hash function is chosen. This process requires the re-shuffling of data, which needs to be done *obliviously*. This is the most complex component of the construction, and is the main factor in its (amortised) complexity overhead.

Based on the constructions above, the original scheme from [4] requires $O(N \log N)$ in storage, and has amortised computation overhead of $O(\log^3 N)$ per query using the AKS algorithm for the oblivious sorting [7]. Due to the very large constants, the complexity is considered $O(\log^4 N)$ in practice (by using Batcher's sorting network [8]).

In the past few years, several improvements have been proposed and many applications using ORAM schemes have been proposed, for example [6], [9]–[20]. Improvements typically arise from the use of different data structures and hash function schemes, more efficient sorting algorithms (for the oblivious shuffling), and the use of secure local (client) memory. Currently, the best amortised overhead is $O(\log^2 N/ \log\log N)$ presented in [16] whose security is based on the one-way function.

One can show that the combination of the scanning method described above, and frequent oblivious re-shuffling can provide a high level of memory access pattern privacy protection. In particular, the re-shuffling following level overflow ensures that a data item is not visited twice in the same level (for $2 \le i \le n$) using the same hash function h_i. We note however that it was shown in [16] that the choice of hash function may still leak information to adversaries.

Despite much recent progress, where the asymptotic efficiency as well as the constant terms of ORAM solutions have been improved (making it particularly attractive for remote storage access pattern protection), current solutions remain inefficient for preventing 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. This motivates the proposal of other methods, which while not achieving the same level of protection as ORAM, offer low computation (and storage) overheads, and may therefore achieve both security and performance levels acceptable in practice.

Suppose that we are accessing the data 'a' several times in the ORAM. The Read(a) process searches for 'a' in the set of $(A_t \cup P_t)$ where A_t indicates all element in the top level and P_t indicates the path which is determined by the hash function $h(a)$ and $h(dummy)$. At the next access to 'a', the read operation searches for 'a' in the set of $(A_{t+1} \cup P_{t+1})$. Here, we have to consider two cases: 1) re-shuffle has been done or 2) re-shuffle has not been done.

1. Since the re-shuffle has not been done, we are using the same hash functions. But, 'a' must be in the top level, and search dummy location for the lower levels. Hence P_t and P_{t+1} have no correlation.
2. Since the re-shuffle has been done, new hash functions are chosen. Therefore P_t and P_{t+1} have no correlation.

As shown, P_t and P_{t+1} are indistinguishable. And A_t and A_{t+1} are also indistinguishable, since the all data always is accessed in the top level. Therefore the adversary cannot tell if 'a' is being accessed more than once. Re-shuffling the elements is critical for the security of ORAM, but it is also critical for the cost of ORAM. The best amortised overhead is $O(\log^2 N/ \log\log N)$ presented in [16].

Williams et al. [21] introduced a collection of new techniques, which are supporting parallel queries and a new de-amortisation construction, to improve their performance. By applying the collection to ORAM proposed in [22], they implemented which is called PD-ORAM ("Parallel De-amortised ORAM"). Supporting parallel queries from multiple client can decrease a response time, especially when the client is accessing the database through a relatively high latency network. A new de-amortisation construction enabled the database to process queries simultaneously with re-shuffling. As the re-shuffling process is one of the heaviest task of ORAM, this new construction contributed for the improvement.

Recently the third construction called the tree construction is proposed by Shi et al. [14]. It has an N-element database in a binary tree of depth $\log N$. Each node in the tree has a bucket which can store k data items. Their scheme uses $O(N \log N)$ storage at the server. The client needs $O(1)$

Table 1 Comparison of ORAM and related schemes. r is a small constant of $r > 1$. B is a size of data block ($B = \chi \log N$). ℓ_m and ℓ_h are respectively the size of buffer and history table. k is the security parameter whose typical setting may be $k \in [50, 80]$.

	Computational Overhead	Server Storage	Client Storage	Security
GO [4]	$O(\log^3 N)$	$O(N \log N)$	$O(1)$	random oracle
PR [11]	$O(\log^2 N)$	$O(N)$	$O(1)$	random oracle
BMP [12]	$O(\sqrt{N})$	$O(N)$	$O(\sqrt{N})$	random oracle
DMN [13]	$O(\log^3 N)$	$O(N)$	$O(1)$	information theoretic
GM [24]	$O(\log N)$	$O(N)$	$O(N^{1/r})$	one-way function
GMOT [15], [25]	$O(\log N)$	$O(N)$	$O(N^{1/r})$	random oracle†
SCSL [14]	$O(\log^3 N)$	$O(N \log N)$	$O(1)$	random oracle
SO [26]	$O(\log N)$	$O(N)$	$O(1)$	one-way function
KLO [16]	$O(\frac{\log^2 N}{\log \log N})$	$O(N)$	$O(1)$	one-way function
SSS [18]	$O(\log^2 N)$	$O(N)$	$O(\sqrt{N})$	random oracle
WS [19]	$O(\log^2 N \log \log N)$	$O(N)$	$O(\log N)$	one-way function
SDSFRY [20]	$O(\log^2 N / \log \chi)$	$O(N)$	$O(\log^2 N / \log \chi) \cdot \omega(1)$	randomised encryption
GGHJRW [23]	$O(k \log^2 N / \log k)$	$O(N)$	$O(1)$	random oracle
ZZLP [27]	2	$O(N)$	$O(1)$	probabilistic
Ours 1 [1]	$3 + 2\ell_h$	$O(N)$	$O(1)$	probabilistic
Ours 2 [1]	$2(\ell_m + \ell_h + 1)$	$O(N)$	$O(1)$	probabilistic

† This can be realised without a random oracle by using [13]

memory and computation complexity is $O(\log^3 N)$. The scheme is proven secure given the access to a random oracle. Later, the tree construction is optimised by Gentry et al. [23]. The optimisations can reduce the storage overhead by an $O(k)$ factor and computation complexity by an $O(\log k)$ factor, where k is a security parameter.

Comparison of computation overhead, storage overhead and security is summarised in Table 1. As shown in Table 1, the security of SDSFRY [20] is based on the randomised encryption. It is an encryption algorithm which uses randomness on the encryption and resulting ciphertexts correspond to the same plaintext are indistinguishable from one another.

2.2 Private Information Retrieval

Emerging of cloud storage services enables users to store and access their data very easily. Moreover, users can store huge data which is too large to store locally. However, it also raises a new security challenge, which is how to protect user's privacy. When the user request data to the server, the server will notice which data the user wants. The server might be curious about the user's request and try to extract user's private information. Encryption can provide confidentiality of data, however, sometimes it is not sufficient. For instance, if the particular file is accessed very often, it implies that file is more important for the user than the ones accessed less often. An adversary may try to delete those files.

Private Information Retrieval is a technology that allows clients to query a database in a way that even the database cannot learn anything about the clients' queries. A trivial solution is to download everything every query. This solution, however, is impractical due to a high communication overhead and a high storage overhead at the client side.

There are two PIR schemes: computational PIR (CPIR) [28], [29] and information theoretic PIR (IT-PIR) [30]. In the CPIR, the client queries the database an encrypted query and the server returns the encrypted result to the client in order to prevent the computationally limited server from learning anything about the query. For example, Chor and Gilboa's scheme [28] and Kushilevitz and Ostrovsky's scheme [29] require $O(N)$ server storage for storing data of size N, the same as evaluated in ORAM schemes, and communication overhead is $O(N^{\epsilon})$. We evaluate the efficiency of PIR schemes in terms of the communication cost, not the computational cost, as the efficiency is generally measured by the communication cost. The security of Chor and Gilboa's scheme is based on an one-way function and Kushilevitz and Ostrovsky's one is based on the hardness of deciding quadratic residuosity. An integer q is called quadratic residue (QR) if there exists an integer x such that $x^2 = q \mod n$, and q is called quadratic non-residue (QNR) otherwise. It is considered hard to predicate if q is QR or QNR when n is a product of two distinct prime numbers of equal length. On the other hand, the ITPIR can offer perfect security, that is, the server cannot acquire any information about the client's query even if the server has unlimited computational resources and unlimited time. In order to achieve the ITPIR, we usually need multiple servers and an assumption that these servers do not collude. The t-private ℓ-server PIR can information theoretically guarantee the privacy of the query even if up to t out of ℓ servers collude. Beimel and Stahl [31] introduced a notation called t-private k-out-of-ℓ PIR in which k out of ℓ servers need to respond and up to t servers may collude without compromising the security. In addition they examined a situation that v out of k servers can return incorrect answers, due to a malicious servers or database failure, which is termed as t-private v-Byzantine-robust k-out-of-ℓ PIR. Chor et al. [5], [30] have proposed several schemes. Their proposals enable the client to retrieve one bit with $O(\sqrt{N})$ communication cost in a simple ℓ-server scheme, $O(N^{1/\ell})$ communication in a general ℓ-server scheme and $\frac{1}{3}(1+o(1))\cdot\log^2 N\cdot\log\log(2N)$ communication cost in $\frac{1}{3}\log N + 1$-server scheme. Their scheme is information theoretic secure.

After the first proposal of PIR in [5], several improvements in terms of communication cost have been shown [10], [31]–[37]. The scheme proposed in [35] requires $O(N)$ server storage and $O(1)$ communication overhead, and is information theoretic secure. This scheme was implemented as an open-source project on SourceForge [38]. Smith et al. [32], [33] proposed a scheme with a tamper-proof device. The client sends the secure coprocessor (SC) with encrypted query. The SC receives the query and decrypt it. Then the SC reads the entire database and get the requested data item. When returning the data item to the client, the SC encrypts the item and send it back to the client. Williams and Sion [10] proposed a single-server PIR with an ORAM and a secure coprocessor. The communication and computational complexities of their scheme is $O(\log^2 N)$ and $O(\sqrt{N})$ client storage is required. The security of their scheme is proven when it has the access to a random oracle. Devet et al. [37] improved the scheme presented by Goldberg [35] and evaluated the performance of the scheme. They showed that their implementation was several thousands times faster than the scheme of [35].

Bao et al. [39] and Schnorr et al. [40] independently proposed similar PIR schemes using homomorphic encryption. Their schemes requires only $O(1)$ computation to be done on-line. The idea of the schemes is to do as many operations as possible off-line to achieve practical on-line overhead. However, because of heavy off-line computation, the client has to wait until the server is ready to be queried, and this latency may make the schemes less practical. The schemes work as follows: When the client wants to obtain a data item, which is encrypted with the servers secret key and publicly available, first the client download the item and encrypts it with the client's secret key. After the encryption at the client, the client sends the item to the server and asks to remove the server's encryption. Finally, the client obtain the data item by decrypting one's own encryption.

Asonov and Freytag's scheme [34] also assumes the SC inside the server and SC first shuffles the entire database according to a random permutation π. When the client request the data item i_1, the SC fetches the item from $\pi(i_1)$. This only requires $O(1)$ of computation and communication. For the second query of requesting the item i_2, the SC first has to read $\pi(i_1)$ and then read $\pi(i_2)$. When $i_1 = i_2$ (the second and the first query request the same data item), the SC reads $\pi(i_1)$ and a random item in order to hide the fact that the client is reading the same data item twice. Therefore, for the n-th query, the SC has to read all previously read items before reads $\pi(i_n)$. At a certain point, the SC has to pick a new permutation π' and shuffle the database with π'.

The PIR schemes can protect user's privacy from an honest-but-curious servers. However, PIR schemes can not offer a protection from a dishonest user. Gertner et al. [41] proposed a symmetric private information retrieval (SPIR) that prevents the user from learning additional information. Henry et al. [42] considered an application of SPIR for e-commerce and proposed a protocol that extended the PIR scheme [35] to a priced symmetric private information retrieval (PSPIR). Their PSPIR scheme maintain user's anonymity and does not leak any information about the record of user's purchases.

Recently implementations of CPIR and evaluating performances on real environments are attracting more attentions. Melchor and Gaborit [43], [44] proposed a lattice-based new scheme with a reasonable communication cost and with computational complexity being improved by a factor of one hundred. Later, Olumofin and Goldberg [45] implemented the lattice-based scheme and evaluated the performance. They demonstrated that the overhead of Olumofin and Goldberg's scheme was 10 to 1000 times smaller than the trivial scheme (i.e. downloading entire database).

Recent schemes can outsource the database to untrusted servers and yet can protect both the privacy of the database owner and clients. Huang and Goldberg [46] proposed a scheme for outsourcing Private Information Retrieval. Their scheme requires $O(\sqrt{N})$ computational overhead and the server stores $O(\sqrt{N})$ data. The security of the scheme is proven when it has the access to a random oracle. They also implemented their scheme and evaluated the performance. When the client updates 1MB record in the 1 TB database, an amortised end-to-end latency is smaller than 300 ms.

The comparison of PIR schemes is summarised in Table 2. In Table 2, we compared communication overhead of schemes since they are generally evaluated by communication overhead[†], while ORAM schemes are evaluated by computational overhead.

2.3 Hardware-Assisted Control Flow Obfuscation

ORAM constructions remain too expensive to be implemented on embedded processors. In [27], 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. We give a brief description of the scheme below; for more details, refer to [27].

Let n be the size (in blocks) of memory, and $m \ll n$ be the size of the shuffle buffer. The shuffle buffer is within the CPU trusted boundary, i.e. it is considered secure local storage (cache), and an adversary is not able to observe access

[†] All literatures referred in Table 2 evaluate their own scheme in terms of the communication overhead, except Huang and Goldberg's scheme [46].

Table 2 Comparison of PIR. ℓ is the total number of servers and k is the minimal number of servers which are available at the time of retrieval. c is a constant of $c \geq 2$ and ϵ is a small constant of $\epsilon \geq 0$.

	Communication Overhead	Server Storage	Client Storage	Security
CG [28]	$O(N^{\epsilon})$	$O(N)$	$O(1)$	one-way function
KO [29]	$O(N^{\epsilon})$	$O(N)$	$O(1)$	quadratic residuosity problem
CKGS [5], [30]	$O(\log^2 N \log\log N)$	$O(N \log N)$	$O(1)$	information theoretic
AF [34]	$O(1)$	$O(N)$	$O(1)$	secure coprocessor
G [35]	$O(1)$	$O(N)$	$O(1)$	information theoretic
BS [31]	$O(N^{1/k})$	$O(N)$	$O(1)$	information theoretic
WS [10]	$O(\log^2 N)$	$O(N \log N)$	$O(\sqrt{N})$	random oracle
GMOT [36]	$O(1)$	$O(N)$	$O(N^{1/c})$	one-way function
HG [46]	$O(\sqrt{N})^{\ddagger}$	$O(N)$	$O(1)$	random oracle

‡ They evaluated the scheme in terms of computational overhead.

pattern in the shuffle buffer. As in other parts of this paper, we assume that data is stored encrypted and access operations consist of sequential `read` and `write` operations. A random permutation is initially applied to data before loading it to RAM. The scheme then works as follows.

1. The first m blocks in memory are moved to the shuffle buffer.
2. When making a request for a data item, if the block is found in the shuffle buffer, access the block.
3. If the block is not found in the shuffle buffer, pick a random block in the shuffle buffer and swap it with the requested data block in memory (the accessed data item is now in the shuffle buffer).
4. When the program finishes its entire process, the full contents of shuffle buffer are written back into memory.

Note that item 3. implies that the permutation used to map data in memory is *dynamically* modified as the program runs. Although the dynamic secret permutation helps to protect the privacy of individual items being accessed (or being repeatedly accessed), it also means that the scheme needs to make use of a block address table to map data items into memory (describing the permutation at time t). As a result there is the storage requirement of size $O(n)$ within the trusted environment to represent this mapping (albeit with constant < 1).

The scheme trades security for low overhead. Besides the costs of having a secure on-chip buffer, the scheme trivially leaks information about access of data items during execution of program. In fact, the lack of memory access in step 2 indicates that when step 3 is executed, one knows the exact block being accessed (the one in memory, being brought into the buffer). Thus a step 2 followed by a step 3 indicates that the data items being accessed are definitely distinct (i.e. there is no 2-recurrence at this particular stage). Likewise, a step 3 followed by a step 2 indicates a 2-recurrence with probability $1/m$. Furthermore, an access in memory to a data item which was previously swapped out from the buffer indicates with high probability the existence of repeated access to a particular data. These could be confirmed by running the program several times.

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 adapt some of the ideas from this scheme in our proposal.

2.4 Lightweight Memory Access Pattern Protection Scheme

We first defined a new security notion called *δ-length ϵ-security* [1].

Definition 1 *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 $d \leq \delta$.*

Then we proposed a practice-oriented scheme for protecting RAM access pattern and considered two instances, which satisfy δ-length ϵ-security. The first instance is similar to the proposal by Zhuang et al. [27], and 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. The second instance requires no special hardware and can offer the same level of security as the first instance, but as a result leads to a higher overhead.

The main feature of the proposal in [1] is to maintain the *history* of access, which together with the access of dummy data, helps one to hide data access pattern without frequent oblivious re-shuffling. The history table $\mathcal{H}$ is stored inside secure memory, in case of the first instance, along with the buffer $\mathcal{M}$ and random permutation $\mathcal{E}$. As we do not require secure memory for the second instance, all of them are stored in unsecured memory.

When the program is invoked, our scheme first randomly permutes all data blocks according to the permutation $\mathcal{E}$. Note that unlike ORAM, this permutation is done only once. As different mapping is chosen every time the program is invoked, our scheme has to hold the mapping table inside the secure region. After the shuffling, the program starts accessing data blocks on RAM. The scheme swaps data blocks between the secure buffer and main memory for every access. The buffer temporarily holds M data blocks and corresponding addresses, which were recently accessed. As the size of buffer is limited, the buffer will be full at some time, thereafter every time the program accesses a data block, two blocks should be removed from the buffer and kicked back to memory. The addresses of those blocks, which are kicked out from the buffer, are stored in the history table. The history table can hold up to H addresses. When the history table gets full, two random entries are overwritten by new entries.

Let us explain the scheme using an example in which the program accesses the data block 'a' stored in the address i_a. The scheme always brings two blocks from memory into the buffer and evict two blocks instead. Which blocks should be evicted can be randomly determined. The choice of which blocks should be brought is described as follows:

1. if 'a' is in $\mathcal{M}$, we replace two random elements (not 'a') from $\mathcal{M}$ by a random element 'r' from main memory and a random element 'p' from main memory which had already been accessed before (as recorded in the history table), and we 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 'r' from main memory and 'a' (as recorded in the history table). Note that $\mathcal{H}$ holds only addresses and data itself is stored in main memory.
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 'p' from main memory which had already been accessed before (as recorded in the history table).

In words, the choice is determined as one of the blocks always looks randomly chosen and the other always looks one in the history table. Let us take the case 2 as an example, two blocks (r, a) are copied to the buffer. The block 'r' looks as it is randomly chosen, and it is indeed. The block 'a' looks as if it is chosen not because it is requested, but because its address is registered in the history table from the adversary's view point. As the first instance of their scheme assumes secure hardware, the program can access the block 'a' directly after bringing two blocks. In the second instance, the program has to access all data blocks in the buffer otherwise the adversary might be able to notice which block is actually the program is accessing. We have a pseudocode of our scheme in Algorithm 1.

Algorithm 1 Pseudocode of access pattern protection scheme

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

2.5 Difference between ORAM and PIR

The target of both ORAM and PIR is the same: how to efficiently protect the pattern of accesses. The main difference is that ORAM supports both the `read` and `write` operations to RAM (or server) while PIR usually consider only the `read` operation. As the functionality of PIR is limited compared to ORAM, PIR tends to be more practical, in terms of communication and storage cost, than ORAM. PIR works very well when one server or cloud operator provides a large database and many clients want to download part of the database in a privacy preserving manner. However, as PIR does not support `write` operation, it does not work well in some services for example file hosting services where clients upload their files and they often update part of their files. Neither does PIR work well when a software developer wants to protect his/her software from reverse engineering. ORAM is usually less practical than PIR in terms of performance, however, ORAM supports both `read` and `write` operations. Hence ORAM is suitable for protecting access pattern in a database which is often updated and a software protection.

PIRs usually offer less functionality than ORAMs, hence they are lighter than ORAMs and oppose smaller overhead. Because of their lightweightness, PIRs are more attractive for users who wish to hide only their reading pattern. Some ORAM schemes are practical and their applications to realise a private access to remote servers have also been considered. Our scheme is as light as one of the most efficient PIRs, our scheme can be used instead of PIRs without sacrificing performance.

3. Implementation Issues of ORAM

Considering the practical implementation of ORAM, there were several questions to be addressed and improvements to be achieved. We propose practical solutions for these issues. The issues are;

- Management of data blocks in the buffer
- Construction of a secure area
- Size of each block.

In Sect. 3.1, we propose a better management of data blocks. The construction of secure area based only on software is discussed in Sect. 3.2. Finally we discuss better use of each block for saving storage in Sect. 3.3.

3.1 Managing Data in Buffer Using Flags

The square-root based solutions first scan all data blocks in order to know if the accessing block is in the shelter or not at the beginning of the process. If the accessing block is in the shelter, the block from a dummy location will be fetched into the shelter. When the dummy block is being fetched, there is a chance that the block which is already in the shelter is chosen again. If this block is the dummy one, it is not a problem. However, if the block is the real one, which is the one actually accessed by the program, we cannot distinguish which block is newer. As the result, the program may misbehave in the software protection scenario and the database may be ruined in the database access protection scenario. Therefore, we must ensure that all blocks in the shelter are the latest and there is no duplication.

The trivial solution to ensure that is to scan the entire shelter before fetching the data block, which impose large overhead. We can omit this scan by using n 1-bit flags F_i indicating whether the data block is in the buffer or not. We set $F_i = 1$ when data stored in address i is stored in the buffer and $F_i = 0$ otherwise. When bringing two blocks into the buffer, we check flag of each block. If it is 0, we bring the block into the block. If it is 1, we pick different block and check the flag again.

The same situation can happen in the lightweight scheme proposed in [1]. Their scheme always fetches two blocks in to the shelter, the chance of the block which is already in the shelter being chosen is higher than the square-root solution. We have confirmed that this solution can improve the performance by roughly 3 times in many parameter settings with a prototype implementation. When the hierarchical solution is applied, there is no need for the management of newer data as newer data blocks are always upper level. The scheme proposed in [27] does not either require this management as their scheme swaps two blocks between the shelter and main area.

3.2 Constructing Secure Region

ORAM of square-root solution has a shelter of size $\sqrt{N}$ and the scheme has to access all data blocks in the shelter at least twice per one access. As the size of the program or database grows, the size of the shelter also grows, which increase the computational overhead of the implementation. ORAM of hierarchical solution also has the same problem as it also has to access all data blocks in the top level buffer. Though it may not be serious as square-root solution, as the size of top level buffer is fixed and usually smaller than $\sqrt{N}$.

The cost of accessing shelter or top leverl buffer can be reduced by using secure hardware where only the client can access as done in [1], [27]. Though this is looks promising, requiring secure hardware may compromise the practicality. The third option is to construct a secure region with obfuscation. Let $access(K)$ be the complexity of accessing K blocks and $obf(V)$ be the complexity of obfuscating variables V. Also let X be the variables after the obfuscation. Then the obfuscation can offer higher performance than accessing all data blocks in the buffer when the following inequation holds;

$$access(K) \leq obf(V) + access(X).$$

In the square root construction, the size of the shelter tends to large as its size is $\sqrt{N}$. Therefore, it is likely that we can improve the performance of ORAM scheme with this method. Depending on the size of the top level buffer for hierarchical solution and the complexity of obfuscation $obf(V)$, this method is also applicable to the hierarchical solutions. It is also applicable and for the scheme of [1].

We use the scheme proposed in [47] for obfuscation. By using obfuscation with the memory protection scheme, we only need to protect the shelter (square-root), top level buffer (hierarchical) or buffer and history table ([1]), which are much smaller than N. As described in the following, we divide the buffer and history table into smaller ones and apply the obfuscation repeatedly. Hence we can further decrease the overhead due to obfuscation.

By obfuscating data blocks inside the secure region, the accesses to memory is also "obfuscated", that is, the access to a certain data block is transformed into the access(es) to obfuscated block(s). When the protection scheme needs to access one of data blocks, it accesses obfuscated blocks and unlock the obfuscation in order to obtain the real value. As the correspondence between the original access and the "obfuscated" access(es) is secret, an adversary cannot understand which block is actually accessed. The obfuscation does not affect any operation done by the protection scheme as it only encodes the data blocks. The obfuscation also realise less access overhead than that of accessing all blocks in the buffer by appropriately choosing the parameters, which we discuss later this section. Thus, by using obfuscation for data blocks, we can construct a secure area without hardware, and less overhead than accessing all blocks in the buffer.

Each block which requires to be secret is implemented as variables, and these variables are encoded into obfuscated variables. The following is a small example of the obfuscation from a set of variables $\{v_1, v_2, v_3\}$ into a set of obfuscated variables $\{x_1, x_2, x_3, x_4, x_5\}$;

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ d \end{pmatrix} \oplus \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix}, \tag{1}$$

where the matrix is chosen as its rank to be the same as the number of elements in the original variables (in this example, it is 4). Each element of the matrix is randomly chosen except the rightmost column, which is set to all 1s. A random variable d is generated by a pseudo-random number generator and it is attached to the original variables. Finally a secret vector $\{y_1, y_2, y_3, y_4, y_5\}$ is exclusive-ORed, this operation acts like random masking. We repeatedly apply the same obfuscation for the multiple sets of the original values until we obtain enough size of the secure region. In this example, we obtain the secure region of size 4 per one iteration.

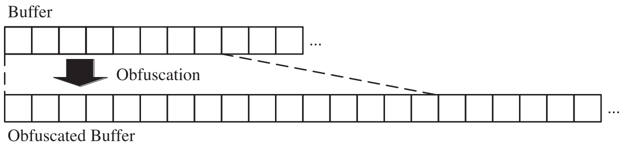

Fig. 1 Constructing secure area using obfuscation.

When one of v_is, say v_1, is requested, we first determine which set of obfuscated variables to access depending on the index of v_1, then access all x_is to decode v_1. Suppose v_1 is updated to v_1' after the operation, then we apply the same obfuscation as given in Eq. (1) with newly generated random number d'. We use different random numbers every time the program accesses to the secure region. As the corresponding column to d is set to all 1s, the change of d will cause the changes of all variables of $\{x_i\}$ even if none of variables $\{v_i\}$ are changed (i.e. `read` operation). Moreover, as the encoding applies the secret matrix, the adversary cannot detect the correspondence between $\{v_i\}$ and $\{x_i\}$. Though the adversary may be able to detect which one of $\{x_i\}$ is being accessed, one cannot understand which one of $\{v_i\}$ is being accessed. Thus, we can construct a secure region using obfuscation.

There is, however, a limitation to use obfuscation for constructing secure area, which is the size and number of all variables must be pre-determined. In order to overcome this limitation, we divide the secure region into smaller units. Figure 1 shows an example when 8 blocks to be obfuscated into 16 blocks with one random number. We can of course use different matrices and vectors for each obfuscation unit, which makes obfuscation more difficult to be analysed, though this will increase the code size. We also note that we can of course choose arbitrary size for the unit of obfuscation.

3.3 Reducing Storage Overhead

In the scheme of [27], the uniform unit of data was defined as a block of size 128-bit. However, some programs uses a byte block or smaller as the smallest unit. When one byte data is allocated to 16 byte block, it wastes remaining 15 bytes. For the better management of storage, we modify the scheme so that one block can store multiple smaller blocks.

This modification not only saves storage, but also improves the performance for operations such as data copy and move. When a program copies for example four integer-type blocks, the original scheme requires four copy operations. On the other hand, our proposal only requires one copy operation when these four blocks are in the same 128-bit block. By carefully choosing smaller blocks on storing them into a large block, we can expect speed-up and saving storage at the same time.

3.4 Implementation Result

We implemented our scheme to which three modifications are applied. Then we measured the required time to write 1MB data on RAM using the protection scheme. The evaluation was done on the PC of CPU: Intel Core i7 3930K and RAM: 8 GB which runs Ubuntu 13.04 x64 and gcc 4.7.3. As our scheme has two parameters (e.g. number of blocks in the buffer and history table) which affects the performance, we measured the required time while we change the number of blocks in the buffer and the history table from 100 to 1000. Figure 2 shows the relation between parameters and performance of our efficient implementation. The x-axis denotes the number of blocks (from 100 to 1,000 for every 100) in the secure buffer, the y-axis denotes required time to load 1 MB data to the protection scheme and each curve corresponds to each size of the history table. In the fastest setting, our scheme only requires 0.125[s] to write 1 MB data. As shown in the figure, the overhead grows linear to the size of buffer and history table. This is because more time required to scan the entire buffer and history table as their size grow. However, growth of overhead is very slow compared to that of size of buffer and history table thanks to the efficient implementation of scanning process.

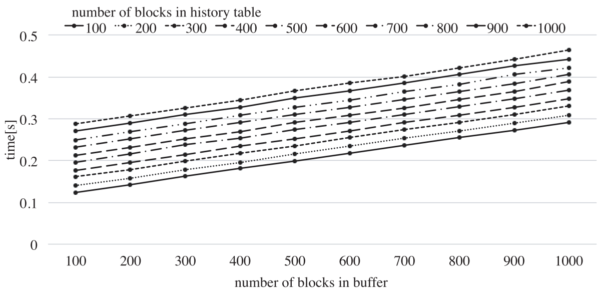

Fig. 2 Performance evaluation in various settings.

4. Conclusion

In this paper, we summarised series of researches on ORAM and PIR, and compared the overhead of each scheme. We also introduced two lightweight scheme. Then discussed three general implementation issues, namely management of data blocks in the buffer, construction of a secure area and size of each block, and their solutions. The solutions can be widely used for implementing ORAM and improving the performance.

Acknowledgement

The last author is partially supported by JSPS-Kaken No.23300027.

References

[1] Y. Nakano, C. Cid, S. Kiyomoto, and Y. Miyake, "Memory access pattern protection for resource-constrained devices," CARDIS, LNCS, vol.7771, pp.188–202, 2012.

[2] Y. Nakano, S. Kiyomoto, and Y. Miyake, "Evaluation of memory access pattern protection in a practical setting," ISA, ASTL, vol.48, pp.1–6, 2014.

[3] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," STOC, pp.182–194, 1987.

[4] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," J. ACM, vol.43, no.3, pp.431–473, 1996.

[5] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," FOCS, pp.41–50, 1995.

[6] J.R. Lorch, J.W. Mickens, B. Parno, M. Raykova, and J. Schiffman, "Toward practical private access to data centers via parallel ORAM," IACR ePrint, vol.2012, p.133, 2012.

[7] M. Ajtai, J. Komlós, and E. Szemerédi, "An $O(n \log n)$ sorting network," STOC, pp.1–9, 1983.

[8] K.E. Batcher, "Sorting networks and their applications," AFIPS Spring Joint Computing Conference, AFIPS Conference Proceedings, vol.32, pp.307–314, 1968.

[9] M. Ajtai, "Oblivious RAMs without cryptographic assumptions," STOC, pp.181–190, 2010.

[10] P. Williams and R. Sion, "Usable PIR," NDSS, 2008.

[11] B. Pinkas and T. Reinman, "Oblivious RAM revisited," CRYPTO, LNCS, vol.6223, pp.502–519, 2010.

[12] D. Boneh, D. Mazieres, and R.A. Popa, "Remote oblivious storage: Making oblivious RAM practical," Tech. Rep. MIT-CSAIL-TR-2011-018, Massachusetts Institute of Technology, 2011.

[13] I. Damgård, S. Meldgaard, and J.B. Nielsen, "Perfectly secure oblivious RAM without random oracles," TCC, LNCS, vol.6597, pp.144–163, 2011.

[14] E. Shi, T.H.H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O((\log N)^3)$ worst-case cost," ASIACRYPT, LNCS, vol.7073, pp.197–214, 2011.

[15] M.T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving group data access via stateless oblivious RAM simulation," SODA, pp.157–167, 2012.

[16] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (in)security of hash-based oblivious RAM and a new balancing scheme," SODA, pp.143–156, 2012.

[17] E. Stefanov and E. Shi, "ObliviStore: High performance oblivious cloud storage," IEEE Symposium on Security and Privacy, pp.253–267, 2013.

[18] E. Stefanov, E. Shi, and D.X. Song, "Towards practical oblivious RAM," NDSS, 2012.

[19] P. Williams and R. Sion, "Single round access privacy on outsourced storage," ACM Conference on Computer and Communications Security, pp.293–304, 2012.

[20] E. Stefanov, M. van Dijk, E. Shi, C.W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An extremely simple oblivious RAM protocol," ACM Conference on Computer and Communications Security, pp.299–310, 2013.

[21] P. Williams, R. Sion, and A. Tomescu, "PrivateFS: A parallel oblivious file system," ACM CCS, pp.977–988, 2012.

[22] P. Williams, R. Sion, and B. Carbunar, "Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage," ACM Conference on Computer and Communications Security, pp.139–148, 2008.

[23] C. Gentry, K.A. Goldman, S. Halevi, C.S. Jutla, M. Raykova, and D. Wichs, "Optimizing ORAM and using it efficiently for secure computation," Privacy Enhancing Technologies, LNCS, vol.7981, pp.1–18, 2013.

[24] M.T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious RAM simulation," ICALP (2), LNCS, vol.6756, pp.576–587, 2011.

[25] M.T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Oblivious RAM simulation with efficient worst-case access overhead," CCSW, pp.95–100, 2011.

[26] S. Lu and R. Ostrovsky, "Distributed oblivious RAM for secure two-party computation," IACR ePrint, vol.2011, p.384, 2011.

[27] X. Zhuang, T. Zhang, H.H.S. Lee, and S. Pande, "Hardware assisted control flow obfuscation for embedded processors," CASES, pp.292–302, 2004.

[28] B. Chor and N. Gilboa, "Computationally private information retrieval (Extended abstract)," STOC, pp.304–313, 1997.

[29] E. Kushilevitz and R. Ostrovsky, "Replication is NOT needed: SINGLE database, computationally-private information retrieval," FOCS, pp.364–373, 1997.

[30] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," J. ACM, vol.45, no.6, pp.965–981, 1998.

[31] A. Beimel and Y. Stahl, "Robust information-theoretic private information retrieval," J. Cryptology, vol.20, no.3, pp.295–321, 2007.

[32] S.W. Smith and D. Safford, "Practical private information retrieval with secure coprocessors." Technical Report, IBM Research Division, 2000.

[33] S.W. Smith and D. Safford, "Practical server privacy with secure coprocessors," IBM Systems Journal, vol.40, no.3, pp.683–695, 2001.

[34] D. Asonov and J.C. Freytag, "Almost optimal private information retrieval," Privacy Enhancing Technologies, LNCS, vol.2482, pp.209–223, 2002.

[35] I. Goldberg, "Improving the robustness of private information retrieval," IEEE Symposium on Security and Privacy, pp.131–148, 2007.

[36] M.T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Practical oblivious storage," CODASPY, pp.13–24, 2012.

[37] C. Devet, I. Goldberg, and N. Heninger, "Optimally robust private information retrieval," Proc. 21st USENIX Conference on Security Symposium, Security '12, Berkeley, CA, USA, 2012.

[38] I. Goldberg, "Percy++ project," SourceForge,
http://percy.sourceforge.net/

[39] F. Bao, R.H. Deng, and P. Feng, "An efficient and practical scheme for privacy protection in the E-commerce of digital goods," ICISC, LNCS, vol.2015, pp.162–170, 2000.

[40] C.P. Schnorr and M. Jakobsson, "Security of signed ElGamal encryption," ASIACRYPT, LNCS, vol.1976, pp.73–89, 2000.

[41] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin, "Protecting data privacy in private information retrieval schemes," STOC, pp.151–160, 1998.

[42] R. Henry, F.G. Olumofin, and I. Goldberg, "Practical PIR for electronic commerce," ACM Conference on Computer and Communications Security, pp.677–690, 2011.

[43] C.A. Melchor and P. Gaborit, "A fast private information retrieval protocol," ISIT, pp.1848–1852, 2008.

[44] C.A. Melchor, G. Castagnos, and P. Gaborit, "Lattice-based homomorphic encryption of vector spaces," ISIT, pp.1858–1862, 2008.

[45] F.G. Olumofin and I. Goldberg, "Revisiting the computational practicality of private information retrieval," Financial Cryptography, LNCS, vol.7035, pp.158–172, 2012.

[46] Y. Huang and I. Goldberg, "Outsourced private information retrieval," WPES, pp.119–130, 2013.

[47] K. Fukushima, S. Kiyomoto, T. Tanaka, and K. Sakurai, "Analysis of program obfuscation schemes with variable encoding technique," IEICE Trans. Fundamentals, vol.E91-A, no.1, pp.316–329, Jan. 2008.

Yuto Nakano received B.E. and M.E in Electrical and Electronic Engineering from Kobe University, Japan in 2006 and 2008, respectively. He joined KDDI and has been engaged in the research on hash functions and stream ciphers. He is currently a researcher at the Information Security Lab. of KDDI R&D Laboratories Inc.

Shinsaku Kiyomoto received his B.E. in engineering sciences and his M.E. in Materials Science from Tsukuba University, Japan, in 1998 and 2000, respectively. He joined KDD (now KDDI) and has been engaged in research on stream ciphers, cryptographic protocols, and mobile security. He is currently a senior researcher at the Information Security Laboratory of KDDI R&D Laboratories Inc. He was a visiting researcher of the Information Security Group, Royal Holloway University of London from 2008 to 2009. He received his doctorate in engineering from Kyushu University in 2006. He received the IEICE Young Engineer Award in 2004. He is a member of JPS.

Yutaka Miyake received the B.E. and M.E. degrees of Electrical Engineering from Keio University, Japan, in 1988 and 1990, respectively. He joined KDD (now KDDI) in 1990, and has been engaged in the research on high-speed communication protocol and secure communication system. He received the Dr. degree in engineering from the University of Electro-Communications, Japan, in 2009. He is currently a senior manager of Information Security Laboratory in KDDI R&D Laboratories Inc. He received IPSJ Convention Award in 1995 and the Meritorious Award on Radio of ARIB in 2003.

Kouichi Sakurai received his B.S. degree in Mathematics from Faculty of Science, Kyushu University and his M.S. degree in Applied Science from the Faculty of Engineering, Kyushu University in 1986 and 1988, respectively. He was engaged in research and development on cryptography and information security at the Computer and Information Systems Laboratory at Mitsubishi Electric Corporation from 1988 to 1994. He received his Doctorate in Engineering from the Faculty of Engineering, Kyushu University in 1993. From 1994, he worked for the Department of Computer Science of Kyushu University in the capacity of associate professor and became a full professor in 2002. His current research interests are in cryptography and information security. Dr. Sakurai is a member of the Information Processing Society of Japan, the Mathematical Society of Japan, ACM, IEEE and the International Association for Cryptologic Research.