Oblivious RAM as a Substrate for Cloud Storage — The Leakage Challenge Ahead

Marc Sánchez-Artigas
Universitat Rovira i Virgili
marc.sanchez@urv.cat

ABSTRACT

Oblivious RAM (ORAM) is a well-established technology to hide data access patterns from an untrusted storage system. Although research in ORAM has been spurred in the last few years with the irruption of cloud computing, it is still unclear whether ORAM is ready for the cloud. As we demonstrate in this short paper, there are still some important hurdles to be overcome. One of those is the standard block-based ORAM interface, which can become a timing side-channel when used as a substrate to implement higher level abstractions such as filesystems, personal storage services, etc., typically found in the cloud. We analyze this form of leakage and discuss some possible solutions to this problem, concluding that thwarting it in an efficient manner calls for further research.

Keywords

ORAM; access pattern; leakage; information flow

1. INTRODUCTION

Oblivious RAM (ORAM) is a privacy tool to conceal data access patterns. Although this concept was proposed decades ago to hide the accesses made to memory by programs [4], the interest in ORAM has been much revived with the advent of cloud computing. Specifically, many cloud services require of secure online storage to host their sensitive data (healthcare data, financial reports, etc.). Privacy concerns for those systems should not only involve protection against data leaks, which could be realized through encryption, but also against side channels such as data access patterns. To address this issue, a bunch of ORAM schemes have been proposed [15, 8, 13, 11], to make this technique more amenable to the cloud.

The problem. Despite the enormous progress made during the last few years, for instance, to cope with multi-client and asynchronous scenarios (see TaoStore [10] and the references thereof), it is still unclear whether ORAM is actually getting to a practical end. Proof of that is the recent work [2], which discusses the existing mismatch between ORAM theory and the performance requirements of cloud applications.

However, there are other fundamental issues that have also been overlooked in the literature. One of them is whether the block-based ORAM interface is indeed suitable for the cloud. Although the "key-value"-like ORAM interface is analogous to that provided by cloud storage services (e.g., Windows Azure, Amazon S3, etc.), it is not readily adequate for many storage systems like (distributed) file systems, personal storage (e.g., Dropbox, UB1, Box, OneDrive, etc.) and document-oriented databases (e.g., MongoDB, CouchDB, etc.), to cite a few. The key reason is that this key-value interface is only oblivious at the block level, but not so at upper levels such as at the file or document levels. For example, this implies that an access to a file is turned into several block accesses in a short period of time, which is seen by the remote storage service as batch of requests originating from the same source. This very simple observable feature can leak bits on the private user input and thus easily break ORAM security.

Contributions. In this paper, we show that this simplistic form of timing channel can leak extra information through a practical example, and we elucidate the existence of a trade-off between performance and information leakage that arises from the interface mismatch between cloud applications and ORAM. Finally, we discuss some possible solutions to such a problem, which call for further research. To our knowledge, this work is the first that studies the effects of the interface mismatch between ORAM algorithms and cloud storage.

2. PRELIMINARIES

"Oblivious" Storage Interface. From the seminal work by Goldreich and Ostrovsky [4], the atomic unit of storage and access in ORAM algorithms has been the *block*. This has led to the definition of a simple block-based interface that allows an ORAM client to `read` and `write` to block addresses $[N]$. This block interface is said to be oblivious, i.e., the untrusted storage server cannot learn the plaintext of user content, the requested addresses, nor the relationships between requested addresses. Technically, this interface offers two primitives: 1) $(\texttt{read}, v_i, \texttt{data})$, to read a block `data` with unique address v_i from a storage repository, and 2) $(\texttt{write}, v_i, \texttt{data})$, to write a block `data` with unique address v_i to the remote server. This basic interface, originated from memory protection, does not consider all types of modern cloud storage such as object and file storage, becoming a source of privacy leakage.

Security Model. As commonly agreed, we consider that the remote server is "honest but curious", that is, it behaves

CCSW'16, October 28 2016, Vienna, Austria
© 2016 ACM. ISBN 978-1-4503-4572-9/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2996429.2996430

correctly in following the protocol, but it attempts to gain as much knowledge as possible by direct observation of the data access pattern. The network connection between the clients and the server is assumed to be secure, e.g.,through SSL.

However, we assume that the communication between the client and the server is asynchronous, just because the client can issue multiple block requests at any time, to fetch all the blocks that belong to a file or object. And thus, it is easy for the server to learn *when* requests are related to one another, information that can also be visible by a potential intruder in the datacenter. Notice that we implicitly assume that block requests are never intentionally delayed by the client in order to not slow down the application needing these accesses. The rest of the security definition is the standard one, which can be found elsewhere (see Path ORAM [15], for instance).

Path ORAM. Path ORAM [15] is a tree-based ORAM. It organizes the server storage as a tree where each node (a.k.a. bucket) holds a few blocks. To fulfill a request for address v, the ORAM client looks up a position map that links block v to a random tree leaf. From the leaf, all blocks on the path to the root are retrieved to find the block with address v. An eviction procedure on the read path is then ran to insert all the read blocks (temporarily stored in a client stash) back to the tree, and as close as possible to the leaf.

Path ORAM requires negligible computation. However, it can incur substantial bandwidth overhead. Let W denote the bandwidth cost of an ORAM scheme defined as *the number of blocks transferred per block of useful data*. In practice, the bandwidth cost W of Path ORAM is of $\mathcal{O}(\log N)$. For each `read` and `write` operation, the client reads a path of $Z \log N$ blocks from the server, writing them back. This amounts to a total of $2Z \log N$ blocks bandwidth per request, where Z is the bucket size in blocks. For complete details, see [15].

3. METRICS, STORAGE SIDE-CHANNELS

Bandwidth Efficiency. Although the bandwidth cost is a useful metric to compare an ORAM algorithm to a baseline, it can be misleading for investigating the efficiency of ORAM in cloud storage systems. The major reason is that the block size is chosen offline in presumably all known schemes, that is, before any piece of data is stored in the system, and thus, it is subject to *internal fragmentation*: some space is wasted within each valid block because of the rounding-up from the actual requested file size to the allocation granularity.

Since network-level efficiency is central for cloud providers, we propose a new metric to quantify the efficiency of ORAM named *Bandwidth Efficiency* (BE). It is defined as:

$$BE = \frac{\text{Total data to access a set of files}}{\text{Total size of files}}.$$

More formally, given a set of M files $\mathcal{F} = \{f_1, f_2, \ldots, f_M\}$, let $s(f_i)$ be the file size of file f_i in bits. Let $\mathbb{P}(f_i)$ be the access probability of file i (we assume $\sum_{i=1}^{M} \mathbb{P}(f_i) = 1$). Then,

$$BE = \frac{W\left(\sum_{i=1}^{M} \lceil s(f_i)/D \rceil \, \mathbb{P}(f_i)\right)}{\sum_{i=1}^{M} s(f_i)\mathbb{P}(f_i)}, \tag{1}$$

where W is the bandwidth cost incurred by an ORAM access in blocks and D is the size of the ORAM blocks in bits. Since our metric accounts for the retrieval of *useful* information, it better captures the real efficiency of ORAM in cloud storage systems. In particular, for Path ORAM, we have that

$$BE = \frac{2Z \lceil \log_2(N) \rceil \left(\sum_{i=1}^{M} \lceil s(f_i)/D \rceil \, \mathbb{P}(f_i)\right)}{\sum_{i=1}^{M} s(f_i)\mathbb{P}(f_i)},$$

where N is the total number of blocks required to store the M files and is given by $\sum_{i=1}^{M} \lceil s(f_i)/D \rceil$.

Privacy Leakage. A limitation in current ORAM schemes is that they operate at the block level. And hence, they may leak bits of information about the secret input when accesses occur at a higher granularity (file). To measure this form of leakage, we will use the *min-entropy* metric proposed in [12]. The reason is that classical metrics like the Shannon entropy yield bounds for the expected effort for recovering secrets by brute-force search. However, as illustrated in [12], even if the average effort can be proven to be significant, the probability of guessing the secret with a small number of guesses can be high. This better matches the reality of cloud storage, and it is expected strong security guarantees.

Concretely, we consider an implementation of a storage application that manages data at the file level. Built on top of ORAM, every file access in this application translates into a *batch* of random block accesses. Because the block size D is of fixed size (typically, of at least $\Omega(\log N)$ bits), the batch size can be captured by a function $g : \mathcal{F} \to \mathcal{B}$ such that $g(f)$ returns the number of blocks to be fetched to read and store a file f. Here $\mathcal{F}$ is the finite set of files of a user, a file system or a file store, and $\mathcal{B}$ denotes the set of possible batch sizes in number of blocks.

Clearly, such a deterministic model induces an equivalence relation $\sim$ on $\mathcal{F}$: Two files f and f' are equivalent $f \sim f'$, iff $g(f) = g(f')$. For clarity, let $\mathcal{F}_b$ denote the equivalence class $f^{-1}(b)$: $\mathcal{F}_b = \{f \in \mathcal{F} \mid g(f) = b\}$. The importance of the equivalence classes is that they bound the knowledge of the server $\mathcal{S}$: If $\mathcal{S}$ sees a batch of size b blocks, then it knows that f belongs to class $\mathcal{F}_b$, and this can tell $\mathcal{S}$ some information about the access pattern.

To put it in more formal terms, let F denote the random variable that describes the next file to be accessed. Similarly, say that random variable B outputs the batch size. We then quantify the amount of information flowing from F to B by assuming that a server $\mathcal{S}$ that wishes to guess the requested file F. It is natural then to measure bit leakage by comparing the uncertainty of $\mathcal{S}$ about F before and after observing B:

$$\text{leakage} = \text{initial uncertainty} - \text{remaining uncertainty}.$$

The meaning of "uncertainty" here is the vulnerability of F to being guessed correctly in one shot by $\mathcal{S}$, and is given by $V(F) = \max_{f \in \mathcal{F}} \mathbb{P}_F(f)$. In a similar way, we can define the *a posteriori* vulnerability $V(F \mid B)$ as [12]:

$$V(F \mid B) = \sum_{b \in \mathcal{B}} \mathbb{P}_B(b) V(F \mid b) = \sum_{b \in \mathcal{B}} \max_{f \in \mathcal{F}} \mathbb{P}(b \mid f) \, \mathbb{P}_F(f).$$

Since the mapping between files and batches is deterministic, we have that $\mathcal{F}$ is split into $|\mathcal{B}|$ equivalence classes $\mathcal{F}_b$. Then, $V(F \mid B) = \sum_{b \in \mathcal{B}} \max_{f \in \mathcal{F}_b} \mathbb{P}_F(f)$.

It is trivial to convert from vulnerability to uncertainty by taking the negative logarithm, giving *Rényi min-entropy* [9]. Therefore, $H_\infty(F) = -\log V(F)$ (initial uncertainty), and $H_\infty(F \mid B) = -\log V(F \mid B)$ (remaining uncertainty). The *min-entropy* leakage from F to B, denoted by $\mathcal{L}_{FB}$, can be finally defined as: $\mathcal{L}_{FB} = H_\infty(F) - H_\infty(F \mid B)$, which is

$$\mathcal{L}_{FB} = \log \left(\frac{\sum_{b \in \mathcal{B}} \max_{f \in \mathcal{F}_b} \mathbb{P}_F(f)}{\max_{f \in \mathcal{F}} \mathbb{P}_F(f)} \right). \qquad (2)$$

As can be easily inferred from (2), min-entropy leakage is therefore strongly dependent on the a priori distribution $\mathbb{P}_F$. And actually, when $\mathbb{P}_F$ is the uniform distribution, i.e., each file is accessed with equal probability $1/|\mathcal{F}|$, it is trivial to see that min-entropy leakage is maximal and equal to $\log|\mathcal{B}|$ bits (the notion of *channel capacity* in information theory):

LEMMA 1. *The maximal min-entropy leakage is* $\log|\mathcal{B}|$*, and it is realized by a uniform distribution on* $\mathcal{F}$.

PROOF. Using (2), we have that

$$\mathcal{L}_{FB} \leq \log \left(\frac{|\mathcal{B}| \max_{f \in \mathcal{F}} \mathbb{P}_F(f)}{\max_{f \in \mathcal{F}} \mathbb{P}_F(f)} \right) = \log|\mathcal{B}|.$$

The above upper bound is reached if and only if the following condition holds: For all $b \in \mathcal{B}$, there exists $f^* \in \mathcal{F}_b$ such that $\mathbb{P}_F(f^*) = \max_{f \in \mathcal{F}} \mathbb{P}_F(f)$. This condition is fulfilled when F is uniformly distributed. But it is also valid for non-uniform distributions, as long as every equivalence class $\mathcal{F}_b$ includes the maximum of the distribution. $\square$

From an information theory viewpoint, Lemma 1 shows that a larger set of potential batch sizes $\mathcal{B}$ to observe will result in a greater amount of data leakage. In the next section, we will show to what extent the batch size can leak bits about the private files using a real trace from a cloud storage service.

4. TRADE-OFF ANALYSIS OF THE ORAM INTERFACE

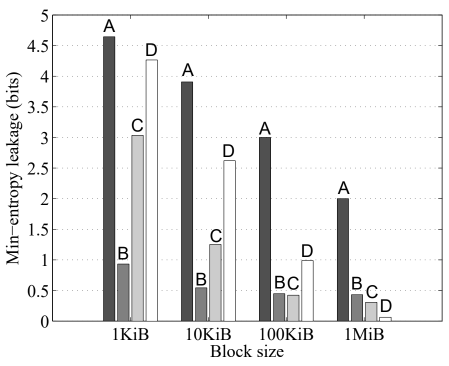
In this section, we analyze how realistic workloads can leak information on the accessed private files, just because the cloud server $\mathcal{S}$ can observe all the block requests corresponding to a file. We also discuss potential strategies to reduce leakage and how it affects performance, unveiling the existence of a trade-off between efficiency and security.

To better inform this argument, we quantify the leakage of several real user access patterns from the UbuntuOne (UB1) personal storage service according to the model developed in the previous section. UB1 was the personal storage service of Canonical Ltd. It was integrated by default in Linux Ubuntu OS, and as a result, it had a user base of over 1 million users until its shutdown on July 2014. The main positive of UB1 is that the observed patterns in this service are representative of those found in many real storage systems. From its recent measurement [6], we selected four typical access patterns on personal files[1]. We named A through D for quick reference throughout this text: Workload A is read-only with a single access per file; workload B exhibits a Zipf-like access pattern; C is read-heavy with a slightly skewed pattern; and D has a balanced read/write ratio with moderate skew. These traces capture the real access pattern for $1,000$ files issued by four distinct users. For the experiments, we will assume a bucket size of $Z = 3$ for Path ORAM.

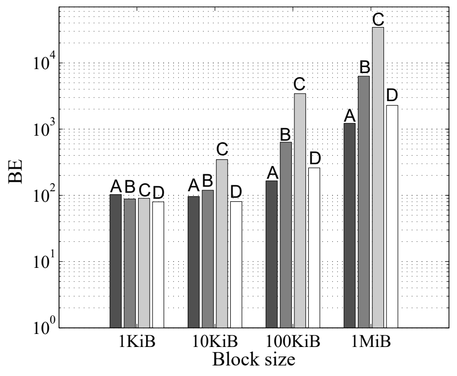
4.1 Leakage vs. Eff ciency

We first study the trade-off between leakage and efficiency caused by the mismatch in the access granularity of ORAM and cloud storage applications for personal storage, network

[1]The traces will be made available after the workshop.



(a)



(b)

Figure 1: Min-entropy leakage (a) and Bandwidth Efficiency (b) as a function of the block size for workloads A to D.

file systems, etc. To do so, we calculate the min-entropy and BE for each workload and depict the results in Fig. 1. Since the block size D is a sensitive parameter, we compute both measures for four different values of D, ranging from 1 KiB to 1 MiB, each representing an order-of-magnitude increase in the block length. These values are good representatives of existing blocks sizes in modern storage systems, from the few kilobytes of networked file systems like NFS and CephFS up to personal cloud storage systems with chunks of a few MBs.

The first observation to be made is that by increasing the block size, information leakage reduces as shown in Fig. 1(a). This figure succinctly demonstrates that an increment by an order of magnitude in the block size does not always result in an important reduction on the amount of information leaked (see Patterns A and D, for instance). For small block sizes, the leakage can be significant. For $D = 1$ KiB, the leakage in workload D is of $\mathcal{L}_{FB} = 4.265$ bits, which is large given that the number of distinct files is only 1K (≈ 10 bits). While one can allege that min-entropy leakage is a rather crude metric, it is unclear whether a personal storage service may leverage additional information on the protected files. And further, it has been shown in the literature that it is possible to identify files via side channels like deduplication [7], for we believe it is certainly an appropriate measure.

For completeness, we detail the computation of $\mathcal{L}_{FB}$ when the block size is $D = 1$ KiB in Table 1, as it is the case that exhibits the best efficiency. After observing the batch size B, the remaining uncertainty on the accessed user file becomes between 35% to 66% of the initial uncertainty $H_\infty(F)$. And

Pattern	$\lvert\mathcal{B}\rvert$	$H_\infty(F)$	$H_\infty(F \mid B)$	$\mathcal{L}_{FB}$
A	25	9.966	5.322	4.644
B	92	2.687	1.784	0.903
C	104	6.695	3.660	3.035
D	207	6.592	2.327	4.265

Table 1: ORAM leakage of various access patterns at the file level. Block size is assumed to be $D = 1$KiB.

Idle time	Percentiles				
	25%	50%	75%	90%	99%
Pareto(1.38, 9.62)	11.85	15.89	26.24	50.92	269.4
Pareto(1.13, 9.17)	11.83	16.94	31.26	70.30	538.4
Pareto(0.79, 6.78)	9.75	16.30	39.20	125.1	2306.4

Table 2: Number of dummy accesses d during inactivity periods in Dropbox.

consequently, the knowledge of B increases largely the one-guess vulnerability of F in all the workloads.

An interesting observation is that in practice, it is possible to encounter users that exhibit uniform access patterns, e.g., by copying or moving an entire folder to a new place, thereby leaking a large volume of information as shown by Theorem 1 and empirically exemplified by Pattern A. Concretely, in this case the bit leakage is the highest one, of $\log|\mathcal{B}| = 4.644$ bits, despite that Pattern A presents the smallest set of potential batch sizes $\mathcal{B}$ (equivalence classes).

Another important question is whether the increase in the block size translates into a huge bandwidth waste. To better understand this, Fig. 1(b) depicts BE against the block size. As shown in this figure, BE can become three, or even four, orders magnitude higher than the amount of useful content. This suggests that increasing the block size to reduce leakage does not pay off due to the poor bandwidth usage.

In conclusion, although several recent works have studied how to make ORAM practical in an outsourced cloud storage scenarios [14, 16, 8, 13, 17], arguing that a primary hurdle for cloud ORAM deployment is its high bandwidth overhead, we view that the block-oriented interface of ORAM can be also problematic for cloud applications in terms of leakage, which is "paradoxical", given that the interest in ORAM has been reinvigorated for its strong security guarantees on protecting access patterns.

4.2 Zero Leakage → High Overheads

As we discuss in this section, completely preventing ORAM leakage comes at high overheads in practice. We investigate two simple solutions to this problem, and demonstrate that achieving zero leakage incurs high overhead.

Solution 1 [**Maximizing block size**]. One naive strategy could be to restrict the size of all batches to 1 by choosing as the block size the size of the largest file. With this approach, all accesses will yield exactly 1 output (i.e., batch size), thus leaking no knowledge over this channel. This is trivial to see using information theory:

LEMMA 2. *The smallest possible amount of min-entropy leakage is met if the set of possible batch sizes $\mathcal{B}$ has size* 1.

PROOF. Here we have $|\mathcal{B}| = 1$, so from (2),

$$\mathcal{L}_{FB} = \log\left(\frac{\sum_{b\in\mathcal{B}} \max_{f\in\mathcal{F}_b} \mathbb{P}_F(f)}{\max_{f\in\mathcal{F}} \mathbb{P}_F(f)}\right)$$
$$= \log\left(\frac{\max_{f\in\mathcal{F}} \mathbb{P}_F(f)}{\max_{f\in\mathcal{F}} \mathbb{P}_F(f)}\right) = \log 1 = 0.$$

The second equality follows from the fact that if there is only a single equivalence class, it had to include the maximum of the a priori distribution $\mathbb{P}_F$. □

This strategy, though, is only useful when the variation in size across files is small. When the variation is large, the fact of padding all files to the largest size can result in huge delay to read and write small files and significant bandwidth waste due to internal fragmentation. This situation is aggravated by the fact that Path ORAM needs to read an entire path to access the target (potentially padded) block of D bits, which may raise the bandwidth overhead by more than a $\mathcal{O}(\log N)$-factor.

Unfortunately, personal storage systems such as Dropbox exhibit high variability in file size, being well fitted by heavy-tailed distributions [5]. Similar behavior has been observed in UB1 [6], and although 90% of files are smaller than 1MiB, some of them are very large in size, making it impractical to predict the size of the largest file in advance, and thus, of the block size, as it is chosen offline, i.e., before the user puts any file into his synchronization folder. This is clearly visible in Fig. 1(b) when the block size $D = 1$ MiB. The BE is between three to four orders of magnitude higher than what would be expected by just fetching the small files of < 1 MiB in their entirety as Dropbox, UB1, etc., do. In Dropbox, all the files of less than 4MiBs are downloaded as a single chunk.

Solution 2 [**Periodic ORAM access**]. A second approach could be to force ORAM to be accessed at a single, periodic rate as proposed in [3] to protect against timing attacks. By accessing the ORAM at a periodic rate, it is possible to fully obfuscate the actual number of file blocks, because the server cannot tell *when* a request for a file starts and terminates. If no file access is triggered when the next periodic access to the ORAM must be made, an indistinguishable "*dummy*"[2] access is issued instead, thereby preserving obliviousness at the file access level.

While this approach leaks no knowledge about the access pattern, the main problem with it is its high overhead. If the offline-selected, periodic rate is too small, it can take a long time to access a file —hurting performance. On the contrary, if the chosen rate is too high, huge bandwidth can be wasted due to dummy accesses. The latter is particularly important, since network-level efficiency is "pivotal" for cloud storage. It suffices to think of the rich variety of data reduction methods put in place like compression, deduplication, etc., to realize the importance of efficient traffic usage in the cloud.

To give a sense of this, consider a personal storage service like Dropbox or UB1. In these systems, traffic is bursty, with long periods of inactivity. In Dropbox, for instance, the time between two consecutive transfers is distributed according to a Pareto distribution [5] with CDF $1-\left(\frac{k}{x}\right)^\alpha$ for $x \geq k$, where $\alpha > 0$ is the shape parameter that determines the thickness of the tail. Under this distribution, most clients exhibit short inactivity periods while a handful of them stay idle for much longer time. Assuming a periodic rate r, and a desired target probability ϵ, it is easy to derive an upper bound on the

[2]A dummy access is an access made to a random leaf in the tree. By ORAM's security definition, this access appears to be indistinguishable from a real access.

total number of dummy accesses d performed by $100\epsilon\%$ of clients. Concretely, it can be easily seen that $d \leq r\frac{k}{(1-\epsilon)^{\frac{1}{\alpha}}}$. Now taking the three typical configurations reported in [5], Table 2 reports the number of dummy accesses d performed during the idle periods when no request is needed for $r = 1$ sec. As shown in the table, the number of dummy accesses d can be very high even for that small rate — the client must wait 1 sec. before fetching the next block in the file. At 99th percentile, for the third configurations, the ORAM software might need to run more than 2K dummy accesses to leak no information on the protected files. This will result in a large bandwidth wastage in addition to a high latency file access. That is, accessing a file split into s blocks will take s seconds in this scenario, thereby yielding a poor trade-off. As before, this solution is not practical for cloud storage —as it is in the domain of secure processors [3, 1].

4.3 What to do?

Given that full protection is prohibitively expensive, a wise path to follow should be, in lieu of blocking leakage entirely, limiting it to a small, controllable constant — as it has been done in the literature (see [3], and the references thereof).

From a theoretic standpoint, this can be easily performed by virtue of Lemma 1, which bounds the maximal leakage to $\log |\mathcal{B}|$. That is, given a $L-$bit leakage bound, it apparently seems that it suffices to restrict the number of allowed batch sizes to $|\mathcal{B}| \in \mathcal{O}(2^L)$ in order to let the leakage bound to hold.

In practice, however, the leakage due to the timing channel created by the interface mismatch is not so simple to control without increasing the bandwidth costs. The main reason is the large variability in file sizes, which makes it very difficult to determine the set $\mathcal{B}$. To better understand this, consider that a file is split into b blocks. A simple strategy to access that file would be: First, to take the smallest batch size b^* in $\mathcal{B}$ that exceeds b, and then dispatch b^* requests to the server, adding $b^* - b$ dummy accesses if necessary to obfuscate the real number of blocks to the server.

Although simple, the above approach could become very inefficient if the allowed batch sizes do not represent well the current access pattern. In other words, if the difference $b^* - b$ was usually large due to a bad composition of $\mathcal{B}$, the presence of dummy accesses would be the norm, hugely increasing the bandwidth costs. In practice, this behavior is expected due to the impossibility to adjust to the different file sizes during different phases. This example evinces that this problem is not trivial to solve, which constitutes a promising avenue of further research.

5. CONCLUSIONS

In this research, we have shown that the standard block-based ORAM interface can create a timing side-channel when it is utilized as a substrate to build higher level abstractions such as filesystems, object storage, personal storage services, etc., of frequent use in cloud computing. We have elucidated the existence of a trade-off between information leakage due to this channel and performance, and shown the limitations of the simple solutions to this problem, opening a new vein of research.

Acknowledgements

This work was partially supported by European Commission via the H2020 IOStack project (ICT-07-2014), and the *Cloud Services and Community Clouds* Spanish project (TIN2013-47245-C2-2-R).

6. REFERENCES

[1] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *CCS'10*, pages 297–307, 2010.

[2] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang. Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward. In *CCS '15*, pages 837–849, 2015.

[3] C. Fletcher, L. Ren, X. Yu, M. van Dijk, O. Khan, and S. Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *HPCA'14*, pages 213–224, 2014.

[4] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, 1996.

[5] G. Goncalves, I. Drago, A. Couto da Silva, A. Borges Vieira, and J. Almeida. Modeling the dropbox client behavior. In *ICC'14*, pages 1332–1337, 2014.

[6] R. Gracia-Tinedo, Y. Tian, J. Sampé, H. Harkous, J. Lenton, P. García-López, M. Sánchez-Artigas, and M. Vukolic. Dissecting ubuntuone: Autopsy of a global-scale personal cloud back-end. In *IMC'15*, 2015.

[7] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.

[8] J. R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *FAST'13*, pages 199–213, 2013.

[9] A. Rényi. On measures of entropy and information. In *4th Berkeley Symposium on Mathematics, Statistics and Probability*, pages 547–561, 1960.

[10] C. Sahin, V. Zakhary, A. El Abbadi, H. R. Lin, and S. Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *S&P'16*, pages 198–217, 2016.

[11] M. Sanchez-Artigas. Toward efficient data access privacy in the cloud. *IEEE Communications Magazine*, 51(11):39–45, 2013.

[12] G. Smith. On the foundations of quantitative information flow. In *FOSSACS'09*, pages 288–302, 2009.

[13] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE S&P*, pages 253–267, 2013.

[14] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious ram. In *NDSS'12*, 2012.

[15] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path o-ram: An extremely simple oblivious ram protocol. In *CCS*, pages 299–310, 2013.

[16] P. Williams, R. Sion, and A. Tomescu. Privatefs: A parallel oblivious file system. In *CCS'12*, pages 977–988, 2012.

[17] N. Wolfe, E. Zou, L. Ren, and X. Yu. Optimizing path oram for cloud storage applications. *arXiv preprint arXiv:1501.01721*, 2015.