MemCloak: Practical Access Obfuscation for Untrusted Memory

Weixin Liang
Zhejiang University

Kai Bu*
Zhejiang University

Ke Li
Zhejiang University

Jinhong Li
Zhejiang University

Arya Tavakoli
Simon Fraser University

ABSTRACT

Access patterns over untrusted memory have long been exploited to infer sensitive information like program types or even secret keys. Most existing obfuscation solutions hide real memory accesses among a sufficiently large number of dummy memory accesses. Such solutions lead to a heavy communication overhead and more often apply to the client/server scenario instead of the CPU/memory architecture. Sporadic obfuscation solutions strive for an affordable memory bandwidth cost at the expense of security degradation. For example, they may have to obfuscate accesses over a limited range of memory space to control the overhead.

In this paper, we present MemCloak to obfuscate accesses throughout the entire memory space with an $O(1)$ communication overhead. We advocate leveraging data redundancy to achieve extremely efficient obfuscation. Loading multiple duplicates of a data block in memory, MemCloak enables the CPU to fetch the same data by accessing different memory locations. This breaks the condition for snooping the access pattern. Moreover, we leverage data aggregation to improve memory utilization. It enables the CPU to fetch the same aggregated data block times from the same memory location but each time for a different data block therein. This further prohibits an attacker from correlating memory accesses. We propose a series of optimization techniques to compress the position that tracks memory layout. The optimized position map is hundreds of times smaller than the traditional position map. It takes only several megabytes for protecting a 4 GB memory and can fit in an on-chip cache or buffer. We implement MemCloak using the gem5 simulator and validate its performance using highly memory-intensive MiBench benchmarks.

CCS CONCEPTS

• **Security and privacy → Hardware security implementation**; **Hardware-based security protocols**;

KEYWORDS

Access pattern obfuscation, Oblivious RAM, side-channel attack

*Corresponding Author: kaibu@zju.edu.cn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACSAC '18, December 3–7, 2018, San Juan, PR, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274695

ACM Reference Format:
Weixin Liang, Kai Bu, Ke Li, Jinhong Li, and Arya Tavakoli. 2018. MemCloak: Practical Access Obfuscation for Untrusted Memory. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3274694.3274695

1 INTRODUCTION

The pattern of memory accesses has long been exploited for side-channel attacks. Specifically, the pattern refers to the sequence of addresses accessed during program execution [18]. It can be used to construct the control flow graph (CFG) of a program [3]. An extensive measurement study demonstrates the uniqueness of CFGs across different programs [46]. For example, among 1,334 procedures of Alpha compiler's Standard C library, only 0.05% of all possible pairs of their CFG match for procedures with 15 or more blocks. An attacker can thus monitor a program's memory access pattern, construct the corresponding CFG, and then identify exactly which program is running. If a CFG reveals a cryptographic function in use, the attacker may even compromise the secret key [25, 46]. For example, Diffie-Hellman [12] and RSA [28] involves a loop of conditional branches using the value of each bit of the secret key for condition check [46]. A bit one directs the program execution to an IF-branch code segment while a bit zero to an ELSE-branch code segment. Using the snooped CFG, the attacker can easily infer two possible secret keys—one is the secret key per se and the other is its complement.

However, how to obfuscate accesses for untrusted memory in a practically efficient way remains unsolved. Different from emerging trusted memory capable of cryptographic computation [2, 5, 31], widely-deployed conventional untrusted memory[1] cannot perform computation and the addresses sent to memory have to remain in plain text. A fundamental obfuscation technique called Oblivious RAM (ORAM) [18] therefore hides a real access among many dummy accesses. Apparently, ORAM imposes a high communication overhead on memory bandwidth and gains few practical implementations in the CPU/memory scenario. With the recent blossom of cloud computing where network bandwidth might be of less concern, ORAM gradually applies more to obfuscating data-access patterns on a remote server and attracts many improvements in the client/server scenario [8, 10, 11, 13, 15, 18, 26, 27, 34, 37, 38, 42, 43]. Sporadic attempts at the CPU/memory scenario trade security for efficiency [24, 40, 45, 46]. For example, HIDE [46] needs to fetch all blocks in the same chunk (i.e., one or more continuous pages) of a previously fetched and cached block. Then it permutes block locations within the chunk, records the new block-address mapping

[1] For ease of presentation, we hereafter use the terms of untrusted memory and memory interchangeably whenever no confusion arises.

in the position map, and writes all blocks back to memory. HIDE has to limit the overhead by confining the obfuscation to only a small range of the memory space.

In this paper, we take the challenge and present the design and implementation of MemCloak. It obfuscates accesses for untrusted memory with an $O(1)$ communication overhead and an on-chip cacheable/bufferable position map. We achieve such a minimum communication overhead by leveraging data redundancy. The root cause for leaking the access pattern is that currently the CPU always accesses the same memory location for fetching the same data block therein. If we can enable the CPU to access different locations for the same block, access pattern leakage is avoided. To this end, the data redundancy technique we propose loads multiple differently-encrypted copies of each block in memory. We leverage data aggregation to improve memory utilization. The data aggregation technique XORs multiple data blocks into one. When all but one of the blocks inside the aggregate are already fetched and cached, the CPU can fetch the aggregate and extract the remaining block therein by lightweight XOR computation. With both data redundancy and data aggregation techniques, the CPU can fetch the same block by accessing different memory locations and access the same memory location for fetching different blocks. This prohibits an attacker from correlating memory accesses and thus protects the access pattern.

We further propose a series of optimization techniques to compress the size of the position map over hundreds of times. We use computation-based address mapping to remove the destination addresses from the position map. We also remove keys (for data encryption/decryption) from the position map by deriving them from the addresses on the fly. Given that one key reason for the possibly giant map size is that it dedicates an entry for each block, we restructure the position map in a page-level fashion and each page-oriented entry maintains much less information for blocks in that page. Another challenge arises when we replenish new data duplicates by piggybacking them in dummy writes. Each read access should be associated with a dummy write and vice versa to protect the access type (i.e., read or write) [5]. If we simply find a feasible computation function that can map a new duplicate to an empty location, we have to deal with frequent address collisions and cumbersome map update. We address this challenge by using another address as an address's mapping alias. This way, we can randomly map an address across the entire memory space without heavy re-computation. Based on the preceding optimization techniques, MemCloak can compress a 1 GB traditional position map to a several-megabyte one, practically fittable in an on-chip cache.

In summary, we make the following contributions to obfuscating accesses over untrusted memory.

- Obfuscate memory accesses with a minimum $O(1)$ communication overhead (Section 3). The proposed techniques of data redundancy and data aggregation break access correlation by enabling the CPU to access the same data from different locations and to access the same location for different data.
- Compress the traditional position map over hundreds of times to fit in an on-chip cache or buffer (Section 4).
- Implement MemCloak using gem5 [7], a widely used simulator for computer architecture research (Section 5). Our modification accounts for 2,000+ lines of C/C++ code over gem5's 250K lines. We execute MemCloak over three highly memory-intensive benchmarks—dijkstra, susan, and jpeg encode—from the MiBench benchmark suite [19] (Section 6). The results demonstrate that MemCloak can nearly randomize access patterns with a minimized overhead.

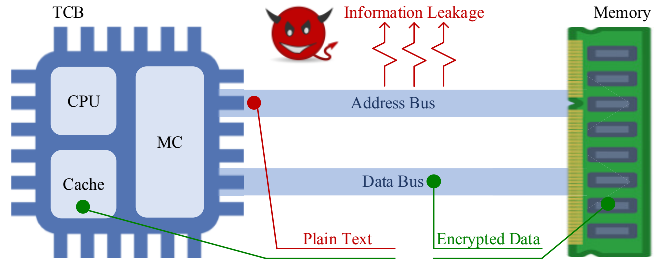

Figure 1: Side-channel attack over memory access patterns [36]. The CPU, cache, and memory controller (MC) reside in a trusted computing base (TCB). A passive attacker snoops both the plain-text address bus and the encrypted data bus to infer the pattern of memory accesses.

2 PROBLEM

In this section, we raise a question about the practicality of how to efficiently obfuscate access patterns over untrusted memories. We first review known side-channel attacks over memory accesses. Existing countermeasures introduce many more dummy accesses to hide a real access, leading to a high overhead.

2.1 Side-Channel Attack over Memory Accesses

The pattern of memory accesses (i.e., the sequence of addresses) has long been exploited for side-channel attacks. As shown in Figure 1, while CPU and memory chips are secure, both the address bus (for the CPU to transmit a memory address to the memory) and the data bus (for data transmission between the CPU and memory) are vulnerable to eavesdropping attacks. Shielding the data bus from eavesdropping attacks simply requires data encryption [4, 9, 17, 22, 32, 35, 39, 41]. Such a cryptographic protection, however, fails to harden the address bus. This is because traditional memory chips do not support cryptographic computation. Addresses sent to memory have to remain in plain text [31]. An attacker can easily discover memory access patterns and exploit them for side-channel attacks. Such attacks may reveal the reused code of a running program [46] or even secret keys of AES encryption [25].

The first example attack uses memory access patterns to infer what program is running. This is achieved by control-flow graph (CFG) matching [46]. A CFG graphically represents all paths that might be traversed through a program during its execution [3]. Zhuang *et al.* conducted an extensive measurement and demonstrated that most programs have a unique CFG [46]. Take, for example, the Standard C library of the Alpha compiler including 1,334 procedures, each with at least 5 blocks. When comparing all possible pairs of CFGs generated for all these procedures, only 5% of the comparisons match. The more blocks a procedure has, the less likely its CFG matches with that of other procedures. For example, only 0.1% of the comparisons match for procedures with 10 or more blocks. The number drops to 0.05% if the comparisons focus on

only procedures with at least 15 blocks. Leveraging this property of CFG uniqueness, an attacker can passively monitor the address bus, detect jumps upon accesses to discontinuous locations, and construct the CFG. Once the constructed CFG matches with that of a procedure in standard libraries, the attacker identifies the running program with a high probability.

Even worse, memory access patterns may leak critical data such as secret keys [25, 46]. Zhuang *et al.* [46] use the private-key operations of Diffie-Hellman [12] and RSA [28] for example. Such operations involve a loop of conditional branches using the value of each bit of the secret key for condition check. A bit one directs the program execution to an IF-branch code segment while a bit zero to an ELSE-branch code segment. Given that an attacker can identify such operations through CFG matching [46], it can further infer from the observed CFG that which bits of the secret key are identical. Although the attacker cannot distinguish between the IF-branch and ELSE-branch code segments, it can determine two values using the observed CFG—one is the secret key per se and the other is its complement. A little more efforts like exercising these two values over snooped encrypted data will help the attacker nail down the exact secret key.

2.2 Burdensome Obfuscation of Access Pattern

Obfuscation solutions for untrusted memory usually base themselves on ORAM technique. ORAM associates with a high overhead as it protects the access pattern through hiding a real access among a sufficiently large number of dummy accesses [18]. Following ORAM, the CPU sends to memory a number of addresses including the one of the real interest. Memory then responds with data blocks stored on the received addresses. Note that each data block should be encrypted before being loaded into memory. Otherwise, data correlation leaks memory access patterns. After the CPU receives these encrypted data blocks, it first decrypts the interested one and then operates on it. To invalidate the correlation of two accesses to the same location, the CPU shuffles the addresses of received data blocks, re-encrypts each of them, and writes the re-encrypted data back to the corresponding locations. Intuitively, fetching all n data blocks in the entire memory upon each memory access promises the most secure obfuscation. However, this incurs a high communication overhead of $O(n)$.

The major design technique toward efficient ORAM is modeling memory layout with a certain structure for ease of hiding where fetched data blocks are written back. For example, ORAM uses a hierarchical structure to achieve an $O(\text{polylog } n)$ communication overhead [18] while Path ORAM uses a tree structure to achieve an $O(\log^3 n)$ overhead [34]. The state-of-the-art Floram uses dual memory chips to achieve an $O(\log n)$ overhead [13]. We refer interested readers to Floram [13] for a comprehensive review of ORAM evolvement [10, 11, 15, 18, 26, 27, 34, 37, 38, 42, 43] and to SEAL-ORAM [8] for an experimental evaluation of typical ORAM solutions for remote access of cloud data.

No matter how efficient an ORAM-based solution can be, it has to impose a sufficiently large number of dummy accesses to hide the access pattern. This is why ORAM solutions have been considered more practical to a server-client environment such as remote access of cloud data [8, 36] instead of the CPU-memory communication as in Figure 1. It is also worth mentioning that, Path ORAM [34], as the base of many ORAM solutions, needs to store many dummy blocks in memory to mitigate system deadlocks when reshuffling cannot proceed because all buckets along a tree path are full. Even if wasting 50% of memory capacity for storing dummy blocks (i.e., 100% memory overhead), system deadlocks can still occur.

2.3 Toward Practically Efficient Obfuscation

In this paper, we take on the challenge of obfuscating memory access patterns with an $O(1)$ communication overhead. This low overhead is essential for protecting CPU-memory communication in a practically efficient way. Ideally, we expect that an extremely efficient obfuscation solution require zero additional memory access. That is, for each read request, the solution still simply lets the CPU send the read address to memory and then let memory send back the corresponding data. Similarly, for each write request, the CPU sends both the write address and the corresponding data to memory. A memory access operation thus requires two communication messages, one on the address bus and the other on the data bus. Without further protection, however, a memory request can easily leak the type (i.e., read or write) that might be leveraged for improving attacking probability. Therefore, most previous obfuscation solutions for either untrusted memory [34] or trusted memory [5] use a different-type dummy request to hide the type information. Specifically, each memory request should be at least transformed to a pair of read-then-write requests [5]. If the original request is a read, it is followed by a dummy write. Otherwise, it is preceded by a dummy read. Taking type protection into account, an obfuscation solution thus introduces a communication overhead lower bounded by $O(1)$. The minimum would be only one additional dummy request with two communication messages.

Although the above minimum is achievable on trusted memory because of the support of encrypted addresses [5], it is never practically achieved on untrusted memory [13]. Sporadic initial attempts strive for this goal via mounting an additional hardware buffer to the CPU chip [24, 40, 45, 46]. We observe that they are associated with high communication overhead and even security degradation. For example, HIDE [46] buffers all blocks fetched from memory. Whenever a block need be evicted from the buffer or written back to memory, HIDE needs to 1) fetch all blocks in the same chunk (usually containing one or more continuous pages) with the block to the CPU chip, 2) permute all blocks to different locations within this chunk, 3) record the new block-address mapping, and 4) write all permuted blocks back to memory. Otherwise, if, as usual, a block is read into the buffer and then evicted, the subsequent use of the block will access the same memory location and leak the access pattern. HIDE efficiency can be improved by mixing the permutation with the read accesses [45]. The idea is that after some read access, a buffered block can be written back to the location of the just read block. This way, block permutation is achieved without much expensive transmission of blocks in a large chunk. Security guarantee of this idea, however, requires a sufficiently large buffer size. Consider an extreme case with a buffer of size one and a recursive accesses of two memory locations l_1 and l_2. The access sequence would be $(l_1, l_2, l_2, l_2 ...)$. This enables an attacker to infer two possible real access patterns—$(l_1, l_2, l_2, l_2 ...)$ and $(l_1, l_2, l_1, l_2 ...)$—with the latter one exactly matching the real access pattern.

3 OVERVIEW

In this section, we present MemCloak, the first practical obfuscation solution for protecting access patterns over untrusted memory with an $O(1)$ communication overhead. The major idea of MemCloak is leveraging data redundancy to reinvigorate dummy accesses and dummy blocks, which otherwise are the main source of bandwidth waste and memory waste, respectively.

3.1 Motivation

Essentially, if we could fetch the same data from different locations and the data are differently-encrypted on each location, the access pattern would not be leaked. This instantly motivates us to leverage data redundancy. Current memory is usually used to accommodate one copy per data block therein. Repetitive accesses on the same location for the same data easily leak the access pattern. It is also easily correlated if we simply write the fetched data back to a different location and read it from that location afterwards. Therefore, previous obfuscation solutions have to introduce dummy accesses to hide a real access. This leads to an intrinsically high overhead on both the address and data buses. They may also further introduce dummy blocks (e.g., across 50% of the memory space) to mitigate system deadlocks [13]. In contrast, we use the space otherwise taken by dummy blocks more wisely. We reinvigorate dummy blocks by filling them with duplicate data. This little twist yields a significant decrease in communication overhead. Using no dummy requests, we can simply fetch the same data from two or more different locations without revealing the access pattern.

Let us use an example to quantify how data redundancy promises a significant efficiency improvement over previous obfuscation solutions. For ease of understanding, we in this example consider repetitive read accesses to only one data block among n memory locations. We temporarily do not consider protecting the access type (i.e., read or write) either. A more comprehensive MemCloak design involving write accesses and type protection will be presented shortly. Since only one data block is concerned, we can create n differently-encrypted copies of it and load them into memory. Making the example more challenging, we assume that there is no cache to buffer fetched data. Then every time the CPU needs to operate on the data, it has to access memory. Given a differently-encrypted copy at each memory location, the CPU can read the data block up to n times without repetitive accesses to the same location. This way, MemCloak protects the access pattern with *zero* communication overhead. In contrast, existing obfuscation solutions might store only one copy of the data block and use the other $n-1$ locations for storing dummy blocks. For each memory access, the state-of-the-art solution imposes a communication overhead of $O(\log n)$ dummy accesses [13]. Furthermore, the CPU needs to re-encrypt some or all of the fetched data blocks and write them back to different memory locations than where they are fetched. The gap between the overhead of MemCloak and that of the state-of-the-art solutions demonstrates a promising leap in efficiency.

3.2 Challenge

However, we cannot efficiently implement MemCloak without addressing a series of challenges. Key challenges include how to improve memory utilization while duplicating data, how to compress the position map while providing one-to-many mappings, and how to replenish duplicate data while serving continuous accesses.

Memory utilization. Intuitively, the more duplicates we associate with a data block, the more consecutive accesses to different duplicates we can use to get the data block without accessing any location more than once. More duplicates per data block, however, waste more memory space. One may consider this memory overhead as a necessary tradeoff for securing access patterns, especially given the extremely efficient $O(1)$ communication it promises. But to encourage the deployment of MemCloak, we should make every effort to improve memory utilization.

Position-map compression. Obfuscation solutions require a position map to track data placement in memory. For each entry in the position map, two key fields should specify what metadata to identify a data block and which data blocks to fetch alongside for access obfuscation. Take the tree-based ORAM for example [34]. These two fields are a block index to identify the requested data block and a path index to specify on which tree-path all data blocks therein should be fetched, respectively. While to our CPU-memory scenario as in Figure 1, a position map should maintain address mappings, each linking one virtual address to multiple physical addresses. To construct such a position map, one may suggest simply extending the page table where originally contains only one-to-one mappings. Consider an entry mapping virtual address l_v^0 to physical address l_p^0 for example. Assume that MemCloak loads $m-1$ more copies of the data currently located at l_p^0 to locations l_p^i $(1 \le i \le m-1)$. Then we can accordingly add $m-1$ more entries to the page table. Each added entry maps l_v^0 to a different l_p^i. Once after an entry has been used as a reference for memory access, it should be invalidated or deleted to avoid repetitive references and access pattern leakage. This leads to frequent modification of the page table. Furthermore, invalidating or deleting the referenced entry means that we cannot buffer it in the TLB and its corresponding data block in the cache. All the above limitations suggest that MemCloak may not simply extend the page table as the position map.

Creating an independent position map faces challenges as well. First, it demands space. Given that each entry in it maps to a different physical address, the required space can be upper bounded by the number n of memory locations. The position map thus can be as large as the page table. Second, it should not affect the efficiency provided by the TLB and caches. This requires that the position map be a transparent layer between the last-level cache and memory.

Data replenishment. Given a number d of duplicate blocks loaded in memory, we can fetch this block up to d times without repetitive access on the same location. We need to replenish more duplicates to serve the $(d+1)$th access and those afterwards. Otherwise, repetitive accesses over some memory locations will appear and leak the access pattern. A possible replenishment strategy follows HIDE [46]. That is, we fetch a number of blocks from memory to the CPU chip, permute/shuffle their locations, and then write them back to the newly assigned locations. This strategy costs CPU time and induces communication overhead. Then one may suggest periodically loading more duplicate blocks from the disk via I/O. However, both strategies need to halt memory accesses while replenishing blocks. We expect an efficient replenishment alongside accessing memory as in [24, 40, 45, 46] without its dependence on a large buffer.

3.3 Methodology

MemCloak addresses the preceding challenges by three lightweight techniques presented in this paper. First, we use XOR to aggregate data blocks to improve memory utilization. Second, we use computation based address mapping to eliminate destination addresses in the position map. Third, we reinvigorate dummy writes required for protecting the access type to piggyback data replenishment.

Memory utilization: XOR in memory. We use XOR-based data aggregation to save memory space without reducing duplicates. Consider a motivating example that needs to load two duplicates for both block a and block b. Originally, this invokes four memory allocations, two for block a and the other two for block b. Using XOR, we use only three blocks—a, b, and $a \oplus b$—to represent the information of four blocks. After a is fetched and cached, the CPU can access either b or $a \oplus b$ for fetching b. Such a data aggregation actually promises more than memory efficiency. It enables the CPU to fetch the same data from different locations and fetch different data from the same location. This significantly discourages an attacker from inferring the access pattern.

Position-map compression: Map addresses on the fly. We map a requested address to one of the duplicates on the fly. The position map no longer stores all mappings from one address to multiple duplicates. Instead, it stores source addresses and the functions that are used to compute the destination addresses for duplicates. Since functions can be shared by all source addresses, each entry in the position map uses only function indices. We can thus avoid repeating source addresses and remove destination addresses toward a highly compact position map.

Data replenishment: Piggyback new duplicates in dummy writes. Inspired by [45], we replenish new duplicates and reshuffle address mappings using dummy writes that otherwise contain only dummy data for protecting the access type. Specifically, we fill a dummy write with a duplicate. Where in memory to put the dummy write imposes another challenge. We cannot simply write the new duplicate back to the location that has just been read as in [45]. That location may store an XORed block, which still can serve subsequent accesses for different data requests. We cannot arbitrarily compute a new location. First, this demands a sufficiently large number of functions to make sure that an address can be mapped to many addresses. Second, the newly computed location might be currently occupied by another data block. We cannot simply select an empty location either. This introduces destination addresses back to the position map and refrains the efficiency improvement by computation-based address mapping. We propose using another address as an address's *mapping alias*. Then an address can use its mapping alias's destination addresses. By careful control of alias selection, we randomly map each address across the entire memory space without cumbersome management of the position map.

4 DESIGN

In this section, we detail the MemCloak design. MemCloak logics only reside in the memory controller, serving as a transparent layer between the last-level cache and memory, without modification to the operating system, programs, page table, and TLB. The key challenge is how to optimize the size of the position map while securing access obfuscation. We propose a series of optimization techniques to compress the traditional position map by hundreds of times. The result position map takes only several megabytes and can be practically fitted into a cache or buffer.

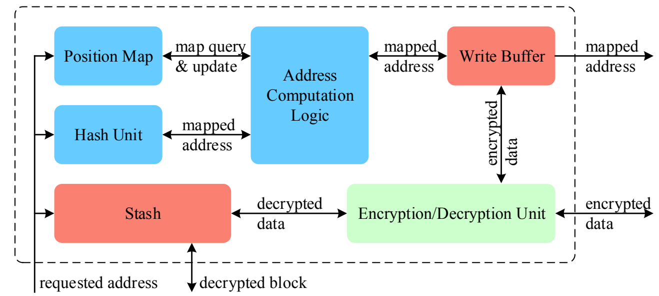

Figure 2: MemCloak architecture in the memory controller.

4.1 Architecture

As with existing memory access obfuscation solutions, we implement MemCloak logics in the memory controller as a transparent layer between the last-level cache and memory. The CPU need not beware of the address manipulation by MemCloak. It simply follows the original memory access scheme that sends out a memory request to the first-level cache. The memory request fetches the requested data block to registers upon cache hits and is cascaded to lower-level caches upon cache misses. If the memory request still encounters a cache miss on the last-level cache, it is directed to the memory controller, where MemCloak obfuscation is enforced before re-directing it to memory. (Note that the initial memory request uses a virtual address, which should be translated to a physical address on a certain level of cache before entering the memory controller.) A key component for MemCloak obfuscation is the position map. It maps the requested physical address to more than one physical addresses holding different encrypted copies of the same data block. Once a data block is fetched from memory, the memory controller decrypts it and feeds the plain-text data block back to the CPU. If the data block will be cached, the memory controller also needs to update its address field to the original requested physical address. Otherwise, subsequent requests to that physical address will encounter cache misses even though the requested data block is cached. Moreover, the address update further avoids modification over the page table.

We present the architecture of the memory controller by MemCloak in Figure 2. When the computer system loads data blocks from the disk to memory via I/O operations, it initializes each data block with multiple different encrypted copies. Some of these copies may be XORed into aggregated blocks. We can adopt existing techniques such as prefetching for compensating memory access delay. The position map tracks all the key information for extracting the original data, such as where it locates in memory, how it is encrypted, and which other data it might be XORed with. Besides the position map, an obfuscation solution needs also a *stash* buffer to temporarily store some recently fetched data blocks. If the requested data block can be found in the stash, it is directly transmitted to the CPU. Otherwise, memory access will take place by first looking up the position map for determining which physical address to access. This process may involve some lightweight computation using the address computation unit and hash units. To

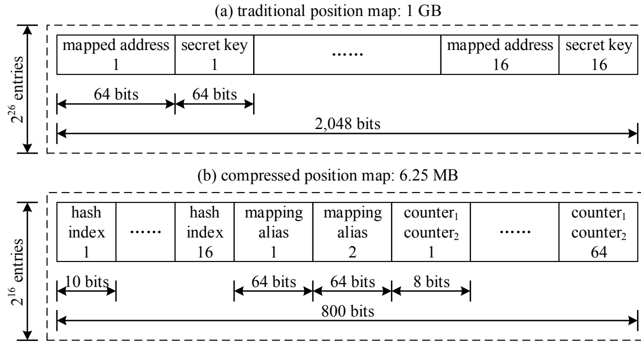

Figure 3: Comparison of (a) the traditional position map and (b) the MemCloak-compressed position map under example settings of 64-bit physical addresses, 64-bit secret keys)[2], 64-byte data blocks, 4 KB pages, and 4 GB memory. MemCloak employs $h = 1,024$ hash functions for address computation and assign $c = 16$ copies per block. We provide the structure of only one map entry for simplicity.

guarantee data consistency, we search the write buffer for data with the calculated physical address before accessing memory. Once the requested data block is read from the write buffer or memory, it is decrypted by the encryption/decryption unit. Then the plain-text data block is transmitted to the stash as well as the CPU. Since accessing the same memory location for fetching the same data leaks the access pattern, we need to reshuffle data placement to avoid such repetitive accesses. This is done by replenishing newly encrypted copies to carefully selected memory locations. The position map should also be accordingly updated such that subsequent requests can find the correct data.

4.2 Position Map Compression

We first analyze the structure and maintenance cost of a traditional position map. Then we propose how MemCloak compresses the position map step by step. Alongside, we present key design principles of MemCloak. For ease of understanding and comparing efficiency gains yielded by each design technique, we in Figure 3 illustrate the traditional map as well as our compressed version.

The giant traditional position map. Following the traditional structure, each entry in a feasible position map for MemCloak should contain at least a requested physical address, a mapping physical address where locates an encrypted copy of the requested data block, and the decryption key for decrypting the encrypted copy. Let l_{addr} and l_{key} denote the length in bits of a physical address and a key, respectively. The size of each entry is $2l_{\text{addr}} + l_{\text{key}}$. Using entry indices to represent source addresses, the size of each entry is shrunk to $l_{\text{addr}} + l_{\text{key}}$. Given that the number of entries is upper bounded by the number n of blocks the memory can accommodate, the size of the position table approximates $(l_{\text{addr}} + l_{\text{key}})n$ bits. To guarantee security, l_{key} should be sufficiently large enough. Inspired by [2], we can use a global private key for all entries and a shorter counter for each entry. Both the private key and counter are used for decrypting an encrypted block. But each entry only needs to track its corresponding counter. Consider a practical example with 64-bit physical addresses, 64-bit counters [2], 64-byte data blocks, and 4 GB memory. The number of blocks supported by the memory is $n = \frac{4\text{ GB}}{64\text{ B}} = 2^{26}$. The corresponding position map then takes a size of $(64 + 64) \times 2^{26}$ bits = 1 GB. This is impractically large for the memory controller to cache or buffer.

Compression technique 1: Computation-based address mapping. Using lightweight computation to map source addresses to destination addresses, we can combine multiple entries into one and remove destination addresses from the position map. In the current design, we adopt hashing for computing destination addresses. To optimize the cost for implementing hash functions, we do not need to implement totally different hash functions. We can simply implement a limited number or even only one hash function and use a different seed for each instance. Let h denote the number of hash functions/instances and c denote the average number of copies per block. Then we can combine the c entries corresponding to the same block in the original position map into one with c fields. Each field contains the index of the hash function used for computing the destination address and the key used for decrypting data therein. The size of each entry is then $(l_{\text{hash}} + l_{\text{key}})c$, where we have $l_{\text{hash}} = \log h$ as the length of a hash-function index. The size of the compressed position map becomes $(l_{\text{hash}} + l_{\text{key}})c \times \frac{n}{c} = (l_{\text{hash}} + l_{\text{key}})n = (\log h + l_{\text{key}})n$. This promises a smaller position map as long as we have $\log h < l_{\text{addr}}$. We observe that $\log h = l_{\text{addr}}$ when the number of hash functions is equal to 2^{64} in a 64-bit addressed system. Given that each block associates with a limited number of copies, we do not need that many hash functions to map these copies to different addresses. Moreover, a uniform hash function can evenly map different inputs across the entire memory space. We thus do not need many hash functions to arbitrate address collisions either. Based on these observations, we expect that the computation-based address mapping technique yield a much smaller position map.

Compression technique 2: Address-derived decryption keys. Leveraging the uniqueness of destination addresses, we derive decryption keys from addresses on the fly in the encryption/decryption unit. We can thus remove keys from the position map and further compress its size. Note that the usage of addresses should be limited to key derivation. Addresses should not be directly tweaked into the messages sent to the memory. Otherwise, keys are vulnerable to inference due to chosen-plaintext attacks and cipher block move attacks [14]. Now each entry in the position map contains only c indices of hash functions, taking a size of $c \log h$. When $c \log h > h$, we can replace each entry with a bit vector of length h. We set the ith bit as 1 if the address computed by the ith hash function accommodates a copy, and set it as 0 otherwise. Using this technique, the size of the compressed position map approximates $\min\{c \log h, h\} \times \frac{n}{c} = \min\{n \log h, \frac{nh}{c}\}$.

Compression technique 3: Circular-based page-level address mapping. Although the preceding compression techniques can greatly decrease the size of the position map, they still have to deal with the very large coefficient n. Remember that in the previous example with 64-bytes blocks in 4 GB memory, we have the number of supported blocks up to $n = \frac{4\text{ GB}}{64\text{ byte}} = 2^{26}$. Even though each block contributes 1 bit to the position map, the coefficient n will lead to a component of 2^{26} bits = 8 MB. Inspired by the virtually tagged and physically indexed technique used for structuring a page table, we propose structuring the position map based on page-level address

mapping instead of the preceding block-level based one. Given a source address, we feed only its page index to the chosen hash function to compute the page index of the destination address. This makes the number of entries upper bounded by the number of pages instead of the number of blocks. How we use the block offset to locate a data block within a page also matters. If we simply equate the block offsets in the source and destination addresses as the virtually tagged and physically indexed technique does, an attacker can still possibly correlate two memory accesses and infer the access pattern. We address this concern using a circular-based in-page mapping technique. Let l_{page} and l_{block} represent the size of a page and the size of a block, respectively. Then the number of blocks in a page is $\frac{l_{\text{page}}}{l_{\text{block}}}$. Let s denote the seed of the chosen hash function for mapping the page index. The circular-based in-page mapping computes the block offset of the destination address as $b_{\text{dst}} = (b_{\text{src}} + s) \mod \frac{l_{\text{page}}}{l_{\text{block}}}$, where b_{src} represents the block offset in the source address. This way, the same block offset might be mapped to a different block off set in different pages and therefore breaks correlation.

However, another challenge arises upon invalidating accessed addresses. According to the page-level address mapping, all the blocks in a page use the same hash function for mapping them to a different page. After a block accesses the address computed by the hash function, that address should be invalidated from future accesses for the same block. This leads to invalidating the hash-function index just used for address computation. Invalidating a hash-function index means that subsequent memory requests corresponding to this entry cannot use it for address mapping. However, besides the just accessed block, all the other blocks in the same page have not been accessed. Invalidating unaccessed blocks degrades performance as we have to load many more data blocks in memory than what are actually used.

Against the preceding challenge, we have to re-introduce block-level information back to each entry but in an efficient way. What each block needs to track is which hash functions have been used and invalidated. Remember that all blocks in the same page use the same set of hash functions. This observation motivates us to more compactly encode the hash-function tracking for each block, without repeating all hash-function indices for each block. The idea is to introduce a vector v for each entry. The vector contains $\frac{l_{\text{page}}}{l_{\text{block}}}$ items. Item $v[i]$ corresponds to the ith block in the page. With a size of $\log c$ bits, item $v[i]$ aims to track how many hash functions out of c choices have already been used by the ith block for address mapping. Formally speaking, $v[i] = k$ means that the ith block has accessed k addresses computed by the first k hash functions indexed in this entry. These used hash functions are invalidated from future use by the ith block. The next request for the ith block will use the $(k + 1)$th hash function for address mapping. Based on this design, each block maintains only a $\log c$-bit counter up to the value of c. While in the original block-level position map, each block maintains c hash-function indices. Position-map size can thus be further reduced.

We now analyze the position-map size after using circular-based page-level address mapping. Each entry now maintains c indices of hash functions as well as a vector of $\frac{l_{\text{page}}}{l_{\text{block}}}$ items. It takes a size of $c \log h + \frac{l_{\text{page}}}{l_{\text{block}}} \log c$. Given that we map a block to c copies, then $\frac{n}{c}$ original blocks can be accommodated by memory. The number of original page, that is, the number of entries in the position map, is therefore $\frac{n}{c} / \frac{l_{\text{page}}}{l_{\text{block}}}$. Then the size of the position map approximates $(c \log h + \frac{l_{\text{page}}}{l_{\text{block}}} \log c) \times \frac{n}{c} / \frac{l_{\text{page}}}{l_{\text{block}}} = (\frac{l_{\text{block}}}{l_{\text{page}}} \log h + \frac{\log c}{c})n$. Now let us compare this size with that of the giant 1 GB traditional position map under the scenario of 64-bit physical addresses, 64-bit counters [2], 64-byte data blocks, 4 KB pages, and 4 GB memory. In this case, the number of blocks in a page is $\frac{l_{\text{page}}}{l_{\text{block}}} = 64$. Then the ratio of the size of the giant traditional position map to that of the compressed position map is $\frac{128}{\frac{\log h}{64} + \frac{\log c}{c}}$. It is straightforward that this ratio can be easily greater than 100 with practical settings of h and c. Take $h = 1,024$ and $c = 16$ for example. We have $\frac{128}{\frac{\log h}{64} + \frac{\log c}{c}} = \frac{128}{\frac{\log 1024}{64} + \frac{\log 16}{16}} = 315$. It indicates that our compressed position map have an over 300x smaller size, which is about $(\frac{\log 1024}{64} + \frac{\log 16}{16}) \times 2^{26} = 3.25$ MB.

Compression technique 4: Data aggregation to leverage hash collisions. Given the intrinsic collision property of hashing, it is normal that more than one address is mapped to the same address using a certain hash function. This usually requires introducing more hash functions such that each address has sufficient choices. More hash functions, however, lead to higher implementation overhead. Moreover, more hash functions associate with a larger h and thus a larger $\log h$, which leads to a longer entry and thus a larger position map. We address this concern by aggregating collided data using XOR. If two or more blocks are hashed to the same address, we can XOR them into an aggregated block and store the aggregated block into that address. Such data aggregation not only saves memory space but also enhances security. Atop accessing the same data from different locations supported by data duplication, data aggregation enables accessing different data from the same location. Such obfuscation leaves an attacker with no clues for correlating memory accesses. Alongside the position map, we use an independent *aggregation map* to track XORed blocks. The memory controller can decide when to access an XORed block based on whether all but one of its ingredient blocks are in the stash. The size of the aggregation map should be much smaller than the position map. We omit quantifying its storage overhead.

4.3 Position Map Update

As data blocks are accessed and invalidated, we need to replenish more data blocks into memory for future use. Data replenishment piggybacks data in dummy writes (Section 3.3) and requires updating the position map as well as the aggregation map. Since the aggregation map is small, we simply replace the invalidated entries with entries corresponding to newly replenished XORed blocks. However, it is more challenging to update the position map. In the current position map design, each block corresponds to only a counter that indicates how many selected hash functions are used. Each block needs to select a different hash function to map its new copy to a different location. A straightforward way is associating each block with the indices of newly selected hash functions. This method is ineffective due to two drawbacks. First, each block needs to maintain the indices of up to c hash functions. This inflates the position map back to the giant traditional one. Second, the addresses

a block can be mapped to is upper bounded by the number of available hash functions. If we expect that a block be evenly mapped across the entire memory space, an impractically large number of hash functions would be needed.

We propose an alias-based mapping technique to efficiently update the position map. As discussed in Section 3.3, we can use a different address as an address's mapping alias. That is, after an address's originally selected hash functions are all used, it can direct to its alias's entry and use the hash functions therein for address mapping. After its alias's hash functions are all used, it can choose another alias. This makes each address uniformly mapped across the entire memory space without cumbersome and insecure re-selection of hash functions. Specifically, the alias-based mapping introduces two aliases to each entry and one more counter for each block in that entry. This way, all blocks in the same page share same aliases but each of them has two propriety counters. One counter tracks how many hash functions of the current alias is used. The other counter tracks how many new copies are replenished to addresses computed using the next alias and its hash functions. Apparently, the current alias is initialized as an address per se. The next alias to choose should have higher access frequency than that of the current alias. Otherwise, replenished data may override some unaccessed data. For pages with the highest access frequency, we can leave some blank pages in memory for their aliasing. Furthermore, we can use well-designed caching/buffering strategies of the stash and caches to balance memory access frequency of each page. That is, highly frequently accessed blocks usually stay in cache. They thus may not impose too many real memory accesses or position-map updates.

We now analyze the position-map size after integrating the alias-based mapping technique. The overhead it introduces is two alias (of size $2l_{\mathrm{addr}}$) per page and one counter (of size $\log c$) per block. Following the analysis of previous compression technique 3, the size of the position map becomes $(c \log h + \frac{l_{\mathrm{page}}}{l_{\mathrm{block}}} \log c + 2l_{\mathrm{addr}} + \frac{l_{\mathrm{page}}}{l_{\mathrm{block}}} \log c) \times \frac{n}{c} / \frac{l_{\mathrm{page}}}{l_{\mathrm{block}}}$. We consider the same settings of 64-bit physical addresses, 64-byte data blocks, 4 KB pages, 4 GB memory, $h = 1,024$ hash functions, and $c = 16$ copies per block on average as in previous analysis. The position map then takes a size of 6.25 MB. It is compressed over 100x than the traditional position map and can be practically fitted into a cache or buffer.

5 IMPLEMENTATION

As with related obfuscation solutions [5, 43], we implemented MemCloak using the gem5 simulator [7]. Gem5 is widely used for computer architecture research. Its emulation encompasses CPU and memory modules and supports tracing memory accesses, which are exactly the operations MemCloak aims to protect. MemCloak implementation aims to enforce the proposed obfuscation techniques over the conventional memory accesses in gem5. As discussed in Section 4, key components include the position map (including the aggregation map), the address computation unit, the hash unit, and the encryption/decryption unit.

Our modification over gem5 remains as a transparent layer between the last-level cache and memory. A memory request in gem5 is first transmitted from the CPU to the first-level cache. If it enjoys a cache hit therein, the requested data is transmitted to the CPU. Otherwise, the first-level cache redirects the memory request to the second-level cache. Upon a cache hit on the second-level cache, the requested data is first transmitted to the first-level cache and then transmitted to the CPU. Generally speaking, gem5 directs a data block from the lowest-level cache where the data block is first found through all its higher-level caches to the CPU. Although this may take a longer access time when the requested data block is found for the first time, caching it in some higher-level cache will save its subsequent access time. If a requested data block cannot be found until the last-level cache, the memory request will be eventually transmitted to memory. A Cache class uses the `CpuSidePort` interface to communicate with its higher-level cache (or the CPU if the Cache class represents the first-level cache) and uses the `MemSidePort` to communicate with its lower-lever cache (or memory if the Cache class represents the last-level cache). Since MemCloak intercepts memory access requests from the last-level cache and may redirect these requests to memory, we implement MemCloak by inheriting from a Cache class in gem5. We use AES counter mode for the encryption/decryption unit. We adopt the random function `rand()` in the standard C library for designing the hash unit. In hardware, such random number generators can be implemented using circuit white noise [21]. MemCloak performs address mapping before sending a memory request to memory and address update before forwarding a fetched data block to the last-level cache. Our modification accounts for 2,000+ lines of C/C++ code while the original code base of gem5 contains over 250,000 lines.

To evaluate the performance of MemCloak, we run highly memory-intensive workloads from MiBench [19]. MiBench is a representative benchmark suite for embedded systems. It collects multiple benchmarks for six embedded application areas, that is, Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecommunications. As embedded systems usually have memory restrictions, it is commonly used to validate efficiency and practicality of access obfuscation solutions [45, 46]. The three highly memory-intensive benchmarks we use are dijkstra, susan, and jpeg encode from the areas of Networking, Automotive and Industrial Control, and Consumer Devices.

6 EVALUATION

In this section, we evaluate security and efficiency of MemCloak. First, the security performance is measured by the randomness of the address sequence accessed during the execution of a benchmark [45]. The randomness is tested using the NIST Statistical Test Suite [23]. Second, the efficiency performance is measured by the execution time and memory usage of a benchmark on the emulated computer architecture. The gem5 simulator tracks the number of clock cycles taken by a benchmark. Then it estimates the execution time by multiplying the number of clock cycles and the configured clock period. The results demonstrate that MemCloak can significantly randomizes memory accesses with approximately 4% time overhead and comparative memory overhead with that of ORAM.

Experiment setup. Since gem5 estimates the relative execution time of a benchmark using the number of clock cycles taken in the emulated computer architecture, the estimation is insensitive to the running environment. We currently run our MemCloak prototype

Table 1: Comparison of memory access randomness of insecure (without MemCloak) and secure (with MemCloak) execution. We test the access randomness using the NIST Statistical Test Suite [30] with 14 randomness analysis tools supported. The tools corresponding to each row are Frequency (Monobit) Test, Frequency Test within a Block, Cumulative Sums (Cusum) Test 1, Cumulative Sums (Cusum) Test 2, Runs Test, Test for the Longest Run of Ones in a Block, Binary Matrix Rank Test, Discrete Fourier Transform (Spectral) Test, Overlapping Template Matching Test, Maurer's "Universal Statistica" Test, Approximate Entropy Test, Serial Test 1, Serial Test 2, and Linear Complexity Test [30].

dijkstra		susan		jpeg encode	
insecure	secure	insecure	secure	insecure	secure
0/283	249/528	14/318	162/397	112/1133	1777/1813
0/283	511/528	19/318	381/397	28/1133	1137/1813
0/283	267/528	11/318	167/397	28/1133	1137/1813
0/283	266/528	11/318	169/397	23/1133	1134/1813
0/283	424/528	2/318	272/397	11/1133	1333/1813
1/283	515/528	15/318	371/397	112/1133	1787/1813
281/283	525/528	316/318	389/397	1124/1133	1805/1813
20/283	526/528	221/318	392/397	279/1133	1792/1813
264/283	521/528	313/318	386/397	1116/1133	1783/1813
0/283	0/528	0/318	0/397	0/1133	0/1813
0/283	309/528	0/318	150/397	0/1133	841/1813
1/283	483/528	1/318	391/397	0/1133	1692/1813
26/283	506/528	5/318	395/397	329/1133	1757/1813
273/283	510/528	307/318	385/397	1107/1133	1751/1813
average					
62/283 = 22%	401/528 = 76%	88/318 = 28%	286/397 = 72%	305/1133 = 27%	1409/1813 = 78%

on a MacBook Pro with a 4-core 2.9-GHz Intel Core i7 processor and 16 GB memory. Following [45], we use two 8-way associative 32 KB caches (one for instructions and the other for data) to strengthen the memory intensiveness of the selected benchmarks. We observe that another reason to limit the cache size during emulation is that the benchmarks may not have sufficiently large workload. If the size of the cache(s) is relatively large, most data can be cached after the first read. The cases when the CPU accesses the same location twice for reading certain data would be rare. This does not necessarily require access obfuscation. In other words, it is hard to comprehensively evaluate the performance of an obfuscation solution when few repetitive memory accesses exist. Since the benchmark workload is fixed, we need to restrict the cache size to indirectly yield more memory accesses. Other settings of the emulated computer architecture include a 2 GHz processor, a 4 GB memory, an 8 KB stash, 4 KB pages, and 64-byte blocks.

6.1 Memory Access Randomness

Following [45], we evaluate the obfuscation efficacy of MemCloak by measuring the randomness of the access sequence. We test the access randomness using the NIST Statistical Test Suite (version 2.1.2, updated in 2010) [30]. It is originally designed to validate random number generators and pseudo-random number generators [29]. Then it is widely used for randomness analysis as well. We refer interested readers to [30] for the technical details. Toward measuring the access randomness, we first collect the access sequence while running a selected benchmark and snooping on the address bus. We then convert the access sequence into a bit stream by concatenating all of the block addresses in the same order as they are collected. We input the bit stream to the NIST Statistical Test Suite. It will be divided into same-length bitstrings, each is analyzed by all supported randomness analysis tools in the NIST Statistical Test Suite. A higher randomness of the input bit stream depends on the following two properties.

- For each randomness analysis tool, more bitstrings of the bit stream can pass the test.
- For all randomness analysis tools, more of them can accept most of the bitstrings of the bit stream.

We compare the access randomness of benchmark execution with or without MemCloak obfuscation. Table 1 reports the comparison using benchmarks of dijkstra, susan, and jpeg encode; each data block has 10 differently-encrypted copies into the memory. In each column, each row reports the randomness test result using one of the 14 supported randomness analysis tools. For most of the tools, the insecure execution without MemCloak obfuscation can barely pass the test. In contrast, MemCloak significantly randomizes memory access in that most bitstrings can pass the randomness test. Take, for example, the results of the dijkstra benchmark using the Frequency (Monobit) Test tool (i.e., the two values on the top left corner). Without access obfuscation, $\frac{0}{283} = 0\%$ of the bitstrings of the bit stream corresponding to the access sequence can pass the randomness test. Armed with access obfuscation by MemCloak, $\frac{249}{528} = 47\%$ of the bitstrings can pass the test. Note that MemCloak induces more memory accesses because of dummy writes. As shown in Table 1, MemCloak significantly increases the access randomness from below 30% to over 70% on average. Furthermore, MemCloak outperforms the existing $O(1)$–communication-overhead access obfuscation solution [45], in which the average randomness test result ranges from 13% to 44%.

Table 2: Comparison of execution time in seconds of insecure (without MemCloak) and secure (with MemCloak) execution.

dijkstra		susan		jpeg encode	
insecure	secure	insecure	secure	insecure	secure
5.551	5.778	492.475	509.718	7.542	7.843

6.2 Execution Time

Table 2 reports the execution time corresponding to different benchmark executions in Table 1. Since protecting the access type necessitates dummy writes, MemCloak requires more memory accesses and therefore takes more time. The average time overhead is about 4%, estimated using the time measurements in the "insecure" and "secure" columns as $\frac{secure - insecure}{insecure}$.

6.3 Memory Usage

Finally, we evaluate how the number of copies per data block affects memory usage and obfuscation randomness. Intuitively, the more copies each data block has, the more memory space MemCloak consumes. Given c copies per data block, the memory overhead is upper bounded by $\frac{c-1}{c} = 1 - \frac{1}{c}$. Although we can aggregate

Table 3: Comparison of memory access randomness of insecure (with only the original data, i.e., one copy per data block) and secure (with MemCloak) execution with various number of copies per data block.

benchmark	number of copies per data block					
	1	2	4	6	8	10
dijkstra	22%	70%	74%	76%	76%	77%
susan	28%	68%	70%	70%	71%	72%
jpeg encode	27%	75%	76%	76%	77%	77%

data blocks to mitigate the memory overhead, it is challenging to quantify or evaluate the effect in a generalized way. It highly depends on how many blocks to aggregate into one and how many copies of a data block to select for aggregation. This is easy to regulate upon initialization but tends to vary during execution.

Fortunately, our evaluation results show that MemCloak can guarantee a satisfactory access randomness with only a limited data redundancy. Furthermore, the increase of access randomness does not significantly increase with the number of copies per data block. This simplifies how to find a tradeoff between memory usage and obfuscation security. Table 3 reports the average access randomness of MemCloak with different levels of data redundancy in comparison with that of the traditional memory access without obfuscation. Specifically, the case of one copy corresponds to the traditional memory access. The cases with two or more copies correspond to MemCloak. MemCloak gains limited randomness improvement as data redundancy increases. When MemCloak uses only two copies per block, it can already improve the access randomness from under 30% to over 65% for each benchmark. In this case, using two copies per data block lead to up to 50% memory overhead, which is comparative to that of ORAM [34].

7 DISCUSSION

Timing attack. As with ORAM, MemCloak does not protect memory access against timing attacks. A timing attack exploits fine-grained timing measurements snooped on the address bus [6, 16]. For example, the access number and frequency, and the gaps between observed memory requests can leak information of program characteristics [44]. A straightforward countermeasure is that the memory controller enforces fixed-rate memory accesses. Enforcing a constant access rate, however, impairs scheduling flexibility and increases the difficulty of tuning tradeoff between security and performance. A better way is to shape the rate of CPU-memory communication into a pre-determined distribution [16, 44]. When genuine traffic solely cannot satisfy the distribution, additional fake traffic is injected. Traffic shaping against timing attacks is complementary to access obfuscation [44]. Both schemes need to be deployed when necessary.

Statistics-based attack. Originally targeting searchable encryption, a statistics-based attack enables an honest-but-curious server to infer encrypted keywords using search pattern and occurrence frequency over its hosted encrypted data [20, 33]. For example, "Thanksgiving" can be a hot keyword on the Thanksgiving Day. The encrypted Thanksgiving-related information stored on the server must be frequently hit as responses to queries. Then if a new encrypted query arrives and the frequently-hit data is hit by the query, it is reasonable for the server to conjecture that the said query includes the keyword "Thanksgiving". Mounting a statistics-based attack on memory access, an attacker needs a fixed data placement in memory. This requirement is, however, exactly what ORAM and MemCloak break to protect memory access patterns. Therefore, MemCloak as well as existing access obfuscation solutions are robust against statistics-based attacks.

Trace-driven attack. Such an attack exploits the traces of cache hits and misses during AES encryption to infer the secret key [1]. That is, the attacker should be empowered with two types of information. One is that the program under execution is AES. The other is the corresponding trace of cache of hits and misses. Without memory access obfuscation, both types of information can be inferred via snooping on the address bus. First, access patterns can be used to profile the program and therefore reveal its type [46]. Second, a memory request indicates a cache miss; this eases the measurement of cache hits and misses. Fortunately, both types of inference are throttled by MemCloak alike obfuscation solutions, which disguise memory access patterns against profiling. Furthermore, MemCloak introduces dummy reads and writes to protect the type of memory access. A read operation thus does not always represent a real memory request. This way, MemCloak further impedes the measurement of cache traces and therefore protects memory access from trace-driven attacks.

8 CONCLUSION

We have designed, implemented, and evaluated MemCloak, a practically efficient solution for obfuscating accesses over untrusted memory. It achieves $O(1)$ communication overhead by leveraging data redundancy. Specifically, MemCloak preloads multiple differently-encrypted copies of each block in memory. This enables the CPU to fetch the same data by accessing different memory addresses and therefore leaks no access pattern. Furthermore, it improves memory utilization by introducing data aggregation. XORing two blocks into one, we can access the XORed block for fetching one of the two blocks when the other is fetched and cached/buffered. This not only saves memory space but also strengthens security. The CPU can now access the same address for fetching different data. With this, MemCloak leaves an attacker with no clue for correlating memory accesses. A common challenge for access obfuscation design is limiting the size of the position map that tracks memory layout. We propose a series of optimization techniques without sacrificing security. MemCloak can compress a giant traditional position map of size up to 1 GB into a significantly smaller one of only several megabytes. Such a compression over hundreds of times makes the position map practically fit in an on-chip cache/buffer. We implement MemCloak on the gem5 simulator [7] and validate its performance using memory-intensive MiBench benchmarks [19]. For future work, we plan to extend MemCloak to obfuscate data accesses over cloud, where memory is more affordable than communication.

ACKNOWLEDGEMENT

This work is supported in part by the National Natural Science Foundation of China under Grant No. 61402404. We would like to thank ACSAC 2018 Chairs and Reviewers and our shepherd, Evangelos Markatos, for their review efforts and helpful feedback. We would also like to extend our gratitude to Tao Li and Baiqiang Leng for their help with implementation of MemCloak.

REFERENCES

[1] Onur Acıiçmez and Çetin Kaya Koç. 2006. Trace-driven cache attacks on AES (short paper). In *ICICS*. 112–121.

[2] Shaizeen Aga and Satish Narayanasamy. 2017. InvisiMem: Smart Memory Defenses for Memory Bus Side Channel. In *ISCA*. 94–106.

[3] Frances E Allen. 1970. Control flow analysis. In *ACM Sigplan Notices*, Vol. 5. 1–19.

[4] Amro Awad, Pratyusa Manadhata, Stuart Haber, Yan Solihin, and William Horne. 2016. Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers. In *ASPLOS*. 263–276.

[5] Amro Awad, Yipeng Wang, Deborah Shands, and Yan Solihin. 2017. Obfusmem: A low-overhead access obfuscation for trusted memories. In *MICRO*. 107–119.

[6] Chongxi Bao and Ankur Srivastava. 2017. Exploring timing side-channel attacks on path-orams. In *HOST*. 68–73.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[8] Zhao Chang, Dong Xie, and Feifei Li. 2016. Oblivious ram: a dissection and experimental evaluation. *VLDB*, 1113–1124.

[9] Siddhartha Chhabra and Yan Solihin. 2011. i-NVMM: a secure non-volatile main memory system with incremental encryption. In *ISCA*. 177–188.

[10] Kai-Min Chung, Zhenming Liu, and Rafael Pass. 2014. Statistically-secure ORAM with $\tilde{O}(log^2 n)$ Overhead. In *Asiacrypt*. 62–81.

[11] Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. 2016. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *TCC*. 145–174.

[12] Whitfield Diffie and Martin Hellman. 1976. New directions in cryptography. *IEEE Transactions on Information Theory* 22, 6 (1976), 644–654.

[13] Jack Doerner and abhi shelat. 2017. Scaling ORAM for Secure Computation. In *CCS*. 523–535.

[14] Z.-H. Du, Z. Ying, Z. Ma, Y. Mai, P. Wang, J. Liu, and J. Fang. 2017. Secure Encrypted Virtualization is Unsecure. *ArXiv e-prints* (Dec. 2017). arXiv:cs.CR/1712.05090

[15] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. 2015. Freecursive oram:[nearly] free recursion and integrity verification for position-based oblivious ram. In *ASPLOS*. 103–116.

[16] Christopher W Fletchery, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas Devadas. 2014. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *HPCA*. 213–224.

[17] Blaise Gassend, G Edward Suh, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. 2003. Caches and hash trees for efficient memory integrity verification. In *HPCA*. 295–306.

[18] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473.

[19] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization*. 3–14.

[20] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *NDSS*, Vol. 20. 12.

[21] Benjamin Jun and Paul Kocher. 1999. The Intel random number generator. *Cryptography Research Inc. white paper* (1999).

[22] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural support for copy and tamper resistant software. In *ASPLOS*. 168–177.

[23] Kinga Marton and Alin Suciu. 2015. On the interpretation of results from the NIST statistical test suite. *SCIENCE AND TECHNOLOGY* 18, 1 (2015), 18–32.

[24] Yuto Nakano, Carlos Cid, Shinsaku Kiyomoto, and Yutaka Miyake. 2012. Memory access pattern protection for resource-constrained devices. In *International Conference on Smart Card Research and Advanced Applications*. 188–202.

[25] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers'Track at the RSA Conference*. 1–20.

[26] Benny Pinkas and Tzachy Reinman. 2010. Oblivious RAM revisited. In *CRYPTO*. 502–519.

[27] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. 2015. Constants Count: Practical Improvements to Oblivious RAM.. In *USENIX Security Symposium*. 415–430.

[28] Ronald L Rivest, Adi Shamir, and Leonard Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (1978), 120–126.

[29] A Ruk et al. 2001. A statistical test suite for the validation of random number generators and pseudo-random number generators for cryptographic applications. *NIST Special Publication* (2001).

[30] Andrew Rukhin, J Soto, J Nechvatal, M Smid, M Levenson, D Banks, M Vangel, S Leigh, S Vo, and J Dray. 1999. A statistical test suite for the validation of cryptographic random number generators. *NIST Computer Security Division/Statistical Engineering Division Internal Document* (1999).

[31] Ali Shafiee, Rajeev Balasubramonian, Mohit Tiwari, and Feifei Li. 2018. Secure DIMM: Moving ORAM Primitives Closer to Memory. In *HPCA*. 428–440.

[32] Weidong Shi, Hsien-Hsin S Lee, Mrinmoy Ghosh, Chenghuai Lu, and Alexandra Boldyreva. 2005. High efficiency counter mode security architecture via prediction and precomputation. In *ISCA*, Vol. 33. 14–24.

[33] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. 2000. Practical techniques for searches on encrypted data. In *S&P*. 44–55.

[34] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*. 299–310.

[35] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. Efficient memory integrity verification and encryption for secure processors. In *MICRO*.

[36] Rujia Wang, Youtao Zhang, and Jun Yang. 2017. Cooperative Path-ORAM for Effective Memory Bandwidth Sharing in Server Settings. In *HPCA*. 325–336.

[37] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *CCS*. 850–861.

[38] Xiao Shaun Wang, Yan Huang, TH Hubert Chan, Abhi Shelat, and Elaine Shi. 2014. SCORAM: oblivious RAM for secure computation. In *CCS*. 191–202.

[39] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. 2006. Improving cost, performance, and security of memory encryption and authentication. In *ISCA*. 179–190.

[40] Jun Yang, Lan Gao, Youtao Zhang, Marek Chrobak, and Hsien-Hsin S Lee. 2010. A low-cost memory remapping scheme for address bus protection. *J. Parallel and Distrib. Comput.* 70, 5 (2010), 443–457.

[41] Vinson Young, Prashant J Nair, and Moinuddin K Qureshi. 2015. DEUCE: Write-efficient encryption for non-volatile memories. *ASPLOS* (2015), 33–44.

[42] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. 2016. Revisiting square-root ORAM: efficient random access in multi-party computation. In *S&P*. 218–234.

[43] Xian Zhang, Guangyu Sun, Chao Zhang, Weiqi Zhang, Yun Liang, Tao Wang, Yiran Chen, and Jia Di. 2015. Fork path: improving efficiency of oram by removing redundant memory accesses. In *MICRO*. 102–114.

[44] Yanqi Zhou, Sameer Wagh, Prateek Mittal, and David Wentzlaff. 2017. Camouflage: Memory traffic shaping to mitigate timing attacks. In *HPCA*. 337–348.

[45] Xiaotong Zhuang, Tao Zhang, Hsien-Hsin S Lee, and Santosh Pande. 2004. Hardware assisted control flow obfuscation for embedded processors. In *CASES*. 292–302.

[46] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. 2004. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *ASPLOS*. 72–84.