ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories

Amro Awad
Sandia National Laboratories
aawad@sandia.gov

Yipeng Wang
North Carolina State University
ywang50@ncsu.edu

Deborah Shands
National Science Foundation
deborah.shands@gmail.com

Yan Solihin
North Carolina State University
solihin@ncsu.edu

ABSTRACT

Trustworthy software requires strong privacy and security guarantees from a secure trust base in hardware. While chipmakers provide hardware support for basic security and privacy primitives such as enclaves and memory encryption. these primitives do not address hiding of the memory access pattern, information about which may enable attacks on the system or reveal characteristics of sensitive user data. State-of-the-art approaches to protecting the access pattern are largely based on Oblivious RAM (ORAM). Unfortunately, current ORAM implementations suffer from very significant practicality and overhead concerns, including roughly an order of magnitude slowdown, more than 100% memory capacity overheads, and the potential for system deadlock.

Memory technology trends are moving towards 3D and 2.5D integration, enabling significant logic capabilities and sophisticated memory interfaces. Leveraging the trends, we propose a new approach to access pattern obfuscation, called *ObfusMem*. ObfusMem adds the memory to the trusted computing base and incorporates cryptographic engines within the memory. ObfusMem encrypts commands and addresses on the memory bus, hence the access pattern is cryptographically obfuscated from external observers. Our evaluation shows that ObfusMem incurs an overhead of 10.9% on average, which is about an order of magnitude faster than ORAM implementations. Furthermore, ObfusMem does not incur capacity overheads and does not amplify writes. We analyze and compare the security protections provided by ObfusMem and ORAM, and highlight their differences.

CCS CONCEPTS

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

KEYWORDS

Access Pattern Ofuscation, Hardware Security, ORAM, Emerging Memory Technologies

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080230

ACM Reference format:
Amro Awad, Yipeng Wang, Deborah Shands, and Yan Solihin. 2017. ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 13 pages.
https://doi.org/10.1145/3079856.3080230

1 INTRODUCTION

Security and privacy breaches frequently impact a wide range of systems that individuals, businesses, and governments depend upon for day-to-day activities, as well as life-critical operations. Protecting operational software requires stronger privacy and security guarantees from a trust base in hardware, as attacks against hardware undermine system foundations. Physical attacks that passively snoop or actively tamper with the memory bus or scan the DRAM or Non-Volatile Memory (NVM) chips are particularly insidious as they very difficult to detect. By observing memory access patterns [25, 27, 40, 54], an attacker may gain information that further enables a devastating attack against the operational system or breach of confidential user data.

State-of-the-art approaches to obfuscating the access pattern are largely based on *Oblivious RAM* (ORAM) [24]. ORAM reshuffles the memory location of a datum (typically a cache block) after each access to prevent observers from learning anything useful from the address stream of memory accesses. The state-of-the-art hardware implementations of ORAM include Path ORAM [47] and its variants [16, 20, 40, 52]. While it offers better performance than other hardware implementations, Path ORAM's overhead limits its practicality. Path ORAM uses a block-level address translation table (PosMap) and each block in memory is associated with a path in a tree. Reading a memory block requires reading (and decrypting) all blocks in the path of the tree from the root to the leaf (100 blocks for 8GB memory), relocating the block to a new path, updating the PosMap, and writing back the block to the new path, in addition to any tree nodes that are evicted in the process. As a result, ORAM implementations incur significant performance overheads: bandwidth increase by 24× and 120× in Ring and Path ORAM, respectively [40]. Such significant bandwidth overhead translates into significant execution slowdown. Furthermore, at least 50% of memory capacity is wasted (i.e., 100% memory overhead) through dummy blocks in order to achieve an acceptable failure rate [16, 20, 40, 47, 52]. Finally, even with 100% memory overhead, failures can still occur, where reshuffling cannot proceed because all buckets along a tree path are full, resulting in system deadlock.

ORAM was designed with an assumption of conventional memories with very limited computational power. It is time to revisit this assumption. The confluence of new memory technologies offers appealing opportunities for access pattern obfuscation in the memory bus. First, memory interfaces are becoming smarter, increasingly incorporating logic that receives request packets instead of specific memory commands. Second, 3D and 2.5D stacked memory, such as Hybrid Memory Cube [2], High Bandwidth Memory [6], and Wide I/O 2 [49], will have at least one logic layer [12] containing memory controller logic, buffering, and interconnection in the same chip as memory banks.

3/2.5D memory integration and increased logic capabilities enable new access pattern obfuscation approaches. In this paper, we propose *ObfusMem*, which obfuscates the access pattern through encrypting memory requests (commands, addresses, and data) before they are placed on the memory bus. ObfusMem expands the trusted computing base to include memory chips and adds a cryptographic engine to the memory to support memory request decryption. Continuous memory location reshuffling becomes unnecessary, eliminating the associated performance, storage, and write amplification overheads and deadlock problem.

The design of ObfusMem presented several challenges. First, in ORAM, read and write operations are indistinguishable. To similarly obfuscate the types of memory requests in ObfusMem requires an additional mechanism. Second, a realistic memory system has multiple memory channels, so access patterns must be obfuscated across multiple memory channels. Third, the ObfusMem design interacts in interesting ways with memory encryption and integrity verification. For example, data encrypted for the main memory must be encrypted again for transmission over the memory bus, and the way the message authentication code (MAC) is computed has significant performance implications. We discuss these issues and provide our insights. Finally, we compare the security protection provided by ObfusMem to that provided by ORAM.

To evaluate the performance characteristics of our design, we implemented ObfusMem on gem5, a full-system cycle-accurate simulator, and ran several memory intensive workloads from SPEC 2006. Our evaluation showed that ObfusMem performance overhead is only 10.9% on average, inclusive of the overheads from regular memory encryption and communication authentication. ObfusMem does not increase the wearout of the memory (if NVM). We believe that ObfusMem's simplicity, low performance overheads, and zero memory overheads, make it an attractive alternative to ORAM.

The rest of the paper is organized as follows. Section 2 discusses the threat model, logic in memory, and overview of ORAM. Section 3 discusses ObfusMem trust architecture and design. Section 4 describes our evaluation methodology, while Section 5 presents quantitative evaluation of ObfusMem. Section 6 discusses security analysis and comparison of ORAM and ObfusMem. Section 7 discusses the most relevant work. Finally, Section 8 concludes this work.

2 BACKGROUND AND ASSUMPTIONS

2.1 Threat Model

Much previous work in secure processor studies [9, 14, 16, 40, 50–52] assume that a die provides a secure perimeter. An attacker can *passively observe* and *actively tamper* with communication on exposed wires between dies, but not within a die. Consequently, the data stream and access pattern (read vs. write, and address stream) on the *memory bus*, i.e. the bus connecting the processor chip and memory chips, is vulnerable to attacks. Furthermore, it assumes that the memory is outside the secure perimeter, and hence is vulnerable to attacks. Memory can be attacked without physical access (e.g. *row hammer* attack [13, 45]), or with physical access (e.g. beaming high-energy neutrons on memory cells [48]). The proliferation of edge/fog computing, where computation is pushed to small servers near the clients, increases the risk of physical attacks, because strong physical security may be prohibitively expensive compared to that in large data centers. Finally, a secure processor also provides a secure execution environment in hardware, to protect against malicious or compromised system software. Intel SGX [30] is an example of a secure processor architecture.

Given that commercial secure processors are already widely available, we assume a basic secure processor as a starting point. Specifically, we assume a processor that includes encryption and integrity verification of data stored in memory and communicated on the processor-memory bus. Such a processor requires memory encryption and Merkle Tree integrity verification [43]. On top of that, ObfusMem adds obfuscation of the access pattern that appears on the processor-memory bus. While some secure processors, such as Sanctum [15], share a similar addition, our next assumptions differ. On the memory side, we assume that the logic layer of the 3/2.5D memory system is not susceptible to physical attacks, i.e., tampering with the internal logic/wires of the memory chip is not feasible. We also assume that all internal wires of processor and memory modules are well-buried inside the silicon wafer and difficult to probe. This is achievable due to the advances in packaging technologies, 3D stacking and Through Silicon Vias (TSVs) technologies. Finally, memory accesses and communication with the processor leak side channel information such as power, timing, and electromagnetic signals [10, 19, 26, 28, 29, 31, 36, 37]. Addressing these side channel information leakages is not a goal for ObfusMem design and is left for future work.

2.2 Logic in Memory Modules

Memory interfaces are becoming smarter, increasingly incorporating logic that receives request packets instead of specific memory commands. DDR4 interfaces account for control logic inside the Dual-Inline Memory Modules (DIMMs) that deploy internal logic for buffering, and registers that can be configured by the memory controller [3, 4]. Registered DIMMs [3, 4] and Fully-Buffered DIMMs [33] deploy internal logic to decode commands and addresses before sending them to memory chips. For performance scalability, many servers use a new industry standard memory module: Load-Reduced DIMMs (LR-DIMMs). In LR-DIMM, an isolation buffer is used to buffer the data and commands inside the DIMM [34].

Memory technologies are expected to follow the trend of increasingly smarter memory interfaces. For example, Non-Volatile Memory (NVM) products, such as Intel 3D XPoint, have TBs of capacity and write endurance problem, requiring logic for wear leveling, scheduling, and failure remapping logic that need to be incorporated

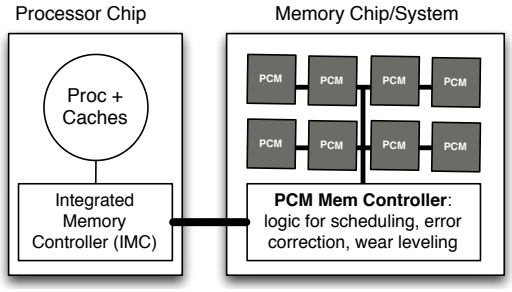

Figure 1: An example implementation of PCM-based DIMM [5].

into the NVM, rather than in the processor-side memory controller. Intel's patent application [5] describes a PCM memory controller that includes logic with such capabilities, as shown in Figure 1. Similar logic can also be observed in recent PCM prototypes [8].

Another important trend is 3/2.5D stacked memory, which not only has substantial logic budget on a separate die, but also buries this logic die layer under memory die layers [12]. The communication between logic and memory in the stacked memory system all occur within the chip, unexposed to attackers. In this context, only the communication between the processor chip and the memory chip is vulnerable to attack as it is transmitted on exposed wires. The logic layer is expected to have moderate power budgets: up to 42 W for commodity servers and up to 55 W for high-end servers [18]. This presents an opportunity for simplifying access pattern obfuscation.

2.3 Access Pattern Obfuscation

While secure processors encrypt memory contents [9, 14, 50, 51], memory devices require memory addresses to be transmitted plainly over the memory bus. An attacker with physical access to the computer can snoop the memory bus and observe the access pattern (address stream and type of requests) [27], which has been shown to leak control flow and data that enables leakage of cryptographic keys [54]. ORAM prevents such snooping and observation by reshuffling the memory location of a datum (typically a cache block) after each access to prevent observers from learning useful information from the address stream of memory accesses. State-of-the-art hardware implementations of ORAM, Path ORAM [47] and its variants [16, 20, 40, 52], use an address translation table (called PosMap) to translate a physical address to an actual address in memory, similar to a page table but operating at the block level. After the memory access completes, the block is randomly assigned a new memory address, using the aid of a tree. Each memory block is assigned to a leaf of the tree, ensuring a unique path from the root of the tree to the block at the leaf. When a block's memory address is reshuffled, it is re-assigned from one tree path to a new randomly chosen tree path. The PosMap is then updated to reflect the new memory address. Each node contains a bucket of blocks, some real and some dummy. Path ORAM ensures that the following invariance is maintained [47]:

> If a block is mapped to leaf l, then it must be either in some bucket on path l or in the stash. Blocks are stored in the stash or ORAM tree along with their current leaf and block address.

When there is a memory access, all blocks along the path are read and decrypted into the *stash* (a secure temporary storage in the processor chip). The path length determines the number of blocks read for each memory access, and is dependent on the size of the memory, e.g. 100 for 8GB memory. All the real blocks are added to the stash. After the block is assigned a new leaf, as many blocks in the stash from the old path are evicted and encrypted. Any remaining space is filled with encrypted dummy blocks.

ORAM obfuscates the access pattern in two ways: First, the memory location of blocks get reshuffled after each access. Second, dummy blocks may be read along with the intended block to provide additional obfuscation. Thus, the deployment of ORAM comes at the cost of many additional memory accesses, due to fetching blocks from a tree path, and the additional address translation. Significant efforts have been made to reduce the overhead of ORAM. However, Ring ORAM [41] and Path ORAM still suffer from multiple times execution slowdown and bandwidth overhead of 24× and 120×, respectively [40]. Furthermore, at least 50% of memory capacity is wasted in order to achieve a reasonably acceptable failure rate [16, 20, 47, 52]. Reshuffling may be prevented if buckets are full along a randomly-chosen new tree path. This could lead to system deadlock.

Non-Volatile Memory (NVM) increases the cost of Path ORAM. NVM writes are expensive because NVM cells have limited write endurance (a few hundred million writes for PCM cells), a write takes much more time to perform than a read, and a write takes much higher energy to perform than a read. Path ORAM and other ORAM implementations cause write amplification: every read or write access to memory results in reading a whole path of blocks (about 100 cache blocks for 8GB memory for $L = 24$ and $Z = 4$) and then writing them back after reshuffling [47]. This write amplification may reduce the memory lifetime by orders of magnitude, significantly increase power consumption, and slow down execution time. The high number of writes and its associated overheads (execution time and energy), and the reduction in write endurance, significantly reduce ORAM's attractiveness for NVM usage.

2.4 Memory Encryption

Prior work considers two different approaches for encrypting data in memory, depending on assumptions about the threat model. i-NVMM [14] describes self-encrypting NVM to match the non-volatile retention time of DRAM, under the assumption that only data remanence attacks are possible. [9, 50, 51] assumes that passive bus snooping, physical bus traffic tampering, and memory content readout and modifications were possible. This leads to a design in which data in memory is encrypted at all times and only the processor chip is trusted. Any data sent off chip to the main memory is encrypted. Any data brought into the processor is decrypted and its integrity is verified.

The state-of-the-art memory encryption uses *counter mode* encryption [50], where the encryption algorithm is applied to an *initialization vector* (IV) to generate a one-time pad. This is illustrated in Figure 2. Data is then encrypted and decrypted via a simple bit-wise XOR with the pad. The decryption latency is overlapped with the LLC miss latency, and only the XOR latency is added to the

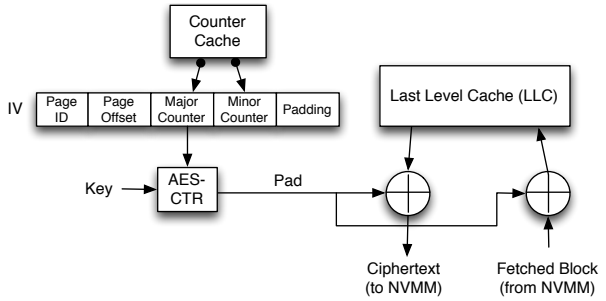

Figure 2: Example of counter mode memory encryption [9, 50].

critical path of an LLC miss. In state-of-the-art design, the IV of a counter mode encryption consists of a unique ID of a page (similar to a page address, but unique across the main memory and swap space in the disk), page offset (to distinguish blocks in a page), a per-block *minor* counter (to distinguish different versions of the data value of a block over time), and a per-page *major* counter (to avoid overflow of counter values). For the rest of the paper, we assume counter mode (processor-side) memory encryption, similar to prior studies [43, 50, 51].

3 OBFUSMEM DESIGN

3.1 Trust Architecture

An ORAM system's processor is trusted, meaning that we rely on the correct operation of the processor to ensure ORAM's security and privacy characteristics. ObfusMem relies not only on the correct operation of the processor, but also on the logic in memory to perform essential cryptographic operations. Thus, ObfusMem's *trusted computing base* (TCB) includes both the processor and the memory. The manufacturers of both components are trusted. The rest of this section discusses how the trust architecture of an ObfusMem system is bootstrapped from its TCB.

First, we assume that the processor and the memory are both trustworthy components, in the sense that they are manufactured to the specifications and do not contain design errors and faults that may introduce security vulnerabilities. Next, the processor and memory must be able to communicate securely. ObfusMem uses a cryptographically protected channel for this purpose. To facilitate the creation of this channel within an integrated system, the manufacturers of the processor and memory must generate a public/private cryptographic key pair for each component and burn those keys into every chip they produce. Manufacturers must ensure that only the public keys can be read out from the chip pins. In effect, the OEM certification process is augmented, as each manufacturer serves as a certification authority for the cryptographic keys it burns into the components it produces. The processor and memory manufacturers do not need to coordinate key generation or binding with one another; they do not even need to know one another.

After a system is built by a system integrator, powered on and begins to boot, the processor and memory must establish a cryptographically authenticated and private communication channel. This requires each to learn the public key of the other. There are several approaches to achieve this, ranging in implementation complexity and trust assumptions. In the first (*naive*) approach, trust between the two components is bootstrapped during execution of a BIOS protocol in which the two components exchange public keys in the clear. However, this assumes the bootstrapping process to be isolated from any physical attacks. Since bootstrapping may occur at the user's operational site, it may be difficult to ensure such isolation.

In the second (*trusted system integrator*) approach, the system integrator programs the processor's public key into the memory chips and the public keys of the memory chips into the processor chip. These keys can be burned into write-once non-volatile registers inside the processor and memory chips. If the chips have Trusted Platform Modules (TPMs), the keys may be placed inside the TPMs of their respective chips. The system integrator is trusted to (1) install ObfusMem-capable chips, and (2) correctly program the public keys into the corresponding components. When the system is shipped to a user, the processor and memory chips already know the cryptographic identities of the components with which they should communicate in order to establish secure communication. The approach limits, but does not prevent component upgrades after system integration. By provisioning spare registers, a system designer can enable a limited number of component upgrades during the system lifetime. Component upgrades require that the system integrator responsible for the upgrade burn additional public keys into the spare registers to identify the newly introduced components to their counterparts.

In the third (*untrusted system integrator*) approach, we use subsystem attestation to verify that an ObfusMem-capable processor is communicating with an ObfusMem-capable memory. This approach relies on both processor and memory having an attestation capability similar to that provided by Intel SGX [30]. While this is realistic given today's processors, the assumption about memory capabilities is very aggressive. The untrusted system integrator performs the same key burning activities as would a trusted system integrator in the second approach. On system boot, both processor and memory use an attestation process to verify that the provided keys match those belonging to their counterparts. If the system integrator had maliciously or erroneously burned the wrong keys into the registers, the attestation process would fail, alerting the components to the problem and leading to system failure. Roughly, the processor measures itself, signs the measurement and makes the signed measurement available for other system components to view. The measurement includes the following items that are critical to bootstrapping trust in ObfusMem: the hardware and firmware characteristics of the chip that indicate their ObfusMem capability and the public key of the processor (installed by the processor manufacturer). The memory can use the measurement to verify that the public key of the processor is the same as the public key that the untrusted system integrator burned into its own register. An identical process is followed by the memory and processor in the "opposite direction" (i.e., memory measures and signs, then processor verifies). Component upgrades are handled in the same way as in the trusted system integrator approach.

We leave the possibility of a fourth (*boot-time key generation*) approach to future work. In this approach, the public/private key pairs identifying the components are derived by the processor and by the memory as each component boots independently. This requires

that both processor and memory have SGX-like key generation capabilities and may enable the system to take advantage of key management, key revocation and component upgrade capabilities. While such an approach could be tremendously flexible, independent boot capability is incompatible with the boot model of current systems.

Of the first three approaches we discussed above, we do not recommend the naive approach, as the assumption of physical isolation is inconsistent with the fundamental motivations for using ObfusMem. If the system integrator is assumed to be trustworthy, the trusted system integrator approach is relatively simple to implement and does not rely on specialized hardware. In order to use the untrusted system integrator approach, one must have SGX-like attestation capabilities in both processor and memories and bear additional implementation complexity and cost. If the system integrator cannot be trusted, then this approach is necessary.

We also do not recommend alternative approaches by which the trusted systems integrator generates a symmetric key and installs it in the various components, as this places even more trust in the integrator. A shared secrets approach also prevents the system owner/user from verifying the authenticity of the system components, offering weaker security and privacy guarantees.

The final step in establishing the ObfusMem trust architecture is for the processor and memory to establish an authenticated and private communication channel. During BIOS execution, the hardware controller in the processor will initiate a Diffie-Hellman key exchange [35] with each of the memory controllers in the memories to establish a different *shared session secret key* with each controller. Each shared secret key enables a private communication channel between processor and memory. If there is more than one memory chip, then the processor must keep track of multiple secret keys, one corresponding to each memory chip. The shared secret key enables a private, authenticated communication channel that persists until the system shuts down. When the system is re-booted, the BIOS execution results in a new shared secret key for the communication channel.

Note that the public key cryptographic operations to bootstrap a trustworthy communication channel between components are executed only onceâĂŤat system boot time. The components generate and exchange symmetric session keys encrypted with the public keys. This establishes private communication channel and then normal system operation is accomplished via higher performance symmetric key cryptography.

The ORAM TCB includes the processor, but not the memory, so the step of establishing authenticated, private communication between these components is not needed. While ObfusMem requires a larger TCB with trust bootstrapping during BIOS execution, it offers the benefits of simplicity in obfuscating the access pattern, zero storage overheads, suitability for emerging memory technologies, and very low performance overheads.

3.2 Access Pattern Obfuscation

So far, we have described how a processor and memory establish a secure communication channel. Now, we discuss how the memory access pattern is encrypted in this channel. Our goal is to obfuscate four aspects of the access pattern: *spatial pattern* (an access to a neighboring block should not be distinguishable from an access to a faraway block), *temporal pattern* (an access to a previously accessed block should not be distinguishable from an access to a different block), *type of access* (a read or write accesses must not be distinguishable), and *memory footprint* (the number of distinct blocks accessed should not be inferable in the long run).

Several approaches can be used to encrypt the addresses and commands on the memory bus. The first approach uses a simple *Electronic Code Book* (ECB) encryption mode, where encrypting address X results in address $Y = E_{Key}(X)$. ECB is sufficient for obfuscating spatial locality across blocks because block addresses are randomized through encryption. However, ECB does not hide the temporal access pattern, because two requests showing the same ciphertext address point to the same plaintext address. Furthermore, memory footprint is also leaked because the number of unique ciphertext addresses appearing on the bus reveals the actual number of accessed cache blocks. Finally, ECB is vulnerable to dictionary attacks; the attacker can use the frequency distribution of accesses to different addresses in ciphertext to match it against the that of the plaintext.

A second approach uses non-ECB memory encryption modes. While many modes are possible, *counter mode* is especially appealing because we can pre-generate encryption pads for future use, since future counter values are known ahead of time. In an AES-CTR mode, an *Initialization Vector* (IV) is encrypted to generate a pad. The IV is implemented as a single use counter called a *nonce*. Later, the generated pad is XORed with the address and command. The resulting ciphertext Y is $Y = E_{Key}(IV) \oplus X$. The IV (counter) changes after every memory access, as shown in Figure 3(a). Note that a 64-bit counter is large enough that it will not overflow for millennia.

Figure 3(a) shows how a request resulting from the last level cache (LLC) write back, consisting of command and address, and data, is obfuscated. Data is passed to the processor encryption (Step 1a) to obtain its ciphertext. In the case of LLC read/write miss, there will not be data involved. In parallel, the address of the request is used to index the Session Key Table to extract the session key for the memory chip which will be handling this request (Step 1b). The command and address are also passed to the next step (Step 1c), while a dummy request generator generates a matching request consisting of dummy command and address, and dummy data (Step 1d – to be explained later). The session key is fetched from the table and is used as an encryption key for the encryption engine (Step 2). The counter value Ctr and subsequent $Ctr+1, Ctr+2, \ldots, Ctr+5$ were generated and input to the encryption engine to generate six 128-bit pads (Step 3). The reason for six pads is that one pad will be used to encrypt the original request address by XORing the pad and command and address (Step 4a), one pad will be used to encrypt a dummy request (Step 4b), and four pads (16-byte each) are used to encrypt the 64-byte block encrypted data (Step 4c). Finally, the counter is increased by six to prepare it for the next encryption (Step 4d).

Note that in ObfusMem, the processor-encrypted data is encrypted a second time before it is sent out to the memory. Skipping this second encryption allows observers to probabilistically infer possible

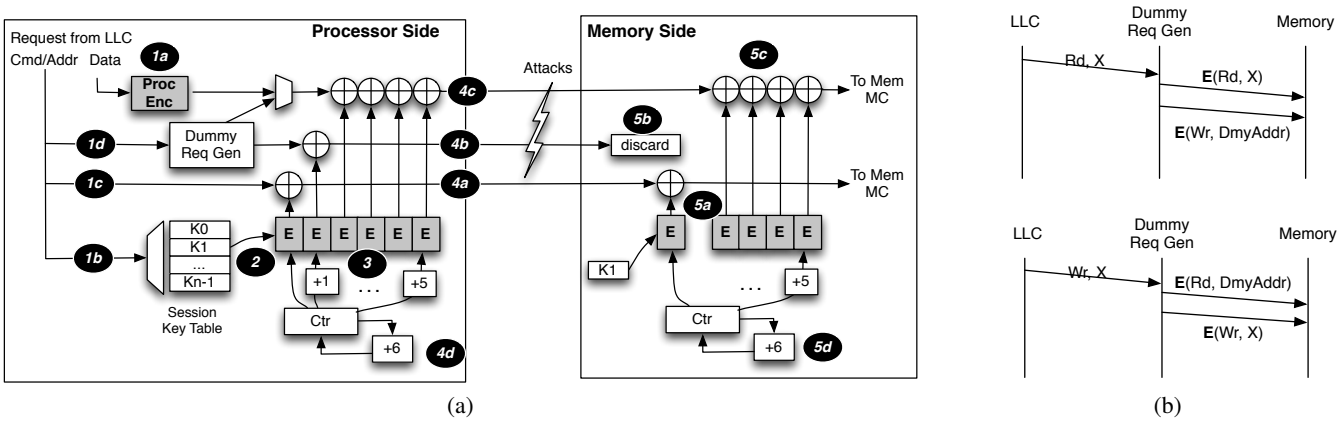

Figure 3: Part (a): ObfusMem access pattern obfuscation for a write request (a write back of the LLC block). A read request due to read/write miss of the LLC is handled similarly in the reverse direction (not shown). Part (b): Illustration of dummy request generation.

temporal reuse of data: two different requests with the same ciphertext being transmitted on the bus indicates a higher than random chance that the requests are to the same address, especially if data is unmodified. By encrypting data for the second time, even unmodified data will appear different every time it appears on the memory bus.

OBSERVATION 1. *Encrypting the access pattern is not sufficient for obfuscation, because accesses to encrypted data can still reveal temporal reuse between requests. To avoid revealing temporal reuse, ObfusMem re-encrypts data ciphertext before transmitting on the memory bus.*

At the memory side, encrypted data sent from the processor is decrypted. The session key and counter values are used to generate pads. One pad is used to decrypt the command and address of the original request (Step 5a), the dummy request is discarded (Step 5b – to be explained later), and the data is decrypted using four pads (Step 5c). Finally the counter is incremented by six to keep it synchronized with the processor-side counter and to prepare it for the next decryption (Step 5d).

3.3 Obfuscating the Type of Memory Requests

While we are not aware of security attacks solely based on the type of memory requests, observers may improve their odds of attack success using knowledge of the request type. In typical Path ORAM designs, the operation types are indistinguishable because every access is treated equally by reading a path and then filling the path from the stash. To address this, ObfusMem piggybacks every read operation with a *dummy write* operation. Similarly, every write operation is preceded by a *dummy read* operation. This is illustrated in Step 1d in Figure 3(a). Accordingly, every access now appears as a *read-then-write* request to an external observer. The alternative, *write-then-read*, is also possible. However, write operations are typically due to block evictions from Last-Level Cache (LLC) and hence are not in the critical path delay of instruction execution. In contrast, reads are usually in the critical path. Thus, read-then-write for a read operation can proceed faster and return the data to the LLC after it is fetched from memory.

Another design question is what address should be assigned to a dummy request. In a *random address* design, a dummy request is assigned a random address. However, an access to a random address reduces performance and wastes energy due to the loss of temporal locality. In a second *original address* design, the dummy request uses the same address as the original request. Because ObfusMem uses counter mode encryption, the original address will be encrypted into different ciphertext addresses in the original vs. dummy requests. With this approach, the NVM temporal locality is not reduced. Unfortunately, it is still expensive: every read now incurs an actual write to the NVM, reducing the NVM lifetime and wasting energy. To overcome this problem, we choose a third *fixed-address* design, where we set aside a location of the memory to hold the dummy (Figure 3(b)). Specifically, every memory module will reserve one 64-byte block as dummy. Thanks to counter-mode encryption, the dummy address appears different each time it is encrypted due to the change of the encryption pad. Furthermore, the fixed-address design enables *request dropping*: when the memory module decrypts the request and finds that the request is to a dummy address, it discards the actual writing of the data to the dummy location (Step 5b of Figure 3(a)), hence saving write energy and avoiding premature wear-out. Likewise, if instead the request is a dummy read, random data can be returned and discarded at the processor.

OBSERVATION 2. *Dummy write (or read) requests must be generated to hide whether a real request is a read (or a write). To eliminate energy and endurance overheads caused by dummy write requests, we can use a fixed dummy address for dummy requests and drop them upon arrival at the memory.*

An alternative approach to ours is making read and requests indistinguishable by requiring a read request to provide dummy data (just as a write), and requiring data reply for a write request (just as a read). While this can achieve the same security guarantee as our

design, it incurs higher bus bandwidth utilization compared to our design. Specifically, our split request allows an optimization where dummy reads/writes can be substituted with real requests, which removes bandwidth overheads in such cases, whereas the former scheme incurs bandwidth overheads regardless.

3.4 Obfuscating Inter-Channel Access Patterns

So far, our discussion has assumed a single memory channel. However, a realistic 3/2.5D stacked memory may have multiple channels, where channels operate independently and use different pins. For example, High Bandwidth Memory allows 4-8 independent channels using separate pins [6]. While our discussion in this section is not specific to a particular 3/2.5D stacked memory, we will address the security challenge that comes from having multiple memory channels.

The spatial access pattern may leak across memory channels, because obfuscation in ObfusMem is performed independently on a channel. In a multi-channel memory, addresses are interleaved across channels to improve memory-level parallelism, bandwidth utilization, and achieve an error correction target. Accesses to different channels appear on different physical pins so they cannot be obfuscated easily. Knowing the interleaving granularity, an attacker can infer spatial access patterns by observing accesses being sent out to different channels as they appear in different pins, without knowing anything about addresses or data.

One approach to obscuring access patterns across channels is to generate dummy requests on other channels. To illustrate, suppose that there are four channels and that there is a request on channel 2. Dummy requests can be created on channels 0, 1, and 3, so that observers cannot tell which channel is receiving a real memory access request. However, note that this *full channel dummy replication* scheme has a cost that increases linearly with the number of channels. Due to the cost, we explore another approach. Note that observers cannot infer spatial locality if all the channels are accessed simultaneously and equally often. From our example above, if channels 0, 1, and 3 already have requests, there is no need to generate dummy requests for them. Dummy requests are needed only for idle channels. With this *idle channel dummy replication* scheme, at any time there is a memory request to be serviced on one channel, the processor-side ObfusMem controller checks the status of all other channels and injects dummy requests on the idle ones.

OBSERVATION 3. *Observers may infer access patterns across memory channels unless channels are accessed simultaneously and equally often. Injecting dummy requests to fill idle channels at the same time a real request is issued on one channel prevents such inferences. When memory channel bandwidth utilization is high, few dummy requests are needed. Write amplification overhead is avoided by discarding dummy writes at each channel in the memory.*

3.5 Integrity Verification of Processor-Memory Communication

So far, we have discussed access pattern obfuscation when the attacker can only passively observe anything communicated between the processor and memory. However, an attacker may have an additional active capability for communication tampering via changing the request type, address, or data of the request. The attacker may drop a request completely, inject a bogus request, replace a request with a bogus one, or replay a request from the past. Access pattern obfuscation makes meaningful attacks more difficult because the attacker does not know the type, address, or data of a request that can lead to accessing a specific location. Also an attacker cannot easily select a request for injection whose content will be meaningful when decrypted. If a request or data reply is removed, counters in the processor and memory may not match, preventing the processor and memory from any further meaningful communication. Since ObfusMem uses counter mode encryption and counters are never reused, valid requests from the past cannot be replayed meaningfully either. Thus, an attacker's impact is limited to a denial of service (DoS) by sabotaging ObfusMem.

While a successful active attack against an ObfusMem-protected system is difficult, *detecting* the incidence of attempted active attacks (especially if promptly) may be valuable to protect user data from a determined attacker that will seek alternative avenues of attack. We now consider attack detection. Note that if the integrity of memory is protected using a Merkle Tree [43]. A Merkle Tree can detect unauthorized modifications to data or counters stored in memory, when data is brought onto the processor to be verified. Tampering to memory requests can be indirectly detected if they lead to unauthorized changes in data. Ideally, detection of tampering should be immediate and cover any modifications to memory requests. To achieve that, we can employ an additional integrity verification mechanism protecting memory requests.

An *encrypt-then-MAC* approach for protecting a message from tampering is to apply a one-way hash function, such as MD5 [42] or SHA-1 [17], over a message. In our context, the message sent from/to the processor to/from the memory is the encrypted memory request. Mathematically, the message is $M = E_K \oplus (r|a|D)$, where E is encryption, K is the encryption key, r is the request type, a is the request address, and D is the processor-encrypted data block of the request. If the MAC function is H and $\alpha = H(M)$, then the processor sends M and α to memory. Upon receiving them, the memory computes $H(M)$ and compares it against α. Any mismatch indicates tampering of M or of α.

Note that the encrypt-then-MAC approach adds significant latency to the critical path of the request to memory and reply from memory, as the encryption must be completed before the MAC can be computed. Thus, we adopt an alternative *encrypt-and-MAC* approach, where the MAC function is not applied over the message, but applied to the plaintexts of components of the message. In this case, the MAC is $\beta = H(r|a|c)$, where c is the counter value. The MAC can be computed early since its components are available early. For example, we can anticipate that a dirty block near the least recently used (LRU) position in the LLC will generate a memory write request soon. A stream or stride predictor can be used to anticipate a future LLC read/write miss, which will generate a memory read request. So, we can compute β before the memory request is generated, allowing overlap of request encryption and MAC generation. At the memory side, when M and β are received, it can decrypt M to obtain r and a, and use its own counter c to recompute $H(r|a|c)$. A mismatch between $H(r|a|c)$ and β indicates tampering. One drawback of the approach is that tampering of data is relegated to Merkle Tree verification. This means that tampering of data brought into the

processor chip will be detected nearly instantly. However, tampering of data that is written to memory will not be detected until the data is eventually read into the processor chip.

Now let us examine several tampering scenarios. If the attacker changes the message to M', the memory will obtain a wrong request r' or address a' after decryption. When the memory recomputes the MAC, it will calculate $H(r'|a|c)$ or $H(r|a'|c)$, which does not match β, so tampering is detected. If the attacker deletes a request or a reply, the counter in the processor and memory will not match. Suppose that the processor counter is c while the memory counter is c'. In this case, the memory computes $H(r|a|c')$ which does not match β, so the tampering is also detected. Finally, if the attacker replays an old valid message, the computed MAC will again fail to match β because the memory will use its fresh counter value to compute the MAC, while β reflects a stale counter value. Overall, any types of request/command tampering will be detected immediately, unless the system crashes prior to detection. Because the attacker must know the original components of the encrypted command to successfully tamper with communication between the processor and memory, a lightweight MAC function is sufficient to detect tampering. We assume MD5 in our implementation, however, other MAC functions could be used, instead.

OBSERVATION 4. *The encrypt-and-MAC approach enables higher performance through overlapping MAC generation and request encryption, compared to the encrypt-then-MAC approach. However, detection of tampering of data written to the memory is delayed until the data is eventually brought back into the processor.*

4 METHODOLOGY

To evaluate ObfusMem, we constructed a Gem5-based full-system cycle-accurate simulator [11]. To enable performance comparison with ORAM, we chose representative benchmarks from the SPEC CPU2006 suite [1]. The characteristics of the benchmarks, such as Instructions Per Cycle (IPC), Last-Level Cache Misses Per Kilo Instructions (LLC MPKI) and average latency gap between consecutive memory requests are shown in Table 1. Eight of the benchmarks have average memory request gaps of less than 100ns, indicating very high memory request rates, while seven of the benchmarks have average memory request gaps higher than 100ns. We warmed-up the simulator and fast-forwarded the workloads to skip the initialization phase, then simulated 200 million instructions.

Table 1: Characteristics of the evaluated benchmarks.

Benchmark	IPC	LLC MPKI	Avg Gap (ns)
bwaves	0.59	18.23	44.32
mcf	0.17	24.82	74.95
lbm	0.35	6.94	67.97
zeus	0.53	4.81	63.56
milc	0.42	15.56	51.54
xalan	0.52	0.97	945.62
omnetpp	4.30	0.10	1104.74
soplex	0.25	23.11	69.06
libquantum	0.33	5.56	146.82
sjeng	0.95	0.36	1382.13
leslie3d	0.49	9.85	58.91
astar	0.70	0.13	5660.18
hmmer	1.39	0.02	2687.60
cactus	1.05	1.91	128.09
gems	0.40	11.66	66.25

To ensure a fair comparison of ObfusMem to ORAM, we model ORAM performance by choosing optimistic assumptions for ORAM implementations. We assume a base Path ORAM of 25 levels, 4 blocks per bucket and a capacity waste of 50%, similar to previous studies [16, 52]. For ORAM, we assume a fixed memory access latency of 2500 *ns*, obtained by extrapolating the ORAM access latency from [20]. Our latency assumption is optimistic, as we include the process of reading and evicting a full tree path to the memory assuming unlimited bandwidth and unconstrained phase-change memory (PCM) write power budget. Note that the ORAM access latency depends on the memory layout, the probability of finding a block in the leaf, applying bandwidth reduction techniques (e.g., XOR) and many other parameters (bucket metadata size, row buffers locality, etc.), hence our optimistic estimate.

The machine configuration is shown in Table 2. We model a 4-core processor with a three-level cache hierarchy and a DDR-interfaced PCM memory with PCM parameters from [32]. Similar to the design in [32], writes to PCM cells are incurred only when evicting data from dirty row buffers.

Note that memory encryption is orthogonal to access pattern obfuscation and is required in all protected systems, including ORAM. In both cases, we overlap fetching data with encryption pad generation.

Table 2: Configuration of the simulated system.

Processor	
CPU	4 core, each 2GHz, out-of-order x86-64
L1 Cache	private, 2 cycles, 32KB, 8-way, 64B block
L2 Cache	private, 8 cycles, 512KB, 8-way, 64B block
L3 Cache	shared, 17 cycles, 8MB, 8-way, 64B block
Coherence	MESI protocol
DDR-based PCM Main Memory	
Capacity	8 GB
# Channels	1 (base), 2, 4 and 8
Channel bw	12.8 GB/s
PCM Latencies	60ns read, 150ns write, based on [32]
Organization	2 ranks/channel, 8 banks/rank, 1KB row buffer, Open Adaptive page policy, RoRaBaChCo address mapping
DDR Timing	tRCD 60ns, tRP 150ns, tBURST 5ns tCL 13.75ns, 64-bit bus width, 800 MHz Clock
Operating System	
OS	Gentoo, Linux, kernel v2.6.22.9
Encryption Parameters	
Counter Cache	5 cycles, 256KB size, 8-way, 64B block

For the memory encryption, we model a counter cache with 5-cycle access latency and a size of 256KB. For ObfusMem, we model ObfusMem's AES-CTR latency based on our synthesize results using 45*nm* libraries and a publicly available pipelined AES-128 implementation [38]. The overall AES encryption latency is 24 cycles (cycle time of 4*ns*) and the AES unit can produce a 128-bit on each cycle. The AES engine consumes 15.1 mW. The required area is 0.204 mm^2. Each channel requires an AES engine on both sides (processor and memory). To enable detection of communication tampering on the channel, we use a publicly available 64-stage pipelined implementation of MD5 [39]. Our synthesize results show that the MD5 unit consumes about 12.5 mW and has an area overhead of 0.214 mm^2.

5 EVALUATION

5.1 Performance Overhead

ObfusMem and ORAM provide similar protection to the memory bus, hence our first evaluation results compare their overhead for such protection. Table 3 shows the execution time overheads of ORAM and ObfusMem with authenticated communication (ObfusMem+Auth) over unprotected execution, and the speedup ratio of ObfusMem+Auth over ORAM. We can see that the overheads of ORAM vary across benchmarks depending on their LLC MPKI: benchmarks with high LLC MPKI suffer from large ORAM execution time overheads. Across all of the benchmarks, ORAM adds 946.1% to the execution time of programs, or a slowdown ratio of slightly over one order of magnitude. This is in line with results reported for the common benchmarks in the literature [20]. In contrast, ObfusMem+Auth adds 10.9% to the execution time, on average, and 32.1% in the worst case. Note that these overheads already include counter-mode memory encryption, ObfusMem's access pattern obfuscation, and authenticated memory bus communication. ObfusMem+Auth achieves speedups ranging from 1.3× to 17.1×, with an average of 9.1×. This very substantial improvement makes a significant difference in the feasibility of incorporating obfuscation of access patterns in general purpose computer systems.

Table 3: Execution time overhead comparison of ORAM vs. ObfusMem

Benchmark	ORAM	ObfusMem+Auth	Speedup
bwaves	1561.0%	18.9%	14.0×
mcf	1133.3%	32.1%	9.3×
lbm	1298.6%	12.5%	12.4×
zeus	1644.3%	14.9%	15.2×
milc	1846.6%	28.4%	15.2×
xalan	137.7%	0.8%	2.4×
omnetpp	64.96%	1.2%	1.6×
soplex	1878.6%	15.7%	17.1×
libquantum	604.8%	2.9%	6.8×
sjeng	152.5%	1.1%	2.5×
leslie3d	1626.6%	15.1%	15.0×
astar	30.7%	0.1%	1.3×
hmmer	86.6%	0.0%	1.9×
cactus	784.8%	5.2%	8.4×
gems	1340.9%	14.3%	12.6×
Avg	**946.1%**	**10.9%**	**9.1×**

Next, Figure 4 breaks down the execution time overheads of ObfusMem into various levels of security, starting from memory encryption only without access pattern obfuscation, plain ObfusMem, and ObfusMem with authenticated communication.

OBSERVATION 5. *Overall, ObfusMem+Auth is almost one order of magnitude faster than ORAM. Furthermore, roughly a quarter of ObfusMem overheads come from the memory encryption (2.2% vs. 8.3%), while communication authentication increases the overhead of ObfusMem only slightly, from 8.3% to 10.9%, as authentication can be partially overlapped with encryption.*

As discussed in Section 3.4, one of the key design aspects of ObfusMem is to obfuscate the inter-channel access pattern, through

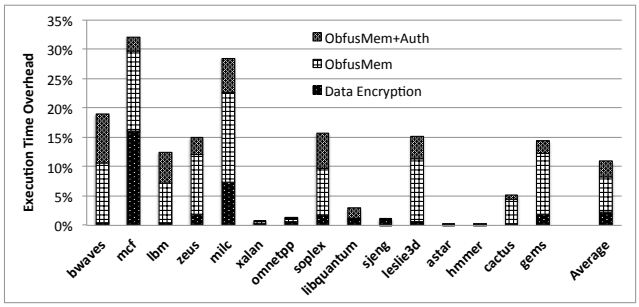

Figure 4: The execution time overhead of ObfusMem, normalized to unprotected system.

dummy request injection across all channels (ObfusMem-UNOPT) or only on idle channels (ObfusMem-OPT). The execution time overheads of these two injection strategies are evaluated with various numbers of channels, with and without communication authentication. Figure 5 shows these results. The overhead of ObfusMem-UNOPT reaches up to 18.8% and 16.3% (with eight memory channels), with and without authentication, respectively. In contrast, the overhead of ObfusMem-OPT reaches up to 13.2% and 10.1% with and without authentication, respectively.

OBSERVATION 6. *Inter-channel access pattern obfuscation is necessary, but potentially costly in bandwidth usage. Performance overhead is minimized by using optimized ObfusMem which injects dummy requests only when necessary. As the number of memory channels increases, this optimization is increasingly critical to limit performance overhead.*

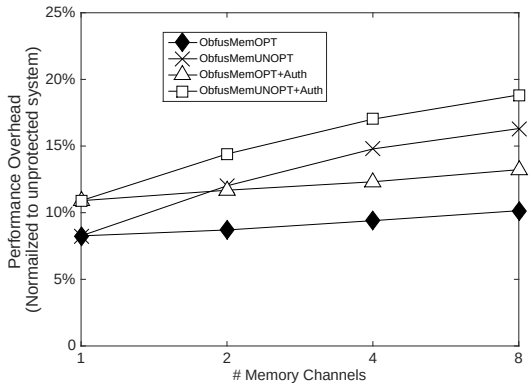

Figure 5: The impact of the number of channels on ObfusMem performance, compared to unprotected system with equal number of channels.

5.2 Impact on Memory Energy and Lifetime

One of the key concerns with main memory NVMs is limited and energy-consuming writes. However, a typical Path ORAM implementation is based on reading a whole tree path that consists of about

100 blocks (assuming $L = 24$ and $Z = 4$) and then evicting it later. Assuming PCM cells' write energy is 6.8× the read energy [32], a basic ORAM implementation may incur $(1 + 6.8) \times 100 = 780\times$ the read energy, while ObfusMem incurs only $\frac{1+6.8}{2} = 3.9\times$ the read energy, assuming 50:50 read:write distribution (a 200× PCM energy reduction). More importantly, ObfusMem does not incur any extra writes, which can improve the lifetime of the system by roughly 100× compared to base ORAM, which requires 100 blocks to be evicted after each access, regardless of the access type (read or write).

However, the savings are not limited to the memory read/write energy, but also include the encryption energy. Path ORAM requires 100 blocks to be read and decrypted, then later encrypted and written back. This costs $200 \times 4 = 800\times$ the energy for 128-bit encryption. In contrast, ObfusMem requires two 128-bit address encryption pads (dummy and real), and four 128-bit encryption pads for data point-to-point encryption, in addition to the four 128-bit pads used for data encryption. This adds up to ten 128-bit pads for the processor side of the channel and six 128-pads for the memory side, i.e., 16 pads per channel. Even though having a high density NVM DIMM will reduce the number of required channels, having a 4-channel system will require 64 128-bit pads in the worst case, which is 12.5× fewer than that required for a typical ORAM (800 128-bit pads). When other channels are already busy with real requests (the best case). ObfusMem only requires 16 128-bit pads (a 50× reduction in the number of pads)

6 SECURITY ANALYSIS AND DISCUSSION

6.1 Security Analysis

We will now analyze the security protections provided by ObfusMem and compare them with those provided by ORAM. As discussed in Section 3.2, access pattern information leakage includes spatial patterns, temporal patterns, type of requests (read vs. write), and memory footprint. Table 4 shows a comparison of ORAM and ObfusMem. ORAM obfuscates the temporal pattern exhibited by a program by randomly remapping the block to another path in the tree. Spatial pattern is hard to infer as well due to mapping sequential addresses randomly on tree paths. However, this is predicated on ORAM PosMap initialized randomly at start up, and PosMap content being secret. PosMap secrecy and random initialization require additional mechanisms, such as memory encryption, or placing it on a separate ORAM. In contrast, ObfusMem obfuscates the spatial and temporal patterns by encrypting addresses with counter-mode encryption. No meta-data is needed in ObfusMem beyond what is required for memory encryption and integrity verification. Furthermore, while the role of memory encryption is central in ORAM (to keep PosMap content secret), in ObfusMem it is only required if the threat model includes attacks on data within memory chips. Attacks on exposed wires on processor-memory bus is already protected through ObfusMem.

Read accesses are indistinguishable from write accesses in both ORAM and ObfusMem: ORAM treats both access types the same way, by reading/writing from/to old/new paths, while ObfusMem inserts a dummy read/write for a write/read request. There is a difference in the treatment of a non-temporal store. In ORAM, the entire path for the block must be brought on chip, just like a temporal store instruction. In ObfusMem, a non-temporal store does not cause data blocks to be read on chip, avoiding cache pollution.

Table 4: Comparing ORAM and ObfusMem.

Aspect	ORAM	ObfusMem
Spatial pattern	Full	Full
Temporal pattern	Full	Full
Read vs. write	Full	Full
Memory footprint	Full	Full
Command authentication	No	Yes
TCB	Proc only	Proc+Mem
Exe time overheads	946%	11%
Storage overheads	100%	0%
Write amplification	100×	None
Deadlock possibility	Low	Zero
Component upgrade	Easy	Harder

The memory footprint size may be used by attackers to infer characteristics of a program, e.g. to infer the number of data records or the graph size. Both ORAM and ObfusMem hide the memory footprint: if the application makes n accesses, memory footprint size M is bounded by $1 \leq M \leq n$. As the program's run time increases, the accuracy of the attacker's estimate of M decreases. In theory, both schemes can generate a large number of dummy path accesses to eliminate this issue.

As discussed in Section 3.5, ObfusMem can be equipped with command authentication with roughly 2% additional overhead. This allows ObfusMem to detect immediately the incidence of tampering of memory requests. Typical ORAM implementations do not include an equivalent protection. Such a protection may be added to ORAM, but at an additional cost to already high execution time and bandwidth overheads.

6.2 Discussion

Summary comparison of ObfusMem to ORAM. Our comparison of ObfusMem to ORAM is summarized in the bottom half of Table 4. The Trusted Computing Base (TCB) of ORAM includes the processor but not the memory, while ObfusMem adds the memory to its TCB. The execution time of programs on ORAM is roughly one order-of-magnitude slower than unsecured processors but only 11% slower on ObfusMem. The storage overhead of ORAM is $\geq 100\%$ (i.e., $\geq 50\%$ of the memory capacity is unusable) vs. zero in ObfusMem (excluding the storage of counters needed for memory encryption, which is not a part of ObfusMem). ORAM amplifies writes by about 100× which is harmful for NVM life time and energy consumption, versus zero write amplification for ObfusMem. Whole system deadlocks are possible (but can be made unlikely) in Path ORAM due to tree bucket overflows; such deadlocks are not possible in ObfusMem [1]. Memory component upgrade is easier in ORAM since the TCB includes only the processor. Memory component upgrade is possible but harder with ObfusMem due to a larger TCB and system integrator involvement. However, as most computer

[1] A deadlock is distinct from a denial of service (DoS): deadlock arises when a system cannot make forward progress under normal system operations, while DoS stalls or slows down forward progress under attacks. ObfusMem does not suffer from deadlocks but it is not designed to prevent DoS.

systems do not get component upgrades during their service life, most systems are unaffected by upgrade complexity.

Multi-socket and multiprocessor systems. For systems with multiple processor chips and sockets, the operating system (OS) treats the memory attached to each socket as a separate pool, arranged as a Non-Uniform Memory Access (NUMA) system. Each memory node may be accessed by multiple processors, causing at least two problems. First, the cache coherence protocol may cause a processor to communicate directly with other processors, without involving memory. Second, accesses by any of the processors to different memory modules may cause the inter-socket access pattern to leak. Neither ObfusMem nor ORAM explicitly address these problems, which may require a processor-to-processor communication protection scheme such as [44]. Adapting such a scheme to ObfusMem is left for future work.

Thermal and timing side-channel leakage. Side channels may allow attackers to correlate system timing, power, thermal, or electromagnetic characteristics with the access pattern or data. We will very briefly discuss thermal and timing side channels, but largely leave it for future work. That ObfusMem does not reshuffle data locations in the main memory is its key advantage (resulting in low overheads) but also allows attackers to thermally analyze the memory chips to infer which rank, bank, row, etc. are activated. ORAM's reshuffling incurs great costs but makes thermal side channel analysis harder. Timing side-channel attacks include the observation of the asymmetric timing of row hit or miss, bank conflict, etc. Neither ObfusMem nor ORAM deal with them. However, there is no fundamental reason why ObfusMem cannot be paired with timing protection techniques such as [19, 21, 28]. For example, ObfusMem accesses can be made timing oblivious by spacing timing of requests, assuming worst timing case, and not dropping dummy requests. Since ObfusMem performance overheads are small, there is room for incorporating timing protection techniques.

7 RELATED WORK

Exploiting on-die logic for security. Gundu et al. [46] proposed using near-data computing capability in future memory technologies to implement data authentication. Our work is different in that we obfuscate the access pattern. Our work is orthogonal to [46] and can potentially be integrated with it.

Secure processor designs. Much prior work considered designing efficient secure processor systems where the TCB only includes the processor, and memory content is encrypted and protected with integrity verification [9, 43, 50, 51]. ObfusMem uses memory encryption to protect data, and additionally provides access pattern obfuscation.

Obfuscating access pattern. Obfuscating the memory access pattern has been a long time research challenge [23, 24]. The concept of Oblivious RAM was proposed by Goldreich et al. [23, 24]. Later, a practical realization of the concept of ORAM was proposed in Path ORAM [47]. Path ORAM has been a target of optimization for several state-of-the-art implementations of ORAM [16, 20, 40, 52]. We contrasted ObfusMem with ORAM and compare their security properties in Section 6.1 and characteristics in Section 6.2. Prior to Path ORAM, there were studies looking into hardware support for permuting the address space at chunk-level granularity [22, 53, 54]. Such approaches obfuscate the blocks within a specified small chunk (typically 64KB) using hardware support. Because of their small granularity, their performance overheads are much smaller than those of ORAM. In contrast, ObfusMem obfuscates the access pattern of the whole memory but at a low performance overhead. A concurrent and most closely related work to ours is InvisiMem [7]. InvisiMem shares a similar approach of adding the logic layer of smart memories to the TCB and obfuscating the memory access pattern using memory request encryption. While InvisiMem uses the same packet size for read and write requests with providing dummy data to hide the actual operation type, ObfusMem uses split dummy requests with the option of dummy request dropping, thus enabling a better bus utilization when there is a heavy load of read and write requests. Finally, our paper discusses trust architecture choices and security analysis in greater details.

Other studies focused on preventing information leakage due to access timing [19, 21, 28, 46]. These proposals study efficient ways to hide the information leak due to the timing of the memory accesses through scheduling mechanisms and memory controller modifications. ObfusMem is orthogonal to them and can potentially be integrated efficiently with them.

8 CONCLUSION

Obfuscation of memory access patterns offers important hardware-level protection for both system security and user privacy. This paper has proposed ObfusMem, a memory access pattern obfuscation system, which offers overhead advantages in comparison with ORAM. ObfusMem relies on cryptographic logic in both processor and memory to encrypt and authenticate commands, addresses, and data that are communicated via the exposed wires connecting the processor and memory. Our evaluation showed that ObfusMem incurs only 11% execution time overhead on average above the cost of unprotected memory accesses, making memory access pattern obfuscation feasible for commodity computer systems. ObfusMem is nearly an order-of-magnitude faster than ORAM. Critically, for non-volatile memory, ObfusMem does not cause early wear-out.

9 ACKNOWLEDGEMENT

This work was done when Amro Awad and Yipeng Wang were PhD students at North Carolina State University. Shands's and Solihin's work was supported by (while they were serving at) the National Science Foundation. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Finally, we thank Edward Suh, Dmitry Ponomarev, and anonymous reviewers, for their feedback.

REFERENCES

[1] 2000. SPEC, SPEC CPU2000 and CPU2006, http://www.spec.org/. (2000).
[2] 2011. HMC Consortium. http://www.hybridmemorycube.org/technology.html. (2011).
[3] 2014. 4RCD0124K DDR4, http://www.idt.com/. (2014).
[4] 2014. CAB4A DDR4, http://www.ti.com/lit/ds/symlink/cab4a.pdf. (2014).
[5] 2014. Published U.S Patent Application. Dynamic partial power down of memory-side cache in a 2-level memory hierarchy, PCT/US2011/066302. (2014). https://www.google.com/patents/US20140304475
[6] 2015. JEDEC Specifications for HBM, JESD235A. http://www.jedec.org/standards-documents/docs/jesd235a. (2015).

[7] Shaizeen Aga and Satish Narayanasamy. 2017. InvisiMem: Smart Memory Defenses for Memory Bus Side Channel. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. ACM.

[8] Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. 2011. Onyx: A Protoype Phase Change Memory Storage Array. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'11)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2002218.2002220

[9] 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 *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 263–276. https://doi.org/10.1145/2872362.2872377

[10] Shivam Bhasin, Paolo Maistri, and Francesco Regazzoni. 2014. Malicious wave: A survey on actively tampering using electromagnetic glitch. In *Proceedings of the IEEE International Symposium on Electromagnetic Compatibility, Tokyo (EMC'14/Tokyo), 2014*. IEEE, 318–321.

[11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[12] BRYAN BLACK. 2013. DIE STACKING IS HAPPENING!, http://www.microarch.org/micro46/files/keynote1.pdf. (2013).

[13] Wayne Burleson, Onur Mutlu, and Mohit Tiwari. 2016. INVITED: Who Is the Major Threat to Tomorrow's Security? You, the Hardware Designer. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 145.

[14] Siddhartha Chhabra and Yan Solihin. 2011. i-NVMM: A Secure Non-volatile Main Memory System with Incremental Encryption. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 177–188. https://doi.org/10.1145/2000064.2000086

[15] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security. Vol. 16. 2016.*

[16] 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 *Theory of Cryptography Conference*. Springer, 145–174.

[17] D. Eastlake, 3rd and P. Jones. 2001. US Secure Hash Algorithm 1 (SHA1). (2001).

[18] Yasuko Eckert, Nuwan Jayasena, and Gabriel H. Loh. 2014. Thermal Feasibility of Die-Stacked Processing in Memory. Workshop on Near-Data Processing (WoNDP). (2014).

[19] Andrew Ferraiuolo, Yao Wang, Danfeng Zhang, Andrew C Myers, and G Edward Suh. Lattice Priority Scheduling: Low-Overhead Timing-Channel Protection for a Shared Memory Controller. In *Proceedings of the IEEE 22nd International Symposium on High Performance Computer Architecture (HPCA), 2016*.

[20] 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 *ACM SIGPLAN Notices*, Vol. 50. ACM, 103–116.

[21] Christopher W Fletcher, 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 *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 213–224.

[22] Lan Gao, Jun Yang, Marek Chrobak, Youtao Zhang, San Nguyen, and Hsien-Hsin S Lee. 2006. A low-cost memory remapping scheme for address bus protection. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation techniques (PACT)*. ACM, 74–83.

[23] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. 1986. How to Construct Random Functions. *J. ACM* 33, 4 (Aug. 1986), 792–807. https://doi.org/10.1145/6490.6503

[24] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on Oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[25] Yufei Gu, Yangchun Fu, Aravind Prakash, Zhiqiang Lin, and Heng Yin. 2012. OS-Sommelier: memory-only operating system fingerprinting in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 5.

[26] Jeff Hao and Valeria Bertacco. 2009. PowerRanger: Assessing circuit vulnerability to power attacks using SAT-based static analysis. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop, 2009. HLDVT 2009*. 54–59. https://doi.org/10.1109/HLDVT.2009.5340174

[27] Andrew Huang. 2003. Keeping Secrets in Hardware: The Microsoft Xbox Case Study. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '02)*. Springer-Verlag, London, UK, UK, 213–227. http://dl.acm.org/citation.cfm?id=648255.752707

[28] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. 2015. Understanding contention-based channels and using them for defense. In *Proceedings of IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015*. IEEE, 639–650.

[29] Michael Hutter and Jörn-Marc Schmidt. 2013. The temperature side channel and heating fault attacks. In *International Conference on Smart Card Research and Advanced Applications*. Springer, 219–235.

[30] Intel. 2013. Intel Software Guard Extensions. *https://software.intel.com/en-us/sgx* (2013).

[31] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. 2011. Introduction to differential power analysis. *Journal of Cryptographic Engineering* 1, 1 (2011), 5–27.

[32] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable DRAM alternative. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 2–13.

[33] Ziyi Liu, JongHyuk Lee, Junyuan Zeng, Yuanfeng Wen, Zhiqiang Lin, and Weidong Shi. 2013. CPU Transparent Protection of OS Kernel and Hypervisor Integrity with Programmable DRAM. *SIGARCH Comput. Archit. News* 41, 3 (June 2013), 392–403. https://doi.org/10.1145/2508148.2485956

[34] Krishna T Malladi, Benjamin C Lee, Frank A Nothaft, Christos Kozyrakis, Karthika Periyathambi, and Mark Horowitz. 2012. Towards energy-proportional datacenter memory with mobile DRAM. In *ACM SIGARCH Computer Architecture News*, Vol. 40. IEEE Computer Society, 37–48.

[35] Ueli M Maurer and Stefan Wolf. 2000. The Diffie-Hellman protocol. In *Towards a quarter-century of public key cryptography*. Springer, 77–101.

[36] Thomas S Messerges. 2000. Securing the AES finalists against power analysis attacks. In *International Workshop on Fast Software Encryption*. Springer, 150–164.

[37] Thomas S Messerges, Ezzy A Dabbish, and Robert H Sloan. 1999. Investigations of Power Analysis Attacks on Smartcards. *Smartcard* 99 (1999), 151–161.

[38] OpenCores. 2012. Tiny AES, http://opencores.org/project/. (2012).

[39] OpenCores. 2014. MD5 Pipelined, http://opencores.org/project/. (2014).

[40] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. 2015. Constants count: practical improvements to Oblivious RAM. In *24th USENIX Security Symposium (USENIX Security 15)*. 415–430.

[41] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2014. Ring ORAM: Closing the Gap Between Small and Large Client Storage Oblivious RAM. *IACR Cryptology ePrint Archive* 2014 (2014), 997.

[42] Ronald Rivest. 1992. The MD5 Message-Digest Algorithm. (1992).

[43] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. 2007. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40), 2007 (MICRO 40)*. IEEE Computer Society, Washington, DC, USA, 183–196. https://doi.org/10.1109/MICRO.2007.44

[44] Brian Rogers, Chenyu Yan, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. 2008. Single-level integrity and confidentiality protection for distributed shared memory multiprocessors. In *Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture, 2008*. IEEE, 161–172.

[45] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. Black Hat (2015).

[46] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. 2015. Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 89–101.

[47] 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 *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 299–310.

[48] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe Navaux, and others. 2015. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015*. IEEE, 331–342.

[49] Hung Vuong. 2013. Mobile Memory Technology Roadmap. *JEDEC* (2013).

[50] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. 2006. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In *Proceedings of the 33rd International Symposium on Computer Architecture, 2006. ISCA '06*. 179–190.

[51] Vinson Young, Prashant J Nair, and Moinuddin K Qureshi. 2015. DEUCE: Write-efficient encryption for non-volatile memories. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 33–44.

[52] 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 *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 102–114. https://doi.org/10.1145/2830772.2830787

[53] Xiaotong Zhuang, Tao Zhang, Hsien-Hsin S Lee, and Santosh Pande. 2004. Hardware assisted control flow obfuscation for embedded processors. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM, 292–302.

[54] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. 2004. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 72–84. https://doi.org/10.1145/1024393.1024403