2018 IEEE International Symposium on High Performance Computer Architecture

Secure DIMM: Moving ORAM Primitives Closer to Memory

Ali Shafiee, Rajeev Balasubramonian, Feifei Li
School of Computing
University of Utah
Salt Lake City, Utah, USA
Email: {shafiee,rajeev,lifeifei}@cs.utah.edu

Mohit Tiwari
Dept. of ECE
University of Texas, Austin
Austin, Texas, USA
Email: tiwari@austin.utexas.edu

***Abstract*—**
As more critical applications move to the cloud, there is a pressing need to provide privacy guarantees for data and computation. While cloud infrastructures are vulnerable to a variety of attacks, in this work, we focus on an attack model where an untrusted cloud operator has physical access to the server and can monitor the signals emerging from the processor socket. Even if data packets are encrypted, the sequence of addresses touched by the program serves as an information side channel. To eliminate this side channel, Oblivious RAM constructs have been investigated for decades, but continue to pose large overheads. In this work, we make the case that ORAM overheads can be significantly reduced by moving some ORAM functionality into the memory system. We first design a secure DIMM (or SDIMM) that uses commodity low-cost memory and an ASIC as a secure buffer chip. We then design two new ORAM protocols that leverage SDIMMs to reduce bandwidth, latency, and energy per ORAM access. In both protocols, each SDIMM is responsible for part of the ORAM tree. Each SDIMM performs a number of ORAM operations that are not visible to the main memory channel. By having many SDIMMs in the system, we are able to achieve highly parallel ORAM operations. The main memory channel uses its bandwidth primarily to service blocks requested by the CPU, and to perform a small subset of the many shuffle operations required by conventional ORAM. The new protocols guarantee the same obliviousness properties as Path ORAM. On a set of memory-intensive workloads, our two new ORAM protocols – Independent ORAM and Split ORAM – are able to improve performance by 1.9× and energy by 2.55×, compared to Freecursive ORAM.

***Keywords*-Memory Systems; Privacy; Oblivious RAM.**

I. Introduction

Many applications execute in datacenters or on smart handheld devices. This makes a hardware attack more feasible, especially on a passive memory system where adversaries can observe both data and addresses emerging from the processor socket. While data can be protected using encryption, hiding the address pattern is not trivial. This is because a passive memory device does not have encryption/decryption logic and must receive the address in plaintext. The addresses touched by a program can therefore be observed by an attacker that has physical access to the hardware.

To close this side channel, researchers have proposed different Oblivious RAMs that make two different access patterns indistinguishable. In spite of decades of progress, state-of-the-art ORAM proposals continue to suffer from very high overheads. An ORAM converts a single memory access into more than a hundred memory accesses, so its performance is very much dictated by the memory bandwidth available to the system. Modern processors have only a handful of memory channels that are quickly saturated by the bandwidth demands of ORAM.

In this work, we try to boost the bandwidth available to ORAM by creating a number of memory access channels that do not burden the processor's limited pin budget. These memory access channels are on the DIMM and are controlled by a custom buffer chip on the DIMM. The buffer chip is part of the trusted computing base (TCB). We shift most of the ORAM controller functionality from the secure CPU to the secure buffer chip on each DIMM. Thus, most of the data movement required for an ORAM access now happens locally on the DIMMs. We refer to this new DIMM architecture as a *secure DIMM* or *SDIMM*. It has the following advantages:

1. Memory Capacity and Cost: By relying on commodity DDR-compatible products and a DIMM interface, SDIMMs are able to provide higher memory capacity and lower cost than currently available active memory architectures (e.g., Micron HMC [1]). High memory capacity at low cost is critical for many cloud applications [2], [3].

2. Performance: The ORAM protocol now has as many memory channels at its disposal as the number of DIMMs. This improves the memory parallelism that can be achieved for ORAM accesses. Additionally, it clears ORAM traffic from the shared main memory channels, which improves the latency for non-secure accesses.

3. Energy: By localizing most ORAM traffic within a DIMM, the energy cost of ORAM data movement is significantly lowered.

4. Privacy: The CPU vendor does not have to trust the DRAM chip vendor or the manufacturer of an active memory device. The CPU vendor can design its own trusted buffer chip and SDIMM, while using non-trusted commodity DRAM chips. As we show later, our new ORAM protocols offer the same privacy guarantees as state-of-the-art ORAM.

Next, we design ORAM protocols that can be efficiently

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00044

IEEE computer society

distributed across multiple SDIMMs in different ways. We first create an *Independent ORAM* per SDIMM, where each SDIMM is responsible for a subtree of the full ORAM. One can consider this memory model as a queuing system with multiple servers. The more SDIMMs in the system, the more parallelism that is available to service ORAM requests. While this Independent model reduces bandwidth pressure on the memory bus, it achieves a service time on an SDIMM that is almost the same as a regular single channel ORAM. For applications with low memory level parallelism, a regular Path ORAM might outperform an SDIMM-based memory system, as Path ORAM can use all channels to reduce ORAM service time. We therefore propose a second distributed protocol, a *Split ORAM* where each bucket in one ORAM tree is decomposed into multiple equal parts and distributed across multiple SDIMMs. The Split ORAM protocol only moves metadata to the CPU, and most data block shuffling is performed locally within each SDIMM. The collective internal memory bandwidth of multiple SDIMMs can be harnessed for a single ORAM request, thus lowering latency per access. Finally, we show a memory layout that localizes ORAM data accesses to one rank for each ORAM request. As a result, we can keep most of the memory ranks in low power mode.

In short, this work offers the following contributions:

1. We propose an SDIMM that shifts the ORAM controller to the DIMM buffer and increases memory parallelism. We discuss the architectural requirement for SDIMMs to guarantee secure address obfuscation.
2. We show how an ORAM protocol can be decomposed across multiple SDIMMs using two different approaches. The first approach, Independent ORAM, achieves a dramatic reduction in main memory channel bandwidth. The second approach, Split ORAM, consumes a moderate amount of main memory channel bandwidth, but reduces latency by spreading every ORAM request across multiple SDIMMs.
3. We show a low power scheme for SDIMMs that keeps most of the ranks in low power mode and localizes all accesses per ORAM request to one rank.
4. Finally, we evaluate our approach and compare it against the state-of-the-art Freecursive ORAM [4]. Our results show that SDIMM-based systems can improve system performance by 1.9× and energy by 2.5× for a 32 GB memory system.

II. Background

A. DRAM Basics

A commodity DRAM-based memory system consists of one or more channels that each have a data bus and a command/address bus. Each channel can have up to three dual inline memory modules (DIMMs). Every DIMM has multiple ranks consisting of multiple DRAM chips. Every cache block is scattered across chips in a rank. All the chips receive the same commands through a shared command/address bus, while they send to and receive from their own part of the data bus. Each rank also consists of multiple banks that can be accessed nearly independently to increase memory parallelism. To fetch a cache line, the memory controller sends a RAS signal to open a row, followed by a CAS signal to send the selected part of the row to the CPU.

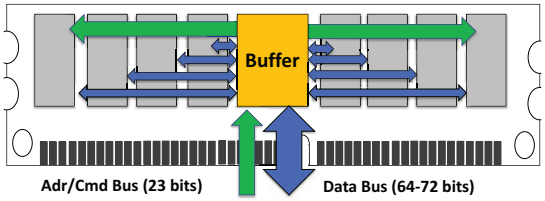

Figure 1. A baseline LRDIMM.

While there are different types of DIMMs, in this work we focus on load reduced DIMM (LRDIMM [5]), which supports high bandwidth and high capacity, and is popular in server machines. In an LRDIMM, the data bus and the command/address bus between DRAM chips and the CPU are relayed through the LRDIMM buffer, which is a chip on the DIMM that currently has no processing capabilities. The buffer chip improves signal quality and the maximum frequency of the memory channel (see Figure 1).

B. Threat Model

In this work, we assume that the CPU is secure and the adversary cannot tamper with it. However, they can deploy physical attacks on the memory system. Such attacks may involve a logic analyzer that can monitor visible signals on printed circuit boards (including the motherboard and DIMMs). The attacker can thus passively observe the data and addresses of the memory access stream that emerges in/out of the processor and memory chips. As a result, they may deduce crucial information about the running programs and their inputs. Prior work [6] has shown that sensitive information can be deduced even if the data is encrypted. This information can be later used to also actively tamper with data (an active attack). To withstand such physical attacks, the memory system must be augmented to not only provide confidentiality (encryption support), but also data integrity (e.g., with Merkle Trees) and access pattern indistinguishability.

C. Path-Oblivious RAM

For decades, many have pursued Oblivious RAM [7], [8], [9], [10], [11], [12], [13], [4], [14] implementations to guarantee indistinguishable access patterns. The key idea behind ORAM is to randomly change the address of each memory block, whenever it is accessed. One of the recent proposals for ORAM, with the least bandwidth and storage overhead, is Path ORAM [11]. In this type of ORAM, data

blocks in memory are logically organized as a balanced binary tree with root at level 0 and leaves at level L. Each node in the tree is called a bucket and contains Z encrypted blocks (typically $Z = 4$ [4]). Some of these blocks may be dummy blocks.

In addition to the tree organization, ORAM has two other key components: PosMap and stash. PosMap is a lookup table that associates a leaf ID, from 0 to $2^L - 1$, to each data block. The stash is a small (typically 200 entries [4]) storage buffer in the memory controller that temporarily holds data blocks that are read from the tree. ORAM guarantees that at any moment, the block with leaf ID l is either kept in the stash or in a node on the path from the root node to the leaf node l. In every access to the ORAM, the leaf ID associated with the accessed block is updated to a random value. In order to hide the access type (i.e., read or write) and to randomly change the leaf ID, ORAM is accessed through an $accessORAM(a, op, d')$ interface, where a, op, and d' are the block's physical address, the operation type, and the new value for the block (for write operation), respectively [4]. In every call to $accessORAM(a, op, d')$, the memory controller performs the following steps:

1. It looks up the PosMap and finds the leaf ID l associated with a. It also updates the PosMap entry for address a with a randomly generated value l'.
2. It fetches and decrypts all the cache lines along the path from the root node to the leaf node l, and adds them to the stash.
3. In the case of a read operation, it finds the block a, and sends a copy of it to the last level cache. For write operations, it updates the block's content with d'.
4. Finally, it stores back as many blocks as possible from the stash to the path from the root node to the leaf node l.

To facilitate these steps, each node in the tree has some additional fields. More precisely, each block is augmented with a field for its physical address and a field for its leaf ID. The memory controller uses these fields to identify the requested block and to store the blocks from the stash back in to the memory. In addition to these two fields, a counter is maintained for the entire bucket, which is used for encryption and decryption.

D. Freecursive Path Oblivious RAM

The PosMap in Path ORAM imposes a significant storage overhead on the secure CPU. This is because the PosMap capacity grows linearly with the size of the tree. To alleviate this overhead, the PosMap is also stored in the memory [4]. The memory space for PosMap is also treated as a separate smaller ORAM to avoid information leakage. To distinguish between these different ORAMs, we call the data ORAM as $ORAM_0$ and the PosMap ORAM as $ORAM_1$. Note that the PosMap for $ORAM_1$ might not fit in the secure CPU as well; hence it will be kept in the memory in $ORAM_2$. In general, $ORAM_k$ keeps the PosMap for $ORAM_{k-1}$. The algorithm recursively stores these PosMaps in the memory, until it becomes small enough to fit on the chip. To find a cache line, the memory controller starts with the on-chip PosMap, say $ORAM_n$, and finds an address to perform $accessORAM$ in $ORAM_{n-1}$. This process continues until the $accessORAM$ is called for $ORAM_0$.

The recursive process of calling $accessORAM$ imposes a severe overhead for each data request. A recent paper addressed this problem and proposed Freecursive Path ORAM [4]. The key idea in this approach is to cache entries from $ORAM_i$ ($i > 0$) in an on-chip space called PosMap Lookaside Buffer (PLB). Upon receiving a new request, the memory controller checks the PLB iteratively for the entries in $ORAM_1$ to $ORAM_n$ associated with this request. The procedure stops if there is no hit in the PLB or for the first ORAM leading to a hit. Fletcher et al. [4] advocate that all the $ORAM_0$ to $ORAM_n$ be stored in the same ORAM tree in memory to avoid information leakage.

The Freecursive architecture has two parts, backend and frontend. The backend has the stash and the memory controller, and takes care of performing the $accessORAM$ function. The frontend, on the other hand, queues up LLC requests and has the PLB cache to determine how many $accessORAMs$ are needed per LLC request. The service time of the backend depends on the available resources, especially memory bandwidth.

E. Active Memory Solution

It is worth highlighting that recent work [15], [16] is considering the use of active memory devices to eliminate the leakage of the address stream. These works rely on logic capabilities (primarily, encryption and decryption) within the memory package, as may be possible with devices like the Micron HMC [1]. Placing active components within the memory package can have significant implications on cost. This cost is incurred by all applications running on this server, regardless of whether they are sensitive or not. Instead, a server with low cost-per-bit DIMMs can benefit all applications that run on this server, while the overheads of an ORAM protocol are experienced only when executing sensitive phases in applications.

The use of active memory devices can impact cost and performance in many ways. First, commodity DRAM chips keep costs down by avoiding extraneous logic and catering to high-volume markets. Second, while capabilities can be added to a separate logic die in a 3D package, such 3D packages are inherently more expensive because of their use of through-silicon vias (TSVs). Third, devices like the HMC support a limited capacity of 4 GB per package. When supporting large memory capacities in servers, it is unlikely that the entire memory system will be constructed with a large network of HMCs that can impact board layout and introduce long network latencies. It is more likely that 3D-stacked active memory packages will be used to construct an

off-chip cache that is backed up with a traditional low-cost and passive memory system. This larger memory system will still need support for ORAM to guarantee privacy. Such an ORAM will also suffer from long latencies since part of the processor pin budget is allocated to the HMC cache and not available to boost ORAM bandwidth.

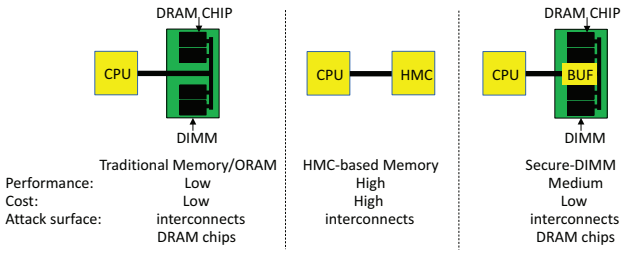

Figure 2. Landscape of solutions: baseline ORAM (left), active memory (center), SDIMM (right).

To some extent, our proposals here are creating an "active" memory system, but are doing so in a cost-aware manner that enables large memory capacities. We are continuing to use low-cost commodity memory chips that are connected to an active logic unit on the DIMM. Since these connections do not rely on exotic packaging and are visible to the attacker, ORAM semantics are required to eliminate side channels. While the use of an active memory unit like the HMC requires the system vendor to trust the memory vendor, in our proposed architecture, the system vendor can create their own active logic unit, thus exercising full control over the trusted computing base.

The design space is depicted in Figure 2. The attack surface (shown by black components) in our model (right) is the same as in traditional ORAM solutions (left). Our solution expands the trusted computing base (shown by yellow components) – by offloading some ORAM functionality to buffers on the SDIMM, we can increase performance at low cost. The active memory solution (center) lowers the attack surface and improves performance, but pays a steep penalty in memory cost-per-bit.

III. Secure DIMMs and New ORAM Protocols

A. Secure DIMM

In this work, we propose a novel *Secure-DIMM* or *SDIMM* that is used to distribute the ORAM tree and reduce its bandwidth impact. In an SDIMM, the central LRDIMM buffer is replaced with a secure buffer that can perform an *accessORAM* operation (see Section II-C) on the DIMM and in close proximity to DRAM chips. Figure 3a depicts the high-level architecture of an SDIMM. As shown, the TCB includes both the CPU and the secure buffer, and the communication between them is encrypted. However, DRAM chips and the on-DIMM bus between the secure buffer and these DRAM chips are not trusted. A secure buffer has two main components:

1) It has an ORAM memory controller that guarantees obliviousness on the DIMM. Therefore, each SDIMM can be considered as a single-channel ORAM.
2) It has an interface logic that enables secure encrypted communication between the secure buffer on the DIMM and the secure CPU. The encryption applies for both data and command/address buses.

SDIMMs can be used to create a distributed implementation of a Freecursive ORAM backend with the same encryption and integrity verification mechanism (PMMAC). Informally speaking, in an SDIMM-based architecture, the CPU sends an encrypted request to an SDIMM, the SDIMM performs an *accessORAM* operation to fetch the requested block, and the SDIMM finally encrypts and sends the block back to the CPU. We try to abide by the DDR protocol as much as possible, i.e., we do not introduce any new pins on the memory channel. As described later, we do introduce new commands that can be placed on the DDR bus. Since an ORAM protocol requires a custom-designed controller anyway, this does not introduce significant additional design effort.

In brief, SDIMM has the following primary advantages:

1) In an ORAM, system throughput is almost directly proportional to the effective memory bandwidth available to the processor. Each SDIMM can be viewed as an independent channel that performs ORAM accesses in parallel, without burdening the shared memory channels. Therefore, the effective available memory bandwidth scales up linearly by deploying multiple SDIMMs.
2) An SDIMM has the same DRAM chips as an LRDIMM, and its secure buffer has almost the same pins as the LRDIMM buffer[1]. Therefore, an SDIMM does not require significant changes to the design and wiring of the DIMM. This has favorable implications for DIMM manufacturing at scale.
3) An SDIMM uses a DIMM form factor. The CPU-side memory controller can be modified to enable SDIMMs and LRDIMMs to co-reside on the same memory channel. This allows virtual machines (VMs) with different levels of security requirements to run on the same CPU. Since an SDIMM handles most data movement locally, it does not negatively impact the bandwidth available to a co-resident VM.
4) In contrast to active memory, such as the Hybrid Memory Cube [1], SDIMMs can offer both address obfuscation and high capacity. As a result, an SDIMM is a suitable choice for in-memory applications (e.g.,

[1]In this work, we assumed a DDR3 topology for the SDIMM. The SDIMM buffer has the same pins as the LRDIMM buffer, so adapting a DDR3 LRDIMM into an SDIMM is straightforward. However, in a DDR4 topology, the LRDIMM data buffer is decomposed into multiple small buffers. Adapting the DDR4 LRDIMM to an SDIMM would require a few additional pins to each buffer chip.

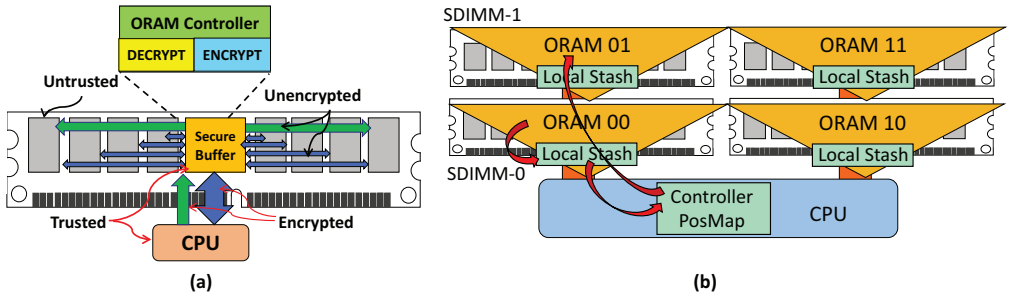

Figure 3. (a) SDIMM overall design. (b) The Independent ORAM protocol.

Oracle TimesTen database) that requires high capacity. In addition, an SDIMM-based system can easily morph between a secure and non-secure memory.

5) An SDIMM gives the system or DIMM manufacturer control over the security of the memory system, and eliminates reliance on the memory vendor or on an active memory device. The system manufacturer must create an SDIMM with commodity DRAM chips and their own trusted secure buffer[2]. This server can then be placed in the cloud and an authentication mechanism can confirm that the hardware has not been tampered with.

To realize the benefits of a secure DIMM, we will answer the following questions in the upcoming sub-sections: (i) When using multiple SDIMMs, how can the ORAM tree be distributed across SDIMMs such that the memory latency is minimized? (ii) How is a secure link between the CPU and an SDIMM initialized? (iii) How can data be transferred between an SDIMM and the secure CPU without leaking any information? (iv) How do we design an SDIMM interface without requiring changes to a DDR channel?

B. CPU-SDIMM Communication

For secure communication, the CPU should authenticate and initialize the connections to the SDIMMs. At boot-up time, the secure CPU has to confirm that it is communicating with a trusted secure buffer on the SDIMM. This authentication of the device can be done in one of many possible ways, using industry best practices. For example, when the system boots up, the CPU can communicate with its secure buffers to obtain their IDs; the CPU then contacts a third-party authenticator, similar to Verisign, to obtain a public key. Once the CPU has a public key for its secure buffer, it goes through the standard practices to establish a secure connection, typically involving multiple messages to agree on upstream/downstream session keys and counters. To ensure secure and low-latency data transfer between the CPU and an SDIMM, we use counter-mode AES, which XORs the plaintext message with a frequently-changing pad that is a function of the key and counter.

[2]It is not uncommon for CPU vendors to design custom DIMMs [17].

C. A Distributed Independent ORAM

In an optimized baseline ORAM memory system, the ORAM binary tree is re-organized as a tree of smaller subtrees. The buckets in each subtree are placed in adjacent memory locations to increase row buffer hit rate. In addition, the cache lines of a bucket are also scattered between multiple channels to utilize channel parallelism [10]. Based on this arrangement and memory address mapping, buckets are distributed over different banks, ranks, DIMMs, and channels.

In the proposed ORAM implementation, as shown in Figure 3b, the address space is partitioned across all SDIMMs based on the most significant bits of the leaf ID. Each SDIMM is only aware of the ORAM sub-tree that it manages. A requested data block will likely transfer from one SDIMM to another, although most data movements are within an SDIMM. The CPU maintains a PLB and on a memory access, it generates the necessary *accessORAM* operations. These are sent to the relevant SDIMM. In other words, the CPU manages the frontend of ORAM while SDIMMs accelerate the backend. The steps are explained in more detail below.

1. The memory controller checks the PLB and determines which request must be issued next. Based on the leaf ID of that request, it is sent to one of the SDIMMs (SDIMM-0 in our example in Figure 3). The request is encrypted and sent to the secure buffer on SDIMM-0 with an *ACCESS* command (we will later explain how these commands are set up in a DDR compatible manner). To obfuscate the operation type (read or write), an *ACCESS* command is always followed by one block of data; in case of a read operation, this block is a dummy that is discarded by the receiving SDIMM.
2. The secure buffer on SDIMM-0 receives the request and decrypts the message. It then fetches all the buckets in the path from its root to the associated leaf ID, decrypts them, and puts them into the local stash, located in the secure buffer of SDIMM-0.
3. In case of a write, the relevant block is updated with its new value. Regardless of the operation (read or write), SDIMM-0 then generates a random new leaf ID for the block requested by the CPU. If the leaf ID belongs to the current

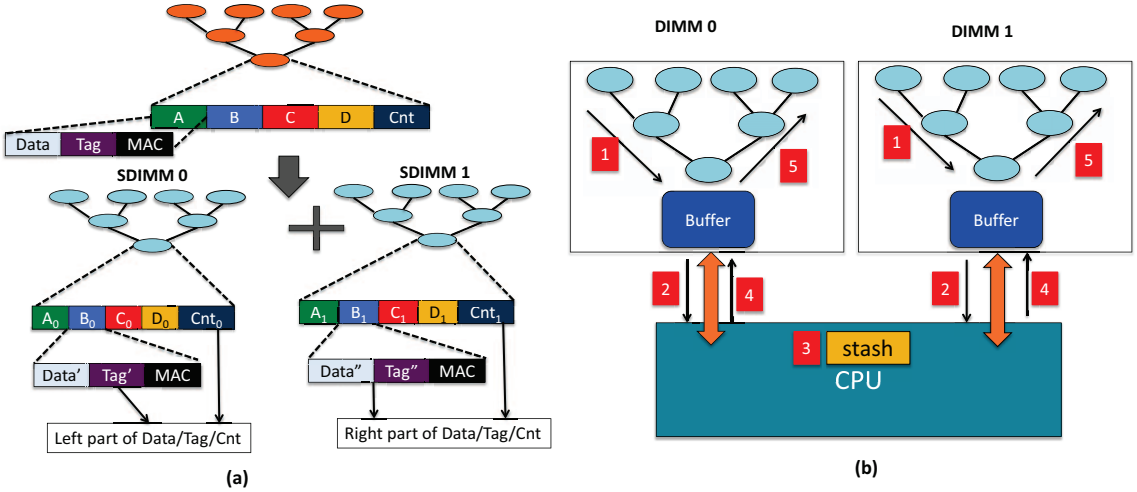

Figure 4. (a) Split ORAM memory layout. (b) Split ORAM protocol steps.

SDIMM, the block remains in the local stash. Otherwise, the block is removed from the stash in SDIMM-0.

4. The secure buffer then moves as many blocks as possible from its local stash to buckets on the path from the root to the old leaf ID.

5. In case of a write operation where the new leaf ID is in SDIMM-0, the secure buffer sends a dummy block to the CPU-side memory controller. In all other cases (read or write), the secure buffer encrypts the pertinent cache block, and sends it to the CPU-side memory controller. The block is augmented with its new leaf ID (mapped to say SDIMM-1). Note that in a DDR channel, only the CPU-side memory controller can control the memory channel. Therefore, SDIMM-0 cannot initiate a data transfer. The CPU-side memory controller has to periodically send a *PROBE* command to SDIMM-0 to determine if a response is ready. When the command's response is positive, it issues a *FETCH_RESULT* command to fetch the requested data block.

6. Upon receiving the data block, the CPU-side memory controller may have to move this data block to its new SDIMM. If the CPU does this with a single write to the destination SDIMM, it is evident where the block has moved. This can disrupt the *indistinguishability* guarantee in ORAM. If the access pattern has high temporal locality, i.e., a single block A is touched repeatedly, the sequence of touched SDIMMs would take the form: read from SDIMM-i, write to SDIMM-j, read from SDIMM-j, write to SDIMM-k, read from SDIMM-k, and so on. Therefore, to obfuscate the new destination of this block and preserve indistinguishability of access patterns, the CPU sends one block to every SDIMM using the *APPEND* command. The command sent to SDIMM-1 in our example contains the pertinent block while the ones destined to other SDIMMs carry dummy blocks. The APPEND command simply adds the block to the destination's local stash, if it is not a dummy.

Since block addresses are random, the number of blocks removed and appended to each SDIMM local stash averages close to zero. However, the proposed protocol can cause overflows in local stashes. If a local stash overflows, it can initiate local reads and writes to drain its stash, similar to conventional stash overflow strategies such as background eviction [10]. We later quantify the probability of stash overflow when transferring data between SDIMMs and show a simple solution to make it almost zero (see Section IV-C).

We refer to this design as the *Independent* architecture because most ORAM operations, including the assignment of new leaf IDs to fetched blocks, are in the hands of each SDIMM. The CPU is largely hands-off; it only manages the PLB, issues *accessORAM* requests, and issues *APPEND* commands to move the requested block to a new SDIMM. This design places a much smaller bandwidth demand on the memory channel than the baseline ORAM because only blocks requested by the CPU (and occasional dummies) are transmitted on the memory channel. The memory channel escapes dealing with the many blocks on an ORAM tree path that are shuffled on every ORAM access.

D. The Split Architecture

In the Independent architecture, a single SDIMM is responsible for most of the data shuffling required by an *accessORAM* operation. Given the SDIMM's limited buffer chip pin bandwidth, the latency for this operation is similar to that of a baseline ORAM operating with a single memory channel. Thus, even though the Independent architecture can efficiently process multiple *accessORAM* operations in parallel on different SDIMMs, the latency for each operation is relatively high.

Therefore, to further reduce latency for an ORAM access, we must spread a single ORAM access across multiple SDIMMs to leverage the collective memory bandwidth in multiple SDIMMs. This is especially useful if the workload has limited memory level parallelism and multiple SDIMMs

are sitting idle while a few SDIMMs are performing accesses in the Independent ORAM architecture.

The *Split Architecture* tries to mitigate the *accessORAM* latency by relying on a different data layout in the memory. Informally speaking, one ORAM tree is decomposed into multiple trees of the same height, but each of them has less capacity per bucket. Here, without loss of generality, we will explain splitting for two SDIMMs (or 2-way splitting). Each bucket in the original ORAM tree has four cache blocks, and corresponding metadata (four tags, four leaf IDs, four MACs, one shared counter). In 2-way splitting, each bucket has one half of each data block, one half of each tag, one half of each leaf ID, half the counter, and its own MAC. Similar to PMMAC [4], MACs are generated based on the compact counters and the data portions available in each bucket. Therefore, in n-way splitting, the MAC overhead is n times that in Freecursive ORAM. However, this overhead is small, relative to the high overhead of dummy blocks in the baseline ORAM. Figure 4a shows the data layout due to 2-way splitting. Note that we are not assigning half the blocks (and corresponding metadata) to one SDIMM; we are assigning half of every block/metadata to each SDIMM.

An accessORAM operation now entails the following steps (shown in Figure 4b):

1. The CPU-side memory controller looks up the PLB to find the appropriate leaf ID for the next *accessORAM* operation. For all buckets on the path from the leaf to the root, the CPU-side ORAM controller sends two types of fetch commands to both SDIMMs. First, it sends a *FETCH_DATA* command that reads the data part of each bucket on the path from the leaf to the root. This data is not transmitted back to the CPU, but is stored in each SDIMM's local stash, i.e., it is a largely local operation. Note that each SDIMM only handles half the bits for each data block.
2. Next, the CPU-side memory controller sends regular read commands, i.e., conventional RAS and CAS signals, and retrieves the metadata (tags, leaf IDs, and counters) for the entire path in both SDIMMs. Note that each SDIMM is responsible for providing half the bits of each metadata block back to the CPU.
3. On the CPU side, the ORAM controller re-assembles all the metadata received from different SDIMMs for the same bucket position and reconstructs tags, leaf IDs, and the counter. Having all the tags, the ORAM controller can find the requested block. The CPU stash is also designed to shadow the local stash in each SDIMM; the key difference is that the CPU stash only has tags and the local SDIMM stashes have data blocks.
4. The CPU knows the exact location of the requested block in the SDIMM stashes. It issues *FETCH_STASH* commands to fetch the pieces of the requested data block from the SDIMMs. This command sends the local stash index to identify the block that must be returned. Having all the leaf IDs, the CPU-side ORAM controller also determines how to write back from the stash. Unlike the Independent architecture, most ORAM decisions in the Split architecture are made by the CPU. The CPU sends this write-back order to all SDIMMs. Note that instead of using the full address to refer to data in the stash, the CPU-side ORAM controllers use the position in the stash. Additionally, the CPU-side ORAM controller sends the re-assembled counters to all the SDIMMs. This is needed as these counters are used for encryption and decryption of the contents of buckets, as well as for MAC verification. All of this information is packaged as two *RECEIVE_LIST* commands to the SDIMMs.
5. The SDIMMs receive the lists of the blocks that should be evicted from their stash into their trees as well as the entire counter bits, through two *RECEIVE_LIST* commands. Using the counters, the SDIMMs re-encrypt the blocks, re-calculate MACs, and update the tree according to the list sent by the CPU.

In these steps, most of the data block shuffling movement happens within the SDIMMs. The memory channel to the CPU is primarily used to send/receive metadata and the specifically requested data blocks. Thus, compared to conventional ORAM, we see lower traffic on the memory channel and lower latency. Compared to the Independent SDIMM architecture, the Split architecture places more traffic on the memory channel, but incurs a lower latency per request by leveraging the collective bandwidth on multiple SDIMMs. In addition, splitting increases the MAC overhead as each split piece requires its own MAC.

Indep-Split Protocol: The Split and Independent protocols can be combined. For example, in a 4-SDIMM design, 2 SDIMMs can be responsible for half the ORAM. This half is managed with the Split protocol, i.e., most of the data shuffle is performed within the half ORAM. The CPU can consider moving the requested block to either half, similar to how the Independent protocol moves the requested block between different SDIMMs. This jump across ORAM halves is performed within the CPU by moving the requested block from one stash to the next. This combined Indep-Split protocol balances parallelism and latency.

E. Low Power ORAM Access

The performance of the ORAM-based memory system depends on available bandwidth. One way to improve bandwidth is to increase memory channel clock frequency. However, DRAM chips consume more background power when frequency is increased. To reduce the power, we propose to re-organize SDIMM's ORAM tree placement in such a way that each rank contains one whole subtree, as shown in Figure 5. Note that the new layout still keeps the buckets in a small subtree close to each other as proposed in [10]. This layout applies to the Independent and Split protocols.

As a result of this new organization, during an $accessORAM$, the SDIMM engages one rank and the other ranks in the SDIMM can be placed in low power mode. Note

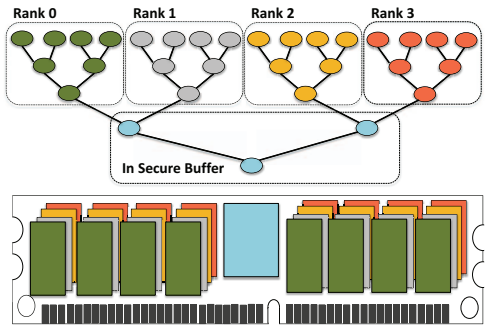

Figure 5. Memory layout for the low power technique.

that wakeup latency (24ns for DDR3 [18]) is much shorter than $accessORAM$ latency. In addition, an SDIMM can turn on the rank required for the next request early enough to hide the wakeup latency. In a quad-rank SDIMM, we can accommodate four large subtrees and place three of them in low power mode. In this case, the first two levels of the tree that are shared by these subtrees are stored in the secure buffer.

F. Extending DDR Commands

In our discussions so far, we have introduced a few new commands that the CPU's memory controller uses to communicate with the secure buffer and orchestrate *accessORAM* operations. Unfortunately, there is no free pin on an LRDIMM to use for these commands. To retain DDR compatibility, we shoehorn our new commands into the existing DDR interface.

There are two types of commands that the CPU sends to an SDIMM: short and long commands. For short commands, the command/address bus is enough to send a request. For long commands, the data bus is also needed. In order to support new commands that can be received by a secure buffer, we use address bits as well to encode commands. To this end, we reserve the first blocks of the SDIMM for commands. This means that the first blocks cannot contain any data. RAS and CAS commands to these reserved addresses are interpreted by the SDIMM as special commands. Since a CAS command can work at the granularity of an 8-byte word, each reserved memory block can be used to construct 8 different commands. Therefore, we use the addresses in Block 0, in the read mode, to express our short commands. For long commands, we use the write mode with address 0. In this case, the data written to this address contains the message. Table I summarizes the formats for these new commands. The first two commands are used during the authentication process to exchange keys.

G. Privacy Analysis

The Freecursive protocol is composed of a frontend and backend. For the Independent protocol, we keep the two components intact, but the frontend executes on the CPU, while the backend executes on the SDIMM. The communication between these two components is protected with counter-mode encryption and does not leak any information. The nature of communication between these two components is also fixed, i.e., it always involves (i) sending an *accessORAM* operation and a data block to a random SDIMM, (ii) receiving a data block from that SDIMM, and (iii) sending blocks to all SDIMMs. The deterministic nature of these additional messages ensures that there is no new information leakage and the indistinguishability properties of the baseline Freecursive ORAM are preserved.

Command	Type	RD vs. WR	cmd/addr Bus
SEND_PKEY	short	RD	RAS(0x0) CAS(0x0)
RECEIVE_SECRET	long	WR	RAS(0x0) CAS(0x0)
ACCESS	long	WR	RAS(0x0) CAS(0x0)
PROBE	short	RD	RAS(0x0) CAS(0x8)
FETCH_RESULT	short	RD	RAS(0x0) CAS(0x10)
APPEND	long	WR	RAS(0x0)) CAS(0x0)
FETCH_DATA	short	RD	RAS(0x0) CAS(0x18)
FETCH_STASH	long	WR	RAS(0x0) CAS(0x18) CAS(idx)
RECEIVE_LIST	long	WR	RAS(0x0) CAS(0x0)

Table I
DETAILS OF COMMANDS USED BY SDIMM.

The Split protocol mirrors the Freecursive baseline, but only exchanges the bare minimum number of data blocks between frontend and backend (with counter-mode encryption). Again, the nature of this communication (number of blocks, source, and destination) is deterministic and does not introduce new leakage. On exposed unencrypted buses, the traffic observed is identical to the traffic that would have been observed with Freecursive.

IV. METHODOLOGY AND RESULTS

A. Methodology

For our evaluation, we use USIMM, a trace-based cycle accurate memory simulator [20]. We modified USIMM to support a last level cache and also implemented Freecursive ORAM [4]. For the backend of ORAM, we used an FR-FCFS memory scheduler [21]. Read requests are prioritized until the write queue size exceeds 40. In our simulator, we fast-forward 1M accesses to warm up the LLC and then measure cycle-accurate performance for the next 1M accesses. For traces, we captured L1 cache misses, for 10 SPEC2006 benchmarks using full system simulator Simics [22]. We did our evaluation for both single channel and 2-channel memory configurations with two DIMMs per channel. For our energy evaluation we relied on Micron

Trace Capturing	
Processor	
ISA	UltraSPARC III ISA
CMP size and Core Freq.	1-core, 1.6 GHz in-order
Re-Order-Buffer	128 entry
Cache Hierarchy	
L1 I-cache	32KB/2-way private, 1-cycle
L1 D-cache	32KB/2-way private, 1-cycle
Cycle-Accurate Simulation	
Cache Hierarchy	
L2 Cache	2MB/64B/8-way shared, 10-cycle
DRAM Parameters	
DRAM Device Parameters	Micron MT41J256M8 DDR3-800 Timing parameters [19] 8 ranks per channel, 8 banks/chip 32768 rows/bank, x8 part
DIMM Configuration	72 bit channel, 9 devices/rank
Row-Buffer Size	8KB
DRAM Bus Frequency	1600 MHz
DRAM Write Queue Size	64 entries
Freecursive Parameters	
PLB Size	64KB
Blocks per Bucket (Z)	4
Data Block Size	64B
Encryption/Decryption Latency	21 cycles
Number of recursive PosMaps	5

Table II
SIMULATOR PARAMETERS.

Power Calculator [23] and CACTI 7 [24] to calculate DRAM chip and memory channel power consumption. Table II summarizes the parameters we use in our evaluation.

B. Results

Evaluating the Baseline: Figure 6 shows the slowdown of the state-of-the-art Freecursive ORAM compared to a non-secure baseline for single channel and 2-channel memory systems, respectively. According to this figure, even with caching 7 levels of ORAM in the memory controller, ORAM, on average, causes 8.8× and 5.2× performance loss for a single and double channel memory, respectively. Additionally, we observe that each LLC miss translates into 1.4 *accessORAM* operations on average because of recursive PosMap look-ups.

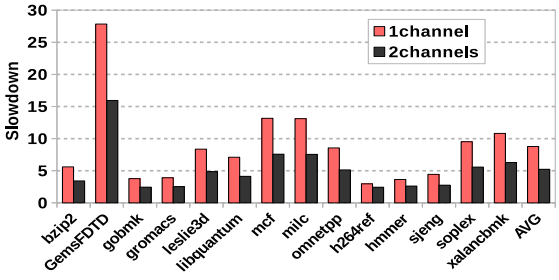

Figure 6. Average slowdown of Freecursive with respect to a non-secure configuration.

SDIMM models: For each memory configuration, there are multiple options for an SDIMM-based design. These different organizations are summarized in Figure 7. For a single channel configuration, we consider two SDIMM-based designs: **INDEP-2** and **SPLIT-2**, respectively representing the Independent and Split protocols on 2 SDIMMs each. For 2-channel memory systems, we consider three SDIMM-based designs: **INDEP-4**, **SPLIT-4**, and **INDEP-SPLIT**: all use 4 SDIMMs, with the Independent protocol, Split protocol, and a combination of the Independent and Split protocols, respectively. We report our results for these designs with and without a 64KB cache that stores the first few levels of ORAM.

Impact on Performance: Figure 8 and Figure 9 show the normalized execution time with respect to Freecursive ORAM for single and double channel systems, respectively. For the single-channel memory, with caching the first few layers of ORAM, these approaches reduce execution time by 32% and 33.5%, with INDEP-2 slightly out-performing SPLIT-2 on average. Without the help of ORAM caching (baseline and proposed), SIDMM-based systems reduce execution time by around 35.7%.

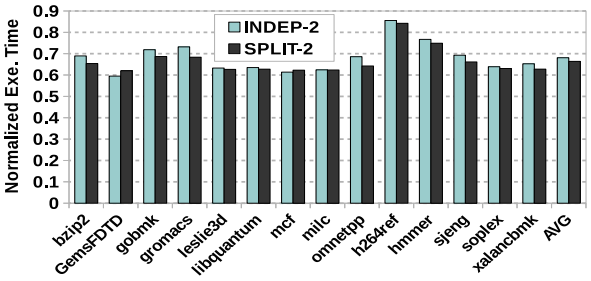

Figure 8. Normalized execution time of single-channel SDIMM-based designs.

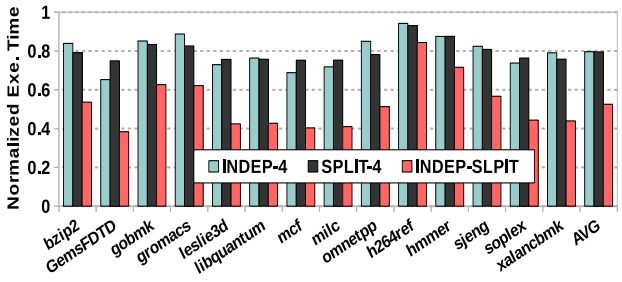

Figure 9. Normalized execution time of double-channel SDIMM-based designs.

For the double channel memory, INDEP-4, SPLIT-4, and INDEP-SPLIT improve performance by 20.3%, 20.4%, and 47.4% on average, respectively. Thus, the 5.2× slowdown (Figure 6) in the Freecursive protocol, relative to a non-secure baseline, has been halved to 2.7× with the INDEP-SPLIT protocol, while using low-cost commodity DRAM chips and not requiring trust in the memory vendor. Applications (gromacs, omnetpp) that have high levels of memory-

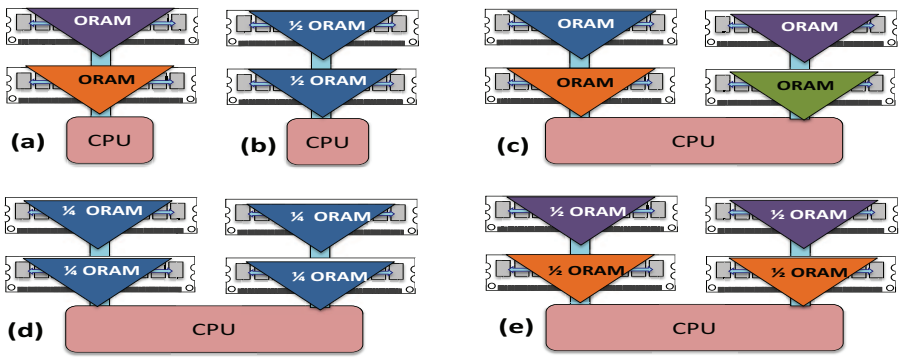

Figure 7. Different SDIMM-based designs (a) INDEP-2 (b) SPLIT-2 (c) INDEP-4 (d) SPLIT-4 and (e) INDEP-SPLIT.

level parallelism do better with the Indep-4 protocol because they can keep all 4 SDIMMs busy. Other applications (GemsFDTD) benefit more from low latency and the SPLIT-4 protocol. We observe that the combined INDEP-SPLIT protocol finds the best balance of latency and parallelism in every benchmark and achieves the best performance.

In Freecursive ORAM, for each *accessORAM* operation, the CPU deals with $2\times(Z+1)\times L$ memory accesses, where Z is the bucket size and L is the number of ORAM tree levels in memory. Meanwhile, in an Independent ORAM protocol, the CPU only deals with 1 read and 5 writes (assuming 4 SDIMMs). In the Split protocol, the CPU requires multiple memory accesses to read and update metadata. For a 28-layer ORAM system with 7-layer ORAM caching, INDEP-2 and INDEP-4 reduce the number of off-DIMM accesses to 4.2% and 7.8% of the baseline ORAM, including PROBE access overheads, respectively. These overheads drop to less than 3.2% when ORAM caching is not used. For the Split and Indep-Split designs, the off-DIMM accesses are reduced to 12% of the baseline ORAM. For the 2-channel case, the Split and Indep-Split models reduce memory latency, relative to Freecursive, by 41% and 63% respectively. In turn, the low ORAM-specific traffic on the main DDR bus can lead to lower latency for memory accesses by other non-secure threads (not evaluated here).

Impact on Energy: SDIMM-based designs reduce energy by keeping most of the memory accesses local to an SDIMM (I/O energy saving), as well as by taking advantage of the low power technique introduced in Section III-E (background energy saving).

We also evaluate our low power technique (Section III-E) and observe no more than 4% performance drop as a result of higher bank conflicts. It is worth noting that localizing accesses to one rank also eliminates the 2-cycle rank-to-rank switching time (T_{RTRS}). Figure 10 shows the memory energy overhead of our best performing organizations (i.e., SPLIT-2 and INDEP-SPLIT) and Freecursive for single and double channel configurations. Compared to Freecursive, SPLIT-2 and INDEP-SPLIT improve memory energy efficiency by 2.4× and 2.5×, respectively.

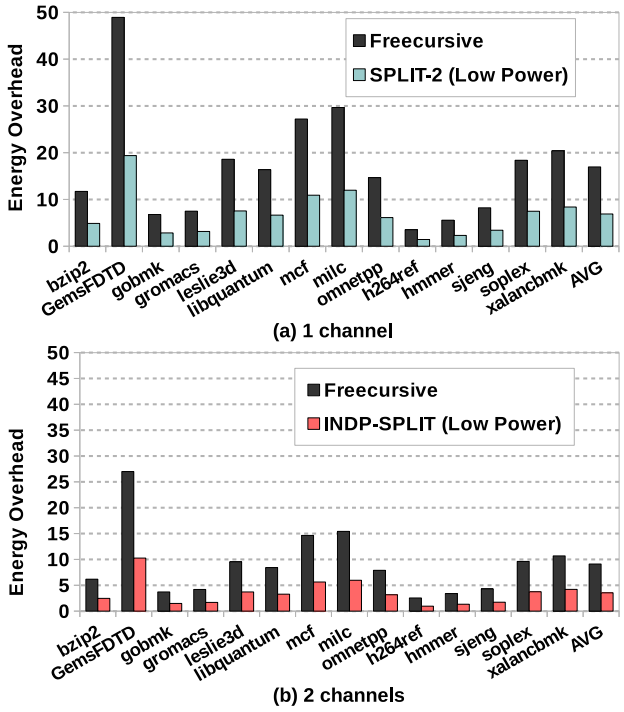

Figure 10. Memory energy overhead normalized to a non-secure baseline for (a) single channel and (b) double channels.

Sensitivity Analysis: We investigate the impact of the number of ORAM layers on the speedup achieved by the best of our SDIMM-based designs (i.e., SPLIT-2 in single channel and INDEP-SPLIT in the double channel memory). Figure 11 shows the average normalized execution time for different numbers of layers. As expected, adding more layers increases the improvements of our designs. Similarly, our designs show slightly higher improvements when ORAM caches are not used. In short, the improvement ranges from 33% to 35% for the single channel memory and 47% to 49% for the double channel memory, respectively.

Area Overhead: The SDIMM buffer chip's main components are an ORAM controller and an 8KB buffer to avoid overflow. Fletcher et al. report 0.47 mm^2 area for the

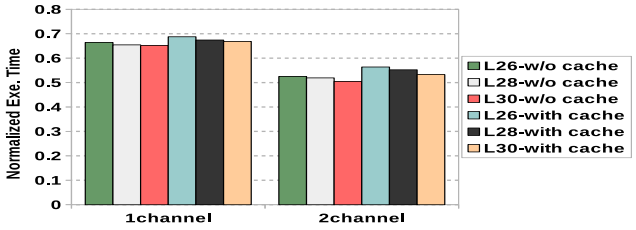

Figure 11. Average normalized execution time for different numbers of layers in ORAM. Lx refers to an x-level ORAM.

ORAM controller in $32nm$. Using CACTI 6.5, we measure the 8KB buffer area to be less than 0.42 mm^2 in the same technology [25]. Therefore, we estimate that the overall area overhead of an SDIMM buffer chip is less than 1 mm^2.

C. Modeling Overflow Rates

In this section, we investigate the impact of transferring blocks between independent SDIMMs on the stash size and stash overflow. Based on our findings, we propose a simple technique to further reduce the stash overflow rate. In our model, we divide the secure buffer into two parts: normal stash and transfer queue (see Figure 12). The normal stash is the part involved in an *accessORAM*, while the transfer queue keeps the block arriving from other DIMMs. We model the size of the transfer queue and show how the probability of transfer queue overflow can be made very small.

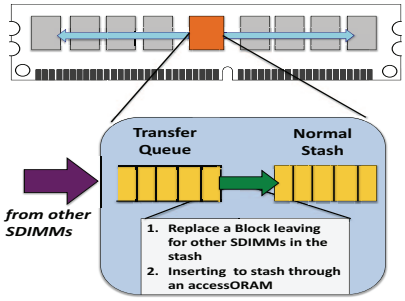

Figure 12. Transfer queue considered in our model.

The transfer queue receives blocks from other SDIMMs. The transfer queue services one block by sending it to the normal stash. The normal stash accepts a block from the transfer queue using two approaches: (1) the transfer queue decides to perform an *accessORAM* operation to service a block, and (2) one block leaves the normal stash for another SDIMM and creates a vacancy. In the latter approach, servicing a block in the transfer queue does not impact the size of the normal stash. In the former approach, the normal stash might overflow due to the *accessORAM* operation. However, prior work has already shown that the probability of this event is extremely small for $Z \geqslant 4$.

The probability of block arrival is $\frac{1}{4}$ in a dual SDIMM system. If we do not use the first approach, the probability of block servicing is also $\frac{1}{4}$. To see the rate of overflow,

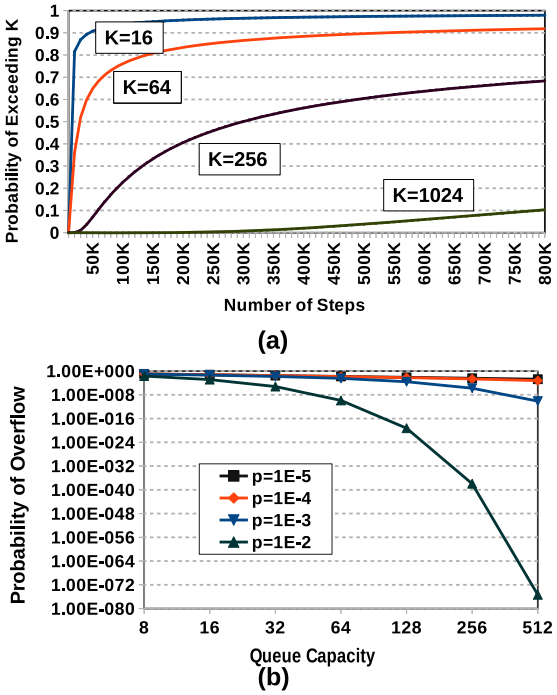

Figure 13. (a) Probability moving more than k steps away from origin in a random walk for different number of steps. (b) The probability of transfer queue overflow when draining it with an *accessORAM* with probability p, for different values of p and different buffer sizes.

we model the size of the transfer queue as a random walk process on a one-dimensional space. Informally speaking, with probability $\frac{1}{4}$, we walk one step to the right, with probability $\frac{1}{4}$, we walk one step left, and with probability $\frac{1}{2}$, we do not walk. To quantify it, let's assume $F(s,k)$ is the probability that, starting from position 0, we are at position k after s steps. Then, $F(s,k)$ is expressed with the following recursive equations.

$$F(s,k) = 0.5 \times F(s-1,k) + 0.25 \times F(s-1,k-1) + 0.25 \times F(s-1,k+1)$$

$$F(s,k) = 0 \quad if(s < k) \& F(0,0) = 1 \quad initial \quad condition$$

Using the above formula, we plotted the chances of piling up more than 16/64/256/1024 blocks in the stash for up to 800K steps. As shown in Figure 13a, the small buffer reaches a 97% chance of overflowing after 100K steps. For 800K steps, the chances of passing the buffer limits are 91%, 70%,and 10% for buffer capacities of 64, 256, and 1024, respectively.

To solve this problem, we must increase service rate so that queue utilization (i.e., ratio of the arrival rate to the service rate) goes under 1 and avoids steady state saturation. We achieve this by leveraging the first approach mentioned above. With a probability p, we insert a block into the ORAM through an extra *accessORAM* operation. In this case, we have utilization $\rho = 0.25/(0.25+p)$.

If we consider each transfer queue as m/m/1/K queue model, i.e., poisson arrival/service probability model with

1 server and K slots in the queue, then the probability of the queue being full is $P_n = \rho^K \times (1-\rho)/(1-\rho^{K+1})$. Figure 13b shows the chance of overflow for different sizes of transfer queue and different values of probability p. We see that even a small queue has a very small overflow rate if we occasionally service an incoming block with an *accessORAM* operation [26].

V. Related Work

The concept of ORAM was first introduced by Goldreich and Ostrovsky to aviod software piracy [7]. Since then, numerous theoretical works have proposed a variety of ORAMs to reduce bandwidth and capacity overheads [27]. One notable proposal is Path ORAM which organizes the ORAM as a binary tree and assumes a small capacity overhead for the client [11]. This approach has been implemented and optimized in Phantom and Ascend hardware prototypes [28], [13]. Recently Nayak et al. introduced HOP which is a hardware prototype with provable security [29]. Fletcher et al. implemented Tiny ORAM on an FPGA that supports variable size data blocks [4].

Besides these implementations, there are multiple proposals to improve the performance of ORAM. Ren et al. [10] explore the design space of ORAM and suggest background eviction and optimized data layout of data in the memory. Yu et al. [30] take advantage of spatial locality in data and use prefetching to get data more efficiently. Fork Path performs multiple ORAM accesses in parallel to avoid reading redundant data. Freecursive ORAM suggests an approach to avoid most of the recursive accesses to ORAM [31]. Fletcher et al. [32] proposed a solution to throttle down the side channel on ORAM-based systems and trade-off performance for security. Ghostrider is a compiler assisted hardware implementation for ORAM [33]. Cooperative Path ORAM optimizes the memory scheduler to mitigate the traffic impact of ORAM access on non-secure threads [34].

Our work is orthogonal to these hardware works and can be applied in conjunction with these works. The major difference between our work and the above approaches is that we create partially active memory components and distribute ORAM functionality across multiple memory modules. Active memory components such as HMC [35] can provide address obfuscation if they have encryption/decryption logic in their logic layer [15], [16]. However, leveraging active memory components requires trusting the memory vendor. Additionally, to support medium to high capacity, many HMCs are needed (at high cost). For example, while one SDIMM can support 64GB, we need 16 4GB HMCs to realize this capacity. HMC-based space-constrained servers will therefore have more challenges supporting the high memory capacity required by cloud applications, e.g., genomic workloads [36], [37].

VI. Conclusions

ORAM constructs impose very high bandwidth penalties. In this work, we make the case that ORAM shuffling should be off-loaded to smart and secure DIMMs so that the processor's progress is not impeded. We introduce two new distributed ORAM protocols that can leverage these SDIMMs to reduce bandwidth and access latency, while not introducing other side channels and while not causing buffer overflows. The Independent protocol has high latency, high parallelism, and low bandwidth penalty, so it favors workloads that exhibit high memory level parallelism. The Split protocol has low latency, low parallelism, and medium bandwidth penalty, so it favors workloads that exhibit low memory level parallelism. Furthermore, we introduce a new low power technique to reduce background power. Our best technique reduces execution time by 47%, CPU-memory bandwidth demand by 8×, and memory energy by 2.5×, relative to the state-of-the-art Freecursive baseline. Relative to a non-secure baseline, our contributions have lowered ORAM overheads to 2.7× in terms of performance and 35× in terms of CPU-memory bandwidth demands.

Acknowledgment

We thank the anonymous reviewers for many helpful suggestions. This work was supported in parts by NSF grants CNS-1302663, CNS-1314709, CNS-1514520, and CNS-1718834.

References

[1] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube – New DRAM Architecture Increases Density and Performance," in *Symposium on VLSI Technology*, 2012.

[2] SAP, "In-Memory Computing: SAP HANA," http://scn.sap.com/community/hana-in-memory.

[3] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazieres, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. Rumble, E. Stratmann, and R. Stutsman, "The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM," *SIGOPS Operating Systems Review*, vol. 43(4), 2009.

[4] C. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas, "Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious RAM," in *Proceedings of ASPLOS*, 2015.

[5] "Load-Reduced DIMMs," http://www.micron.com/products/dram-modules/lrdimm.

[6] M. Islam, M. Kuzu, and M. Kantarcioglu, "Access Pattern Disclosure on Searchable Encryption: Ramification, Attack, and Mitigation," in *Proceedings of NDSS*, 2012.

[7] O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *J. ACM*, 1996.

[8] O. Goldreich, "Towards a Theory of Software Protection and Simulation by Oblivious RAMs," in *Proceedings of STOC*, 1987.

[9] X. Zhuang, T. Zhang, and S. Pande, "HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus," in *Proceedings of ASPLOS*, 2004.

[10] L. Ren, X. Yu, C. Fletcher, M. van Dijk, and S. Devadas, "Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors," in *Proceedings of ISCA*, 2013.

[11] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An Extremely Simple Oblivious RAM Protocol," in *Proceedings of CCS*, 2013.

[12] E. Shi, T. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O((logN)^3)$ Worst-Case Cost," in *Proceedings of ASIACRYPT*, 2011.

[13] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowic, and D. Song, "PHANTOM: Practical Oblivious Computation in a Secure Processor," in *Proceedings of CCS*, 2013.

[14] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation," in *Proceedings of ASPLOS*, 2015.

[15] S. Aga and S. Narayanasamy, "InvisiMem: Smart Memory for Trusted Computing," in *International Symposium on Computer Architecture*, 2017.

[16] D. S. A. Awad, Y. Wang and Y. Solihin, "ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories," in *International Symposium on Computer Architecture*, 2017.

[17] J. Stuecheli, "POWER8 Processor," 2013, https://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.20-Processors1-epub/HC25.26.210-POWER-Studecheli-IBM.pdf.

[18] K. T. Malladi, I. Shaeffer, L. Gopalakrishnan, D. Lo, B. C. Lee, and M. Horowitz, "Rethinking DRAM Power Modes for Energy Proportionality," in *Proceedings of MICRO*, 2012.

[19] *Micron DDR2 SDRAM Part MT47H64M8*, Micron Technology Inc., 2004.

[20] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "USIMM: the Utah SImulated Memory Module," University of Utah, Tech. Rep., 2012, UUCS-12-002.

[21] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, "Memory Access Scheduling," in *Proceedings of ISCA*, 2000.

[22] "Wind River Simics Full System Simulator," 2007, http://www.windriver.com/products/simics/.

[23] "Micron System Power Calculator," http://www.micron.com/products/support/power-calc.

[24] R. Balasubramonian, A. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM TACO*, vol. 14(2), 2017.

[25] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *Proceedings of MICRO*, 2007.

[26] J. H.Dshalalow, *Advances in Queueing: Theory, Methods, and Open Problems*. CRC Press, 1995.

[27] C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs, "Optimizing ORAM and Using it Efficiently for Secure Computation," in *Proceedings of PET*, 2013.

[28] D. S. Fletcher CW, Dijk MV, "A secure processor architecture for encrypted computation on untrusted programs," in *STC*, 2012.

[29] K. Nayak, C. Fletcher, L. Ren, N. Chandran, S. Lokam, E. Shi, and V. Goyal, "HOP: Hardware Makes Obfuscation Practical," in *Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[30] X. Yu, S. K. Haider, L. Ren, C. W. Fletcher, A. Kwon, M. van Dijk, and S. Devadas, "PrORAM: dynamic prefetcher for oblivious RAM," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, 2015.

[31] X. Zhang, G. Sun, C. Zhang, W. Zhang, Y. Liang, T. Wang, Y. Chen, and J. Di, "Fork Path: Improving Efficiency of ORAM by Removing Redundant Memory Accesses," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[32] C. Fletcher, L. Ren, X. Yu, M. van Dijk, O. Khan, and S. Devadas, "Suppressing the Oblivious RAM Timing Channel While Making Information Leakage and Program Efficiency Trade-Offs," in *Proceedings of HPCA*, 2014.

[33] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[34] R. Wang, Y. Zhang, and J. Yang, "Cooperative Path ORAM for Effective Memory Bandwidth Sharing in Server Settings," in *International Symposium on High Performance Computer Architecture*, 2017.

[35] J. T. Pawlowski, "Hybrid memory cube (hmc)," in *Hotchips*, 2011.

[36] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler, "Faster and More Accurate Sequence Alignment with SNAP," *arXiv preprint arXiv:1111.5572*, 2011.

[37] N. A. Miller, E. G. Farrow, M. Gibson, L. K. Willig, G. Twist, B. Yoo, T. Marrs, S. Corder, L. Krivohlavek, A. Walter *et al.*, "A 26-Hour System of Highly Sensitive Whole Genome Sequencing for Emergency Management of Genetic Diseases," *Genome Medicine - 7*, 2015.