Supporting Trusted Virtual Machines with Hardware-Based Secure Remote Memory

Taekyung Heo
NVIDIA
Santa Clara, CA, USA
theo@nvidia.com

Seunghyo Kang
KAIST
Daejeon, Republic of Korea
shkang@casys.kaist.ac.kr

Sanghyeon Lee
KAIST
Daejeon, Republic of Korea
leesh6796@casys.kaist.ac.kr

Soojin Hwang
KAIST
Daejeon, Republic of Korea
sjhwang@casys.kaist.ac.kr

Joongun Park
Georgia Institute of Technology
Atlanta, GA, USA
jpark3234@gatech.edu

Jaehyuk Huh
KAIST
Daejeon, Republic of Korea
jhhuh@kaist.ac.kr

Abstract

Although recent studies have been improving the performance of RDMA-based memory disaggregation systems, their security aspect has not been thoroughly investigated. For secure disaggregated memory, the memory-providing node must protect its memory from memory-requesting nodes, and the memory-requesting node requires the confidentiality and integrity protection of its memory contents in the remote node, even when the privileged software is compromised. To provide protection of remote memory, this study proposes a hardware-assisted memory disaggregation system. The proposed trusted disaggregated memory combines the current trusted hardware-based virtual machine (VM) and a new dedicated hardware engine for trusted memory disaggregation. The processor with supports for trusted VM protects the context of a user VM within the local system, while the proposed hardware engine provides an efficient isolation and protection of remote memory pages, guaranteeing the confidentiality and integrity of remote memory pages. In the secure memory disaggregation system, fast address translation and access validation are supported with the cooperation of the hardware engine and guest OS in a trusted virtual machine. In addition, the proposed system hides the memory access patterns observable from remote nodes, supporting obliviousness. Our evaluation with an FPGA-based prototype implementation shows that such fine-grained secure disaggregated memory is feasible with comparable performance to the latest software-based technique without security support.

ISMM '24, June 25, 2024, Copenhagen, Denmark
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0615-8/24/06
https://doi.org/10.1145/3652024.3665509

***CCS Concepts:* • Software and its engineering → Distributed memory; • Security and privacy → Distributed systems security.**

Keywords: disaggregated memory, trusted computing, memory protection

ACM Reference Format:
Taekyung Heo, Seunghyo Kang, Sanghyeon Lee, Soojin Hwang, Joongun Park, and Jaehyuk Huh. 2024. Supporting Trusted Virtual Machines with Hardware-Based Secure Remote Memory. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management (ISMM '24), June 25, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3652024.3665509

1 Introduction

Memory disaggregation has emerged to enable efficient utilization of memory capacity across system boundaries. Imbalance in memory utilization among virtual machines within a cloud necessitates memory capacity sharing among the nodes in the same cluster. Recent studies showed that RDMA-supporting networks can allow effective expansion of memory beyond a single node with their low latency and high bandwidth data transfers [2, 15, 35]. Such memory disaggregation reduces the total cost of ownership of data centers by avoiding over-provisioning of memory capacity for each node.

Although there have been recent studies to improve the performance of the disaggregated memory systems [2, 7, 35], its security aspect has not been thoroughly investigated. Unlike local memory, the disaggregated memory system opens the memory boundary beyond the conventional system limit, and thus the memory contents of a user application can exist across multiple nodes. In addition, a node must allow other nodes to access part of its memory, being forced to trust the behavior of other nodes.

Such secure disaggregated memory poses several new challenges. First, the confidentiality and integrity of memory pages stored in remote nodes must be protected under vulnerable privileged software in the remote nodes. Second, when a node donates part of its memory for other nodes, its memory must be protected from any malicious attempt to access illegitimate regions of memory. Third, the address translation and permission validation from the local system to the remote memory must be efficient, while supporting fine-grained page-level management. Finally, memory access patterns of memory user nodes must be indistinguishable from the donor node.

Meanwhile, the trusted virtual machine (TVM) technology, such as Intel TDX and AMD SEV, allows a user virtual machine (VM) to be isolated from the hypervisor or other VMs in a physical system. The hardware layer protects the context and memory of a user VM from compromised privileged software. The memory is protected from certain types of physical attacks by the hardware-based encryption and integrity validation. Such TVM provides the basis of current confidential computing on clouds. With the growing need for flexible memory expansion, TVM must also be extended to support secure and efficient disaggregated memory.

To harden the disaggregated memory system, this paper proposes a hardware-assisted disaggregated memory system, called Trusted Disaggregated Memory (TDMem). It proposes to combine the hardware-based TVM support with a new hardware engine for RDMA-based memory disaggregation. A user virtual machine (VM) is protected by the local TVM hardware, and its remotely located memory pages are protected by the *trusted disaggregated memory engines* (TDMe) in the local and remote nodes.

Figure 1 shows the proposed TDMem architecture. The trusted computing base (TCB) is limited to TVM and TDMe, and the hypervisors in the local and remote nodes are not trusted. In TDMem, the hardware engine (TDMe) cooperates with the guest OS inside a TVM for trusted disaggregated memory. To allow low latency page accesses, the guest OS of a user TVM maintains the target page address in the remote node. However, TDMe in the remote node checks the source VM of page requests and validates accesses at page granularity. Such decoupling of address translation and permission validation allows fine-grained access validation without degrading the performance.

In our design, the guest OS in a TVM encrypts all pages to be stored in remote nodes. The encryption key and message authentication codes (MAC) for remote pages remain in the local memory protected by TVM. Therefore, the confidentiality and integrity of remote memory pages are guaranteed even from compromised privileged software in the remote node or from physical attacks in the remote DRAM.

Another important aspect of security is the patterns of page accesses. Prior work showed that such coarse-grained fault address patterns can leak critical information [42]. To hide the memory access patterns, remote page addresses from a user TVM are passed to the local TDMe in ciphertext, and only the local and remote TDMe can identify the remote

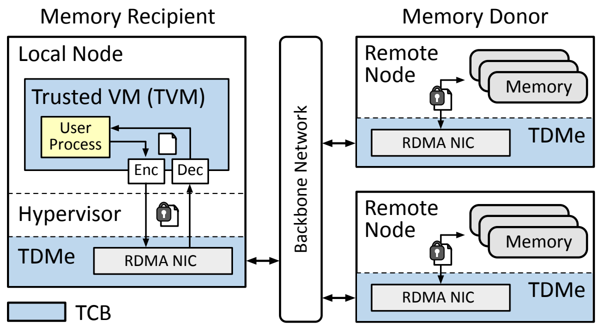


Figure 1. Trusted disaggregated memory overview

addresses. In addition to such address hiding, this study proposes the support for hiding reuse distances. The privileged software in the remote node can identify reuse distances of pages, inferring a certain level of access patterns. TDMe randomizes the page locations to hide such reuse distance profiles too.

The new hardware-assisted system is prototyped in a KVM system equipped with the Xilinx Alveo U50. Currently, the prototype system uses a conventional virtual machine due to the implementation limitation of the current TVM with an FPGA, but TDMe is implemented to an FPGA board. However, the required software changes are mostly in the guest OS, and the guest Linux kernel has been modified for the tight integration of TDMem with the existing virtual memory system. Our evaluation shows that even with the security supports with fine-grained memory management, the performance degradation is minimized to 4.4%, compared to the prior best-performing RDMA-based disaggregated memory system without support for confidentiality, integrity, and pattern obfuscation.

- This paper discusses the potential security problem of disaggregated memory and proposes a new TVM extension with the secure disaggregated memory hardware.
- The hardware engine allows fast address translation and access control for remote memory accesses.
- The hardware-assisted mechanism allows fast page transfers comparable to the prior mechanism based on region allocation.
- It obfuscates the memory access pattern in the remote node by exploiting the existing swap subsystem and our new hardware, so the data leaks by page access patterns are avoided.

2 Background

2.1 RDMA-Based Memory Disaggregation

Memory disaggregation allows data to reside in remote memory. A disaggregated memory system has several nodes connected with a high-speed network [2, 15, 35]. In this paper, we denote a node that donates its memory as a *donor* and a node that utilizes the memory of other nodes as a *recipient*. A node can be a donor and a recipient at the same time. From the perspective of a memory-consuming application, there are two types of memories: local memory and remote memory. The local memory is the memory in a recipient that can be accessed without additional network latency and address translations. The remote memory is the memory donated by a donor.

Remote Direct Memory Access (RDMA), which enables direct memory access to the memory in a remote node, has been widely used in prior memory disaggregation studies [2, 15, 35]. The communication channel of RDMA is called a queue pair. A queue pair is composed of a send queue and receive queue. The communication between hosts can be done with the abstraction called *verbs*. Verbs are classified into two types depending on the involvement of a remote CPU: one-sided verbs and two-sided verbs. One-sided verbs do not require a remote CPU's involvement, providing more scalable performance than the two-sided verbs. For the one-sided verbs, a memory region must be registered to access memory. The memory registration takes tens of microseconds [14].

In the RDMA-based memory disaggregation, to access remote memory pages, the remote pages are migrated to the local memory via fault handling, and thus the remote memory pool acts as a huge memory-based swap space of the local guest OS [15]. For virtualized systems, the hypervisor is responsible for safely migrating pages between the local and remote memory, while the guest OS, with better knowledge of applications, commonly determines the victim pages to be evicted to the remote memory pool. In our design, the guest OS picks victim pages and also encrypts those pages before leaving the local memory protected by TVM. The PCIe-attached hardware engine handles the access control and remote page mapping for remote memory accesses.

2.2 Trusted Virtual Machine

Trusted Execution Environment (TEE) is a secure and isolated environment within a computing system that provides a secure execution environment for applications and data. The granularity of TEE can vary, but recently trusted virtual machines have become an efficient TEE model for clouds [9, 37]. Intel TDX is a TEE introduced by Intel [9]. It aims to enhance the security of virtual machines (VMs) by providing hardware-based isolation and encryption for the memory of each virtual machine. It enables the creation of trusted, encrypted virtual machines called Trusted Domains (TDs), protecting sensitive data even from malicious hypervisors or other software running on the host system. With TDX, virtual machines can benefit from the hardware-based security for integrity and confidentiality of their memory contents.

TDX uses extended-page table (EPT) to allocate and map memory used by the TDs. To ensure secure translation of guest physical addresses (GPA) to physical addresses (PA), TDX employs a data structure called the Physical-Address-Metadata Table (PAMT). The PAMT guarantees that a page mapped into the EPT of a TD cannot be mapped into the any other TD. Within a TD, there are two types of EPT: Secure EPT and Shared EPT. The Secure EPT is responsible for providing secure execution within the TD by encrypting and protecting the integrity of the memory mapped to it.

On the other hand, the Shared EPT is utilized to establish communication with untrusted entities outside of the TD. [18] Operations such as hypercalls, networking, and disk I/O are performed through the memory covered by the shared EPT. Note that the memory content within this region is not protected by default, unlike the secure EPT. In our design, a user application is running in a TVM, and its memory is protected in secure EPT. However, to transfer data to the remote node via TDMe, page migration uses shared EPT, requiring encryption.

2.3 Requirement for Secure Memory Disaggregation

This section discusses four requirements of secure disaggregated memory: memory protection, secure and fine-grained memory allocation, fast and secure address translation, and memory access pattern obfuscation.

Confidentiality and integrity: The confidentiality and integrity of pages stored in remote memory must be guaranteed. As the privileged software such as hypervisor in the donor has a full control over donated memory, it may read or write any stored pages. Moreover, if a disaggregated memory system fails to isolate nodes due to design flaws or bugs, a malicious recipient may read or write unauthorized pages. Therefore, pages should be encrypted before being swapped out to remote memory. In addition, any modifications to pages have to be detected to prevent reading contaminated pages. Reading contaminated pages may result in a malfunction of applications or privilege escalation.

Secure and fine-grained memory allocation: First, memory allocation metadata must be securely protected from adversaries. The metadata include the start address, size, and ownership of allocated memory. As these metadata are used for memory access control, tampering of memory allocation metadata may result in unauthorized reads or writes of memory. Second, memory allocation at fine-granularity can improve memory utilization. Coarse-granular memory allocation may cause internal and external fragmentations when memory demands are highly fluctuating in dynamic cloud scenarios such as serverless systems or spot instances.

Fast and secure address translation: Memory disaggregation involves mappings between a local address space and a remote address space. The local address space is used by a recipient's CPUs to access data in local memory, but the page is located in the remote memory by a different address. As the address translations from a local address to a remote address are in the critical path of accessing data, fast address translation is essential to achieve high performance. In addition to fast identification of remote addresses, secure address translation is critical to prevent a compromised donor from providing corrupted pages and to prohibit a malicious recipient from reading unauthorized pages.

Memory access pattern obfuscation: The memory access patterns of recipient must be hidden from a donor node. Even if a donor node can observe the memory access patterns, the patterns should not convey any critical information by obfuscating the memory access pattern. Many prior studies have shown that the disclosure of memory access patterns may leak critical information such as the access frequency of data and correlation between them [19, 24, 44]. Moreover, several practical software attack schemes based on memory access pattern detection were proposed [19, 29]. A malicious observer can extract information about the location of private data from memory access patterns, which can result in the revelation of private data.

2.4 Threat Model

This study assumes that a user virtual machine is protected from a malicious hypervisor by the trusted virtual machine (TVM) architecture. It also assumes that the confidentiality and integrity of local memory is supported by TVM. Trusted computing base (TCB) of TDMem is the TVM hardware of the recipient node and TDMem devices (TDMe) on both donor and recipient nodes. The vulnerabilities in the user applications and guest OS within a TVM are not considered and orthogonal to the VM-based trusted computing. Although TVM protects the application memory, the memory region for DMAs with TDMe is not protected as the current TVM does not provide a secure DMA. Side-channels through caches and memory channels in CPUs are not considered in this study, as the current TVM technology does not provide such protection.

The confidentiality and integrity of local and remote memory pages are supported under compromised hypervisors and physical attacks on DRAM. The local memory protection is provided by TVM hardware, and remote pages exists only in ciphertext in the remote memory. When remote pages are migrated to the local guest OS, their integrity is validated by the stored MACs in the guest OS memory. The encryption key and MACs are stored in the guest OS memory, and thus TVM protects those security meta-data. Replay protection for physical DRAM attacks are not provided in the current TVM such as Intel TDX and AMD SEV. Therefore, we do not provide that.

However, unlike confidentiality and integrity protection for both software-based and physical attacks, memory access pattern obfuscation can be supported only for software-based attacks. For software-based attacks, the local guest OS memory cannot be accessed by the hypervisors, as TVM blocks such accesses. TDMe will hide remote page access patterns from the remote hypervisor. However, physical attacks such as DRAM probing can identify addresses on the channel to DRAM, so the current TVM and TDMe cannot hide patterns from such physical probing.

Even if the network between TDMes is compromised, the confidentiality and integrity detection of remote memory pages are still supported, as the local guest OS on a TVM will encrypt pages and validate their integrity. However, memory access pattern obfuscation requires to encrypt messages between TDMes, if the network can leak request contents via software-based attacks. For our FPGA prototype, the FPGA devices have a physically separate network and their packets do not go through the hypervisor software stack. Therefore, the prototype does not use encrypted messages between TDMes to reduce the implementation complexity. Note that even if messages are encrypted, the migrating page itself does not need to be re-encrypted, as it is already encrypted by the guest kernel on a TVM.

3 Design

3.1 Overview

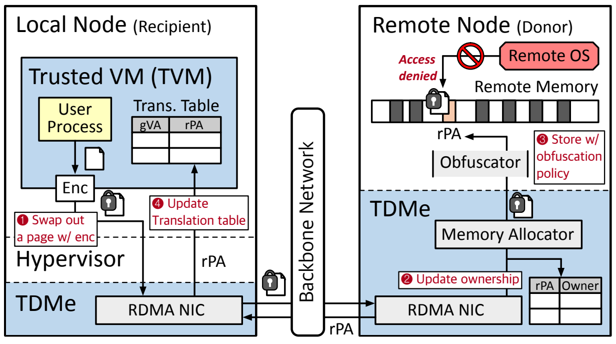
TDMem is a disaggregated memory system with hardware supports to enhance its security support. TDMem has multiple participating nodes in a system, and each node is equipped with a trusted disaggregated memory engine (TDMe). It combines the trusted virtual machine (TVM) technology with TDMe to extend the memory boundary of a TVM to the remote memory.

Figure 2 presents the overview of TDMem. On the donor side, the hardware TDMe with RDMA capability accesses the part of the memory assigned for the remote memory pool. The donor hypervisor assigns the memory pool chunks to TDMe, and TDMe will allocate pages by its own page-level allocation mechanism. In addition to the donor memory allocation, TDMe also validates the access permission from the recipient OS, and randomizes the page locations to hide page access patterns.

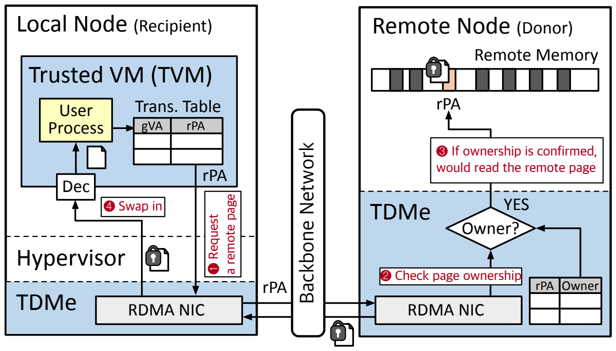
On the recipient side, a user VM is protected by the hardware TVM support. The virtual memory system (VMS) of the guest OS running on the VM is tightly integrated to the secure memory disaggregation engine using the frontswap interface. Frontswap allows the Linux kernel to redirect page swap requests to other subsystems. When the guest OS swaps out a memory page, it is encrypted and its MAC is stored in the guest OS memory protected by TVM.

Figure 2 (a) shows the swap-out process to the remote node. The guest OS on the recipient side (left) encrypts the page and keeps its MAC. On the donor side (right), TDMe assigns a page location with an obfuscation policy and places the incoming encrypted page. The ownership for the remote page location is updated in the remote TDMe. After that, the translation table in the guest OS is updated with the new remote location.

Figure 2 (b) shows the swap-in process from the remote node, initiated by a page fault in the local VM. The guest OS



(a) Remote swap-out process



(b) Remote swap-in process

Figure 2. Overview of TDMem

maintains the remote location for the page for fast translation, but the remote TDMe checks the access permission.

TDMe must support remote attestation from the guest OS on a TVM. During the attestation of a TVM, the local TDMe and connected remote TDMe are attested for the validity of the devices. The ASIC implementation must provide such a remote attestation function. For FPGA-based designs, the security features of FPGAs, such as bitstream encryption and authentication can be used and reprogramming of devices needs to be blocked by disabling a JTAG port [31].

3.2 Confidentiality and Integrity Validation

TDMem guarantees the confidentiality of recipient's pages and validates their integrity with the recipient-side software page encryption. By using TVM-supported encryption APIs, it provides a secure key generation protected from the hypervisor. When a page is evicted from a recipient node, the page is encrypted with AES-GCM in the recipient-side kernel. The encryption key is randomly generated during the initialization of TDMem. The key resides in the kernel and is tied to a node, not shared with others. As encrypted pages cannot be read without keys, the confidentiality of pages is guaranteed. The integrity of pages can be validated by encryption also. As AES-GCM supports the authentication of encrypted data, any modifications to encrypted pages can be detected on

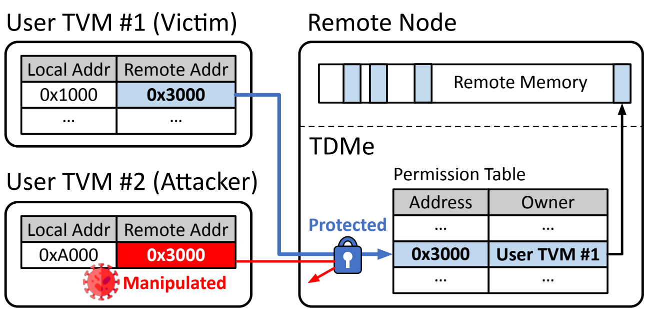


Figure 3. Decoupled translation and permission checks

decryption. On page encryption and decryption, a MAC is generated that is unique to the page content, to detect any integrity violation. MACs are stored in the recipient-side address translation table, protected by TVM.

With the current TVM, the guest OS is responsible for encrypting data before data leaves its trusted boundary [9]. Without the encryption within the protected domain, an untrusted host hypervisor can leak the data directly accessing the memory, since the guest OS uses a shared EPT for I/O data transfer.

3.3 Fine-Grained and Secure Memory Allocation

TDMem supports fine-grained memory allocation with the assistance of TDMe. TDMem has memory allocators in the secure memory disaggregation engines. The memory capacity of a donor node is dynamically allocated to different recipients or its own applications for efficient memory utilization. The page allocator in the donor must be nimble, and the hardware permission validation must be designed to support such page-level validation. TDMem has memory allocators in TDMe to allocate memory at 4KB page granularity on every page swap-out request. Memory allocators are located in TDMe to minimize the latency on memory allocation. Memory allocation requests can be handled immediately once a command reaches TDMe. The memory allocators are securely protected from malicious OSes because both the logic and metadata reside in TDMe and cannot be altered by OSes.

In addition, when TDMe has its own memory such as high bandwidth memory (HBM), it can also place certain pages in the local or remote HBM, depending on the preferred location set by the guest OS. It is used to allow the performance optimization by the guest OS.

3.4 Decoupled Translation and Permission Checks

Address translation must be fast and secure because it is on the critical path and controls memory accesses. TDMem achieves both goals by decoupling address translations and permission checks. Figure 3 illustrates the decoupled address translation and permission checks. To achieve high-performance, TDMEM avoids introducing an additional address space. TDMEM exposes the donor's physical address space to recipient directly and offloads address translations to recipient. Each recipient has a mapping table that maps swap offsets to remote physical addresses. By exposing the remote address space directly to recipient, performance degradation is avoided. As address translations are done in recipient, a compromised donor cannot forge the translation table.

The key problem with offloading address translations to recipient is the risk of recipient's accesses to unauthorized pages. A malicious recipient may try to read illegitimate pages by tampering the translation table. TDMEM tackles the problem with a permission table at the donor TDMe. The table tracks the ownership for all memory pages in the donor. On page loads, the permission table is looked up to validate the load request. The permission table is updated on memory allocation and deallocation. Another problem with having a permission table at the donor side is that a compromised donor may manipulate the permission table. TDMEM blocks this type of attack by having the permission table in TDMe. TDMe is part of TCB, and its operation cannot be tampered by malicious privileged software.

4 Memory Access Pattern Obfuscation

4.1 Page Address Obfuscation

Memory disaggregation systems with direct address translations have a limitation that memory access patterns may leak. If the mappings between the recipient's address space and the donor's address space are fixed, a malicious hypervisor in the donor may track the memory access pattern of a recipient by tracking the access pattern at the donor side. One of the solutions to hide memory access patterns is the Oblivious RAM (ORAM), which is an algorithm that obfuscates memory access patterns from adversaries. Although there have been many studies to improve the performance of ORAM and to scale them [8, 12, 33, 41, 43], the adoption of ORAM still incurs prohibitive performance degradation in the real world.

An alternative way to provide obfuscation is to hide addresses. Prior hardware designs proposed to encrypt addresses between the processor and DRAM modules to prevent physical DRAM probing from revealing memory access patterns [1]. For TDMEM, we employ a similar approach of hiding memory addresses used in the local VM. TDMEM first uses the swap subsystem scheme of the Linux kernel, using separated swap address space for allocating swap slots on swap-out requests, as well as supporting flexible mapping between physical address space and swap address space. In addition, to avoid attackers with the knowledge of the open-source logic of the Linux swap space allocation, TDMEM further obfuscates memory access patterns by changing the remote address of a local page with TDMe.

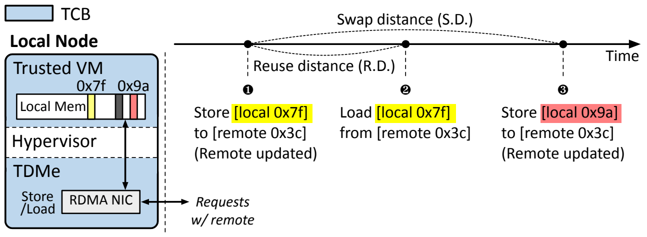

TCB
Local Node
Trusted VM
0x7f
0x9a
Local Mem
Hypervisor
TDMe
Store /Load
RDMA NIC
Requests w/ remote
Swap distance (S.D.)
Reuse distance (R.D.)
Time
❶
❷
❸
Store [local 0x7f] to [remote 0x3c] (Remote updated)
Load [local 0x7f] from [remote 0x3c]
Store [local 0x9a] to [remote 0x3c] (Remote updated)

Figure 4. The potential threat of access pattern leakage

4.2 Reuse Distance Obfuscation

In addition to the address hiding by swap address changes and encryption, reuse distances without knowing page addresses can also leak certain access patterns. [34] For the support for reuse distance obfuscation, we assume that the one-sided RDMA read by TDMe is not detected by the hypervisor. However, the hypervisor can know the occurrence of a page write by checking the memory contents changes. Note that our support for the obliviousness of remote page addresses cannot be supported for physical DRAM probing, although the remote hypervisor cannot detect access patterns.

Figure 4 illustrates how page writes can leak patterns. Note that the content of local memory and remote memory are denoted as `[local page_address]` and `[remote page_address]` in the figure, respectively. ❶ Local page 0x7f is swapped out, written at 0x3c of the remote memory. ❷ After some time, local page 0x7f is swapped in. ❸ More time later, another local page 0x9a is swapped out, written at 0x3c of the remote memory. As mentioned in Section 2.4, a kernel of remote node can read all contents on its memory. Therefore, in this example, a malicious remote node kernel can catch the change of contents at `[remote 0x3c]`: remote kernel knows the distance between ❶ and ❸ - *swap distance*, which is an upper bound of the distance between ❶ and ❷ - *reuse distance*.

By collecting the *swap distance* data for an amount of pages, the malicious remote kernel can approximate the distribution of *reuse distance*. Several prior work have claimed that the distribution of data reuse distance summarizes core information of workload, from the major type of operation [22] to the overall access pattern [11]. Thus, the attacker can use the approximated *reuse distance* information as a gadget of other attacks: for example, estimating the type of workload.

Goal: The remote hypervisor can detect the occurrence of an RDMA write through memory scanning, enabling it to estimate the distribution of *swap distance*. However, an RDMA read does not change the memory content, indicating that it is not detectable by memory scanning. Therefore, TDMEM's primary goal of memory obfuscation is to hide *reuse distance*

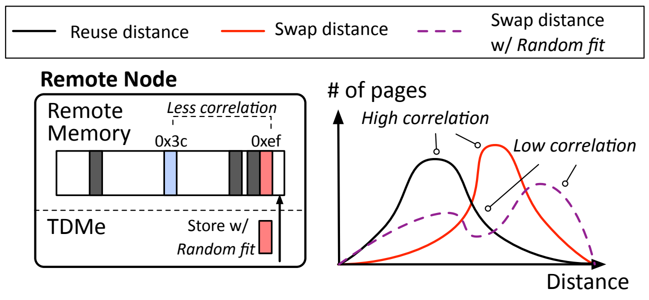


Figure 5. Memory access pattern obfuscation mechanism

Table 1. Maximum R^2 value (vs. Reuse Distance Histogram)

Workloads	Fixed Fit	Next Fit	Random Fit (Proposed)
cactuBSSN	0.951	0.226	0.233
omnetpp	0.938	0.760	0.291
deepsjeng	0.998	0.983	0.008
lbm	0.666	0.865	0.055

pattern by removing any effective correlation between *swap distance* and *reuse distance* distribution.

Main idea: The main idea of hiding *reuse distance* is to alter the location of swapping pages on every write. For each swap-out write, the remote TDMe chooses a random location among free pages to prevent any correlation between the swap distance and actual reuse distance distributions. Since the reallocation and RDMA write are performed by the remote TDMe, it does not incur any significant performance degradation of recipient.

We show that a *random-fit* swap allocation can effectively obfuscate the *reuse distance* information. This allocation scheme randomly selects swap addresses among the available free space. Figure 5 visualizes the obfuscation with the *random-fit* allocation. The red line is *swap distance* without obfuscation that might reflect the actual *reuse distance* - the black line. On the other hand, the purple line is the *swap distance* using a *random-fit* allocation, which has significantly less correlation with the actual *reuse distance* since a swap slot selection has no relation with the page address, access point in time neither.

4.3 Result

To show the effect of obfuscation using *random-fit* swap allocation, we perform a trace-based simulation. We select 5B instructions from four SPEC CPU benchmarks where they load and store actively. The simulated system includes 32MB of local memory and 512MB of remote memory, except for `deepsjeng` - due to its large footprint, we set remote memory capacity as 1GB to make the workload run. Two potentially-unsafe swap allocation schemes are also evaluated as baselines in addition to *random-fit*: *fixed-fit* assumes fixed mapping between local and remote (swap) address, while *next-fit* allocates swap address in round-robin manner from the free space. Note that the definition of 'distance' is cycles between access and re-access for certain remote address.

Table 1 represents the experiment result: *coefficient of determination* (R^2) between distribution of *reuse distance* and *swap distance* with *fixed-fit*, *next-fit* and *random-fit*, parsed in the granularity of 1 million. For more accurate measurement of correlation level, we shifted *swap distance* distribution to find the maximum R^2 value in Table 1. In general, two datasets are considered to be correlated when R^2 value is above 0.5 [28]. While two unsafe choices - *fixed-fit* and *next-fit* swap allocation reaches R^2 value above 0.5 for most cases, our proposed *random-fit* swap allocation always reports R^2 value under 0.3. This implies the necessity of our obfuscation scheme, as well as its effectiveness.

4.4 Discussion

Memory traffic obfuscation: There exists an side-channel attack that uses the amount of memory traffic as a gadget of secure data extraction [16]. Although it is a hardware-based side channel attack of which is not included in the threat model of TDMem, TDMem can prevent this attack by adding dummy read/write with some costs paid by performance degradation.

Detecting RDMA reads by intentional IOMMU faults: Although we assume that the remote hypervisor cannot detect one-sided RDMA reads, a potential way to generate an IOMMU fault, by removing access permission for pages. However, such IOMMU faults can be detectable by TDMe, as a faulted RDMA read will be retried. If such a malicious read fault occurs for the RDMA region used for the memory pool, TDMe can determine that the remote hypervisor is compromised, blocking further services and reporting the security problem.

Some architectures allow access bits in IOPTEs to be set by IOMMU. With the mechanism, the hypervisor can scan IOPTEs to detect accesses via access bits. To support the reuse distance hiding by TDMe, BIOS must support the disabling of such mechanism, and the hypervisor must not be allowed to enable it. It requires a minor change in the processor support. Note that address handling without the reuse distance obfuscation does not require such a change.

5 Prototyping TDMem

5.1 Overview

We prototyped TDMem with a Linux KVM system equipped with the Xilinx Alveo U50. Although TDMem aims to use a trusted virtual machine support by the processor, our current implementation uses a conventional virtual machine. However, the software-side implementation is concentrated on the guest OS, and thus the support for TVM will require neglisible changes. For performance evaluation, except for TVM-specific extra overheads, our prototype includes all major performance costs for the protected access to the disaggregated memory.

The hardware component (TDMe) is implemented in the Xilinx FPGA board. The FPGA board has an FPGA chip, network module, and 8GB on-board HBM. The FPGA chip is used to implement the TDMem logic, and the network module is used to connect nodes with 100Gbps Ethernet. The role of HBM memory is determined by the role of a node. A recipient uses the whole HBM memory as remote memory that can be accessed without network latency. On the other hand, a donor uses the HBM memory for donated memory and memory allocation metadata storage. To summarize, TDMe can use three memory tiers: recipient-HBM, donor-HBM, and donor-DRAM. On a page swap out request, a recipient can specify the target memory tier to store the page. IP cores are written in Vitis HLS and Verilog, and they are integrated in the Vivado flow.

5.2 TDMem Operations

TDMem is integrated to frontswap of the Linux kernel, which is the interface that redirects swap operations to other subsystems. Frontswap has four functions: store, load, invalidate page, and invalidate area. These functions become the default operations of TDMem. On each operation, a command is generated in the kernel driver and forwarded to the recipient engine over PCIe.

The commands generated by the kernel are handled by the FPGA engines. The size of a command is 64B, which is the default transfer size of the DMA engine. A command encodes required information to process the command. The completion of a command is identified by polling. The recipient kernel is responsible for reserving a memory block that is accessible from its FPGA board. The memory block is named as `completion`, and a `completion` contains several metadata required to process the operation. A `completion` for a store command contains the completion status of a request, stored memory tier, and remote memory address. These metadata are kept in the translation table in the recipient. To load the page, the recipient generates a request using the metadata.

5.3 Software Implementation

Address Translation: TDMem offloads the address translation responsibility to recipient. Each recipient has a flat address translation table that maps a local address (swap offset) to a remote physical. Figure 6 presents the format of the translation entry. The `valid` field presents whether the entry has valid translation information. A translation entry becomes valid when a store request completes. The `tier` field presents the memory tier where the page is stored, and the `remote_address` field has the address of the page in the tier. The `store_pending` and `load_pending` fields are used to coordinate with other concurrent swap requests. The

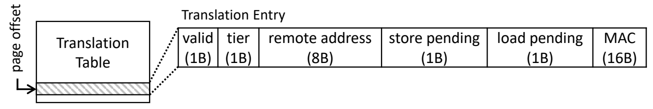


Figure 6. Translation table entry format

fields are used to prevent loading pages before store completion or invalidating pages before page load completion. The `MAC` field is used for the authentication of decrypted pages. The field is updated on encryption. The `MAC` field is compared with the MAC that is generated on the decryption of the page.

5.4 Hardware Implementation

TDMem uses the Xilinx QDMA Subsystem for PCI Express [3] for the communication between the host and card. QDMA is chosen over XDMA because it supports a queue-based submission mechanism, which supports thousands of concurrent requests. This feature is essential to serve concurrent page swaps. The descriptor bypass mode has been enabled to allow the card to write the host DRAM directly. For the networking feature, the UltraScale+ Integrated 100G Ethernet Subsystem [4] is used for the prototyping purpose. The FPGA prototype has two engines: recipient and donor.

5.4.1 Recipient Engine. The recipient engine receives commands from the recipient-side guest OS. It handles them locally or forwards to the donor engine.

Request generator : A request generator parses commands and creates requests to the remote node. The store request generator stores pages in the local HBM or remote memory. The target memory tier is determined by the kernel and encoded to the command field (`target_tier`).

The on-board HBM memory allocator manages the memory status. For store requests whose target is the local HBM, the HBM allocator attempts to find a free page in the local HBM. If a store request to the local HBM fails, the request is silently redirected to the donor-side engine so that memory allocation is done at the donor node.

The load request generator loads a page from the local HBM if the page is at the local HBM. Otherwise, it will forward commands to the donor engine. The invalidate page request generator and invalidate area request generator deallocate pages from the local HBM and send commands to deallocate pages from remote memory.

On-board HBM memory allocator: The on-board HBM memory allocator is responsible for the allocation and deallocation of on-board HBM memory. The memory allocation granularity is 4KB. The memory allocator is implemented with a bitmap. As the size of HBM is 8GB, the size of the bitmap is 2,097,152-bit (256KB). To allow concurrent access to the table, `ARRAY_PARTITION` pragma has been applied to the table with a cyclic option and factor of 32.

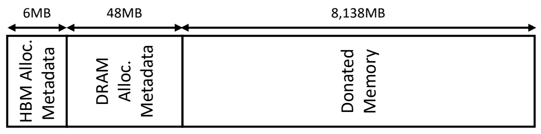


Figure 7. Donor-side HBM memory map

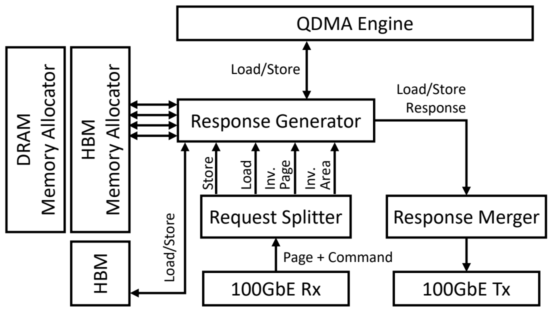


Figure 8. Block design of the donor engine

Response handler: The response handler reads returned responses and writes `completions` or pages to the recipient. Only two commands have responses: store and load. The completion of page invalidation does not have to be identified. The identification of page invalidation results in wasted hardware resource and network bandwidth. A store response is a single-word response that contains the tier and address of the stored page. A load packet has 65 words, where the first word has the metadata, and the following 64 words have the page content. The writes from the store response handler and load response handler are arbitrated in a round-robin manner.

5.4.2 Donor Engine. Unlike the recipient engine, where commands are sent from the host side, the donor engine receives commands from the network interface. Figure 8 shows a detailed design of donor engine.

Response Generator: The response generators are different from the recipient-side request generators in two aspects. First, the response generators coordinate not only with the on-board HBM memory allocator but also with the host DRAM memory allocator. Second, the invalidate page response generator and invalidate area response generator do not generate response packets. The store response generator and load response generator access donated memory, which is reserved by the kernel. The donor reserves memory with the kernel boot parameter, `memmap`. The donor engine knows the starting address and the size of the donated memory.

On-board HBM Allocator & Host DRAM Allocator: The donor-side memory allocators have the same responsibility as the recipient-side's, which is to manage the allocation status of memory. However, there are two differences from the recipient-side memory allocator. First, the donor-side memory allocator has to track the ownership of pages in addition to allocation status. The load response generator looks up the memory allocator to confirm that the current load request is trying to read a valid memory page that is owned by the requester. The ownership of pages is tracked with a machine ID (MID), whose size is 8-bit. As the metadata size becomes eight times of the recipient side's, it is not possible to hold all metadata in the FPGA logic. Instead, the metadata is stored in the lower address of the HBM. Figure 7 presents the HBM memory map. Second, the donor engine has the host DRAM memory allocator in addition to the on-board HBM memory allocator.

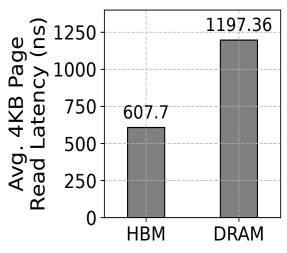


Figure 9. On-FPGA 4KB page read latency, which excludes the overhead of the software and network stack

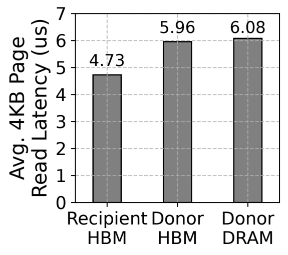


Figure 10. Recipient-side 4KB page read latency, which includes the overhead of the software and network stack

Table 2. System configurations

	Donor Host	Recipient Host	Recipient Guest
Memory	DDR4 188GB	DDR4 125GB	DDR4 32GB
Kernel	5.9.0		5.10.0
Processors	Intel(R) Xeon(R) CPU E5-2630 v4		
OS	Ubuntu 18.04		
QDMA Driver	2020.2		

6 Evaluation

6.1 Experimental Setup

We evaluate the performance of TDMem on a Linux system equipped with an FPGA card and high-performance network card. The evaluation is conducted with two pairs of machines. The first pair of machines is equipped with Mellanox ConnectX-5 for the evaluation of fastswap. The second pair of machines has Xilinx Alveo U50 to evaluate TDMem. In each pair, one machine becomes a donor and the other becomes a recipient. Table 2 presents the machines and their configurations. Table 3 presents the evaluated macrobenchmarks. The table shows their memory footprint and the number of CPUs that they utilize.

6.2 Microbenchmark Results

6.2.1 On-FPGA Page Read Latency. Figure 9 presents the 4KB page read latency where the software overhead

Table 3. Macrobenchmarks and their memory footprint

Workload Name	Mem. Footprint (GB)	Num of CPUs
`tensorflow-inception`	1.5	2
`kmeans`	5.3	8
`quicksort`	8.6	1
`in-memory-analytics`	7.6	8
`graph-analytics`	10.3	8
`xsbench`	5.5	8

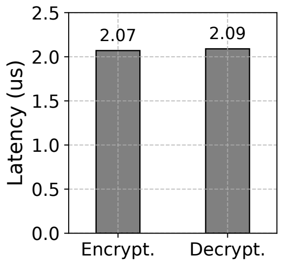


Figure 11. 4KB page encryption latency and decryption latency

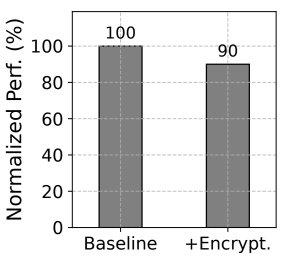


Figure 12. The impact of memory encryption on the microbenchmark performance

is not included. The page read latency is measured by designing and deploying a microbenchmark in the FPGA. The microbenchmark module reads 4KB pages sequentially for a given range of memory addresses. The total elapsed cycles are measured, and the elapsed cycles are divided by the number of pages read. As the microbenchmark is designed to operate at 250MHz, the clock period is 4ns. The average 4KB page read latency is calculated by multiplying the clock period by the average elapsed cycles. On average, HBM takes 607.7ns, and DRAM takes 1197.36ns to read a 4KB page.

6.2.2 Recipient-Side Page Read Latency.

Unlike the previous experiment, which excludes the overhead of the software and network stacks, this experiment includes them by measuring the page read latency in the recipient-side Linux kernel. Figure 10 presents the average page read latency in three types of memory tiers: recipient-HBM, donor-HBM, and donor-DRAM. The page read latency has been measured by sequentially reading 2,097,152 pages (8GB) from the target memory tier. The pages are stored in the target memory before running experiments, and the elapsed time has been measured with jiffies in the Linux kernel. `CONFIG_HZ`, which is the kernel configuration that defines the timer resolution, is set to 250. The experiment result implies that the performance gap between HBM and DRAM presented in Figure 9 is mostly hidden, and the performance overhead comes from the network latency and page fault handling.

6.2.3 Page Encryption Latency.

We measured the page encryption latency in the Linux kernel. The vanilla Linux kernel has the `tcrypt` module, which evaluates the performance of encryption algorithms. We evaluated the performance of `gcm(aes)` with the key size of 128-bit. Figure 11 presents the average page encryption and decryption latencies. It takes 2.07us for 4KB page encryption and 2.09us for 4KB page decryption. Although the latency seems relatively high considering that the latency of page read is between 4-6us, most of the latency can be hidden by the readahead mechanism of the Linux kernel.

Figure 12 shows the normalized performance of a microbenchmark that loads 262,144 pages (1GB) sequentially from the recipient HBM. The normalized performance is defined as the performance normalized to the baseline without encryption. The performance is the reverse of the elapsed time. The performance of the microbenchmark with page encryption shows 90% of the performance without encryption. Most pages are readahead and decrypted in the swap cache, effectively hiding the memory decryption latency.

6.3 Macrobenchmark Results

Geomean of normalized performance: Figure 13 presents the geomean of normalized performance of workloads for a given configuration. The normalized performance is defined as the performance of a workload with a given configuration divided by the performance of the workload run with 100% local memory. The geomean of normalized performance is the geomean of all normalized performance of workloads. Workloads are run with five configurations: `fastswap`, `TDMem-HBM-plain`, `TDMem-HBM-crypt`, `TDMem-DRAM-plain`, and `TDMem-DRAM-crypt`. The experiments are run while varying the local memory ratio between 40% and 100% with the 10% step size. Fastswap shows 63.1% performance compared to the non-swapped run, and TDMem experiences negligible performance loss, presenting the 3-7% lower performance compared to fastswap. Please note that the performance overhead from software encryption is mostly hidden because of the readahead mechanism in the Linux kernel, as we have shown in Section 6.2.3.

Normalized performance (`TDMem-DRAM-crypt`): Among the experiment configurations presented in Figure 13, we choose `TDMem-DRAM-crypt` and illustrate workloads' performance degradation while varying the local memory ratio in Figure 14. The performance loss from losing local memory differs for each workload. While `quicksort` presents 75.48% of its performance with the 40% local memory ratio, `kmeans` shows 7.19% of its performance. We qualitatively analyze the root cause with the tools presented in prior studies [17, 30], which allow us to analyze the memory access frequency. We found that the reason behind

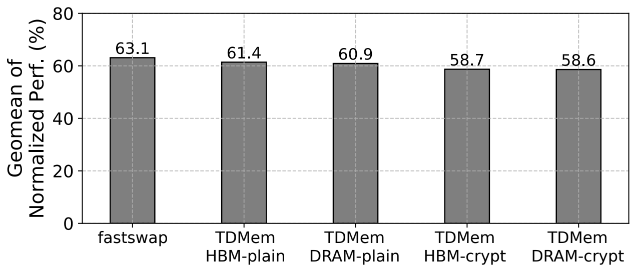


Figure 13. Geomean of normalized performance of workloads for each configuration

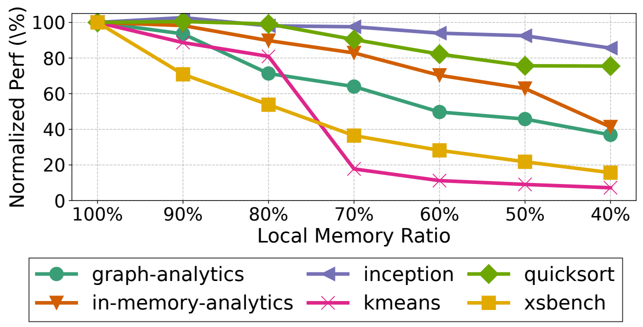


Figure 14. Normalized performance of workloads with various local memory ratios with TDMem. Swapped out pages are stored in the donor DRAM.

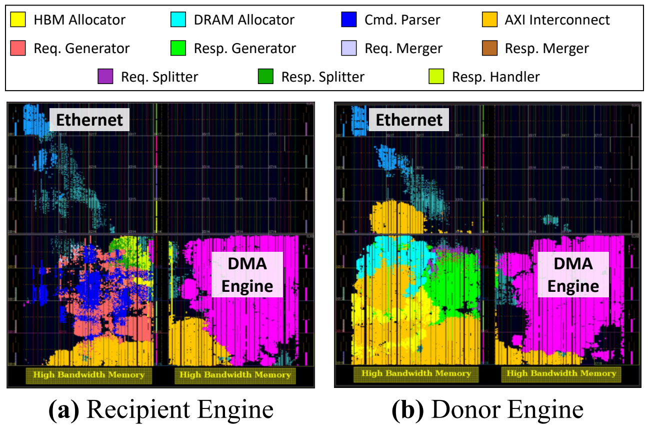


Figure 15. Floorplan of TDMe

the different sensitivity on the local memory ratio is the various memory access patterns and intensity. While the memory footprint of quicksort is 8.6GB, only a small part of memory is intensively accessed. On the other hand, the total allocated memory of kmeans is intensively accessed, being more sensitive to the local memory loss.

6.3.1 Max Memory Bandwidth. In this experiment, we measure the memory bandwidth of TDMem and compare it

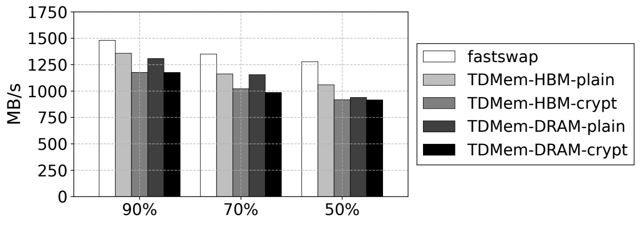


Figure 16. Memory bandwidth measured with the STREAM Triad benchmark

with fastswap's. The memory bandwidth is measured with the STREAM benchmark. STREAM allocates 4GB memory and runs several sub-benchmarks to measure the bandwidth. Among the sub-benchmarks, we use the Triad to measure the average memory bandwidth. We measure the memory bandwidth of five configurations: `fastswap`, `TDMem-HBM-plain`, `TDMem-HBM-crypt`, `TDMem-DRAM-plain`, and `TDMem-DRAM-crypt`. `fastswap` presents the case where the swapped-out pages are stored in remote memory with RDMA. The configurations starting with `TDMem` is run with TDMem. The `HBM` and `DRAM` keywords show the target memory tier where the swapped-out pages are stored at. The `crypt` keyword means that the encryption latency is added on page loads. The experiments are run with various local memory ratios, which include 90%, 70%, and 50%. The local memory ratio is defined as the memory size in the recipient DRAM divided by the maximum memory footprint size. The memory footprint has been measured with the cgroup's `memory.usage_in_bytes`.

Figure 16 shows the memory bandwidth with the configurations. `fastswap` performs the best with the 90% and 70% local memory ratios. At the 90% local memory ratio, `fastswap`, `TDMem-HBM-crypt`, and `TDMem-DRAM-crypt` present 1480.5MB/s, 1176.4.4MB/s, 1175.9MB/s, respectively. Please note that the memory bandwidth gap between `TDMem-HBM-crypt` and `TDMem-DRAM-crypt` is negligible, implying that the major bottleneck for the memory bandwidth is not the memory itself. The memory bandwidth loss of `TDMem-DRAM-crypt` compared to `fastswap` are 21%, 23%, and 28% at the 90%, 70%, and 50% local memory ratio, respectively. The maximum memory bandwidth is reduced for our prototype, requiring further optimizations on the guest OS change and data path implementation on the FPGA board.

6.4 FPGA Resource Utilization

Figure 15 illustrates the floorplan of the recipient engine and donor engine, and each component is filled with different colors. The recipient engine consumes 18% of LUT, 5% of LUTRAM, 12% of FF, 30% of BRAM, and 2% of URAM. The donor engine accounts for 19% of LUT, 9% of LUTRAM, 17% of FF, 24% of BRAM, and 2% of URAM.

In both engines, DMA engines are the major consumer of logic resources. In the recipient engine, the HBM memory allocator accounts for 15.8% of the BRAM consumption of the recipient engine. BRAM is used to manage the memory allocation status of HBM memory. On the other hand, as the donor engine manages the memory allocation status in the on-board HBM, the BRAM usage of memory allocators is relatively low. The response generator consumes BRAM for the free lists in the store response generator. All BRAMs in the response generator are consumed by the store response generator.

7 Related Work

Disaggregated Memory System: As disaggregated memory system is an emerging technique for modern memory-intensive workloads, several HW/SW approaches were proposed as prior work. InfiniSwap introduces a paging system designed for RDMA [15]. DeACT proposes a virtual memory support for a specific disaggregated memory system [21]. Kona, on the other hand, exploits cache coherence instead of virtual memory [7]. ThymesisFlow presents interconnect stack for huge scale disaggregated memory [32]. AIFM enhances the performance of remote memory through application-centric approach [35], while Fastswap focuses on swapping mechanism [2].

To protect RDMAs from network attackers, sRDMA incorporates the encryption support for network interface cards [39]. However, our approach provides the protection of the entire VM accessing the remote memory with obliviousness support.

Trusted Execution Environment: Several important studies have been conducted to enable the execution of native applications within trusted execution environments (TEEs). Haven ensures application integrity and confidentiality by running it within an enclave and isolating it from the underlying system [6]. Graphene-SGX is a library operating system that securely runs unmodified applications within Intel SGX enclaves, offering strong isolation and security guarantees [40]. SCONE enables secure and confidential execution of containerized applications [5]. To reduce memory management overhead in Intel SGX's limited memory space, various software approaches have been investigated. Eleos addresses performance issues by introducing exit-less system calls and customized paging with a user-level library [27]. ShieldStore, an in-memory key-value store designed for Intel SGX, minimizes SGX's page swapping with fine-grained memory encryption of unprotected memory [20]. Vault proposes a variable arity unified tree that combines MAC sharing and compression to reduce the paging overhead [38]. MMT proposes migratable merkle tree, a design that utilizes an integrity forest and allows secure delegation of memory subtrees between enclaves without the need for software re-encryption [13]. There are recent studies to support obliviousness with TEEs. Oblix proposed an oblivious search index structure with Intel SGX [26]. ZeroTrace provides ORAM supports for SGX enclaves [36].

Disaggregation with CXL: Compute eXpress Link interface (CXL) has been proposed to be used in memory disaggregation system [10]. The interface emerges to enhance the speed of exchanging data from connected devices, especially remote nodes in disaggregated system. Kona has shown a possibility of disaggregated memory system using a future FPGA that supports CXL interface [7]. By leveraging the full benefit of CXL, Kona shortens the critical path in loading remote data and reduces dirty data amplification to prevent memory waste. TPP proposes a system that identifies and allocates hot and cold pages to appropriate memory tier [25]. TPP ensures memory headroom for new short-lived hot page allocations, and promoting hot pages from slow CXL-memory to fast local memory. Pond structures CXL communication layers in a NUMA system to estimate the latency of accessing data and builds an emulated NUMA system, with a prediction model to reduce the performance impact of the pool memory [23].

8 Conclusion

This paper proposed a new hardware-assisted memory disaggregation system, TDMem. Combined with trusted virtual machines, it allows fine-grained page-level management of memory pools in donor nodes, while access validation is enforced by the secure hardware engine protected from vulnerable privileged software. In addition, it further secures the confidentiality of memory pages with page address obliviousness supports as well as encryption. We prototyped TDMem with a FPGA-based implementation. The security features of TDMem cause a minor performance overhead, which causes 4.4% performance degradation compared to the latest page-granular far memory system, fastswap without the security supports for remote memory pages.

Acknowledgement

This work was supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) (IITP2017-0-00466 SW StarLab and IITP2021-0-01817) and National Research Foundation of Korea (RS-2024-00347114) funded by the Ministry of Science and ICT (MSIT), Korea.

References

[1] Shaizeen Aga and Satish Narayanasamy. 2017. InvisiMem: Smart Memory Defenses for Memory Bus Side Channel. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. 94–106.

[2] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can Far Memory Improve Job Throughput?. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*.

[3] AMD. 2021. QDMA Subsystem for PCI Express. https://www.xilinx.com/products/intellectual-property/pcie-qdma.html. [Online; accessed 6-April-2021].

[4] AMD. 2021. UltraScale+ Integrated 100G Ethernet Subsystem. https://www.xilinx.com/products/intellectual-property/cmac_usplus.html. [Online; accessed 6-April-2021].

[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. Scone: Secure linux containers with intel sgx.

[6] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* (2015).

[7] Irina Calciu, Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking Software Runtimes for Disaggregated Memory. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[8] T-H Hubert Chan and Elaine Shi. 2017. Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs. In *Proceedings of the Conference on Theory of Cryptography*.

[9] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. 2023. Intel TDX Demystified: A Top-Down Approach. *arXiv preprint arXiv:2303.15540* (2023).

[10] CXL. 2019. Compute Express Link (CXL). https://www.computeexpresslink.org/. [Online; accessed 4-August-2019].

[11] Chen Ding and Yutao Zhong. 2003. Predicting Whole-Program Locality through Reuse Distance Analysis. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programing language design and implementation*.

[12] Jack Doerner and Abhi Shelat. 2017. Scaling ORAM for Secure Computation. In *Proceedings of the Conference on Computer and Communications Security (CCS)*.

[13] Erhu Feng, Dong Du, Yubin Xia, and Haibo Chen. 2023. Efficient Distributed Secure Memory with Migratable Merkle Tree. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[14] Philip Werner Frey and Gustavo Alonso. 2009. Minimizing the Hidden Cost of RDMA. In *2009 29th IEEE International Conference on Distributed Computing Systems*.

[15] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[16] Jiaxi Gu, Jiliang Wang, Zhiwen Yu, and Kele Shen. 2019. Traffic-Based Side-Channel Attack in Video Streaming. *IEEE/ACM Transaction on Networking* 27, 3 (2019), 972–985.

[17] T. Heo, Y. Wang, W. Cui, J. Huh, and L. Zhang. 2020. Adaptive Page Migration Policy with Huge Pages in Tiered Memory Systems. *IEEE Trans. Comput.* (2020), 1–1. https://doi.org/10.1109/TC.2020.3036686

[18] Intel. 2022. Intel® Trust Domain Extensions. https://cdrdv2.intel.com/v1/dl/getContent/690419. *White Paper, Febuary* (2022), 9.

[19] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *Proceedings of the Symposium on Network and Distributed System and Security (NDSS)*.

[20] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference 2019*.

[21] Vamsee Reddy Kommareddy, Clayton Hughes, Simon David Hammond, and Amro Awad. 2021. DeACT: Architecture-Aware Virtual Memory Support for Fabric Attached Memory Systems. In *Proceedings of the 27th International Symposium on High Performance Computer Architecture (HPCA)*.

[22] Tamara Silbergleit Lehman, Andrew D. Hilton, and Benjamin C. Lee. 2018. MAPS: Understanding Metadata Access Patterns in Secure Memory. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[23] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*.

[24] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. 311–324.

[25] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*.

[26] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *2018 IEEE Symposium on Security and Privacy (SP)*. 279–296. https://doi.org/10.1109/SP.2018.00045

[27] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*.

[28] Eva Ostertagova. 2012. Modeling using Polynomial Regression. *Procedia Engineering* 48 (2012), 500–506.

[29] Dag Arne Osvic, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT–RSA 2006*.

[30] SeongJae Park, Yunjae Lee, and Heon Y Yeom. 2019. Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality. In *Proceedings of the 20th International Middleware Conference Industrial Track*.

[31] Ed Peterson. 2013. Developing Tamper Resistant Designs with Xilinx Virtex-6 and 7 Series FPGAs. *Application Note. Xilinx Corporation* (2013).

[32] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H Peter Hofstee. 2020. ThymesisFlow: A Software-Defined, HW/SW co-Designed Interconnect Stack for Rack-Scale Memory Disaggregation. In *Proceedings of the 53rd International Symposium on Microarchitecture (MICRO)*.

[33] 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 *Proceedings of the 24th USENIX Security Symposium*.

[34] Nirjhar Roy, Nikhil Bansal, Gourav Takhar, Nikhil Mittal, and Pramod Subramanyan. 2020. When Oblivious is Not: Attacks against OPAM. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/roy

[35] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI).*

[36] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace : Oblivious Memory Primitives from Intel SGX. *IACR Cryptol. ePrint Arch.* 2017 (2018), 549. https://api.semanticscholar.org/CorpusID:3202596

[37] AMD Sev-Snp. 2020. Strengthening VM isolation with integrity protection and more. *White Paper, January* (2020), 8.

[38] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems.*

[39] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. 2020. SRDMA: Efficient NIC-Based Authentication and Encryption for Remote Direct Memory Access. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20).* USENIX Association, USA, Article 47, 14 pages.

[40] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX.. In *USENIX Annual Technical Conference.*

[41] Rujia Wang, Youtao Zhang, and Jun Yang. 2017. Cooperative Path-ORAM for Effective Memory Bandwidth Sharing in Server Settings. In *Proceedings of the 23rd International Symposium on High Performance Computer Architecture (HPCA).*

[42] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the Symposium on Security and Privacy.*

[43] Xiangyao Yu, Syed Kamran Haider, Ling Ren, Christopher Fletcher, Albert Kwon, Marten Van Dijk, and Srinivas Devadas. 2015. PrORAM: Dynamic Prefetcher for Oblivious RAM. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA).*

[44] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. 2004. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

Received 2024-03-22; accepted 2024-05-10