*u*MMU: Securing Data Confidentiality with Unobservable Memory Subsystem

Hajeong Lim
hajeong.lim@skku.edu
Sungkyunkwan University
Suwon, Korea

Jaeyoon Kim
jena9925@skku.edu
Sungkyunkwan University
Suwon, Korea

Hojoon Lee*
hojoon.lee@skku.edu
Sungkyunkwan University
Suwon, Korea

Abstract

Ensuring data confidentiality in a computing system's memory hierarchy proved to be a formidable challenge with the large attack surface. Diverse and powerful attacks threaten data confidentiality. Memory safety is notoriously hard to achieve with unsafe languages, thereby empowering adversaries with unauthorized memory accesses, as represented by the HeartBleed incident. More recently, microarchitectural side channel attacks reign as a prevalent threat against data confidentiality that affects program execution including the safeguarded ones inside TEEs.

In this paper, we introduce an in-process memory subsystem called *u*MMU. *u*MMU coherently consolidates the notion of employing processor registers as *unobservable* storage with data confidentiality protection techniques such as memory encryption and Oblivious RAM. *u*MMU creates a new address space called *u*Virtual address space that is unobservable to adversaries. Under the abstraction created by *u*MMU, the processor's spacious extended registers, such as Intel x86's AVX512, are transformed into unobservable and *addressable* physical memory backing. Completing the principles of virtual memory abstraction is the memory management that maintains a secure swap space applied with memory confidentiality policies such as encryption or ORAM. *u*MMU is a versatile and powerful framework that can host data confidentiality policies on sensitive data. Our real-world evaluation indicates that *u*MMU significantly improves the performance of programs with encryption and ORAM schemes for sensitive data protection: an average of 69.93% improvement in encryption-based protection of sensitive data in MbedTLS, and 497.84% for ORAM-based elimination of access patterns on Memcached's hashtable.

CCS Concepts

• **Security and privacy → Systems security**.

Keywords

memory protection, side-channel defense, secure computation

ACM Reference Format:
Hajeong Lim, Jaeyoon Kim, and Hojoon Lee. 2024. *u*MMU: Securing Data Confidentiality with Unobservable Memory Subsystem . In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690340

*Corresponding author

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690340

1 Introduction

Maintaining data confidentiality in memory is increasingly more difficult today with the large attack surface. The memory hierarchy is a shared infrastructure across mutually distrusting entities and potentially vulnerable code. Memory safety remains a daunting challenge in the domain of systems security and is a direct threat to data confidentiality. Adversaries aptly exploit such weaknesses, and HeartBleed [2] stands as a quintessential example of its kind. Side-channel attacks represent more subtle yet powerful adversary primitives that threaten the confidentiality of sensitive data. Recent works have demonstrated the practicality of leveraging microarchitectural side-channel attacks for the extraction of executing program's sensitive data [31, 38]. Even with the latest hardware-supported *Trusted Execution Environments (TEEs)*, the threat persists. Intel SGX has been plagued with memory side-channel attacks that undermine the confidentiality of the memory it protects even in the presence of the hardware security measures in place [10, 12, 57, 66].

Among many approaches, we generalize and focus on the use of *unobservable* storages. Safeguarding data in unobservable storages such as registers isolates the data from the shared memory hierarchy. As such, it has been a versatile primitive that has been adopted in many security measures.

Registers as unobservable storage. In theory, a register-only computation on sensitive data would immensely reduce the attack surface against both memory disclosure and side channel attack models. Register-only cryptography [18, 23, 46, 58] has been developed to protect the crypto keys from memory disclosure and cold-boot attacks. The unobservability of register-only operations is also a key building block for composing inherently oblivious algorithms [6, 44, 47, 56, 59]. However, register-only computation of sensitive data hardly scales for general programs, as the capacity of general-purpose registers is far from sufficient for sensitive data and its by-products. Nevertheless, trust in the security of in-register data has been an underlying premise in many works.

Memory confidentiality policies. The premise that in-register data is secure can be complemented with *Memory Confidentiality Policies (MCPs)*. That is, an MCP can be applied such that the sensitive data is transformed into a safe form before its exportation to untrusted memory. MCPs are often implemented through compiler instrumentation that transforms the target program's load and store instructions. For instance, a line of work such as DynPTA and others [11, 51, 69, 73] instrument the target software's load and store instructions such that data is encrypted before entering

memory. Raccoon [53] is an obfuscated execution engine for TEE for side channel mitigation, and it replaces the target program's load and stores with an ORAM interface. As such, the program's memory data access patterns become uniform and indiscernible to the attacker. Enforcing MCPs, however, imposes a substantial overhead to the memory-bound performance of programs. For instance, PathORAM [60] applied as MCP on load and stores renders memory accesses multiple magnitudes slower (e.g., over 1000×).

Use of extended registers. Another technique often employed to extend the maximum capacity of an in-register protection scheme is to utilize the processor's extended registers. In the current x86 architecture, the AVX512 ISA extension provides 32 512-bit registers, which amount to half the size (2KB) of a memory page. PRIME [18] implements RSA public-key crypto to be completely register-only through the Intel AVX ISA extension [26]. Obfuscuro [5] incorporates the AVX registers for its ORAM's stash which is a sensitive ORAM component whose access patterns may undermine the obfuscation engine's guarantees.

Our proposal. In this paper, we introduce a novel in-process memory subsystem for memory data confidentiality called ***u*MMU**. The key insight is to construct an abstraction that coherently consolidates the register-bound data protection and the concept of MCPs for data confidentiality. The abstraction implements the principles of virtual memory. The processor's spacious extended registers are transformed into an *addressable* physical memory for sensitive data. The process memory, whose throughput is impeded by the MCPs, is analogous to a disk in our subsystem and hence used as a swap space. The resulting subsystem transparently optimizes the use of registers and MCP-enforced memory while providing life-cycle confidentiality for sensitive data.

*u*MMU makes a case for the virtual memory as the most suitable abstraction for combining the unobservable register storage and MCP-enforced memory. An attempt to turn the extended registers into storage through compiler modifications [73] quickly reveals its limitations. If we are to tweak the compiler's register allocation pass to selectively store sensitive data in the extended registers, the data becomes inherently bound to each function. For sensitive data in extended registers to cross the function boundary, a calling convention must mediate their use. This calling convention would force the 2KB of data, regardless of the actual usage, to be spilled to memory before the function returns and loaded during the function prologue.

On the other hand, *u*MMU naturally inherits the advantages of virtual memory by constructing *uVirtual Address (uVA)* Space. The new address space allows the sensitive data to have a process-wide context. Also, the limited capacity of the general-purpose and extended registers are flexibly extended through a MCP-enforced swap space. The faster registers are efficiently filled with just enough data for the ongoing computation due to the inherent locality of the computation in many cases.

However, the extended registers are naturally inapt for serving as a physical memory because they lack *addressability*. For instance, given an arbitrary index N and an offset, accessing the *specified chunk* in `zmm`N is not trivial. The design space for such a *memory controller* is rather wide. In addition to formulating a performant implementation, we found that an imprudent design exposes the controller to the threats of memory-side channels.

To this end, *u*MMU implements a meticulously designed *Register-as-Memory (RasM)* controller mechanism to enable extended registers as an addressable and unobservable physical storage backing which we call *Unobservable Storage (uStorage)*. *u*MMU transforms the extended registers into unobservable memory. In accordance with the security requirements of MCPs, RasM is also specifically designed to respect the security guarantees of MCPs by ensuring the obliviousness of its operations.

The design of *u*MMU is a powerful primitive for security schemes for in-memory data. We adopt and evaluate *u*MMU with three representative MCPs from previous works: (1) memory encryption scheme [51, 73], (2) ORAM [60]-backed storage for TEEs [53, 56], and (3) plaintext salting [69]. These MCPs are adapted into *u*MMU's swap management backend. Our evaluations show that *u*MMU-hosted MCPs significantly outperforms the standalone MCPs implementations in memory-bound algorithms. In general sorting algorithms, *u*MMU-accelerated is Path ORAM shows 192.79× performance of the standalone. Likewise, accelerated sensitive encryption showed 1.69× performance compared to the standalone, and plaintext salting showed 1.82×. Our real-world evaluation demonstrates that *u*MMU boosts sensitive key data encryption in MbedTLS by 69.93%, and ORAM-based hashtable access pattern protection in Memcached by 497.84%.

In all, we summarize our contributions as follows:

- We introduce a virtual memory subsystem that generalizes the registers into unobservable storage that is backed by a secure swap governed with Memory Confidentiality Policies.
- We devise a novel software-based memory controller called RasM to transform extended registers into an addressable physical memory.
- We conduct a comprehensive evaluation to report the potential of the new abstraction for securing data confidentiality in security solutions.
- We publicly release *u*MMU implementation and ported evaluation targets in the hopes of adoption in future works[1].

2 Background and Related Work

In this section, we discuss the background and previous works that are indispensable in explaining the design and contributions of this work.

2.1 AVX512

AVX512 [26] is the latest installment to the Intel x86 *Advanced Vector Extensions* (AVX) ISA extensions for SIMD. The extension enhances computational efficiency in many high data-throughput tasks, including video processing and cryptography. The extension brings 32 512-bit registers called `zmm0` through `zmm31`, providing a total of *2048 bytes* of storage. New instructions have been added to the ISA in accordance to facilitate the computation with the registers; dedicated instructions support arithmetic and logical computation as well as masking operations on the registers. *u*MMU incorporates the processor's extended register set as its *Unobservable Storage (uStorage)*, and its current x86 implementation adopts the AVX512

[1] https://github.com/sslab-skku/uMMU

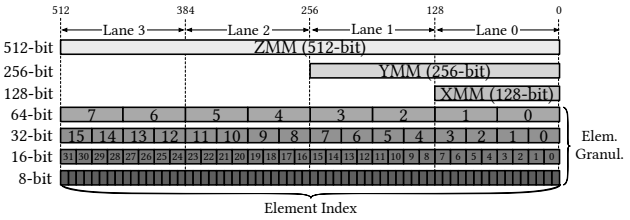

Figure 1: AVX512 registers

ISA extension. The *u*MMU design tackles the challenge of endowing addressability to these registers, a pivotal design component.

AVX512 register structure. Figure 1 shows the structure of AVX512's SIMD registers. The AVX512 registers allow referencing of varying subregister data contents for flexibility and backward compatibility purposes. A `zmm` register content can be referenced through named *sub*registers `ymm` and `xmm`, and also *elements* that refer to indexed data chunks within the named register. Another notable convention used in AVX512 is the *lanes* that divide the 512-bit register width into four segments.

2.2 Memory Confidentiality Policies

We generalize the *Memory Confidentiality Policies (MCPs)* that appeared in the previous works. These MCPs are currently supported in *u*MMU's swap management.

Memory encryption. The memory encryption MCP (MCP_{Enc}) generalizes the encryption-based data confidentiality from previous works [32, 51, 52]. MCP_{Enc} specifically appropriates DynPTA and Palit et al.'s load/store instruction instrumentation model. That is, all program load/store instructions that target designated sensitive data are instrumented to call the interface function to perform encryption or decryption for store and load instructions. The worst-case overhead of such a memory interface can be up to 45.75% in a crypto library (MbedTLS) as the data protection coverage increases [51]. Considering DynPTA's polished dynamic tracking that minimizes unnecessary encryption due to the inaccurate points-to analysis and optimized use of AES-NI, the performance overhead is still substantial. The threat model here is the adversary with arbitrary memory read capability and also side-channel attacks such as Meltdown [38] and Spectre [31] as with the original work [51].

Oblivious RAM. MCP_{ORAM} generalizes the application of the *Oblivious RAM (ORAM)* interface on the program's load and store instructions (e.g., via instrumentation). This type of MCP has been used as a building block in oblivious and obfuscated executions inside TEEs [5, 53]. ORAM algorithms [19, 54, 60, 63, 68, 76] provide provable obliviousness of memory access pattern on the data that they protect, although a very high performance overhead is to be endured. With MCP_{ORAM}, we assume the microarchitectural side-channel threat model against *Trusted Execution Environments (TEEs)* [5, 53, 56]. More specifically, the execution and memory contents themselves are guarded within the trust boundary such as TEE. Nevertheless, the adversary seeks to leak sensitive information by observing the data access patterns.

Memory write salting. Cipherfix [69] (MCP_{Salt}) introduces a type of plaintext salting algorithm to protect sensitive data from the known weakness of AMD's hardware confidential virtual machine support, SEV-SNP [28]. SEV-SNP encrypts the virtual machine's memory with AES in 16-byte chunks using their system physical address as a cryptographic *tweak*. This encryption scheme allows the adversary to perform plaintext-ciphertext correlation attacks since the same plaintext occurrences located at the same physical address would yield identical ciphertexts [36]. Cipherfix, therefore, introduces binary translation that selectively encodes sensitive data with XOR on store and decode on load.

3 Overview

In this section, we introduce *u*MMU's components and their operations at a high level with Figure 2. We follow along the data path from the *u*MMU-enabled program's initial memory access request to the final data retrieval. Along the way, we introduce the key components and our terminologies. Then, we discuss the design and implementation of the two key components, MCP-enforcing paging and *Register-as-Memory (RasM)* controller for *u*Storage in §4 and §5 respectively.

***u*Virtual address space.** *u*MMU creates a new virtual address space called *u*Virtual address space. The *u*Virtual address space, in essence, maps into the processor's *u*Storage but allows *u*MMU to bring *page management* behind the scene to support data larger than the size of *u*Storage. The address space also interfaces the programmer who wants to adapt *u*MMU manually in a program, and also the *u*MMU compiler instrumentation.

Programming interface. The *uVirtual Address (uVA)* is managed similarly to the process heap, and programmer management of *u*VA is facilitated with `umalloc`, `ummap`, and `ufree`. Underneath, *u*MMU implements a simple memory allocator that manages 64-byte *u*VA memory chunks. The allocator tracks the *top* of this *u*VA heap that corresponds to the `brk` in conventional heap management. The management of memory is inspired by the tinyalloc implementation [62]. The function returns a `uVaddr_t`-typed *u*VA.

Program instrumentation. *u*MMU allows programmers to mark the allocation sites of sensitive objects with a function `umalloc`. The instrumentation uses such allocations as the source for sensitivity propagation and subsequent load/store replacements. *u*MMU compiler employs SVF [61]'s context-insensitive and interprocedural Andersen pointer analysis for sensitivity propagation and sensitive load/store instruction identification. The method is similar to many previous works in software security that identify and replace load/store instructions that target designated allocation sites [8, 27, 30, 42, 49, 52]. With `umalloc`, objects are allocated on the *u*VA space, and accordingly, the sensitive load/store are replaced with *u*-{load/store}.

***u*MMU paging.** *u*MMU's virtual instructions invoke *u*MMU's page management (explained in §4) with a *u*VA as shown in ❶ of Figure 2. In turn, the subsequent operations of *u*MMU would walk the *uMMU Page Tables (uPGT)* to resolve the *uVirtual Page Number (uVPN)* portion of the *u*VA (❷) to obtain a *uPhysical Address (uPA)*.

RasM controller. If the walk successfully obtains a *uPhysical Page Number (uPPN)* (❸), we consult the RasM controller to proceed to perform load or store with *u*Storage. The RasM controller is a pivotal *u*MMU component that transforms the extended registers into addressable physical memory. The register selector

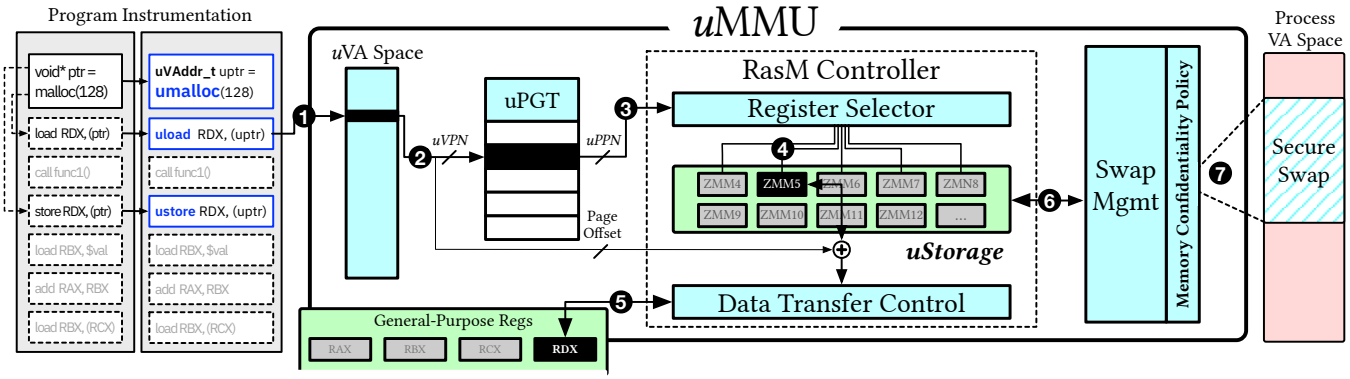

Figure 2: *u*MMU Overview

logic in the controller is responsible for translating a *u*PPN to one of the registers, e.g., `zmm5`, in *u*Storage (❹). As soon as the *u*PPN is resolved to a single register, this information is forwarded to the data transfer control along with *page offset* collected from the initially requested *u*VA. The data transfer control can fetch the data location as requested by the `uload` instruction into the designated *General Purpose Register (GPR)* (❺). Conversely, it would move the data from a GPR into the translated location in the *u*Storage (e.g., subregister-width data into a `zmm` register).

Swap management and MCPs. Flexible and efficient use of the limited *u*Storage is a key notion of *u*MMU. Hence, it must be able to *swap out* to process memory as needed (❻). During data swap-out, the swap management executes the register MCP before storing them in memory. For instance, in the case of MCP_{Enc}, the data would be encrypted before it is exposed to the untrusted memory (❼).

4 Paging and Memory Confidentiality Policies

*u*MMU's paging implements the *u*VA-to-*u*PA translation while coherently bridging the RasM and MCPs. We explain *u*MMU's software MMU scheme and also the supported MCPs in this section.

4.1 *u*Virtual Address address space

The `uload`/`ustore` instructions are substituted with a call to the *u*MMU's entry point function. A software page table walk is performed to obtain the *u*PA used by RasM for subsequent data fetch or store.

***u*MMU Page Tables.** Figure 3 illustrates the *u*VA-to-*u*PA translation. *u*PGT is implemented as a single-level page table. That is, the translation is performed as the following: *u*PGT[*u*VPN] = *u*PPN. Currently, the size of *u*VA is programmer-defined, and so is the size of the *u*VA space. Each *u*PGT entry is an 8-bit value (`char`) that encodes (1) a 5-bit *u*PPN, (2) a dirty bit, (3) an accessed bit, and (4) a pinned bit. The *u*PPN value encoding corresponds to the 28 `zmm` registers (`zmm4`–`zmm31`) that RasM manages. In this sense, each `zmm` register is a *page* in *u*MMU whose size is 64 bytes (512-bit). We make use of the invalid *u*PPN values for management purposes, for instance, a value of `0` and `1` encodes `NOT_PRESENT` and `UNMAPPED`, and `2` signifies `SWAPPED_OUT`, for the queried *u*VPN. *u*PGT and the resulting *u*VA space can be configured with `ummu_init(int uva_bitness)` during the initialization of the application. If the programmer configures *u*MMU with uva with 14, for instance, the *u*VA space spans 2^{14} (16,384) bytes. Accordingly, it requires 256 8-bit *u*PGT entries that each map a 64-*byte* page.

Oblivious page table lookup. *u*MMU always walks the *u*PGT linearly, from beginning to the end, in consideration for attack models such as that of ORAM algorithms. With *u*PGT's entry size of 8-bit, 64 consecutive entries form a single cache line in x86-64. Therefore, the adversary can narrow down the currently access *u*Storage page to one of the 64 entries. Considering *u*PGT's small and fixed size, a linear scan is the most feasible and efficient solution. To be even more efficient, *u*MMU uses SIMD instructions to access *u*PGT in 512-byte strides (512 entries). The linear lookup shows a negligible overhead and therefore is used as a default method for all MCPs regardless of their threat models.

Handling page faults. Handling page faults is straightforward and largely similar to conventional paging implementations. During the address translation, invalid *uPPN* triggers a page fault. When the PPN is `SWAPPED_OUT`, *u*MMU retrieve the page from MCP-enforced swap into *u*Storage, and map the physical page into the virtual address. However, if the PPN is `NOT_PRESENT`, the corresponding virtual address is mapped but not allocated, requiring *u*MMU to allocate a physical page and update the page table accordingly.

4.2 Swap management and Memory Confidentiality Policies

*u*MMU's swap management enables flexible use of *u*Storage through interfacing the swap space and the MCPs. It implements a form of Clock Page Replacement algorithm, which we deem the most suitable for its simplicity and relatively low computation cost regarding

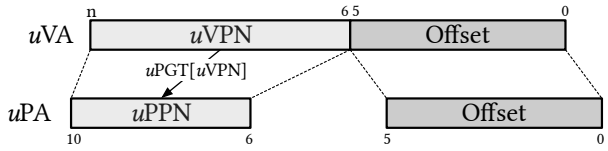

Figure 3: *u*MMU's *u*VA to *u*PA translation

page management. *u*MMU's swap management allows registration of MCPs that governs the protection of swap space.

Optimizations. *u*MMU implements several optimizations that are available in common OS page tables. The existence of the dirty bit allows the elimination of unnecessary data copies, which is the most costly operation in *u*PGT. If the dirty bit is clear during a page swap-out, *u*PGT skips syncing of the *u*Storage data with its counterpart in the swap space. *u*MMU also supports pinning certain *u*Storage pages through the pinned bit. One case where the feature proves to be advantageous is MCP_{Enc}. The AES keys can be pinned in the *u*Storage to prevent their eviction.

Memory Confidentiality Policies. *u*MMU currently supports MCPs as explained in §2, and also summarized again in Table 1.

MCP_{Enc}. For MCP_{Enc}, we directly adapt the implementation from DynPTA [51] that utilizes AES-NI [26] for performant encryption and decryption. The incorporation of AES-NI brings a slight reduction in *u*MMU's *u*Storage capacity. This is because the implementation keeps AES's 10 sets of 128-bit *round keys* in the AVX512 registers. Therefore, *u*Storage's capacity is reduced by a total of 160 bytes when using MCP_{Enc} in the swap management. In addition, we made changes to the implementation regarding the SSE2 compatibility. The original implementation stores the round keys in `xmm0`-`xmm15`, and the usage conflicts with the SSE2 functionality used in many programs. *u*MMU is designed to be compatible with SSE2-enabled programs, thus our MCP_{Enc} uses registers, `zmm4`, `zmm5`, and a portion `zmm6`, to coexist with SSE2.

MCP_{ORAM}. We implemented Path ORAM [60] into MCP_{ORAM} using the implementations of the previous works [5, 53, 56] as references. We configured the Path ORAM according to a known good configuration [55]; MCP_{ORAM} uses 4 blocks for each bucket, and a stash size of 200 blocks. The block size is set to 64 bytes, i.e., cache line size in x86-64, in consideration of cache side-channel attacks. Our Path ORAM implementation follows the practice of using the SIMD intrinsic operations for performing a large bulk memory fetch on the stash and the position map [53]. This way, access patterns on the ORAM components are not visible on the memory hierarchy.

MCP_{Salt}. We implement our own version of CipherFix [69]'s XOR-based data encryption (salting) and decryption for load and stores as a compiler pass. The original work targets binary programs and, therefore, implements its functionality through binary translation. We used the so-called CipherFix-BASE (`rdrand`) [69] model from the work, among the three PRNG sources discussed: `rdrand`, custom AES PRNG, XorShift128+. We selected `rdrand` due to its robustness and security.

5 Register as Memory (RasM) controller

The RasM controller's main objective is to retrofit the x86 AVX512 registers into addressable storage that can serve the role of a *physical memory* in conventional systems for *u*MMU. The unique design requirements entail unique challenges for the implementation of the RasM controller. The first challenge is the *addressability* of registers. *u*MMU repurposes the AVX512 registers as a physical memory whose locations can be referenced by a *u*PA. The second challenge is the *obliviousness* requirement; the controller must also respect the security requirements of the MCPs. We found that the register selection process, if implemented naively, inherently yields input-dependent data access patterns and *Program Counter (PC)* patterns. Lastly, the third requirement is, needless to say, the *performance*. The component is a rather peculiar, purpose-specific code written in assembly that dictates *u*MMU's *u*Storage throughput.

5.1 Background: AVX512 operations

We first discuss the operations of the AVX512 instructions that we employ in the RasM implementations in a concise manner.

Mask registers. AVX512 provides masked operation with eight mask registers (`k0`–`k7`) to be used with AVX512 instructions. The below example illustrates mask registers in action:

```
1 mov r1, 0b00000010
2 kmovb k1, r1
3 ;; Copies bits [127:64] of zmm2 into zmm1
4 vmovdqu64 zmm1{k1}, zmm2
```

The above code first applies the mask on the source register and then copies the resulting value to the destination register. In this example, only the second bit of the mask register is set. Therefore the bits [127:64] of `zmm2` are copied to `zmm1`, and other bits are not changed. The mask registers are powerful feature that enables flexible element manipulation in `zmm` registers, such as selection, which will be described below.

Element manipulation. Figure 4 illustrates the three key instructions that are used in the RasM controller and, therefore, helpful in understanding our implementation.

`vpbroadcastq` takes a single 64-bit GPR and replicates it across all elements in the destination `zmm` register. Also, the instruction can be applied with mask registers (`k1`) to transfer data in GPR into the desired location with crafted mask register values. Most of the instructions that operate on `zmm` registers as operands do not support direct data movement between GPR. For this reason, RasM utilizes the broadcast instruction for moving data from GPR into `zmm` registers efficiently with a single instruction.

`vcmopressq` selectively stores elements that are enabled by a mask. Unlike the other two instructions, `vcompressq` treats mask register differently, the mask registers are used to *select* elements int source register. The instruction selects elements from source `zmm` based on the mask register, which would be continuously stored into destination `zmm`.

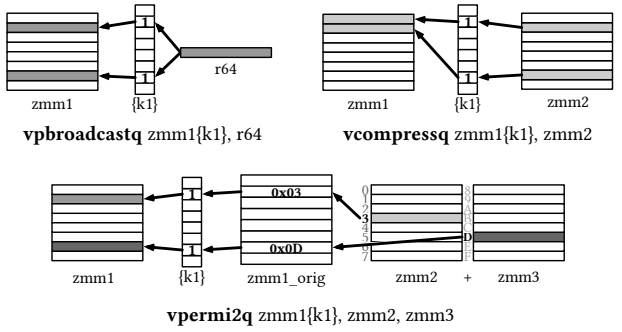

Figure 4: Key AVX512 instructions in RasM

MCP Model	Operation on Sensitive Data	Adversary Capability	Objective
MCP_{Enc} [21, 51, 52, 73]	AES Encryption/Decryption	Arbitrary memory read [2] Spectre [31], Meltdown [38]	In-memory data access prevention
MCP_{ORAM} [5, 53, 56]	ORAM-based *u*Physical page shuffling	Side-channel attacks on TEE [9, 24, 64].	Elimination of sensitive data access pattern
MCP_{Salt} [69]	XOR operation with random bytes	SEV-SNP weak ciphertext [36]	Additional entropy for ciphertexts

Table 1: Memory Confidentiality Policy threat models and operations on data.

`vpermi2q` takes two registers `zmm2` and `zmm3` and performs full permutation on them according to the indices stored in the `zmm1` register. Since the instruction overwrites the result into `zmm1` register, the `zmm1` register acts as both the source and destination register. Since the permutation instruction processes two registers at once, it allows the implementation of efficient register selection, which we describe in §5.3.

`zmm` to GPR. AVX512 provides efficient methods for operating on its registers. However, flexible data movement between the `zmm` registers and GPR are not the intended use cases. As such, the RasM controller, which deviates from conventional uses, must devise unique sequences to mitigate this.

5.2 *u*Physical Address

As shown in Figure 5, *u*MMU RasM controller's *u*PA expresses a single byte location in the *u*Storage with a R_{idx} ([10:6]), a E_{idx} [5:3], and a 3-bit page offset. The 5-bit R_{idx} is the result of *u*PGT translation and directs the memory access to one of the 28 `zmm` registers (`zmm4-31`). This is because *u*MMU uses the 28 `zmm` registers as *u*Storage, and `zmm0-zmm3` as scratch registers during RasM controller operation itself. The 3-bit E_{idx} is directly used by `AVX512` instructions, to select a 64-bit element within the `zmm` register selected by R_{idx}. E_{idx} is shifted, i.e., $E_{mask} = 1 << E_{idx}$, to produce E_{mask} for AVX512 instructions to use it as a value for the mask registers. The E_{off} is a 3-bit value that selects individual bytes within an element.

5.3 Register selection

The register selection process takes R_{idx} as an input to select a single `zmm` register. Then, the data inside the selected register is copied to the common *return* register `zmm0` for all RasM selection implementations. The selection process has quite a large design space and can have multiple plausible candidates. From here on, we introduce a few possible candidates, namely *Indirect jump-based RasM (RasM-indjmp)*, *Conditional move-based RasM (RasM-cmov)*, and *Permutation-based RasM (RasM-perm)*. Then, we draw a comparison among them in terms of performance and security, which will later show that the candidates are suitable for varying circumstances.

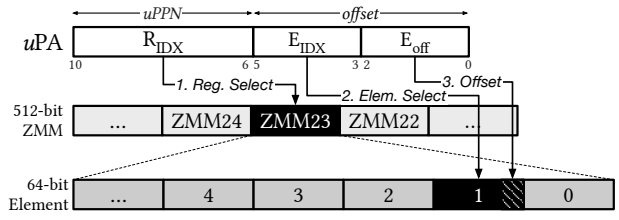

Figure 5: *u*Physical Address translation in RasM.

```
1  regsel_indjmp_load:
2      ; %jmp_addr = .LOAD_ZMM_N + R_idx * 8
3      jmp    %jmp_addr
4
5  regsel_indjmp_store:
6      ; %jmp_addr = .STORE_ZMM_N + R_idx * 8
7      kmovq k1, %E_MASK
8      jmp    %jmp_addr
9
10 .LOAD_ZMM_N+0x00:
11     vmovdqu64 zmm0, zmm4  ;; 6byte
12     ret                   ;; 1byte
13     nop                   ;; 1byte
14 .LOAD_ZMM_N+0x08: ...
15 ...
16 .LOAD_ZMM_N+0xd8: ...
17
18 .STORE_ZMM_N+0x00:
19     vmovdqu64 zmm0{k1}, zmm4 ...
20 .STORE_ZMM_N+0xd8: ...
```

Listing 1: RasM-indjmp

***Impl#1*: RasM-indjmp.** One straightforward and fast implementation is selecting a register through indirect jump instructions as we call it RasM-indjmp. The simplified version is shown in Listing 1. For each `zmm4–zmm31`, there exists a tiny function (`.LOAD_ZMM_N+`*offset*) that is a target for the indirect jump. The `jmp` instruction uses the R_{idx} value as an offset to the target address (Line 2). An alternative approach is to use a direct call/jump (if-else structure), but this results in 28 branch conditions, leading to inefficiency. RasM-indjmp exhibits less overhead associated with multiple branching conditions by enabling a single jump instruction to select the target address based on computed values.

Line 10-13 contain the assembly code for transferring data from `zmm4` registers to `zmm0`. For the range from `zmm4` to `zmm31`, the code includes a total of 28 jump entries. The 6-byte length `vmovdqu64` instruction performs data transfer between `zmm` registers. Following execution, the routine concludes with a return statement. To maintain an 8-byte alignment, a `nop` operation supplements each code block. Such alignment enhances the computation of target jump addresses for the `vmovdqu64` instruction corresponding to a specific R_{idx}.

Code access pattern leakage. However, one quickly observes that RasM-indjmp is inherently susceptible to side-channel attacks.

```
1  regsel_cmov_load:
2     cmp %R_idx, 0
3     cond_vmovdqu64 zmm0, %E_mask, zmm4
4     cmp %R_idx, 1
5     cond_vmovdqu64 zmm0, %E_mask, zmm5
6     ...
7     cond_vmovdqu64 zmm0, %E_mask, zmm30
8     cmp %R_idx, 27
9     cond_vmovdqu64 zmm0, %E_mask, zmm31
```

Listing 2: RasM-cmov. When the R_{idx} is 2, only the highlighted (Line 5) operation is reflected.

For instance, a FLUSH+RELOAD [71] attack or its derivatives [9, 29, 41] may learn which indirect call target was reached during register selection. An adversary monitoring the code cache line (64 bytes) could identify whether the instruction in `.LOAD_ZMM_N+0x00-0xd8` was executed in 64-byte granularity. Since the size of `zmm` registers is 64-byte, and each entry code block size is 8byte, this attack reveals the access patterns of RasM controller with a resolution of $64 \times 8 = 512$ bytes. Another feasible attack is the branch prediction-related attacks [4, 16, 17, 25, 35, 72], where an attacker controlling the CPU's branch prediction can infer the jump target.

***Impl#2*: RasM-cmov.** RasM-cmov shown in Listing 2 is a security-focused implementation for register selection. By leveraging the *conditional move* (`cmov`) instructions, RasM-cmov in fact executes all code pieces that correspond to each `zmm` register, thereby satisfying PC-Security [45] similar to previous works on oblivious execution [8, 34, 45, 53]. Branchless code removes the risk of branch prediction attacks and ensures a constant time code execution, if the execution time for each instruction does not depend on the operand. However, branchless code that executes all code disrupts the intended behavior of the program. To address the issue, conditional move instructions determine whether to execute it or not based on a register, rather than using the branch.

Constructing conditional SIMD `mov` primitive. The `cmov` instruction proves to be a powerful leverage in implementing a side-channel resistant code. Unfortunately, the AVX512 extension does not provide a `cmov` equivalent for the AVX512 vector registers. For this reason, we construct an oblivious conditional `mov` primitive for vector instructions as shown below:

```
1  cond_vmovdqu64 (zmm1, %mask, zmm2):
2     mov  r1, 0b00000000  ; r1 is a GPR
3     cmov r1, 0b11111111
4     kmovb k2, r1
5     kandb k1, %mask, k2 ; k1 = (CF==1)? %mask, 0
6     vmovdqu64 zmm1{k1}, zmm2
```

The above code implements a *conditional* `vmovdqu64` on the `zmm` register by achieving functional equivalence to `cmov`. The key feature used to achieve the objective is AVX512's mask register (§2.1). The conditional move performed on the GPR `r1` is moved into the mask register `k2`. Next, the mask AND instruction (`kandb` at Line 5) conditionally unsets all mask bits of `k1`, which determines the result of the following vector move instruction (Line 6). By chaining

```
1  rasm_perm_load:
2     ; X-R_idx = 0..13
3     ; X-E_idx = 0..15
4     X-R_idx = (R_idx - 4) / 2
5     X-E_idx = (R_idx%2 == 0) ? E_idx, E_idx + 8
6
7     ; Setup
8     kmovw k1, 1
9     kmovw k2, 1
10    vpbroadcastq zmm1, X-E_idx
11    vpbroadcastq zmm2, X-E_idx
12    ; Round 0
13    vpermi2q zmm1{k1}, zmm4, zmm5
14    kshiftlw k1,1
15    ; Round 1
16    vpermi2q zmm1{k1}, zmm6, zmm7
17    kshiftlw k1,1
18    ...
19    ; Round 8
20    vpermi2q zmm2{k2}, zmm18, zmm19
21    kshiftlw k2,1
22    ...
23    ; Round 13
24    vpermi2q zmm2{k2}, zmm30, zmm31
25
26    ; Final Round
27    vpbroadcastq zmm0, X-R_idx
28    vpermi2q zmm0, zmm1, zmm2
```

Listing 3: RasM-perm

the effect of the `cmov` instruction into the mask registers, the code above achieves conditional data move into the `zmm` registers.

RasM-cmov implementation. Utilizing this primitive as a building block, RasM-cmov constructs a secure register selection, whose implementation is shown in Listing 2. The RasM-cmov consists of 28 sub-operations which compare the R_{idx} and conditionally move the data in *u*Storage into `zmm0`. While RasM-cmov provides security, the performance overhead is to be endured as a trade-off. Since the absolute amount of instructions executed in the `cmov`-based operation is inevitably increased compared to simpler implementations such as RasM-indjmp. As our analysis towards the end of this section will show, the executed but discarded instructions and the cost from `cmov` emulation for AVX512 introduce a moderate, but perceptible overhead.

***Impl#3:* RasM-perm.** RasM-perm is the most optimized implementation that provides both obliviousness and enhanced performance. RasM-perm achieves efficiency by simultaneously accessing two paired registers by a single instruction to reduce the executed instructions in a CPU pipeline-friendly manner. To process two registers at once, we utilize the aforementioned `vpermi2q` (§5.1) instruction. The implementation streams through 14 paired registers with 14 `vpermi2q` and stores the result into `zmm1` and `zmm2`, when RasM-cmov executes `vmovdqu64` 28 times. However, due to its inherent nature, it only supports *u*Storage *load*, but not *store*. This is because the `vpermi2q` only supports storing its result into a single register, and there is no alternative instruction that allows storing the result into two destination registers with a single instruction.

RasM-perm has distinct representation of registers compare to previous RasM implementations. Since `vpermi2q` operates on concatenated two registers, RasM-perm has *extended* representation of `zmm` registers(`zmm2N`, `zmm2N+1`). There are 14 paired registers

```
RasM-load %GPR ← %uPA
 ; zmm1←zmm[%R_idx] §5.3
 regsel_load zmm1, %R_idx

 vcompressq zmm0{E_mask}, zmm1
 vmovq %GPR, xmm0
```

```
RasM-store %uPA ← %GPR
 vpbroadcastq zmm0{%E_mask}, %GPR

 ; zmm[%R_idx]←zmm0 §5.3
 regsel_stor %R_idx, %E_mask, zmm0
```

Listing 4: Overview of RasM's load and store operations.

<zmm4,zmm5>, ... , <zmm30,zmm31>), and the number of elements to operate on is doubled from 8 to 16. Listing 3 (Line 2–5) shows the recalculation of the element index and register index according to the paring. For instance, if the X-R_{idx} is 0 and X-E_{idx} is 8, this specifies the ninth element in <zmm4,zmm5>, which is identical to first element in zmm5.

The overall operation of RasM-perm comprises 14 rounds, each loading elements at X-E_{idx} into zmm1 and zmm2 without overlapping with other rounds. To isolate the results of individual rounds, only a single bit of the mask registers is enabled per round operation. The setup phase (Line 7–11) sets the mask registers to 1 and fills up the two scratch registers zmm1,zmm2 with X-E_{idx}. Note that the registers act as both input and output registers; vpermi2q selects an element at X-E_{idx} and overwrites the result back into the registers. Afterward, each round selects elements from the paired register, loads the data in scratch registers, and shifts the mask register to prevent future rounds from overwriting the result. Finally, the resulting zmm1 and zmm2 contains all elements at X-E_{idx}, the permutation instruction selects the register at X-R_{idx} and loads into the return register zmm0.

5.4 Data transfer control

Assuming that we have successfully duplicated the contents of the selected register into zmm0, now we must proceed to element selection and final data load/store. A challenge similar to that of register selection arises again with element selection; dynamically selecting element n also requires a new primitive development. Conventional uses of 512-bit zmm registers often interface directly with memory, while AVX512 instructions only support xmm–GPR data movements. To transfer the element data at E_{idx} into GPR, it should first move it into xmm (=zmm[127:0]), which is the only register that is capable of moving into GPR. The overall load and store of RasM, including element selection, is shown in Listing 4. The load operation used vcompressq instruction to move data in zmm to xmm, by setting the E_{mask}. Finally, vmovq is invoked to transfer data from uStorage into GPR, allowing the CPU to access the required data. To move GPR data into zmm, we utilized the vpbroadcast instruction, with E_{mask}, and further register selection phase would write the value in zmm0 into uStorage.

Handling sub-64bit / ustore. For simplicity, the explanation so far has only shown accessing quad-word (64-bit) data. However, since the GPR register size can be a byte, word, or double data type, uMMU should also support such primitive data types used by the program. For the load operation, supporting only quad-word data types is sufficient; We can simply perform a shift operation to get the sub-64bit of the loaded 64-bit data. For the store operation, RasM should explicitly support smaller than 64-bit data. Since the AVX512 instruction vmovdqu64, in RasM-indjmp and RasM-cmov support smaller data types (e.g., vmovdqu32), we can readily implement more fine-granular stores by replacing the data type of those instructions.

Impl	St.	Ob.	Load/Store (Cycles)		Norm. Exec Time		
			Rand.	Seq.	Qsort	Rsort	Bsearch
indjmp	✓	✗	54.3/57.5	17.1/22.3	3.58×	3.00×	1.74×
cmov	✓	✓	60.8/64.3	62.8/61.0	8.83×	7.89×	4.82×
perm	✗	✓	20.2/–	21.7/–	5.9^{c}× 4.7^{i}×	4.9^{c}× 3.7^{i}×	3.12×

[c] RasM-cmov for store operation. [i] RasM-indjmp for store operation.

Table 2: Comparsion of RasM implementations.

5.5 Choosing Optimal RasM for MCPs

Table 2 compares the three RasM implementations in terms of store capability (**St.**), Obliviousness (**Ob.**), and a set of microbenchmarks with quick sort, radix sort, and binary search. The measurements were taken using a high-precision timer (rdtsc) during one million iterations. As we explained, RasM-perm is the only implementation that is not capable of supporting the store operation. Also, RasM-cmov and RasM-perm are the oblivious computation-aware among the implementations. For both load/store latency and sorting, we used an integer array (int) with 448 random elements, which is the maximum capacity of uStorage.

Load/store latency. We measured the cycle counts of the load-/store operations by averaging the total of one million isolated RasM-based uStorage accesses. We placed measurement probes on the exact interval of the accesses using a high-precision timer (rdtsc). As expected, RasM-indjmp shows the highest performance for both random and sequential accesses. Notably, it outperforms RasM-cmov by a large margin in sequential accesses, which is likely due to the assist from the branch predictor. RasM-perm, although it can only support load operations, outperforms others in random accesses and is on par with RasM-indjmp in sequential.

Performance in sorting algorithms. We also performed an execution time benchmark to estimate a more general performance of the implementations. Since RasM-perm does not support store, we complement it with RasM-indjmp and RasM-cmov and report the measurements from both cases. We again performed the experiments 1,000,000 times for each and measured the times with rdtsc. Then, we normalized the measured times to that of a native execution without uMMU as shown in Table 2.

Verdict. Our analysis indicates that the RasM implementation candidates each have their advantages and disadvantages. In the case of MCP_{Enc} and MCP_{Salt}'s threat model, obliviousness is not for consideration, and we can expect a substantial amount of sequential memory accesses (e.g., cryptographic key access). Therefore, RasM-indjmp would prove to be advantageous over other candidates. On the other hand, the use cases of MCP_{ORAM} accompany obliviousness requirements. As such, RasM-indjmp cannot be considered, and we are left with RasM-cmov and RasM-perm. The optimal solution is RasM-perm for load operations only and RasM-cmov for store operations. While these implementations provide memory

access pattern obliviousness, a slight timing difference (around 40-50 cycles) in `uload` and `ustore` is to be expected. This means that the total execution time of functions or programs can exhibit subtle differences in execution times depending on the ratio of loads vs. stores. Nevertheless, we argue that constant-time execution is rather idealistic for general applications that are clients to *u*MMU. We further discuss the security *u*MMU integration with MCPs in §7.

5.6 SSE/SSE2 Compatibility

*u*MMU's RasM design explained thus far is designed with considerations for optional SSE/SSE2 compatibility. Many amd64 (x86-64) compilers enable SSE2 by default, and SSE2-enabled code would use `xmm0-xmm15`. The System V AMD64 ABI [3] defines `xmm0` to `xmm15` as volatile, meaning that they are not preserved across function calls. Since RasM also uses `xmm4–xmm15` as a part of *u*Storage, they must be preserved before returning from *u*MMU. *u*MMU's solution is to spill the overlapping `xmm4–xmm15` into upper [128:511] bits of `zmm0–zmm3`, the elements that are not visible in the SSE2 convention. This way, *u*MMU stays compatible with the widely used SSE2 feature.

6 Evaluation

In this section, we conduct a comprehensive evaluation of *u*MMU's efficacy and efficiency through the three currently supported MCPs.

Comparison. We evaluate *u*MMU's ability to accelerate MCPs. For each tested program, we prepare two versions: one compiled with *u*MMU using one of the three MCPs as its backend, and the other compiled with the standalone version of MCP. For instance, when the tested MCP is encryption, the two versions would be called $u\text{MMU}_{\text{Enc}}$ and Enc_{SA}. The compiler pass implementations for both were based on LLVM 14.0.0 [1] and SVF 2.7 [61]. For the standalone versions of the program, their MCP enforcement is directly applied to its sensitive load and store instructions. For *u*MMU, the same set of load and store instructions are substituted with `uload` and `ustore`.

Experiments. We conduct three types of experiments: microbenchmarks, algorithm benchmarks, and real-world program benchmarks. In the microbenchmarks (§6.1), we investigate the cost of a page hit vs. page miss for the MCPs as well as the cost of supporting SSE. With algorithm benchmarks (§6.2), we examine how *u*MMU's performance reacts to each algorithm by discussing page fault rates, locality of data accesses, data size, and the overhead from MCPs. The real-world program benchmarks on MbedTLS (§6.3) and Memcached (§6.4) demonstrate how *u*MMU can be applied to popular programs and also the practicality of *u*MMU in terms of performance.

Experiment settings. Throughout the experiments, (1) we configured *u*PGT to support just enough *u*VA space size for the data it protects. For instance, if the data size is 1792, 32 entries exist in *u*PGT to support a 2048-byte address space (2^{11}). (2) SSE2 compatibility mode in RasM was disabled for algorithm benchmark. This is because only the MCP_{Enc} among MCPs require it and the mode can become an interference in the context of the experiment. However, we do enable it for the real-world program benchmarks to measure realistic performance numbers. (3) As discussed in §5.5, we employ RasM-indjmp for MCP_{Enc} and MCP_{Salt}, and a combination of RasM-cmov and RasM-perm for MCP_{ORAM}. All experiments were conducted on a machine with Intel Xeon Silver 4216 (32 cores @ 3.2GHz) and 256GB of RAM running Ubuntu 22.04.3 LTS with kernel version 5.15.0.

Page Hit / Miss	$u\text{MMU}_{\text{ORAM}}$		$u\text{MMU}_{\text{Enc}}$		$u\text{MMU}_{\text{Salt}}$	
	Load	Store	Load	Store	Load	Store
Hit	24.2	68.3	54.1	60.0	24.4	65.8
Miss/Swap In	24520.2	24555.0	693.3	703.3	510.8	551.2
Miss/Swap Out	24217.2	24264.7	698.3	698.5	3384.3	3423.7

Table 3: Microbenchmark result showing cycle count for Page hit vs. Page Miss path for each MCP

6.1 Microbenchmarks

Page hits vs. misses. Table 3 shows the measurements of page hit vs. misses for each MCP. All experiments were conducted on random 64-byte load and store tests on 4096 KB *u*Storage, the page fault and hit overheads are measured using `rdtsc`. Understandably, MCP_{ORAM} was the most costly MCP of all. MCP_{Enc} showed moderate overhead thanks to AES-NI. MCP_{Salt} shows high overhead on store due to the involvement of a random number generation process but shows very low load overhead. These measurements allow a better understanding of the algorithm and real-world experiment results presented in the remainder of this section. Besides, the overhead of RasM was already presented (Table 2), and we found that the page table walk had negligible overhead (~5 cycles).

SSE compatibility overhead. Table 4 shows the performance loss in RasM implementations due to SSE compatibility support from the experiment conducted in the same setting as in §5.5. Overall, the implementations experience a moderate overhead, except RasM-perm. We suspect that the reason is due to the CPU pipelining disturbance, since the *u*Storage spilling creates register dependency among the AVX512 instructions. Our real-world program benchmarks are conducted with SSE2 enabled, and the results serve as an estimation of the performance loss.

6.2 Algorithm benchmark

As shown in Figure 6, we benchmarked *u*MMU's ability to accelerate MCPs. We use three target programs, QuickSort (`qsort`), RadixSort (`rsort`), and Dijkstra (`dijk`), which are originally from GhostRider [39]. The same programs were included in the evaluation of three previous works on oblivious computation [8, 39, 53], as they are well-suited for highlighting memory-bound performance characteristics. For both $u\text{MMU}_{MCP}$ and MCP_{SA}, the array in the sorting algorithms is protected as sensitive data. Additionally, the block size of ORAM (Path ORAM) was 64 bytes for both

RasM Impl.	Disabled	SSE1	SSE2
`cmov`	60.53	65.51	71.83
`indjump`	53.58	57.77	61.65
`perm`	19.66	33.97	53.09

Table 4: RasM performance loss (cycles) due to SSE compatibility

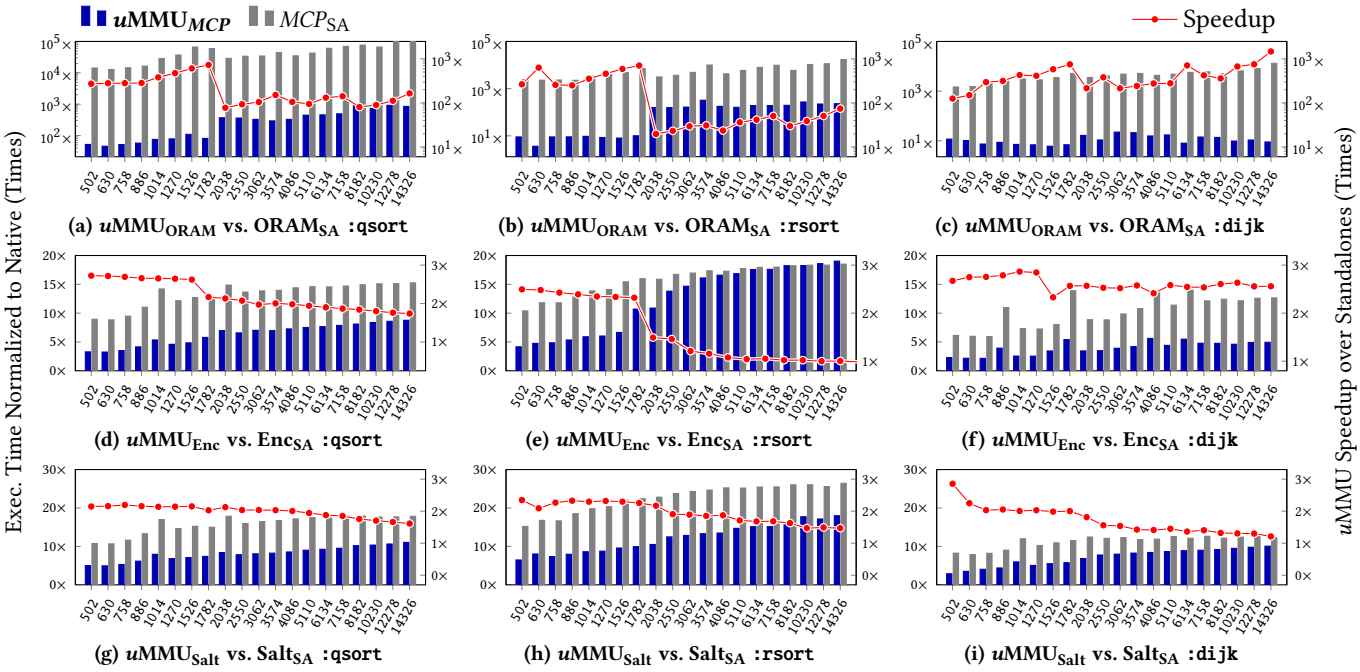

Figure 6: *u*MMU$_{MCP}$ vs. Standalone MCPs in sorting algorithms. Bar graphs (Y1-axis) show execution time normalized to native. Line graph (Y2-axis) shows *u*MMU$_{MCP}$'s execution time normalized to standalone MCPs (Speedup). X-axes show data size in bytes.

*u*MMU$_{\text{ORAM}}$ and ORAM$_{\text{SA}}$ to reflect the 64-byte cache line granularity memory access pattern leakage. The total node count in the ORAM tree is adjusted to support the varied data sizes.

Page fault rates. Figure 7 illustrates the page fault rates observed in each program during the algorithm benchmark. The page fault rates are universal to all MCPs since they only depend on the program's memory access patterns. We observe that the performance of *u*MMU is directly proportional to the page fault rates.

Data size and locality. The varied data sizes in our experiments are designed to test *u*MMU's performance well over its *u*Storage capacity. Also, they well exceed the sensitive data size in the real-world examples that we will explain (§6.3 and §6.4). In general, *u*MMU's performance benefit decreases towards larger data sizes. Nevertheless, *u*MMU improves the performance of the standalone MCPs even in the most extreme case of data size = 14326 in all cases except Figure 6-(e).

We observe that the locality of the computation significantly impacts overall performance. The retained performance advantage of *u*MMU in `qsort` across all three MCPs illustrates the role of high

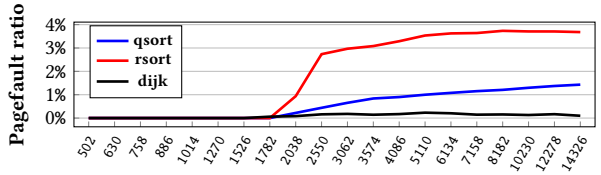

Figure 7: Page fault rates measured during algorithm benchmark in Figure 6. X-axis shows data size in bytes.

locality of data accesses. The inherent high locality of `qsort`'s data accesses allows *u*MMU to keep accelerating the MCP when the protected data size is well over the *u*Storage capacity.

MCP overhead. Along with the locality of the program's protected data accesses, the data size and the cost of MCP enforcement on the data together determine the performance characteristics of *u*MMU. The benefit of *u*MMU's register-only data movement is better manifested when the MCP overhead on in-memory data is higher. For instance, in the case of MCP$_{\text{ORAM}}$ (Figure 6-(a), (b), and (c)), *u*MMU accelerates MCP$_{\text{ORAM}}$ by multiple magnitudes in many intervals. During `rsort`, a near 1,000× speedup was observed in the 502–1782 interval. Also in `dijk`, the acceleration effect ranged from 100× to 1,000×.

6.3 Real-world program benchmark: MbedTLS

We used *u*MMU instrumentation tool (e.g., `umalloc`) to protect the sensitive assets in each MbedTLS [37] 3.5.2's cryptographic algorithms as shown in Figure 8. Similar to the algorithm benchmark, we compile two versions, *u*MMU$_{\text{Enc}}$ vs. Enc$_{\text{SA}}$, for each MbedTLS algorithm that compiles to an executable.

Annotation and instrumentation. We annotated the allocation sites of *input buffer* and *key objects* using `umalloc`. In turn, the points-to analysis in the instrumentation mark all load and store instructions on the objects. The marked instructions are replaced with either `uload`/`ustore` for *u*MMU$_{\text{Enc}}$, or a direct call to the MCP function for the case of Enc$_{\text{SA}}$. We annotated the allocation sites of the *1024-byte* input buffer for all crypto algorithms, such that the incoming plaintexts can be protected. For symmetric-key

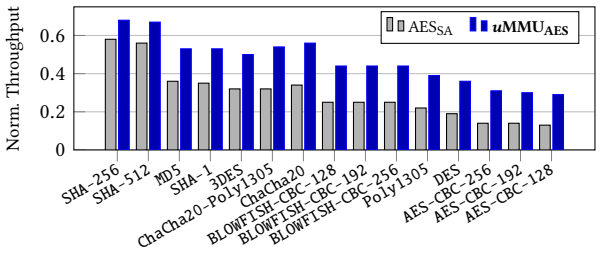

Figure 8: MbedTLS crypto throughput comparison between uMMU$_{\text{Enc}}$ vs. Enc$_{\text{SA}}$. Throughput (Y-axis) is normalized to unmodified MbedTLS.

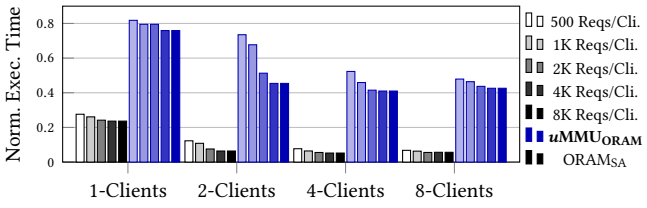

Figure 9: Memcached comparison between uMMU$_{\text{ORAM}}$ vs. ORAM$_{\text{SA}}$. X-axis shows concurrent clients.

crypto algorithms (e.g., `AES-xxx128`) that require key protection, we manually identified the key objects and related intermediate (but sensitive) values and replaced their allocation sites with `umalloc`.

Note that the total size of the protected key and the intermediate values may be different for each algorithm. For hash algorithms such as `SHA-512`, the total *u*VA space usage was 1024 bytes (input buffer). Also, we found that the symmetric-key algorithms retain the key objects, but often allocate (`umalloc`) and deallocate (`ufree`) the intermediate values during its execution. Therefore, the total *u*VA usage fluctuates during execution for symmetric-key algorithms.

Result analysis. Figure 8 shows the throughput for each crypto operation compared to baseline throughput. The speedup was generally higher in the symmetric-key crypto algorithms. *u*MMU on average achieves 1.6× (min: 1.16×, max: 2.15×) speed up over Enc$_{\text{SA}}$. The results reflect how the implications from the algorithm benchmark translate to more complex crypto algorithms. First, the protected data sizes of the crypto algorithms fall well within the range in which *u*MMU can excel, and hence, the observed substantial performance increase. For `AES-CBC-256` as an example, the *u*VA space usage during *runtime* ranged approximately from 1500 bytes to 2500 bytes. This aspect makes a strong case for *u*MMU's *u*Storage *managed* capacity that it is practical for many real-world applications. Second, the operations with the symmetric key tend to have high-locality memory accesses on the keys, as we analyzed through source code analysis. The AES computation, for instance, repeatedly accesses the round-key buffer size of 224 bytes throughout 14 rounds. For this reason, *u*MMU exhibits higher performance gain in the symmetric key crypto algorithms even when the *u*VA space consumption is higher than that of the hashing algorithms.

6.4 Real-world program benchmark: Memcached

We assessed the effectiveness of uMMU$_{\text{ORAM}}$ in shielding sensitive access patterns through comparison with ORAM$_{\text{SA}}$ as illustrated in Figure 9. We conducted the evaluation using Memcached v1.6.14 [14], an in-memory key-value store designed to boost the webserver performance by caching database query results. At its core, it maintains a hashtable, where each key hashed with the Murmur3 algorithm is used as an index.

Attacker model. The adversary is inspired by prior studies such as Membuster [33] that leak queried words in a dictionary program through hashtable monitoring. The adversary is aware that the mail server is integrated with Memcached to support efficient keyword-based querying of emails. In the attack setup, the keywords within each email are hashed and used as indices in the hashtable, where the corresponding values are the identifiers of the emails (e.g., `{"announcements": {email0, email3}, "internal": {email0, email1}}`). Therefore, the accessed offset of the hashtable directly reveals the value of the currently indexed key.

Experiment settings. To hide the hashtable access patterns, we forced the allocation site of Memcached's hashtable (`assoc.c: primary_hashtable`) to use `umalloc`. The hashtable is retained at the default size of 512KB. The same ORAM configuration from §6.2 was used for the experiment. For benchmarking, we used libmemcached project [13]'s command-line benchmark tool called `memslap`. The benchmark tool generates configurable workloads for concurrencies, key/value size, and the proportion of get/set request ratio, which we kept at default settings. We set the benchmark to measure the throughput with concurrent connection counts of 1, 2, 4, and 8. The number of requests for each client is varied with steps in the range of 500 to 8000, , therefore, the maximum number of total required sizes is 64,000. The evaluation was performed locally to prevent the networking overhead from masking the latency.

Result analysis. Figure 9 reports the results from the experiment. The experiment shows that *u*MMU is viable even when the data size and access locality are not in favor if the cost of the MCP is very high. The case of Memcached hash table protection features a large data size (524,288 bytes) and low locality on sensitive data accesses (i.e., randomized key index requests). However, uMMU$_{\text{ORAM}}$'s request processing throughput in the worst-case (8-clients *times* 8000 requests) is still *8*× higher than that of ORAM$_{\text{SA}}$. This throughput amounts to 45% of the native throughput. The results from this experiment confirm that the retained *u*MMU performance over ORAM$_{\text{SA}}$ in larger data from Figure 6 also translates to even larger data sizes in real-world applications.

7 Security Analysis

*u*MMU also must respect the attack model of the MCPs to be practical. Here we discuss the security implications of using *u*MMU with each MCP's security requirements.

Security model in MCP$_{\text{Enc}}$. The threat model of the encryption MCP backend is inherited from DynPTA [51]. In line with the original threat model, arbitrary memory reads and memory disclosure through side-channel attacks are considered as we explained in §2.2. The attack model implicates that the adversary may be able

to read the *u*MMU's *u*PGT. In this attack model, however, the disclosure of the *u*VA space does not satisfy the attacker's goals. Also, the sensitive data is always in either *u*Storage or the encrypted swap in the process address space. Therefore, *u*MMU maintains the security guarantees of DynPTA in its original attack model.

Upholding ORAM guarantees (MCP_{ORAM}). We further discuss the security implications of incorporating $u\text{MMU}_{\text{ORAM}}$ into TEE-guarded sensitive applications. When the data safeguarded with *u*MMU is within the maximum capacity of *u*Storage, the data access pattern is completely unobservable. The aforementioned linear page table walk and oblivious RasM implementations are such examples. Besides the eliminated exposure of sensitive data on process memory, *u*PGT and RasM are designed to yield no discernable access pattern for all *u*VA input as explained. However, if the protected data size may exceed the *u*Storage capacity during runtime, *u*MMU must engage the swap management, and the ORAM storage activation may be observable to the adversary. As a result, the total number of observed ORAM accesses may vary depending on many factors including the size of the input data or the program's control flow variances. Such execution variances are out of the scope of the ORAM algorithms and $u\text{MMU}_{\text{ORAM}}$. Considering that achieving such access timing and frequency obliviousness with general programs is hardly practical without heavy performance sacrifices, we argue that *u*MMU can serve as a primitive for vastly minimizing attacker observability.

8 Related Work

8.1 Leveraging Unobservable storages

A plethora of works have explored protecting sensitive data by relocating them from main memory to unobservable alternative forms of storage.

Register-only computation. To protect sensitive assets involved in cryptographic computation, many works have opted for shielding encryption keys and by-products within the CPU registers to perform so-called register-only computation [18, 20, 21, 23, 46, 58]. In the field of oblivious computing, registers-only computations are used as an alternative to their insecure counterparts that leave discernable traces in memory [6, 44, 47, 59].

Caches and GPU. Alternatively, caches and GPUs have also been discussed as unobservable storage. CaSE [74] employs cache-only data storage within an ARM processor's L2 cache to create an isolated environment. PixelVault [67] secures cryptographic keys in GPU registers and isolates GPU code within its instruction cache to protect the data from the untrusted OS.

8.2 Memory Confidentiality Policies

To complement register-only computation, many works employed data protection mechanisms such as encryption which we referred to as memory confidentiality policy throughout this work. *u*MMU's design objective is to coherently combine the use of extended register and memory confidentiality policies. In turn, *u*MMU provides acceleration to these techniques as we explained.

Encrypting on register-to-memory data movement. Numerous works maintain sensitive data exclusively in registers as prolonged as possible and encrypt the content upon eviction to memory [51, 52, 70, 73]. Notably, DynPTA [51] addressed the imprecision of static points-to analysis due to overapproximation by employing dynamic data flow tracking in runtime. Palit et al. [52] proposed selective data protection with points-to analysis, safeguarding annotated data encrypted in memory. Ginseng [73] is an ARM-based solution that combines sensitive data protection through in GPRs with compiler register allocation, and TrustZone [7]-assisted encryption against untrusted OS.

Interfacing memory accesses with ORAM . MCP_{ORAM} is employed when adversaries can observe, but not read, memory access patterns, such as cache line access or page faults. This is typical in TEE environment; while memory is encrypted, its access patterns are revealed due to its reliance on the host's resource management. [5, 15, 39, 40, 44, 53, 59, 75].

Mitigating ciphertext-plaintext correlation. *Secure Encrypted Virtualization (SEV)* is known for its weakness of deterministic memory encryption based on the physical address. The same plaintext value in the same physical address induces the same cipher text value, allowing adversaries to infer the content of the memory. Cipherfix [69] sought to overcome this problem by masking every write of sensitive data with random salt using binary instrumentation.

8.3 Software-based MMU for security

Software-based MMU approaches for security [48, 49, 75] transform program memory accesses into custom operations to enable secure memory management. Cosmix [49] instruments memory operations to implement secure page fault handlers, such that security policy can be implemented through the emulated self-paging capability inside the enclaves. Klotski [75] also implements a software-defined MMU as a pivotal building block for its obfuscated execution for SGX enclaves. Klotski manages enclave memory as *minipages* (e.g., 2KB emulated pages) backed by ORAM. Eleos [48], proposed the concept of Secure User-managed Virtual Memory (SUVM), application-level paging inside the SGX enclave that reduces the exit cost due to paging. *u*MMU also leverages software-based MMU to virtualize memory accesses on sensitive data to create its *u*VA.

8.4 ORAM and oblivious execution

Numerous microarchitectural side channel attacks proved to be a threat to the confidentiality of in-memory data. Meltdown [38] and Spectre [31] are the quintessential examples. Additionally, various cache side-channles [22, 41, 50, 71] and speculative execution-based side-channels have been reported [4, 16, 43], potentially leaking memory and its access patterns.

Attacks on TEE. TEEs, such as Intel SGX, have been proved to be also susceptible to many such attacks [9, 10, 12, 17, 24, 25, 35, 57, 64, 65]. In response to these security challenges, ORAM has emerged as a key component in constructing oblivious computations [5, 6, 15, 39, 40, 44, 53, 56, 59, 75].

ORAM and Obufscated execution. To address the inherent performance degradation due to the use of ORAM, various ORAM algorithms have been proposed [19, 54, 55, 60, 63, 68, 76] to fit specific needs and accelerate performance. Many works sought to implement obfuscated execution inside TEE to mitigate such side channels. Raccoon [53] linearizes the execution path for sensitive

branches by transforming code to execute both real and decoy paths while obfuscating data access traces with ORAM. Obfuscuro [5] proposed an obfuscated execution engine for Intel SGX enclaves by managing code and data fragments using ORAM and performing all execution and data accesses in fixed locations (i.e., scratchpad). In our work, we showed that *u*MMU can accelerate Path ORAM [60], a key building block in oblivious computing, with $u\text{MMU}_{\text{ORAM}}$.

9 Conclusion

In this paper, we introduced *u*MMU, a novel memory subsystem that integrates the notion of unobservable storage and memory confidentiality policies. By applying the virtual memory principles to unify the unobservable register storage and Memory Confidentiality Policy, *u*MMU provides a powerful primitive for protecting sensitive memory data. We demonstrated the effectiveness and efficacy of *u*MMU by incorporating three MCPs: memory encryption, ORAM, and plaintext salting.

Acknowledgments

We deeply appreciate the anonymous reviewers for their constructive comments and feedback. This work was supported by grants funded by the Korean government: the National Research Foundation of Korea (NRF) grant (NRF-2022R1C1C1010494) and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2022-II220688, RS-2022-II221199, RS-2024-00437306, and RS-2024-00439819).

References

[1] 2012. LLVM. The LLVM Compiler Infrastructure. https://llvm.org/.
[2] 2023. The Heartbleed Bug. https://heartbleed.com.
[3] 2024. x86 psABIs. https://gitlab.com/x86-psABIs/x86-64-ABI
[4] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security* (Singapore) *(ASIACCS '07)*. Association for Computing Machinery, New York, NY, USA, 312–320. https://doi.org/10.1145/1229285.1266999
[5] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. 2019. OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX. In *NDSS*. The Internet Society.
[6] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2018.23284 99 citations (Semantic Scholar/DOI) [2023-02-23].
[7] ARM. 2024. Arm TrustZone Technology. https://developer.arm.com/ip-products/security-ip/trustzone. Last accessed Apr 2 , 2024,.
[8] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. 2021. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *CCS*. ACM, 715–733.
[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser
[10] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck
[11] Scott A. Carr and Mathias Payer. 2017. DataShield: Configurable Data Confidentiality and Integrity. In *AsiaCCS*. ACM, 193–204.
[12] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 142–157. https://doi.org/10.1109/EuroSP.2019.00020
[13] Data Differential. 2011. libmemcached. https://libmemcached.org/libMemcached.html.
[14] Dormando. 2018. memcached - a distributed memory object caching system. https://memcached.org.
[15] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proc. VLDB Endow.* 13, 2 (2019), 169–183.
[16] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783743
[17] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. *SIGPLAN Not.* 53, 2 (mar 2018), 693–707. https://doi.org/10.1145/3296957.3173204
[18] Behrad Garmany and Tilo Müller. 2013. PRIME: private RSA infrastructure for memory-less encryption. In *Proceedings of the 29th Annual Computer Security Applications Conference* (New Orleans, Louisiana, USA) *(ACSAC '13)*. Association for Computing Machinery, New York, NY, USA, 149–158. https://doi.org/10.1145/2523649.2523656
[19] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *J. ACM* 43, 3 (May 1996), 431–473. https://doi.org/10.1145/233551.233553 1650 citations (Semantic Scholar/DOI) [2023-06-16].
[20] Johannes Götzfried and Tilo Müller. 2013. ARMORED: CPU-Bound Encryption for Android-Driven ARM Devices. In *ARES*. IEEE Computer Society, 161–168.
[21] Johannes Götzfried, Tilo Müller, Gabor Drescher, Stefan Nürnberger, and Michael Backes. 2016. RamCrypt: Kernel-based Address Space Encryption for User-mode Processes. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (Xi'an, China) *(ASIA CCS '16)*. Association for Computing Machinery, New York, NY, USA, 919–924. https://doi.org/10.1145/2897845.2897924
[22] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 897–912. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss
[23] Le Guan, Jingqiang Lin, Bo Luo, and Jiwu Jing. 2014. Copker: Computing with Private Keys without RAM. In *21st Annual Network and Distributed System Security Symposium (NDSS '14)*. The Internet Society.
[24] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 299–312. https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel
[25] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. 2019. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020, 1 (Nov. 2019), 321–347. https://doi.org/10.13154/tches.v2020.i1.321-347
[26] Intel Corporation. 2024. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 325462-083US.
[27] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP.2019.00017
[28] David Kaplan. [n. d.]. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. ([n. d.]).
[29] Mehmet Kayaalp, Nael B. Abu-Ghazaleh, Dmitry V. Ponomarev, and Aamer Jaleel. 2016. A high-resolution side-channel attack on last-level cache. In *DAC*. ACM, 72:1–72:6.
[30] Taegyu Kim, Vireshwar Kumar, Junghwan Rhee, Jizhou Chen, Kyungtae Kim, Chung Hwan Kim, Dongyan Xu, and Dave (Jing) Tian. 2021. PASAN: Detecting Peripheral Access Concurrency Bugs within Bare-Metal Embedded Applications. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 249–266. https://www.usenix.org/conference/usenixsecurity21/presentation/kim
[31] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy*. IEEE, 1–19.
[32] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 437–452. https://doi.org/10.1145/3064176.3064217
[33] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-che Tsai, and Raluca Ada Popa. 2020. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 487–504. https://www.usenix.org/conference/usenixsecurity20/presentation/lee-dayeol

[34] Hyun Bin Lee, Tushar M. Jois, Christopher W. Fletcher, and Carl A. Gunter. 2021. DOVE: A Data-Oblivious Virtual Environment. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. https://www.ndss-symposium.org/ndss-paper/dove-a-data-oblivious-virtual-environment/

[35] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 557–574. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho

[36] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 717–732. https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan

[37] Linaro Limited. 2024. Mbed TLS. https://www.trustedfirmware.org/projects/mbed-tls/.

[38] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[39] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine Shi. 2015. 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, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 87–101. https://doi.org/10.1145/2694344.2694385

[40] Chang Liu, Michael Hicks, and Elaine Shi. 2013. Memory Trace Oblivious Program Execution. In *2013 IEEE 26th Computer Security Foundations Symposium*. 51–65. https://doi.org/10.1109/CSF.2013.11

[41] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. 605–622. https://doi.org/10.1109/SP.2015.43

[42] Peiming Liu, Gang Zhao, and Jeff Huang. 2020. Securing unsafe rust programs with XRust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 234–245.

[43] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2109–2122. https://doi.org/10.1145/3243734.3243761

[44] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 279–296.

[45] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *ICISC (Lecture Notes in Computer Science, Vol. 3935)*. Springer, 156–168.

[46] Tilo Müller, Felix C. Freiling, and Andreas Dewald. 2011. TRESOR Runs Encryption Securely Outside RAM. In *20th USENIX Security Symposium (USENIX Security 11)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/usenix-security-11/tresor-runs-encryption-securely-outside-ram

[47] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 619–636. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/ohrimenko

[48] 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* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 238–253. https://doi.org/10.1145/3064176.3064219

[49] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. 2019. CoSMIX: A Compiler-based System for Secure Memory Instrumentation and Execution in Enclaves. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 555–570. https://www.usenix.org/conference/atc19/presentation/orenbach

[50] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology* (San Jose, CA) *(CT-RSA'06)*. Springer-Verlag, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/11605805_1

[51] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. 2021. DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1919–1937. https://doi.org/10.1109/SP40001.2021.00082

[52] Tapti Palit, Fabian Monrose, and Michalis Polychronakis. 2019. Mitigating Data Leakage by Protecting Memory-Resident Sensitive Data. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 598–611. https://doi.org/10.1145/3359789.3359815

[53] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security Symposium*. USENIX Association, 431–446.

[54] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2014. Ring ORAM: Closing the Gap Between Small and Large Client Storage Oblivious RAM. *IACR Cryptol. ePrint Arch.* (2014), 997. http://eprint.iacr.org/2014/997

[55] Ling Ren, Xiangyao Yu, Christopher W Fletcher, Marten Van Dijk, and Srinivas Devadas. 2013. Design space exploration and optimization of path oblivious ram in secure processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 571–582.

[56] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *NDSS*. The Internet Society.

[57] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 753–768. https://doi.org/10.1145/3319535.3354252

[58] Patrick Simmons. 2011. Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In *ACSAC*. ACM, 73–82.

[59] Jeongseok Son, Griffin Prechter, Rishabh Poddar, Raluca Ada Popa, and Koushik Sen. 2021. ObliCheck: Efficient Verification of Oblivious Algorithms with Unobservable State. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael D. Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2219–2236. https://www.usenix.org/conference/usenixsecurity21/presentation/son

[60] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 299–310. https://doi.org/10.1145/2508859.2516660

[61] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266.

[62] thi.ng. 2017. tinyalloc. https://github.com/thi-ng/tinyalloc.

[63] Shruti Tople, Yaoqi Jia, and Prateek Saxena. 2018. PRO-ORAM: Constant Latency Read-Only Oblivious RAM. *IACR Cryptol. ePrint Arch.* (2018), 220. http://eprint.iacr.org/2018/220

[64] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution* (Shanghai, China) *(SysTEX'17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3152701.3152706

[65] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 178–195. https://doi.org/10.1145/3243734.3243822

[66] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *2019 IEEE Symposium on Security and Privacy (SP)*. 88–105. https://doi.org/10.1109/SP.2019.00087

[67] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. PixelVault: Using GPUs for Securing Cryptographic Operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1131–1142. https://doi.org/10.1145/2660267.2660316

[68] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. Association for Computing Machinery, New York, NY, USA, 850–861. https://doi.org/10.1145/2810103.2813634

[69] Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. 2023. Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 6789–6806. https://www.usenix.org/conference/usenixsecurity23/presentation/wichelmann

[70] Jinyan Xu, Haoran Lin, Ziqi Yuan, Wenbo Shen, Yajin Zhou, Rui Chang, Lei Wu, and Kui Ren. 2022. RegVault: hardware assisted selective data randomization for operating system kernels. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 715–720. https://doi.org/10.1145/3489517.3530549

[71] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[72] Jiyong Yu, Trent Jaeger, and Christopher Wardlaw Fletcher. 2023. All Your PC Are Belong to Us: Exploiting Non-control-Transfer Instruction BTB Updates for Dynamic PC Extraction. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 65, 14 pages. https://doi.org/10.1145/3579371.3589100

[73] Min Hong Yun and Lin Zhong. 2019. Ginseng: Keeping Secrets in Registers When You Distrust the Operating System. In *NDSS*. The Internet Society.

[74] Ning Zhang, Kun Sun, Wenjing Lou, and Y. Thomas Hou. 2016. CaSE: Cache-Assisted Secure Execution on ARM Processors. In *2016 IEEE Symposium on Security and Privacy (SP)*. 72–90. https://doi.org/10.1109/SP.2016.13

[75] Pan Zhang, Chengyu Song, Heng Yin, Deqing Zou, Elaine Shi, and Hai Jin. 2020. Klotski: Efficient Obfuscated Execution against Controlled-Channel Attacks. In *ASPLOS*. ACM, 1263–1276.

[76] Xian Zhang, Guangyu Sun, Chao Zhang, Weiqi Zhang, Yun Liang, Tao Wang, Yiran Chen, and Jia Di. 2015. Fork Path: Improving Efficiency of ORAM by Removing Redundant Memory Accesses. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 102–114. https://doi.org/10.1145/2830772.2830787