Object-oriented Unified Encrypted Memory Management for Heterogeneous Memory Architectures

MO SHA[*], Alibaba Cloud, Singapore
YIFAN CAI[*†], University of Pennsylvania, USA
SHENG WANG, Alibaba Cloud, Singapore
LINH THI XUAN PHAN, University of Pennsylvania, USA
FEIFEI LI, Alibaba Cloud, China
KIAN-LEE TAN, National University of Singapore, Singapore

In contemporary database applications, the demand for memory resources is intensively high. To enhance adaptability to varying resource needs and improve cost efficiency, the integration of diverse storage technologies within heterogeneous memory architectures emerges as a promising solution. Despite the potential advantages, there exists a significant gap in research related to the security of data within these complex systems. This paper endeavors to fill this void by exploring the intricacies and challenges of ensuring data security in object-oriented heterogeneous memory systems. We introduce the concept of Unified Encrypted Memory (**UEM**) management, a novel approach that provides unified object references essential for data management platforms, while simultaneously concealing the complexities of physical scheduling from developers. At the heart of **UEM** lies the seamless and efficient integration of data encryption techniques, which are designed to ensure data integrity and guarantee the freshness of data upon access. Our research meticulously examines the security deficiencies present in existing heterogeneous memory system designs. By advancing centralized security enforcement strategies, we aim to achieve efficient object-centric data protection. Through extensive evaluations conducted across a variety of memory configurations and tasks, our findings highlight the effectiveness of **UEM**. The security features of **UEM** introduce low and acceptable overheads, and **UEM** outperforms conventional security measures in terms of speed and space efficiency.

CCS Concepts: • **Security and privacy** → *Distributed systems security*; • **Information systems** → *Main memory engines*; • **Hardware** → *Emerging architectures*; *External storage*.

Additional Key Words and Phrases: Memory Security; Unified Memory Management; Heterogeneous Memory Architecture; Data Confidentiality; Data Integrity

ACM Reference Format:
Mo Sha, Yifan Cai, Sheng Wang, Linh Thi Xuan Phan, Feifei Li, and Kian-Lee Tan. 2024. Object-oriented Unified Encrypted Memory Management for Heterogeneous Memory Architectures. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 155 (June 2024), 29 pages. https://doi.org/10.1145/3654958

[*]Both authors contributed equally to the paper.
[†]Yifan's work was partially done during an internship at Alibaba Cloud.

Authors' Contact Information: Mo Sha, Alibaba Cloud, Singapore, shamo.sm@alibaba-inc.com; Yifan Cai, University of Pennsylvania, USA, caiyifan@seas.upenn.edu; Sheng Wang, Alibaba Cloud, Singapore, sh.wang@alibaba-inc.com; Linh Thi Xuan Phan, University of Pennsylvania, USA, linhphan@cis.upenn.edu; Feifei Li, Alibaba Cloud, China, lifeifei@alibaba-inc.com; Kian-Lee Tan, National University of Singapore, Singapore, tankl@comp.nus.edu.sg.

ACM 2836-6573/2024/6-ART155
https://doi.org/10.1145/3654958

1 INTRODUCTION

As the volume of data continues to grow, it is no longer cost-effective for data management systems to support memory-intensive workloads efficiently. Instead, in recent years, a wide range of *heterogeneous memory architectures*, designed to suit specific scenarios and optimization goals, have been introduced and widely integrated into the development of data management systems [26, 52, 75, 78, 92]. Such systems integrate a variety of storage modalities to form a large-scale memory system to support memory access operations during program execution in a transparent manner. They are manifested in a variety of ways, e.g., remote memory [2, 5, 19, 31, 32], disaggregated memory [30, 35, 51, 58, 87], and non-volatile memory [46], with each specific approach offering its own unique advantages. In general, these systems aim to address the balance between performance and cost that arises from the distinct physical characteristics of various storage tiers. Moreover, heterogeneous memory systems enhance flexibility in resource allocation, optimize utilization, and consequently, mitigate costs.

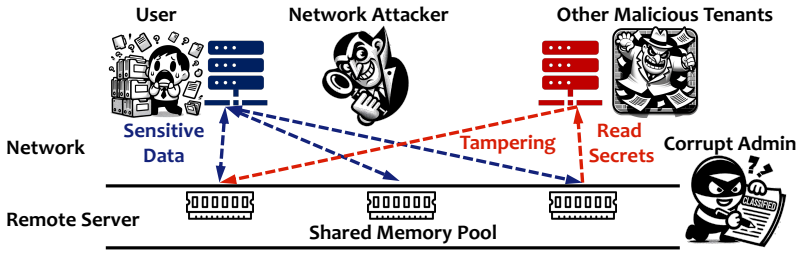

Fig. 1. An example of shared memory pool expanding attack surface in heterogeneous memory architectures.

While there has been extensive research on empowering heterogeneous memory architectures for data processing [50, 53, 107, 110], the focus has largely been on performance, applicability, fault tolerance, and availability. Surprisingly, to our knowledge, there is no reported work that looks at safeguarding the data. In particular, these novel memory architectures introduce additional challenges in safeguarding data. For instance, consider a shared memory pool built on networks [56], as depicted in Fig. 1. Data reliability is inherently lower than in traditional schemes due to the presence of networks and remote servers, which are physically more challenging to control. Attackers or even corrupt administrators of these facilities can pose a greater threat than before, thus expanding the attack surface. Moreover, with dynamic resource allocation typical in heterogeneous memory systems, e.g., clouds or other shared hardware scenarios, multi-tenancy [96] exacerbates threat complexity and diversifies attack vectors. Malicious tenants might exploit vulnerabilities [38] and breach isolation to access or tamper with others' data. We seek to bridge such a gap in this paper.

Motivation. Prior research studies on traditional memory security fall short in effectively addressing the challenges associated with developing a data management platform based on heterogeneous memory architectures. This situation demands a more precise delineation of the domain-specific characteristics of these challenges, thereby emphasizing a robust and clear motivation for investigating innovative and effective technical solutions.

First, in heterogeneous memory architectures, consideration of asymmetric security is imperative due to significant variations in security attributes (e.g., exclusive vs. shared), safeguards (e.g., local vs. remote), or the intrinsic properties (e.g., volatile vs. non-volatile) of different memory tiers. This results in the emergence of intricate and unique attack vectors, which are unlike the security challenges in homogeneous memory environments that generally operate on consistent, uniform assumptions about benign hypotheses or the capabilities of adversaries. In contrast, the asymmetric nature of heterogeneous architectures implies that the overall security is only as strong as its weakest

tier—a compromise in a single tier could jeopardize the robustness of the entire system. Therefore, it is essential to have a deep understanding and effective incorporation of this characteristic to enhance data security during interactions among tiers with disparate levels of security strength.

Second, effectively deploying proactive security solutions is challenging, especially in the context of methodologies based on specific benign conditions. This is because the subordinate memory tier within a heterogeneous memory framework often lacks the capacity to maintain any security assumptions. It is difficult to both reliably ascertain its activation and discern when a failure occurs. Thus, in these cases, it becomes imperative to consider the most pessimistic hypothesis: the application operates on a heterogeneous memory tier that is entirely compromised. Under such conditions, adversaries may have unrestrained and silent access to read or alter the managed memory data within this layer. This situation necessitates the implementation of robust passive security strategies, including encryption and verification mechanisms, to safeguard the applications' memory, even in such critical scenarios.

Third, most of the existing memory security mechanisms are efficient only with page-based memory management. While these coarse-grained page-swapping methods [5, 35] simplify memory allocation and replacement complexities compared to fine-grained management [24, 25, 71, 81], they have significant drawbacks for applications with memory access patterns that lack locality or exhibit global randomness. These drawbacks include notable latency fluctuations and I/O amplification [1, 17] upon page misses. Given that the interconnection bandwidth between memory hierarchies is often the bottleneck in memory-intensive applications [15], this can substantially impact performance. In these scenarios, fine-grained object-based schemes can be a complementary choice, offering superior performance [68, 81, 89], especially for a significant portion of typical data management tasks, such as table joins [11], key-value queries [89], and graph processing [100, 108]. However, achieving robust passive measures for pessimistic scenarios on fine-grained objects is not straightforward. This is because most data security techniques operate on larger data blocks to efficiently amortize the incurred overhead [37]. For example, to ensure the three primary properties of data security—confidentiality, integrity, and freshness—at least three types of metadata are required with traditional approaches: nonces, digests, and timestamps. Further, the corresponding algorithms involve initial computation costs unrelated to data block sizes. For fixed and sufficiently large pages, this may be acceptable. However, if such metadata is at the object level, e.g., a few bytes in length, the extra space overhead would be several times the effective data payload, clearly contradicting the primary goal of heterogeneous memory systems—spatial efficiency.

Our proposal. In this paper, we delve into object-oriented heterogeneous memory architectures, initiating the exploration of their data security challenges. We propose the object-oriented **U**nified **E**ncrypted **M**emory (**UEM**) management for heterogeneous memory architectures. Specifically, **UEM** is developed in C++ and exposes unified object references, allowing developers to effortlessly build data management platforms upon heterogeneous memory architectures. They can focus on data operational logic without becoming entangled in the nuances of memory management. Interactions between **UEM** and specific memory devices occur via a unified interface, ensuring that **UEM** is not tied to any specific hardware, exhibiting impressive scalability. As for data security, when objects are written back to the heterogeneous memory tiers through unified references, their data is encrypted. Similarly, during reads, **UEM** checks for data integrity and freshness at dereference, returning the decrypted original content. This entire security enforcement process remains transparent to developers. Central to our proposal is the way in which **UEM** provides cost-effective data security measures rooted in its design philosophy. Instead of viewing objects in isolation and managing security metadata independently, **UEM** employs centralized data structures.

These designs holistically ensure the security properties of all objects, thereby reducing storage overhead and processing latency more effectively than traditional methods.

Novelty. Our work identifies and addresses a critical yet overlooked issue in standard heterogeneous memory scenarios: the inconsistency in security assurance across memory hierarchies. As memory paradigms within data management platforms evolve, bridging this security gap becomes crucial. This paper pioneers efforts to resolve this discrepancy, proposing a forward-thinking approach to imposing security attributes at the object level within heterogeneous memory. Traditional solutions falter at this granularity due to excessive overheads in runtime and storage. To address these challenges, we introduce an innovative, centralized security strategy for effective and lightweight execution, applicable across standard heterogeneous memory models, irrespective of specific hardware, architectures, or implementations.

The contributions of this paper are summarized as follows.

- We identify a lack of attention to data security in existing work in the field of heterogeneous memory architectures. In response, we propose **UEM**, a unified object access for developers working within heterogeneous memory architectures, which ensures that data security attributes are maintained. To the best of our knowledge, this is the first work of its kind.
- To address the challenges of fine-grained data security, we propose innovative strategies for object-level data assurance and incorporate them into **UEM**. These encompass the *Aggregated Verification Set* (**AVS**) and the *Dynamic Mask Pool* (**DMP**), which are centralized data structures, making data security enforcement lightweight in heterogeneous memory architectures.
- To validate **UEM**'s efficacy across varied heterogeneous memory environments and typical data management computational workloads, we conduct an extensive experimental evaluation. Our experimental setup spans three leading-edge heterogeneous memory architectures: network-based disaggregated memory, non-volatile memory, and trusted execution environments; and three distinct workload types: tabular, key-value, and graph.
- The experiments demonstrate that **UEM** introduces reasonable overhead, ranging from 0.7% to 57.5%, depending on the varying hardware, compared to the state-of-the-art approaches that do not prioritize data security. Moreover, when evaluated against conventional data security methods, **UEM** exhibits exceptional lightweight characteristics, delivering a performance 1.8x - 6.3x faster and significantly reducing additional space consumption.

2 PRELIMINARIES AND RELATED STUDIES

2.1 Heterogeneous Memory Architectures

Heterogeneous memory, crucial in modern high-performance and efficient computing, integrates multiple storage types within a single system, tailored for various workloads and requirements.

Disaggregated Memory. In this paradigm [29, 64], computation and storage are decoupled and connected via a network, allowing remote memory resources to function as local. This architecture enhances resource sharing and scalability. Essential to its success is robust connectivity, with Ethernet excelling in cost-effectiveness and compatibility, among which the Transmission Control Protocol [46, 81] (TCP) is a key communication protocol. Techniques such as InfiniBand [35, 58], which is significant for Remote Direct Memory Access [16, 43] (RDMA), and Compute Express Link [34, 56, 65] (CXL), a new technology with cache coherence, are also integral.

Multi-type Memory Hybrid. This concept combines various memory types, each optimized for specific roles or data types. High Bandwidth Memory [75] (HBM), for instance, offers high bandwidth but poses challenges in manufacturing complexity and scalability [42]. Non-Volatile

Random-Access Memory [3, 33, 52, 79, 97, 98] (NVRAM), including NAND [44] and 3D XPoint [36], presents higher capacity but with slower access and shorter lifespans [74]. In a broader context, this category also includes CPU caches utilizing SRAM [77] and memory page swapping on disks [95].

Dedicated Memory Region. Even in systems with a singular physical storage type, memory is partitioned into regions, each satisfying specific needs like stability, performance, or security. For example, real-time embedded systems [59] may use dedicated regions for consistent memory access timing. Trusted Execution Environments [83] (TEE), such as Enclaves [20], Secure World [4], and their high-level applications [57, 85, 93, 101], epitomize this, safeguarding against software and hardware attacks and isolating sensitive data processing.

► **UEM** adopts a unified interface that abstracts interactions with different memory tiers, thus decoupling from any specific physical implementations. To thoroughly validate its generality, we conduct evaluations in typical scenarios of all three categories of heterogeneous memory architectures mentioned above, with specific experimental settings detailed in Section 6.1.1.

2.2 Memory Security

Modern systems increasingly grapple with memory security, a critical aspect exacerbated by the multitude of attack vectors, demanding a deeper understanding and enhanced protection.

Attack Vectors. Physical attacks pose a significant threat to memory security, involving direct control or access to the target system. These include hardware disassembly, storage media theft, or intercepting communications. In Infrastructure as a Service [62] (IaaS) contexts, hardware suppliers and system administrators may pose risks due to potential data spying [23] and physical device access. Multi-tenant hardware sharing scenarios also present vulnerabilities [96], enabling unauthorized memory access. New attack vectors such as Spectre and Meltdown [38] breach security to access other programs' memory in such environments. The Rowhammer vulnerability [48] exploits DRAM charge leakage, flipping memory bits to manipulate data and inject code. This can reveal encrypted data's side information [9], underlining the need for comprehensive security techniques, especially in insecure memory hierarchies.

Security Attributes and Threats. UEM prioritizes three main aspects of data security: confidentiality, integrity, and freshness. ***Confidentiality*** is crucial to keep information secret and private, preventing unauthorized access. Memory security threats like Spectre and Meltdown represent significant risks as they allow attackers to bypass security and access sensitive data such as passwords or personal profiles in memory. ***Integrity*** relates to maintaining the data's consistency and accuracy during storage, transmission, or processing. Attacks such as Rowhammer, which can alter memory cell contents and cause data corruption, highlight the importance of integrity. ***Freshness*** ensures that data is current and not superseded by outdated or expired information. Rollback attacks, aiming to reset the system to a former state to disguise malicious activities as legitimate, are a direct challenge to ensuring data freshness. These could involve reverting to a previous login session to gain unauthorized access or compelling the system to repeatedly perform a previous action to achieve excessive insights.

Security Defense Techniques. Defense strategies against unauthorized memory access can be broadly categorized into proactive and reactive measures.

(I) Proactive Protection: this involves isolation techniques to prevent illegal memory actions. Operating systems use memory isolation [6] to block cross-process access, but privilege vulnerabilities remain a concern. Sandboxing methods [90] like NaCl [106], gVisor [60], and VC3 [84] limit kernel interaction and verify addresses before access. StackGuard [22] and ProPolice [28] counter stack overflow attacks, while Safe Linking [41] addresses heap overflows. Address Space Layout Randomization [86] and Address Obfuscation [13] further enhance security by confusing attackers.

(II) Reactive Protection: this includes methods that detect certain unauthorized access incidents. Techniques like cryptographic algorithms [70] and AEAD are used to encrypt data and authenticate messages [10], ensuring confidentiality and integrity. However, memory page verification mechanisms have limited scope and can be circumvented by advanced attacks. Advanced protections like Merkle trees [94] face difficulties in concurrent environments due to high storage and processing requirements [8, 55, 109]. Additionally, technologies like TME [47], SGX [20], and TDX [18] provide enhanced memory security, especially with dedicated hardware.

► **UEM** prioritizes reactive security measures like encryption and validation to safeguard data in heterogeneous memory environments, which are inherently less controllable. Given the unpredictability and strength of potential adversaries, including those with physical device access, **UEM** adopts a cautious approach. Proactive measures are less feasible in such varied memory tiers and are more prone to breaches. **UEM** conceals the layout of objects in these memory tiers from potential attackers. It encrypts data upon writing and checks integrity and freshness during reading, guarding against tampering and rollback attacks. We detail **UEM**'s threat model in Section 3.1, and present its innovative security mechanisms that bolster performance in Sections 4.3 and 4.4.

2.3 Object-oriented Memory Management

Traditional page-based data access often leads to I/O amplification, as retrieving a small data portion requires loading a complete page. To address this inefficiency, various studies propose object-oriented memory management techniques such as AIFM [81], FlatFlash [1], and Project PBerry [17]. These methods, while more complex due to the variability in object sizes, offer benefits like detailed data lifecycle observation and enhanced Garbage Collection (GC) efficiency. Techniques include reference counting [40], generational approaches [76], and dynamic address resolution [103], enabling flexible data location management and supporting defragmentation. In this context, log-structured memory management [39, 82, 104], treating memory as a sequential log, has gained prominence. This approach allows for data duplication and relocation to minimize fragmentation. It employs incremental GC, efficiently reclaiming memory and reducing GC-related lags. The emergence of heterogeneous memory architectures introduces complexities in coordinating multiple memory tiers. Solutions like Mako [61] address interconnect bandwidth issues by offloading tasks to remote memory, and MemLiner [99] optimizes GC and application process coordination, using a priority-based algorithm to manage memory eviction. Additional research efforts are directed towards enhancing prefetch techniques [49, 63] to further refine memory management.

► **UEM** supports memory access by exposing unified references to applications. It employs a log-structured memory management scheme to achieve object-level allocation, in an encrypted manner, with only the necessary meta-information to support security properties. The specifics of this implementation are elaborated in Section 4.1.

3 UEM APPROACH

3.1 Threat Model

3.1.1 Targeted Scenarios. Within heterogeneous memory architectures, various memory tiers are ingeniously interwoven to create a cohesive memory system that facilitates the execution of applications. However, these architectures inherently possess asymmetry in security among memory hierarchies. A security weakness in any tier poses a significant threat to the overall system's robustness. In this context, we assume that the execution of the application, especially the memory for loading application processes, is secure, with **UEM** management processes integrated as part of the application. Application owners might take necessary measures to ensure this confidence, such as deploying the application on physically controllable or non-shared dedicated computing nodes,

or implementing security barriers like TEEs. Our study focuses on the standard heterogeneous memory model, thus not assuming any specific deployment forms, i.e., discussions on gaining such trust for particular deployments are beyond this paper's scope. To demonstrate the adaptability of our proposal, we evaluate **UEM** atop three typical deployment instances of heterogeneous memory aligned with the assumptions of asymmetric security, including TCP-based disaggregated memory, DRAM-NVRAM hybrid deployment, and TEEs supported by Intel TDX. This is consistent with the heterogeneous memory categorization investigated in Section 2.1, and a detailed description of these instances is in Section 6.1.1.

3.1.2 Threat Agents. In the scenario described above, the identities of threat agents targeting memory security can be diverse and difficult to pinpoint. This paper assumes that the adversary is malicious. Firstly, in heterogeneous memory architectures, the primary purpose often includes resilient resource allocation. This implies non-exclusive, shared resource consumption, enabling co-tenants to potentially bypass isolation mechanisms or maliciously exploit system vulnerabilities for unauthorized access. Secondly, the scope of threats may also extend to privileged entities abusing their elevated access permissions. Complicated infrastructures typically require a substantial number of specialized operational staff, making it challenging to fully prevent a few from becoming corrupt or acting with malicious intent. Thirdly, particularly in the case of platforms exposed to the public domain, there is a constant threat from unknown external forces that might infiltrate and cause disruptions. For example, a hacker who successfully penetrates a routing node can effectively monitor traffic. Lastly, in extreme situations, threats to memory security can be more fundamental, such as physical intrusions into devices. This may involve eavesdropping on or hijacking device interfaces, or even disassembling and stealing memory storage devices. We note that, regarding data security in runtime memory, attacks can transcend mere data compromise by potentially altering applications' execution, such as modifying conditional branch criteria. Executing these attacks effectively demands extensive knowledge of the targeted application.

3.1.3 Countermeasures. As outlined above, when the application interfaces with diverse memory tiers via **UEM**, all tiers external to the application's boundary are deemed untrusted, regardless of their specific forms. Given the complex dynamics and unpredictable nature of threat agents, it is imperative to consider the most extreme scenarios. Under such pessimistic assumptions, it is imprudent to rely on any benign presuppositions about the untrusted environment, as we presume these facilities to be entirely compromised. Furthermore, the threat agents, distinguished by their diverse origins and varying capabilities, are collectively represented in a uniform, potent adversary model. That is, the adversaries are capable of fully controlling the untrusted memory layers, including communication with the application, granting them unfettered and covert access to and modification of data. In this demanding scenario, **UEM** aims to provide a resource-efficient means to ensure data security, encompassing confidentiality, integrity, and freshness. For confidentiality, **UEM** consistently applies robust encryption to data leaving the trusted environment, preventing adversaries from deciphering any information from the untrusted domain. For integrity and freshness, **UEM** detects any tampering with or rollback attempts on the data upon its retrieval. In the event of unauthorized actions, **UEM** will issue an urgent exception alert to users. These countermeasures ensure that even if heterogeneous memory tiers are fully breached, applications built atop them can still operate stably through **UEM**, while maintaining memory security.

3.2 Rationales and Key Ideas

The foundational concept of unified memory is to provide developers with a seamless interface to read and write data in different physical states, allowing for the transparent management of heterogeneous memory hierarchies within application logic. This necessitates a mediating

"interchange station," which acts as a bridge between the seamless interface and the various physical layers, managing tasks such as scheduling and transformation.

This paper focuses on object-oriented scenarios, emphasizing fine-grained memory management. The "seamless interface" that **UEM** provides for accessing objects across heterogeneous memory is a reference, i.e., `UemRef`. On the application end, ownership and access to data are determined by holding instances of object references. `UemRef` is a kind of smart pointer, akin to `unique_ptr`, encompassing metadata used by the so-called interchange station (i.e., **UEM** Manager), to derive a valid raw address when access is imperative. This process is referred to as "dereferencing."

Within this framework, the primary challenge lies in the additional costs associated with implementing memory security measures. Specifically, in a heterogeneous memory architecture where data is accessed and modified through `UemRef`, data remains confidential and unaltered when scheduled across different memory hierarchies. Conventional methods for maintaining confidentiality and integrity typically perceive data objects as independent entities. For instance, the prevalent AEAD method encrypts data, computes a data tag, and appends all relevant metadata to the ciphertext. This is useful and efficient when communication is distributed among numerous parties without a central trusted entity to manage all required metadata for confidentiality and integrity, or when each entity's size is substantial, so the security mechanisms yield relatively minimal overhead per data unit. However, if AEAD or similar strategies are directly applied to smaller objects, their design is not optimal, because of the considerable overhead they add per data unit in terms of both computation and storage. To mitigate this issue, we recognize the unique nature of **UEM** in its associated context, where the managed objects serve as volatile data in runtime memory. Thus, their decryptability aligns with the persistence of the corresponding process, eliminating the need to understand its content outside the host process. This insight enables centralized protection of all data with substantially reduced overhead.

To safeguard the integrity and freshness of data, **UEM** employs Aggregated Verification Sets (**AVS**), a centralized design, to eliminate the need for maintaining individual tags for each object. The validation of integrity and freshness is not performed with every object access but is asynchronously recorded within **AVS** upon access. **AVS** contains one aggregated set for read operations and another one for write operations. A routine check by a background task verifies **AVS**'s correctness. Discrepancies between the sets indicate data tampering, while consistency suggests the data remains untouched or has not been rolled back. This centralized strategy considerably diminishes computational and storage complexities, which will be further discussed in Section 4.3.

To ensure confidentiality, **UEM** also devises a centralized data structure, called the Dynamic Mask Pool (**DMP**) for data encryption. This method avoids the necessity of creating and storing separate Initialization Vectors (IV) for each object. Similar to the idea of stream ciphers, the encryption/decryption process involves a simple XOR calculation on the plaintext/ciphertext, with a bit string, which is fetched from **DMP** and has the same length as the object. Therefore, the algorithm stores only a mapping that links each object to its position within the **DMP**. A more detailed explanation of this algorithm is provided in Section 4.4.

3.3 Architecture Overview

The architecture of **UEM** is illustrated in Fig. 2. **UEM** assumes applications access heterogeneous memory within a "trusted environment," depicted in light green in the diagram. In contrast, the storage hierarchy of heterogeneous memory is considered risky. When an application constructs an object in heterogeneous memory, it acquires a **UEM** reference through the **UEM** Manager's allocation interface. The **UEM** Manager maintains a local memory pool of a predetermined size and manages resident objects utilizing a log-structured approach. Here, every new object acts as an entry in an ongoing log, meaning each object gets appended at the end of the log. The log is

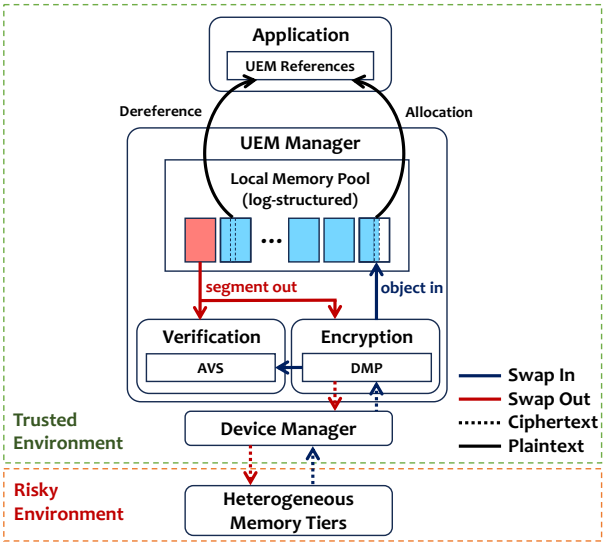

Fig. 2. The overview of **UEM** architecture.

divided into segments by a consistent size. When the successive appends to the log saturate the local memory pool, the **UEM** Manager selects an inactive segment, which does not contain any objects currently in use, to be evicted to free up space.

The pivotal operation within **UEM** is the dereference of **UEM** references. This requires the **UEM** Manager to produce a valid raw pointer and ensure it is accessible within its scope. If the object is still in the local memory pool during this process, its memory address is returned immediately. Otherwise, the object must be retrieved from the remote location and then re-added to the log, replicating the initial allocation.

UEM utilizes a unified interface to abstract interactions with memory tiers. Specifically, for each memory tier, there is an instance referred to as the "Device Manager" that concretely implements specific hardware interaction protocols. Upon the eviction of a log segment, its data is registered to the Data Verification Module to record relevant digests and is subsequently encrypted. Afterward, it is migrated to a heterogeneous memory tier through the Device Manager. Similarly, when an object requires retrieval, it is fetched from the remote side by the Device Manager, decrypted by the Data Encryption Module, integrated into the local memory pool in plaintext, and registered within the Data Verification Module.

Periodically, the Data Verification Module performs asynchronous checks to ensure alignment between incoming and outgoing data. Should any inconsistencies arise, a catastrophic exception will be triggered, notifying the application of tampered memory data. In the absence of such anomalies, the multifaceted process encompassing heterogeneous memory scheduling, data en/decryption, and verification remains imperceptible to the application.

4 DESIGN AND IMPLEMENTATION

4.1 Abstractions and Interfaces

UEM is implemented in C++ and manages application data at the granularity of objects. **UEM** provides the following interfaces for applications, with an example code snippet in Fig. 3.

4.1.1 Device Manager. To accommodate the diversity in memory hierarchies, **UEM** separates the logic of security enforcement and object swapping for varying hardware through an abstraction termed `DeviceManagerInterface`. This abstraction offers three pivotal interfaces. `allocate_memory` can be invoked to allocate a memory block of a specified size on the managed device,

```
class DeviceManagerInterface {
public:
    virtual void* allocate_memory(size_t size) = 0;
    virtual void swap_in(uint64_t addr, size_t len, uint8_t *dest);
    virtual void swap_out(uint64_t addr, size_t len, uint8_t *data);
};

class UemManager {
public:
    UemManager(DeviceManager* dm, size_t local_size);
    template <class T> UemRef<T> allocate();
};

void multiply(UemManager &uemm, UemRef<int> &product,
              UemRef<int> &factor1, UemRef<int> &factor2) {
    UemScope scope(uemm);
    int* f1 = factor1.get(scope);
    int* f2 = factor2.get(scope);
    int* p = product.get(scope);
    *p = (*f1) * (*f2);
    // scope exits due to destruction
}
```

Fig. 3. Code snippet of **UEM** interfaces.

which is subsequently employed to store the swapped-out data. The Device Manager also extends interfaces for both data `swap_in` and `swap_out`. To accommodate various heterogeneous memory tiers, users are required to implement these interfaces. It serves as a driver for **UEM**, ensuring alignment with the precise specifications of the hardware.

4.1.2 Memory Manager. Applications interact with the heterogeneous memory architecture utilizing `UemManager`. To allocate memory space, it is mandatory for an application to first instantiate a `UemManager` instance in C++. Two major arguments are embedded in `UemManager`'s constructor. The application, by the device manager implementation `dm`, determines the device to which the data will be relocated if local memory reaches its capacity. Application developers are relieved from the intricacies of divergent programming interfaces for different remote devices, except for the initial creation of the device manager object with essential configurations, e.g., the physical address of the device. The application also needs to delineate the size of the local memory pool to properly initialize and manage swapping behaviors. Once `UemManager` is initiated, applications can leverage it to allocate memory for varied object types. `allocate<T>()` serves the purpose of allocating memory equal to the size of the object type represented by `T` within the `UemManager`. Akin to smart pointers in C++, objects are automatically deallocated when they become obsolete.

4.1.3 Dereferencing and Scope. The dereferencing of the obtained raw address (i.e., a pointer) is only temporarily valid, as its physical allocation is dynamically managed by `UemManager`. **UEM** utilizes scopes to secure an object in the local memory, ensuring that the background processes neither evict nor move it within the local memory pool during its active use. To dereference a `UemRef`, it must be associated with a `UemScope` object, and the acquired pointer from the `.get()` method is operable solely within the lifespan of `UemScope`. Departing from this scope requires re-dereferencing. Failing to do so and using the obsolete pointer could result in undefined behaviors. Such a mechanism is elucidated with a basic example function `multiply`. This function multiplies two factors, stored in `UemRef` factor1 and factor2, and retains the result in product. Initially, a `UemScope` is generated and bound with `UemRef` dereferencing, yielding the object's pointer in the

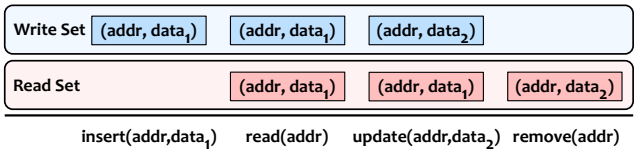

Fig. 4. An example of **AVS** with operations.

local memory pool. Consequently, the three pointers obtained via this scope binding preserve their validity throughout their existence, irrespective of their read-write utility, such as in multiplication calculations, until the encompassed `UemScope` instance is destructed at the function's termination. Though **UEM** offers the choice for programmers to pin desired objects in the local memory to allow faster access, to prevent any impediment to the scheduling mechanism of **UEM**, developers should precisely define the scope to avoid excessive pinning of **UEM** objects.

4.2 Object Swapping

`UemManager` employs the concept of log-structured memory [82] (LSM) to optimize the efficient utilization of valuable local memory space. LSM subdivides the entire memory space into distinct segments. When local memory consumption exceeds a predetermined threshold, `UemManager` implements a FIFO policy to designate a segment and subsequently relocate objects from that segment to managed devices. During the dereference procedure, if the pertinent object is located remotely, `UemManager` allocates a new local space within LSM and liaises with the device manager to transition the object—rather than the entire segment—back into the local memory. `UemManager` also incorporates prefetching for specific data structures characterized by conventional memory access patterns, such as sequential access in arrays. Before the commencement of the data swapping, `UemManager` necessitates synchronization with both the Data Encryption Module and the Data Verification Module to ensure that data is provided with adequate protection. Subsequently, the protected data is dispatched to the device manager. Conversely, when an object is being swapped in, `UemManager` receives encrypted data from the device manager. This then necessitates further interactions with the Data Encryption Module and the Data Verification Module prior to the restoration of the data to the application. This meticulous approach ensures the secure and efficient management of memory spaces, enabling seamless transitions between local and remote memories while maintaining the integrity and confidentiality of the data. The strategic employment of prefetching and structured memory organization further enhances the effectiveness and operational seamlessness of `UemManager` in managing memory spaces and data structures.

4.3 Data Verification Module

UEM enforces write-read consistent memory to ensure the data stored in the untrusted memory is not tampered with. By definition, the memory is write-read consistent if, and only if, for every read at the address *addr*, it returns the data most recently written at the same address *addr*.

4.3.1 Design of **AVS**. The design of **AVS** (aggregated verification sets) is inspired by Blum et al. [14], who introduced an efficient way to verify if the memory is write-read consistent. **AVS** consists of two primitives in the local memory—a set of all read operations (the read set, denoted as $\mathcal{RS}$) and a set of all write operations (the write set, denoted as $\mathcal{WS}$). The observation is straightforward—if the data is not tampered with, the two sets should be the same. To achieve this property, the following additional rules should be enforced to ensure that for each write operation, there is exactly one corresponding read operation, and vice versa: ❶ After any read operation, a write operation on the same address must be performed. ❷ Before any data overwrite or update, a virtual read operation on the same address must be performed.

Algorithm 1 Data Verification Module

```
1: currentScanning = -1
2: h_WS = h_RS = h_RS^new = h_WS^new = 0 # Initialize sets
3: function processRead(addr, data)
4:     if addr > currentScanning then
5:         h_RS = h_RS ⊕ PRF(addr, data)
6:         h_WS = h_WS ⊕ PRF(addr, data)
7:     else
8:         h_RS^new = h_RS^new ⊕ PRF(addr, data)
9:         h_WS^new = h_WS^new ⊕ PRF(addr, data)
10:    end if
11: end function
12: function processWrite(addr, oldData, newData)
13:    if addr > currentScanning then
14:        h_RS = h_RS ⊕ PRF(addr, oldData)
15:        h_WS = h_WS ⊕ PRF(addr, oldData)
16:    else
17:        h_RS^new = h_RS^new ⊕ PRF(addr, oldData)
18:        h_WS^new = h_WS^new ⊕ PRF(addr, newData)
19:    end if
20: end function
21: function verification
22:    for obj ∈ remoteObjects do
23:        currentScanning = obj.addr
24:        h_RS = h_RS ⊕ PRF(obj.addr, obj.data)
25:        h_WS^new = h_WS^new ⊕ PRF(obj.addr, obj.data)
26:    end for
27:    if h_RS ≠ h_WS then
28:        Raise("Verification fails!")
29:    end if
30:    currentScanning = -1
31:    h_RS = h_RS^new
32:    h_WS = h_WS^new # Flush values in new sets to current sets
33:    h_WS^new = 0
34:    h_RS^new = 0 # Clear new sets
35: end function
```

Fig. 4 shows an example of how **AVS** works. Initially, both sets are empty. We first insert $data_1$ to the address $addr$, and insert a tuple $(addr, data_1)$ into $\mathcal{WS}$. Next, we read the data at $addr$, and subsequently, insert $(addr, data_1)$ into $\mathcal{RS}$. According to ❶ above, a virtual write operation is also performed, so that $(addr, data_1)$ is appended to $\mathcal{WS}$. Next, for an update operation, with ❷ above, we first read the original data and append $(addr, data_1)$ to $\mathcal{RS}$, and then append the updated data $(addr, data_2)$ to $\mathcal{WS}$. Finally, for a remove operation, we append the corresponding tuple $(addr, data_2)$ to $\mathcal{RS}$. In the end, both $\mathcal{RS}$ and $\mathcal{WS}$ have two $(addr, data_1)$ tuples and one $(addr, data_2)$ tuple, signifying untampered data.

4.3.2 Representation of **AVS**. Storing all the tuples in the sets, as shown in Fig. 4, is expensive for large systems. An efficient way is to store the collision-resistant hashes of the sets, and the

equivalence of sets can be evaluated by checking the equivalence of the hashes. The hashes of $\mathcal{RS}$ and $\mathcal{WS}$ are defined as the XOR sum of the Pseudo-Random Functions (PRF) of all set elements:

$$h_{\mathcal{WS}} = \bigoplus_{(addr,data)\in\mathcal{WS}} PRF(addr, data)$$
$$h_{\mathcal{RS}} = \bigoplus_{(addr,data)\in\mathcal{RS}} PRF(addr, data)$$

It has been proven that $\mathcal{RS} = \mathcal{WS}$ implies $h_{\mathcal{RS}} = h_{\mathcal{WS}}$, and $h_{\mathcal{RS}} = h_{\mathcal{WS}}$ implies $\mathcal{RS} = \mathcal{WS}$, with high probability p, where $1 - p$ is negligible [14]. Algorithm 1 shows the details of the process for updating the two sets of **AVS**.

4.3.3 Pauseless Verification Process. In the example shown in Fig. 4, the two sets are consistent only when the object is removed. Evidently, we do not want to wait until the program finishes and then remove all data to perform the verification. Instead, during runtime, we perform a virtual remove operation (i.e., adding the tuple to $\mathcal{RS}$), check if the two sets are identical, and then perform a virtual insert operation (i.e., adding the tuple to $\mathcal{WS}$). Concerto [8] further introduces a method for performing the verification without pausing the application, thereby ensuring that the write-read consistent memory adds minimal performance overhead to the applications in runtime. When the verification is in progress, the verifier records the object's address that it is currently scanning (the `currentScanning` variable in Algorithm 1). When a concurrent read or write request from the application occurs, it checks whether the object has already been scanned in the current verification iteration by comparing the address of the object against the cursor (lines 4 and 13). If so, all updates will be performed on the new sets; otherwise, on the current sets. Upon completion of the verification process, the data in the new sets will be flushed to the current sets, and the new sets will be empty until the next verification process. While the pauseless verification process can operate in the background without interrupting the application execution, it does not come for free—memory scanning will incur considerable CPU and I/O resource consumption. To mitigate this, prior studies such as FastVer [7] have proposed leveraging the hot-cold dichotomy in managed memory to reduce the scanning scope. Nonetheless, such approaches do not tackle the problem fundamentally, representing an ongoing research challenge.

4.4 Data Encryption Module

4.4.1 Generation of Masks and Mask Offsets. The Data Encryption Module of **UEM** uses an approximation of one-time pad (OTP), which is proven to have perfect secrecy [88]. When an object needs to be swapped outside the trusted memory, n_m masks/pads ($n_m > 1$) for this object with the same size will be fetched from **UEM**'s **DMP** (dynamic mask pool) with the generated *mask offsets*. **DMP** is a block of secure local memory with a configurable size of s, containing random bytes that are generated during the initialization of any applications running on top of **UEM**, and the mask offset refers to the location of a specific mask in **DMP**. **DMP** is dynamic because: ❶ The mask offsets are generated dynamically during swapping and will not be reused. ❷ The bit string in the pool is dynamic in that it is refreshed periodically. During the swapping process for an object with size s, the Data Encryption Module first generates n_m distinct random numbers from 0 to $s - 1$, which will serve as the offsets in **DMP**. With the offsets, n_m blocks of memory with the same size of the object will be fetched from **DMP** as the masks and be used to perform encryption.

4.4.2 Encryption and Decryption. Fig. 5 shows an overview of how the plaintext `pt` is encrypted to the ciphertext `ct` with $n_m = 3$. First, we generate three random offsets and use them to retrieve three masks, denoted as m_1, m_2, and m_3, from **DMP**. Then, we take the plaintext t_0 as the input for the first round. For each round, we process the input t_{i-1} through the Rijndael S-box [80] to introduce non-linearity, obtaining $S(t_{i-1})$. Next, an XOR operation is applied to $S(t_{i-1})$ with the mask m_i.

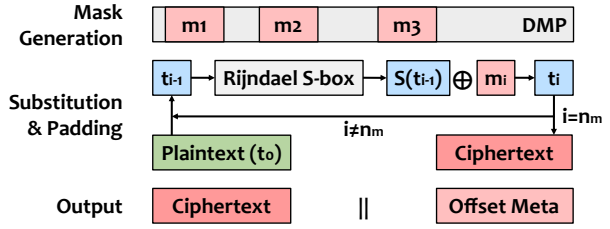

Fig. 5. An overview of data encryption with $n_m = 3$ masks.

This process continues until $i = n_m$, indicating that all n_m rounds of substitution and padding are complete. Finally, the metadata will be appended to the ciphertext before it is written to the remote device, and will be used to look up the mask offsets for decryption. The metadata includes the ID (or address) of the object, and the Data Encryption Module keeps a mapping from all object IDs to the mask offsets. In Section 4.5, we explore how we refine this design to substantially decrease the storage usage for mapping data.

4.4.3 Security Enforcement. To ensure the confidentiality of data, the Data Encryption Module employs specific design strategies as follows.

Multiple masks with secret offsets. We employ $n_m > 1$ masks to encrypt the data through multiple rounds. This is done to prevent potential overlap or collision of mask segments in **DMP**, which might be exploited by adversaries. Moreover, the mask offsets in **DMP** are not written to the untrusted device, so adversaries will not learn any information about the relative position of the masks of two objects in **DMP**. Consequently, even in cases where mask values intersect, adversaries would remain oblivious to such overlaps. A detailed exploration of the implications of these values on the security robustness of UEM is presented in Section 5.1.

DMP rotation. UEM employs **DMP** rotations to prevent long-term mask collisions. Periodically, we refresh the entire **DMP** with freshly generated random bytes, and subsequently, data on the remote device is encrypted using these new masks. This process is co-designed with the verification process and achieves slight performance overhead. More details are provided in Section 4.5.

One-off masks. The masks employed for encrypting an object are single-use and are discarded once the decryption process is completed. In the event that the same object needs to be transferred to an untrusted device again, new masks will be generated. This approach ensures that potential adversaries cannot glean comparative information about different instances of the same object.

4.5 Optimizations

4.5.1 Grouped Mask Management. When a remote object needs to be swapped in, the masks (or the mask offsets) are needed for decryption. On the one hand, storing offsets in untrusted memory leads to security issues. On the other hand, local storage incurs significant storage overhead on the limited trusted memory, a typically constrained resource. To reduce this overhead while maintaining security, we enhance the basic mask offset generation process described in Section 4.4.1 with the following upgraded methodology.

The core idea of the new approach is to group multiple objects and store only one mapping for each group. Meanwhile, we still need to keep the mapping invisible from the untrusted device. Hence, even if in the rare case two offsets collide, adversaries learn nothing about it. To achieve this, we introduce an indirect mapping from objects to an array of mask offsets. First, for each object, we allocate a *(group_id, group_index)* tuple, where *group_index* refers to the index of the object within the group. We store this tuple in untrusted memory with the object. In trusted memory, we only keep a mapping from *group_id* to the mask offsets, and compute the offsets for the objects with the

Algorithm 2 Group Allocation Process for Objects

```
1: nextGroup = 0
2: categories = [16, 32, 64, 128, ...]
3: maxContMask = 65536 # The maximum continuous mask size
4: curGroup = dict() # category -> (group ID, group size)
5: function INIT
6:     for c in categories do
7:         curGroup[c].id = -1
8:         curGroup[c].size = max(1, maxContMask / c)
9:     end for
10: end function
11: function GETSIZECATEGORY(size)
12:     for c in categories do
13:         if size ≤ c then
14:             return c
15:         end if
16:     end for
17: end function
18: function ALLOCATETUPLE(size)
19:     c = GETSIZECATEGORY(size)
20:     if curGroup[c].size * c ≥ maxContMask then
21:         # Allocate new group id
22:         newGroup = nextGroup
23:         nextGroup = nextGroup + 1
24:         curGroup[c].id = newGroup
25:         curGroup[c].size = 1
26:         return (newGroup, 0)
27:     else
28:         # Use existing group
29:         groupId = curGroup[c].id
30:         groupOffset = curGroup[c].size
31:         curGroup[c].size = curGroup[c].size + 1
32:         return (groupId, groupOffset)
33:     end if
34: end function
```

same *group_id*. Algorithm 2 shows how we allocate the *(group_id, group_index)* tuple for an object. To reduce the probability of mask collision or overlapping, we set a maximum continuous mask size for each group, denoted as `maxContMask`. A group grows until the masks in the group reach this size limit. By design, objects with similar sizes will fall into the same group. The algorithm first checks which size category the object falls into with the `getSizeCategory` function. Next, it checks if there is any available group of this size category that is not full (line 20). If so, the object gets the group ID of this group with the next available position in this group as its group index. Otherwise, a new group corresponding to this size category will be created. If the mask offset for *(group_id, 0)* is denoted as k, the corresponding offset for *(group_id, i)* is calculated as $k + i * c$, where c represents the object's size category.

We note that the grouping mechanism does not compromise security. First, objects are still encrypted with multiple masks, and a group ID will map to an array that includes all mask offsets.

Second, by design, mask offsets for objects in the same group do not overlap, since the interval between mask offsets for objects is equal to the size category, which is not smaller than any object. Third, the group-based masks are also one-off—once an object is retrieved and decrypted using its *group_id* and *group_index*, these metadata are no longer used. The retrieved object is treated as newly allocated in the local memory pool and will be re-attempted for group allocation. Upon eviction from the local memory pool to the untrusted tier, it is re-encrypted according to its newly allocated group. That is, consistent with the one-off mask principle, group allocation for an object correlates with its eviction from the local memory pool. Re-allocating groups and group indices for objects may result in "holes" in a group. Compaction is integrated into the **DMP** rotation process. Objects will be migrated to new groups if necessary, before they are encrypted with new masks and written to the remote memory. The mapping from *group_id* to the mask offsets will be stored in the local memory. Compared to the method of retaining mappings for each individual object, the overhead is much smaller. For instance, for objects with a size of 64 bytes, **UEM** only has to store one mapping for every 1024 objects, resulting in an overhead that is merely one-thousandth compared to a strawman solution.

4.5.2 Space Elimination for Timestamps. Rollback attacks are a well-known technique that can potentially bypass the integrity verification system. In such attacks, adversaries capture a copy of the data generated by the system, successfully passing the integrity verification. Subsequently, adversaries replace the current data with a previously obtained version. Without proper design considerations, a system might remain oblivious to this tampering, as the altered data still possesses a valid MAC or PRF output. To protect against rollback attacks, conventional systems [7, 8, 67, 72, 109] typically rely on a timestamp or an equivalent to ensure the freshness of the data. Storing an extra timestamp for each object is relatively inexpensive in some systems where swapping occurs on a page level [67, 72, 109], but it leads to significant relative storage overhead for **UEM**, which optimizes for objects smaller than pages. Recall that in the Data Encryption Module, all masks for encryption and decryption are strictly for one-time use. Once the decryption process is completed, the mask offset is discarded, and a new mask is employed when the object is swapped out again. Consequently, even for two pieces of the same ciphertext, since the mask has been changed, they will be decrypted into a different plaintext, leading to inconsistencies between $\mathcal{RS}$ and $\mathcal{WS}$. Therefore, **UEM** collaboratively designs the Data Verification Module and the Data Encryption Module, eliminating the need for a timestamp.

4.5.3 Batch Verification. The verification process can be made more efficient through batch verification. As a reminder, LSM divides the memory space into segments with a default size of 1MB. During the verification process, instead of processing one remote object at a time, **UEM** fetches the entire segment and processes the objects in that segment locally. With batch verification, we reduce the number of remote memory accesses and improve the performance. We assess the efficacy of batch verification in Section 6.2.3.

4.5.4 Efficient **DMP** *Rotation.* Recall that during the verification process, we read objects from remote memory, perform a virtual remove operation (i.e., update $\mathcal{RS}$), check set equivalence, and then perform a virtual insert operation (i.e., update the new $\mathcal{WS}$). At this stage, if **DMP** rotation is configured to take place, a new **DMP** is generated before processing the first remote object. The verifier reads the remote data, decrypts it with the mask in the old **DMP**, and appends the information to $\mathcal{RS}$. Then, it generates the mask offsets for the same object from the new **DMP**, encrypts the data with the new mask, and writes it back to remote memory. The verifier also updates $\mathcal{WS}$ accordingly. After all the objects are processed, all remote objects will be encrypted with the mask in the new **DMP**, and the old **DMP** will be discarded.

5 SECURITY DISCUSSIONS

5.1 Encryption Strength

The confidentiality of **UEM** is achieved by **DMP**, an approximation of OTP. OTP is proven to have perfect secrecy, but it requires the key (mask) to be the same length as the plaintext, and the key must be randomly generated every time and never reused, making OTP overly expensive for fine-grained objects. **DMP** approximates OTP with two important parameters—the size of **DMP** s and the number of masks n_m. Additionally, the maximum size of the object $|obj|$, and the number of objects n_o, are also considered.

Setup and Notations. Consider a set of objects, $O = \{o_1, \cdots, o_{n_o}\}$. Each object o_i is encrypted with n_m masks, $\mathcal{M}_i = \{m_{i,1}, \cdots, m_{i,n_m}\}$. Let the offsets of the masks $\mathcal{M}_i$ be the *keys*, denoted by $\mathcal{K}_i = \{k_{i,1}, \cdots, k_{i,n_m}\}$. Define $\mathcal{K}_i(1)$ as the first element in $\mathcal{K}_i$, and $\mathcal{K}_i(2)$ as the second element, and so forth. Consider two objects o_i and o_j, along with their masks and keys. Let $\Delta_{i,j,l} = \mathcal{K}_j(l) - \mathcal{K}_i(l)$, where $1 \le l \le n_m$. In the contexts where only two objects o_i and o_j are considered, we omit i and j and use Δ_l to represent $\Delta_{i,j,l}$. Particularly, we assume that s is much larger than $|obj|$; otherwise, the traditional OTP generation for large objects is preferable.

The Adversary's Advantages. Consider two objects $o_i, o_j \in O$. The adversary gains an advantage if and only if all Δ_i ($1 \le i \le n_m$) are equal, and $-|obj| < \Delta_i < |obj|$. In such a case, a segment of o_i and a segment of o_j will be encrypted using the exactly same sequence of masks. Consequently, an attacker might deduce information from this (e.g., if two plaintext messages share the same segment and are encrypted with the same sequence of masks, then the corresponding ciphertexts will also share an identical segment). In other scenarios, due to the randomness of the masks and the non-linear property provided by substitution, no information will be leaked. Drawing a parallel to the birthday problem [45], the probability that for *any* two objects, the final masks encrypting them do not overlap is $p_{no-overlap} \ge \Pi_{i\in[1,n_o-1]}(1 - \frac{(2|obj|+1)i}{s\cdot(s-1)\cdots(s-n_m+1)}) > (1 - \frac{(2|obj|+1)(n_o-1)}{s\cdot(s-1)\cdots(s-n_m+1)})^{n_o-1}$. Using the limit $\lim_{x\to 0}(1-x)^{1/x} = 1/e$, it is further estimated as $p_{no-overlap} \simeq e^{\frac{-(2|obj|+1)(n_o-1)^2}{s\cdot(s-1)\cdots(s-n_m+1)}}$. Since s is much larger than n_m, we have $p_{no-overlap} \simeq e^{\frac{-(2|obj|+1)(n_o-1)^2}{s^{n_m}}}$. As a reference, in a scenario where $s = 2^{27}$, $n_o = 2^{30}$, $n_m = 5$, and $|obj| = 64$, the probability of any overlap occurring, based on Taylor expansion, is approximately $1 - e^{-2^{-68}} < 2^{-68}$, which is negligibly small.

Parameter Selection. From the analysis, it is clear that s and n_m are crucial in determining the adversary's edge against **DMP**. To enhance security, users may opt for larger values of s and n_m, though this increases memory usage and encryption delay, respectively. **UEM** provides interfaces for users to customize these parameters.

5.2 Verification Reliability

The Data Verification Module (**AVS**) of **UEM** employs cryptographic strength and computational difficulty to thwart adversaries. By incorporating a potent Pseudo-Random Function (PRF) along with a hash function resistant to collisions, it significantly challenges the feasibility of attacks, thus ensuring data integrity and freshness.

Write-Read Consistency. The concept of write-read consistent memory, which **AVS** incorporates, originates from Blum et al. [14] and has been subsequently adopted by the database sector in systems such as Concerto [8] and VeriDB [109]. Blum et al. demonstrated that if any data has been tampered with, the write set and read set must be inconsistent. **UEM** guarantees write-read consistency by keeping **AVS** in the trusted local memory and always updating the sets when interacting with the remote memory. **UEM** periodically checks that the read set $\mathcal{RS}$ and write set $\mathcal{WS}$ correspond to

each other, indicating the non-existence of fabricated or stale data. Alterations will cause $\mathcal{RS}$-$\mathcal{WS}$ mismatches, triggering alerts and making sustained undetected attacks virtually impossible.

Continuous Collision Resistance. To avoid storing all elements of the two sets, a collision-resistant hash function is employed over both sets. The hash function must ensure that if $\mathcal{WS} \neq \mathcal{RS}$, then $h_{\mathcal{WS}} \neq h_{\mathcal{RS}}$ with high probability. Blum et al. [14] prove that given a collision-resistant hash function $H(\cdot)$ for set elements, the construction of set hash $h_{\mathcal{S}} = \bigoplus_{s \in \mathcal{S}} H(s)$ is also collision-resistant. With this approach, if the hash function returns k bits, the design of write-read consistent sets detects errors with a probability $p \geq 1 - 1/2^k$. Arasu et al. [8] further show that the hash function $H(\cdot)$ could be replaced by a pseudorandom function, which is indistinguishable from random, to improve performance while maintaining the same security properties. **UEM** assumes that the employed PRF is both unpredictable and indistinguishable from random, reflecting a common choice in the security domain.

5.3 Limitations

First, as mentioned in Section 3.1, applications are responsible for ensuring their own local memory safety. It is important to note that this assumption of trustworthiness may extend to situations such as core dumps, where stored memory snapshots are sensitive and susceptible to attacks from adversaries who might exploit compromised memory tiers. Consequently, users are also responsible for securely handling dump files, including their generation, storage, or ensuring that settings like disabling core dumps are tamper-proof. Second, access patterns raised by **UEM** are non-oblivious due to the untrusted nature of interacting with heterogeneous memory. When consecutive objects are accessed, adversaries may discern the orders, locations, and bit lengths. However, the exposure of orders is confined to the initial access for multiple accesses to a single object, and discerning between reads or writes is unattainable. If heightened security is desired, one could integrate certain oblivious memory techniques [91, 102] on top of **UEM**. Third, the fundamental philosophy of **UEM** revolves around tackling encryption and verification using a centralized approach, ensuring optimal efficiency in handling fine-grained, object-level memory access. This implies that managed objects are exclusively accessible by a singular instance of the **UEM** manager, and are not conducive to extension into multi-instance or distributed scenarios, e.g., cross-process communication via shared memory, or emerging architectures like near-memory computing or Processing-In-Memory (PIM) configurations. Fourth, **UEM** emphasizes detecting data tampering but does not offer recovery solutions. If untrusted heterogeneous storage ceases to operate, it is akin to experiencing a power outage in terms of application data loss. We note that **UEM** manages runtime memory, instead of storage, meaning that the data should inherently be considered volatile. Hence, **UEM** cannot prevent Denial-of-Service (DoS) attacks. Lastly, the detection mechanism employed by **UEM** is asynchronous, making it unsuitable for applications (e.g., ATM) that cannot tolerate a verification delay. However, in most cases, adversaries cannot deliberately influence the application in the way they want via tampering since they are unable to create ciphertext that decrypts to be what they desire. Tampering detection acts more as a deterrence against adversaries and is sufficient for a wide range of scenarios, as any detection of violations would be a formal proof against adversaries, leading to a loss of reputation or lawsuits. In fact, this model has also been used in many studies [7, 8, 109].

6 EVALUATION

6.1 Experimental Setup

6.1.1 Evaluated Heterogeneous Memory Architectures. We evaluate **UEM** on three distinct platforms chosen to provide a comprehensive demonstration of the versatility of our design.

Table 1. Hardware Configurations for Each Testbed

	CPU	Remarks
TCP	Intel E5-2640v4 2 x 10 cores, 2.4 GHz	Mellanox ConnectX-4 25 GB NICs
NVM	Intel Xeon Gold 6326 2 x 32 cores, 2.9 GHz	Intel Optane Persistent 8 x 128GB DCPMM
TDX	Intel Xeon Platinum 8438C 2 x 48 cores, 2.6 GHz	TDX 2.0 enabled VM with 16 vCores

- **TCP**: A network-attached remote memory solution designed to provide applications with seamless execution, mimicking the experience of running on a single machine with abundant and dynamically adjustable memory. However, both its communication links and remote storage facilities are less controllable compared to local memory, and its centralized and shared nature amplifies the intricacy and severity of memory security vulnerabilities.
- **NVM**: As an emerging form of memory that offers greater storage density and lower cost per unit compared to DRAM, a DRAM-NVM hybrid can effectively increase the overall memory capacity of a single machine. Still, NVM is less secure than DRAM because its non-volatile nature increases the potential for physical data attacks, such as forced disassembly, theft, and interpretation.
- **TDX**: Intel Trust Domain Extensions is a cutting-edge TEE solution available in the market. It can be viewed as a virtualization technology enhancement, ensuring that the entire virtual machine operates within a hardware-enforced security perimeter. Conversely, if a heterogeneous memory architecture breaches the VM barriers, external memory cannot benefit from TEE hardware protection, rendering it vulnerable to the OS or adversaries who may exploit escalation vulnerabilities to gain privileges.

6.1.2 Hardware Environment. To assess the three heterogeneous memory architectures discussed above, we establish three corresponding testbeds, as detailed in Table 1. Specifically, for **TCP** evaluations, we utilize **UEM** atop `Shenango` [73], an adept kernel-bypassing scheduler and network stack. We deploy it on the `xl170` machines from `Cloudlab` [27] for experiments. For **NVM** and **TDX** assessments, we use self-purchased machines, where **UEM** is built directly on the Linux kernel scheduler and network stack. This deviation arises from `Shenango`'s tailored drivers for specific hardware, which are incompatible with our **NVM** and **TDX** machines. We note that for a particular testbed, the software stack keeps consistent.

6.1.3 Evaluated Workloads. In addition to varying hardware configurations, we also evaluate **UEM** with three different kinds of workload to demonstrate its generality as follows.

- **Graph Processing**: Our study utilizes the Wikipedia network of top categories sourced from the `SNAP` dataset [54]. This graph encompasses approximately 1.8M nodes and 28.5M edges and is fully connected. Our evaluation of **UEM**'s performance focuses on concurrent BFS with variable sizes of local memory.
- **Key-Value Queries**: We utilize traces of Twitter's requests as presented in [105]. Given that **UEM**'s design is intended to address the I/O amplification challenges with small objects, we standardize the key and value sizes to 50 bytes for both workloads. Our choice for the key-value store system is an in-memory hopscotch hash table implementation. We center our analysis on two distinct workloads: ❶ **Cluster 35**: Characterized as a read-intensive uniform access workload, it comprises 96% of `GET` requests and 4% of `SET` requests. This workload demonstrates a uniform access pattern (Zipf factor = 0). ❷ **Cluster 48**: This workload offers a more balanced

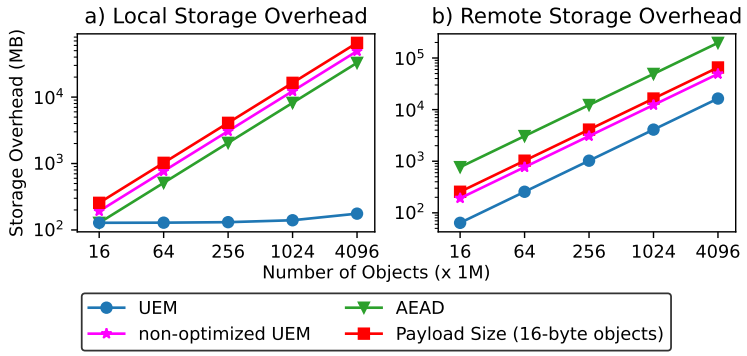

Fig. 6. Storage overhead comparison between **UEM** & **AEAD**.

read-write distribution, with 65% of `GET` requests and 35% of `SET` requests. Additionally, it manifests a more skewed access pattern (Zipf factor = 0.8191).

- **Relational Tabular Analysis**: We incorporate `DataFrame` [69], an in-memory data analysis framework mirroring the `Pandas` [66] interface, on **UEM**. Our performance tests revolve around three salient queries from `TPC-H` [21]: ❶ **Q1** involves scanning and filtering the `lineitem` table and performing extensive computations. ❷ **Q6**, similar to Q1, also scans and filters the `lineitem` table, culminating in a summation of the values from qualifying rows. ❸ **Q17** is distinguished by a join operation between the `part` and `lineitem` tables. The merge join, which requires sorting the join attribute, serves as the joining mechanism. This results in increased computational demand and requires multiple memory access rounds, especially when compared to Q1 and Q6.

6.1.4 Baselines. To our knowledge, **UEM** stands as the pioneering effort in delving into memory safety attributes within the realm of object-oriented heterogeneous memory management. As such, there are currently no direct competitors in this field. To evaluate **UEM**, we first involve **AIFM** [81] as a baseline, representing the state-of-the-art object-oriented disaggregated memory framework without considering data security. Furthermore, we implement AEAD-based encryption and verification upon **AIFM**, termed **AEAD**, as a baseline of naïve memory security mechanisms.

6.2 Mircobenchmarks

6.2.1 Storage Overhead. Fig. 6 compares the extra space (in addition to the objects themselves) occupied in both the local memory and the remote device, among **UEM**, **AEAD**, and non-optimized **UEM** (disabling grouped mask and timestamp elimination proposed in Sections 4.5.1 and 4.5.2). The reference line of payload size assumes an average size of 16 bytes per object. In terms of local memory storage, **UEM** demands approximately an additional 160 MB for around 4 billion objects. A significant portion of this space, 128 MB, is allocated to **DMP**. The remaining space is dedicated to storing mappings from group IDs to mask offsets. Notably, the incremental growth of required extra storage in **UEM** is gradual with the surge in the number of objects. This slow growth is attributed to the fact that many objects are categorized into the same group, necessitating the storage of merely a single mapping for each group. Conversely, the storage overhead of **AEAD** and non-optimized **UEM** is much larger. **AEAD** and non-optimized **UEM** have to store timestamps in both local and remote memory to guarantee data freshness. Further, non-optimized **UEM**, without grouping, needs to store the mapping from the object ids to mask offsets for every object in local memory. Clearly, the **AEAD** scheme incurs a storage overhead matching its effective data payload in the local memory. This substantially deviates from the primary objectives of most heterogeneous memory configurations, rendering AEAD-style methods impractical for such scenarios. In terms

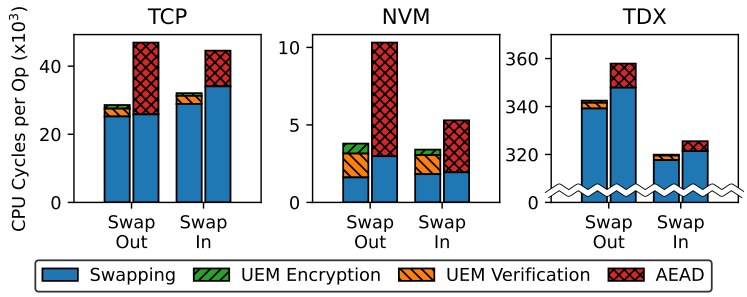

Fig. 7. Performance overhead breakdown of **UEM** & **AEAD**.

of remote device storage overhead, **UEM**'s footprint is merely 8% of that demanded by **AEAD**. Specifically, **UEM** only retains the group ID and offsets, while **AEAD** requires storage for IVs, MACs, and timestamps for each object, individually.

6.2.2 Performance Overhead Breakdown. Regarding 64-byte object swapping, we illustrate the detailed breakdown of performance overhead between various modules of **UEM** in terms of CPU cycles, in Fig. 7. We compare the results of **UEM** (the three-segment bars on the left) against **AEAD** (the two-segment bars on the right) as the strawman solution. Overall, **UEM** outperforms **AEAD** on all devices for both swap-in and swap-out operations.

On **TCP**, data swapping takes the majority of the time, approximately 88.2% for swap-out operations and 90.2% for swap-in operations. Regarding the modules introduced by **UEM**, the Data Verification Module accounts for 8.0% and 7.3% of the time for swap-out and swap-in, respectively. The primary source of this overhead is the computational process linked with the Poly-1305 [12] MAC (or PRF). The Data Encryption Module only consumes 2.5% of the cost for swap-in operations. The cost is slightly higher in swap-out operations, around 3.7%. The additional cost arises from the group allocation process for objects and the mapping management from object groups to mask offsets. **AEAD** introduces 6.3x overhead compared to **UEM** on swap-out and 3.3x overhead on swap-in for its security enforcement. The overhead results from the cryptographic computations and timestamp management for freshness. Notably, the swapping procedure of **AEAD** consumes more cycles than **UEM** since **AEAD** needs to swap a larger amount of data.

On **NVM**, since the data-swapping process becomes faster than that on **TCP**, the relative overhead of security enforcement of both **UEM** and **AEAD** becomes more significant. The time spent on **UEM**'s Data Encryption and Verification Modules accounts for 57.5% and 46.5% of the entire swapping process for swap-out and swap-in, respectively. However, it is still much faster than the strawman solution. In fact, the advantage of **UEM** against **AEAD** is more pronounced when data swapping is less prominent. **AEAD** is 171.7% slower than **UEM** for swap-out and 55% slower for swap-in.

On **TDX**, since the data swapping overhead becomes much larger, the relative overhead introduced by **UEM** becomes negligible. However, it still maintains a noticeable advantage over **AEAD** since it swaps a smaller amount of data.

6.2.3 Verification Latency. This subsection assesses the duration required to verify memory integrity and the efficacy of the proposed optimizations. Table 2 presents the latency for **UEM**'s Data Verification Module when operating on **NVM**, measured per 1GB data. **UEM** specializes in examining objects sourced from potentially compromised, untrusted memory tiers. The latency varies between 2.59s and 7.48s, contingent on object sizes. Implementing batch verification (Section 4.5.3) contributes to performance enhancement, yielding a 2% to 14% improvement. Notably, the throughput for smaller objects is comparatively lower due to the increased number of objects, necessitating more MAC (or PRF) computations. Further, we explore the synergistic benefits of integrating **DMP**

Table 2. Verification Latency for Every 1GB Data

Object Size	16B	32B	64B	128B
Enable Batching (s)	7.36	4.49	3.06	2.27
Disable Batching (s)	7.48	4.99	3.35	2.59

Table 3. **DMP** Rotation Latency for Every 1GB Data

Object Size	16B	32B	64B	128B
Combined Process (s)	9.06	6.17	4.41	3.74
Separate Processes (s)	11.69	7.50	5.32	4.33

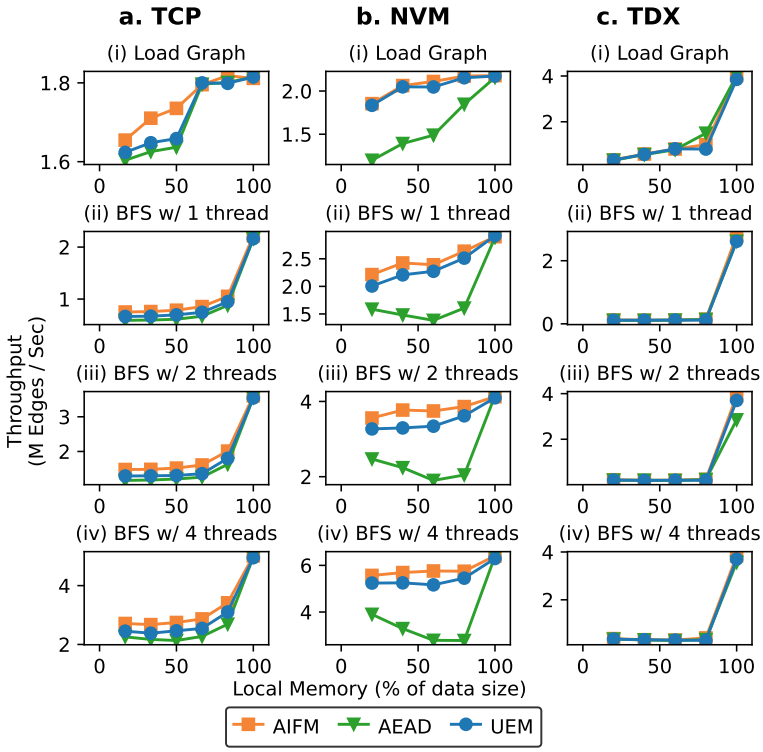

Fig. 8. Performance comparison between **UEM** and baselines on graph processing workloads.

rotation into the verification process (Section 4.5.4), with findings detailed in Table 3. Executed independently, the dual processes, for each 1GB data, incur a cumulative duration of 11.69s for 16-byte objects and 4.33s for 128-byte objects. However, amalgamating these procedures reduces the latencies to 9.06s and 3.74s, respectively. This efficiency gain is attributed to the need for only a single retrieval of data from remote memory. We note that the operation of the Data Verification Module is an asynchronous process running in the background, parallel to the application, and does not block its memory access behaviors.

6.3 Evaluation on Graph Processing

We first evaluate **UEM** on graph processing workloads, which often include a random memory access pattern. We compare our method with **AIFM**, the baseline without security enforcement, and the strawman **AEAD** approach. Fig. 8 shows the results on different devices with varying

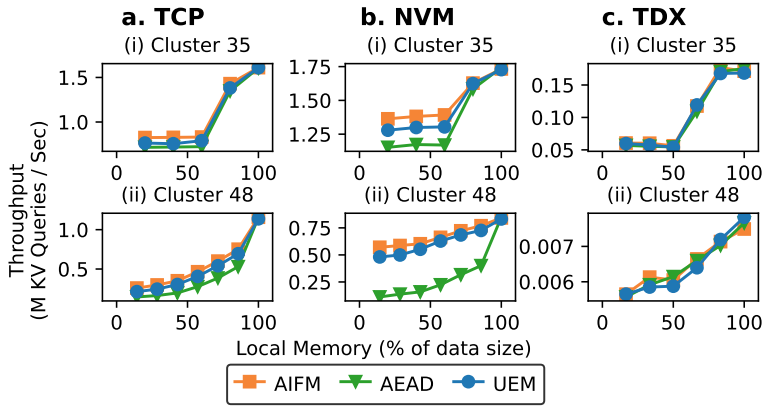

Fig. 9. Performance comparison between **UEM** and baselines on key-value queries.

local memory sizes. In most scenarios, the throughput of task processing increases when the local memory size increases. When the local memory size reaches 100% of the data size, all data can fit into the local memory, and there will be no differences among **UEM**, **AIFM**, and **AEAD**.

On all three devices, **UEM** introduces a small overhead for both graph loading and BFS. On **TCP**, **UEM**'s overhead is about 70% of what **AEAD** brings. On **NVM**, the advantage of **UEM** against **AEAD** becomes larger, because the data swapping process on **NVM** is faster. This matches the observations we have in the microbenchmarks. On **TDX**, since the data swapping process makes up most of the cost, we do not observe a significant difference among the three approaches. In other words, the overhead introduced by **UEM**'s security enforcement is negligible. We also configure our experiments with different numbers of threads, and the results show that the observations above hold regardless of the thread numbers. An intriguing observation specific to **NVM** is that **AEAD** occasionally underperforms as the local memory expands. This is due to NVM's cache-coherence property, implying that data transfers between DRAM and NVM could bypass the actual movement between tiers and instead take place within the CPU cache. We discern this phenomenon to be accentuated when the local memory pool is constrained, during traversal of real-world graphs. We retain the original results without altering the code to optimize the hit rate for this particular circumstance, for a consistent comparison.

6.4 Evaluation on Key-Value Queries

Fig. 9 shows the results on key-value queries. We observe that **UEM** introduces a relatively small overhead in all the settings. The relative overhead compared to the insecure baseline is up to 21.7%. **UEM** also beats **AEAD** in most scenarios. Similar to the graph processing workloads, the advantage of **UEM** over **AEAD** is higher on **NVM**, where the data swapping process is faster. Besides, the relative speedup for **UEM** over **AEAD** is higher in Cluster 48 than in Cluster 35. This is because Cluster 48 involves more `set` operations, while Cluster 35 has more `get` operations. As we observe in microbenchmarks, **UEM** has a more prominent advantage over **AEAD** in swap-out than in swap-in. Regarding performance on **TDX**, since data swapping is much slower and dominates the cost, **UEM**'s security features have a negligible impact on overall performance.

6.5 Evaluation on Relational Tabular Analysis

Fig. 10 shows the performance on TPC-H workloads. **UEM** introduces reasonable overhead compared to the insecure baseline. The exact overhead depends on the specific workload. For computation-intensive workloads, such as Q17 that involves `join` operations on large tables, the overhead of

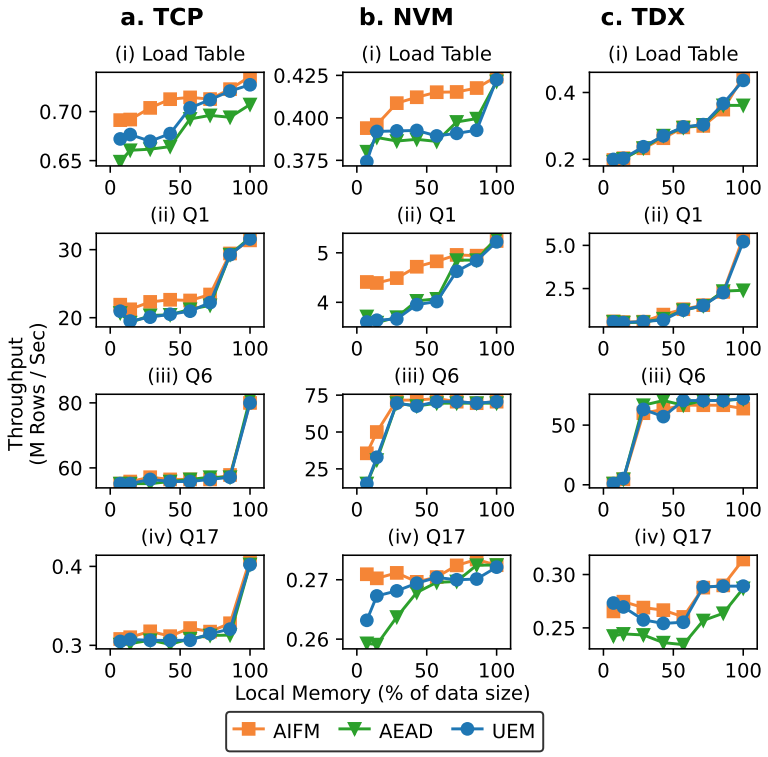

Fig. 10. Performance comparison between **UEM** and baselines on relational tabular analysis workloads.

UEM is small, up to 20k rows/s compared to **AIFM**. Most of the cost comes from the sort operator that the merge-join requires, and the difference in remote memory access does not play a crucial role in this query. On the contrary, for queries like Q1 and Q6, where table scanning makes up most of the query plan, the overhead of **UEM** becomes larger. Compared to **AEAD**, **UEM** is superior in most cases. Overall, **UEM** excels in handling both random access workloads, including graph processing, key-value queries, and table joins (Q17), as well as sequential access scenarios, such as relational tabular analysis with an emphasis on scanning (Q1 and Q6).

7 CONCLUSION

In this paper, we propose **UEM**, a novel approach for object-oriented memory management in diverse architectures. **UEM** enhances security via centralized data structures, ensuring slight computational and storage overhead. Our evaluations show that **UEM** performs consistently well across various devices and workloads, with a performance overhead below 20% in most cases compared to a baseline without considering security. In particular, **UEM** stands out as a pioneering study to enable memory security in heterogeneous memory, bridging a notable research gap. We believe that the slight performance loss is worth the enhanced data security **UEM** brings.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful comments. This work is supported by Alibaba Group through the Alibaba Research Intern Program. Yifan and Linh are partially supported by NSF grants CNS-1955670 and CNS-1750158.

REFERENCES

[1] Ahmed H. M. O. Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-Mei W. Hwu. 2019. FlatFlash: Exploiting the Byte-Accessibility of SSDs within a Unified Memory-Storage Hierarchy. In *ASPLOS*. ACM, 971–985.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote memory in the age of fast networks. In *SoCC*. ACM, 121–127.

[3] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-rationing garbage collection for hybrid memories. In *PLDI*. ACM, 62–77.

[4] Tiago Alves. 2004. Trustzone: Integrated hardware and software security. *Information Quarterly* 3 (2004), 18–24.

[5] 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 *EuroSys*. ACM, 14:1–14:16.

[6] Andrew W. Appel and Kai Li. 1991. Virtual Memory Primitives for User Programs. In *ASPLOS*. ACM Press, 96–107.

[7] Arvind Arasu, Badrish Chandramouli, Johannes Gehrke, Esha Ghosh, Donald Kossmann, Jonathan Protzenko, Ravi Ramamurthy, Tahina Ramananandro, Aseem Rastogi, Srinath Setty, et al. 2021. Fastver: Making data integrity a commodity. In *Proceedings of the 2021 International Conference on Management of Data*. 89–101.

[8] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A high concurrency key-value store with integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 251–266.

[9] Arvind Arasu, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. 2021. Integrity-based Attacks for Encrypted Databases and Implications. In *CIDR*. www.cidrdb.org.

[10] Shahram Bakhtiari, Reihaneh Safavi-Naini, Josef Pieprzyk, et al. 1995. *Cryptographic hash functions: A survey*. Technical Report. Citeseer.

[11] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing using RDMA. In *SIGMOD Conference*. ACM, 1463–1475.

[12] Daniel J Bernstein. 2005. The Poly1305-AES message-authentication code. In *International workshop on fast software encryption*. Springer, 32–49.

[13] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. 2003. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *USENIX Security Symposium*. USENIX Association.

[14] M Blum, W Evans, P Gemmell, S Kannan, and M Naor. 1991. Checking the correctness of memories. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*. IEEE, 90–99.

[15] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*. Morgan Kaufmann, 54–65.

[16] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.* 11, 11 (2018), 1604–1617.

[17] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. 2019. Project PBerry: FPGA Acceleration for Remote Memory. In *HotOS*. ACM, 127–135.

[18] 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. *CoRR* abs/2303.15540 (2023).

[19] Douglas Comer and Jim Griffioen. 1990. A New Design for Distributed Systems: The Remote Memory Model. In *USENIX Summer*. USENIX Association, 127–136.

[20] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* (2016), 86.

[21] Transaction Processing Performance Council. 2023. TPC-H. https://www.tpc.org/tpch Accessed: 2023-10-15.

[22] Crispan Cowan. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*. USENIX Association.

[23] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. 2010. Infrastructure as a service security: Challenges and solutions. In *2010 The 7th International Conference on Informatics and Systems (INFOS)*. 1–8.

[24] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *NSDI*. USENIX Association, 401–414.

[25] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP*. ACM, 54–70.

[26] Subramanya Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *EuroSys*. ACM, 15:1–15:16.

[27] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael

Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19
[28] Hiroaki Etoh and Kunikazu Yoda. 2000. Protecting from stack-smashing attacks.
[29] Mohammad Ewais and Paul Chow. 2023. Disaggregated Memory in the Datacenter: A Survey. *IEEE Access* 11 (2023), 20688–20712.
[30] Michael J. Feeley, William E. Morgan, Frédéric H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. 1995. Implementing Global Memory Management in a Workstation Cluster. In *SOSP*. ACM, 201–212.
[31] Michail Flouris and Evangelos P. Markatos. 1999. The Network RamDisk: Using remote memory on heterogeneous NOWs. *Clust. Comput.* 2, 4 (1999), 281–293.
[32] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, João Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *OSDI*. USENIX Association, 249–264.
[33] Caixin Gong, Chengjin Tian, Zhengheng Wang, Sheng Wang, Xiyu Wang, Qiulei Fu, Wu Qin, Qian Long, Rui Chen, Jiang Qi, Ruo Wang, Guoyun Zhu, Chenghu Yang, Wei Zhang, and Feifei Li. 2022. Tair-PMem: a Fully Durable Non-Volatile Memory Database. *Proc. VLDB Endow.* 15, 12 (2022), 3346–3358.
[34] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *USENIX Annual Technical Conference*. USENIX Association, 287–294.
[35] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *NSDI*. USENIX Association, 649–667.
[36] Jim Handy. 2015. Understanding the intel/micron 3d xpoint memory. *Proc. SDC* 68 (2015).
[37] Michael Henson and Stephen Taylor. 2013. Memory encryption: A survey of existing techniques. *ACM Comput. Surv.* 46, 4 (2013), 53:1–53:26.
[38] Mark D. Hill, Jon Masters, Parthasarathy Ranganathan, Paul Turner, and John L. Hennessy. 2019. On the Spectre and Meltdown Processor Security Vulnerabilities. *IEEE Micro* 39, 2 (2019), 9–19.
[39] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. 2017. Log-Structured Non-Volatile Main Memory. In *USENIX Annual Technical Conference*. USENIX Association, 703–717.
[40] Paul Hudak. 1986. A Semantic Model of Reference Counting and its Abstraction (Detailed Summary). In *LISP and Functional Programming*. ACM, 351–363.
[41] Eyal Itkin. 2020. SAFE-LINKING – ELIMINATING A 20 YEAR-OLD MALLOC() EXPLOIT PRIMITIVE. https://research.checkpoint.com/2020/safe-linking-eliminating-a-20-year-old-malloc-exploit-primitive/ Accessed: 2023-10-15.
[42] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*. IEEE, 1–4.
[43] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *USENIX Annual Technical Conference*. USENIX Association, 437–450.
[44] Sooyong Kang, Sungmin Park, Hoyoung Jung, Hyoki Shim, and Jaehyuk Cha. 2009. Performance Trade-Offs in Using NVRAM Write Buffer for Flash Memory-Based Storage Devices. *IEEE Trans. Computers* 58, 6 (2009), 744–758.
[45] Jonathan Katz and Yehuda Lindell. 2007. *Introduction to modern cryptography: principles and protocols*. Chapman and hall/CRC.
[46] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. 2015. Turning Centralized Coherence and Distributed Critical-Section Execution on their Head: A New Approach for Scalable Distributed Shared Memory. In *HPDC*. ACM, 3–14.
[47] Hormuzd Khosravi. 2022. Runtime Encryption of Memory With Intel Total Memory Encryption - Multi-Key. *Intel, White Paper* (2022).
[48] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*. IEEE Computer Society, 361–372.
[49] Vamsee Reddy Kommareddy, Jagadish Kotra, Clayton Hughes, Simon David Hammond, and Amro Awad. 2020. PreFAM: Understanding the Impact of Prefetching in Fabric-Attached Memory Architectures. In *MEMSYS*. ACM, 323–334.
[50] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. 2022. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. In *CIDR*. www.cidrdb.org.
[51] Samir Koussih, Anurag Acharya, and Sanjeev Setia. 1999. Dodo: A User-level System for Exploiting Idle Memory in Workstation Clusters. In *HPDC*. IEEE Computer Society, 301–308.
[52] Robert Lasch, Thomas Legler, Norman May, Bernhard Scheirle, and Kai-Uwe Sattler. 2022. Cost Modelling for Optimal Data Placement in Heterogeneous Main Memory. *Proc. VLDB Endow.* 15, 11 (2022), 2867–2880.

[53] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. 2022. Hydra : Resilient and Highly Available Remote Memory. In *FAST*. USENIX Association, 181–198.
[54] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[55] Derek Leung, Yossi Gilad, Sergey Gorbunov, Leonid Reyzin, and Nickolai Zeldovich. 2022. Aardvark: An Asynchronous Authenticated Dictionary with Applications to Account-based Cryptocurrencies. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. 4237–4254.
[56] 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 *ASPLOS (2)*. ACM, 574–587.
[57] Mingyu Li, Xuyang Zhao, Le Chen, Cheng Tan, Huorong Li, Sheng Wang, Zeyu Mi, Yubin Xia, Feifei Li, and Haibo Chen. 2023. Encrypted Databases Made Secure Yet Maintainable. In *OSDI*. USENIX Association, 117–133.
[58] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. 2005. Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device. In *CLUSTER*. IEEE Computer Society, 1–10.
[59] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. 2008. Predictable programming on a precision timed architecture. In *CASES*. ACM, 137–146.
[60] Google LLC. 2023. gVisor. https://github.com/google/gvisor Accessed: 2023-10-15.
[61] Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D. Bond, Stephen M. Blackburn, Miryung Kim, and Guoqing Harry Xu. 2022. Mako: a low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *PLDI*. ACM, 92–107.
[62] Sunilkumar S. Manvi and Gopal Krishna Shyam. 2014. Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey. *J. Netw. Comput. Appl.* 41 (2014), 424–440.
[63] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *USENIX Annual Technical Conference*. USENIX Association, 843–857.
[64] Hasan Al Maruf and Mosharaf Chowdhury. 2023. Memory Disaggregation: Advances and Open Challenges. *ACM SIGOPS Oper. Syst. Rev.* 57, 1 (2023), 29–37.
[65] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *ASPLOS (3)*. ACM, 742–755.
[66] Wes McKinney et al. 2011. pandas: a foundational Python library for data analysis and statistics. *Python for high performance and scientific computing* 14, 9 (2011), 1–9.
[67] Ines Messadi, Shivananda Neumann, Nico Weichbrodt, Lennart Almstedt, Mohammad Mahhouk, and Rüdiger Kapitza. 2021. Precursor: A fast, client-centric and trusted key-value store using rdma and intel sgx. In *Proceedings of the 22nd International Middleware Conference*. 1–13.
[68] Ethan Miller, Achilles Benetopoulos, George Neville-Neil, Pankaj Mehra, and Daniel Bittman. 2023. Pointers in Far Memory: A rethink of how data and computations should be organized. *Queue* 21, 3 (2023), 75–93.
[69] Hossein Moein. 2023. DataFrame. https://github.com/hosseinmoein/DataFrame Accessed: 2023-10-15.
[70] Muhammad Faheem Mushtaq, Sapiee Jamel, Abdulkadir Hassan Disina, Zahraddeen A Pindar, Nur Shafinaz Ahmad Shakir, and Mustafa Mat Deris. 2017. A survey on the cryptographic encryption algorithms. *International Journal of Advanced Computer Science and Applications* 8, 11 (2017).
[71] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *USENIX Annual Technical Conference*. USENIX Association, 291–305.
[72] 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*. 238–253.
[73] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378.
[74] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. 2013. Storage Management in the NVRAM Era. *Proc. VLDB Endow.* 7, 2 (2013), 121–132.
[75] Constantin Pohl and Kai-Uwe Sattler. 2018. Joins in a heterogeneous memory hierarchy: exploiting high-bandwidth memory. In *DaMoN*. ACM, 8:1–8:10.
[76] Tony Printezis and David Detlefs. 2000. A Generational Mostly-Concurrent Garbage Collector. In *ISMM*. ACM, 143–154.
[77] Kiran Puttaswamy and Gabriel H. Loh. 2005. Implementing Caches in a 3D Technology for High Performance Processors. In *ICCD*. IEEE Computer Society, 525–532.

[78] Yifan Qiao, Xubin Chen, Jingpeng Hao, Tong Zhang, Changsheng Xie, and Fei Wu. 2020. Architecting Heterogeneous Memory Systems with DRAM Technology Only: A Case Study on Relational Database. In *MCHPC@SC*. IEEE, 25–33.
[79] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *SOSP*. ACM, 392–407.
[80] Vincent Rijmen. 2000. Efficient Implementation of the Rijndael S-box. *Katholieke Universiteit Leuven, Dept. ESAT. Belgium* (2000).
[81] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *OSDI*. USENIX Association, 315–332.
[82] Stephen M. Rumble, Ankita Kejriwal, and John K. Ousterhout. 2014. Log-structured memory for DRAM-based storage. In *FAST*. USENIX, 1–16.
[83] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted Execution Environment: What It is, and What It is Not. In *TrustCom/BigDataSE/ISPA (1)*. IEEE, 57–64.
[84] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 38–54.
[85] Mo Sha, Jialin Li, Sheng Wang, Feifei Li, and Kian-Lee Tan. 2023. TEE-based General-purpose Computational Backend for Secure Delegated Data Processing. *Proc. ACM Manag. Data* 1, 4 (2023), 263:1–263:28.
[86] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *CCS*. ACM, 298–307.
[87] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed shared persistent memory. In *SoCC*. ACM, 323–337.
[88] Claude E Shannon. 1949. Communication theory of secrecy systems. *The Bell system technical journal* 28, 4 (1949), 656–715.
[89] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2023. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *FAST*. USENIX Association, 81–98.
[90] Rui Shu, Peipei Wang, Sigmund Albert Gorski III, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. 2016. A Study of Security Isolation Techniques. *ACM Comput. Surv.* 49, 3 (2016), 50:1–50:37.
[91] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*. ACM, 299–310.
[92] Adam J. Storm, Christian Garcia-Arellano, Sam Lightstone, Yixin Diao, and Maheswaran Surendra. 2006. Adaptive Self-tuning Memory in DB2. In *VLDB*. ACM, 1081–1092.
[93] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building Enclave-Native Storage Engines for Practical Encrypted Databases. *Proc. VLDB Endow.* 14, 6 (2021), 1019–1032.
[94] Michael Szydlo. 2004. Merkle Tree Traversal in Log Space and Time. In *EUROCRYPT (Lecture Notes in Computer Science, Vol. 3027)*. Springer, 541–554.
[95] Rik van Riel. 2001. Page Replacement in Linux 2.4 Memory Management. In *USENIX Annual Technical Conference, FREENIX Track*. USENIX, 165–172.
[96] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. 2015. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *USENIX Security Symposium*. USENIX Association, 913–928.
[97] Chenxi Wang, Ting Cao, John N. Zigman, Fang Lv, Yunquan Zhang, and Xiaobing Feng. 2016. Efficient Management for Hybrid Memory in Managed Language Runtime. In *NPC (Lecture Notes in Computer Science, Vol. 9966)*. 29–42.
[98] Chenxi Wang, Huimin Cui, Ting Cao, John N. Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: holistic memory management for big data processing over hybrid memories. In *PLDI*. ACM, 347–362.
[99] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 2022. MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime. In *OSDI*. USENIX Association, 35–53.
[100] Jing Wang, Chao Li, Taolei Wang, Lu Zhang, Pengyu Wang, Junyi Mei, and Minyi Guo. 2022. Excavating the Potential of Graph Workload on RDMA-based Far Memory Architecture. In *IPDPS*. IEEE, 1029–1039.
[101] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, Xuntao Cheng, Xiaolong Xie, and Yu Zou. 2022. Operon: An Encrypted Database for Ownership-Preserving Data Management. *Proc. VLDB Endow.* 15, 12 (2022), 3332–3345.
[102] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *CCS*. ACM, 850–861.
[103] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. 1995. Dynamic Storage Allocation: A Survey and Critical Review. In *IWMM (Lecture Notes in Computer Science, Vol. 986)*. Springer, 1–116.

[104] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *FAST*. USENIX Association, 323–338.
[105] Juncheng Yang, Yao Yue, and KV Rashmi. 2021. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–35.
[106] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native Client: a sandbox for portable, untrusted x86 native code. *Commun. ACM* 53, 1 (2010), 91–99.
[107] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. 2023. DiLOS: Do Not Trade Compatibility for Performance in Memory Disaggregation. In *EuroSys*. ACM, 266–282.
[108] Daniel Zahka and Ada Gavrilovska. 2022. FAM-Graph: Graph Analytics on Disaggregated Memory. In *IPDPS*. IEEE, 81–92.
[109] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. VeriDB: An SGX-based Verifiable Database. In *SIGMOD Conference*. ACM, 2182–2194.
[110] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *OSDI*. USENIX Association, 55–71.

Received October 2023; revised January 2024; accepted February 2024