Computers, Materials & Continua

Tech Science Press

https://doi.org/10.32604/cmc.2026.081338

ARTICLE

Supporting Memory Safety with a Security-Enhanced Memory Controller

Gen Xu*, Li Lv and Jiayan Dong

Ningbo Institute of Materials Technology and Engineering, Chinese Academy of Sciences, Ningbo, China

*Corresponding Author: Gen Xu. Email: xugen@nimte.ac.cn

Received: 28 February 2026; Accepted: 07 July 2026; Published: 13 August 2026

ABSTRACT: Memory-unsafe languages such as C and C++ remain widely used because they provide low-level control and high performance, but they remain vulnerable to spatial and temporal memory-safety violations such as out-of-bounds accesses, buffer overflows, and use-after-free errors. Prior hardware-assisted defenses reduce software overhead, yet many still rely on CPU-side metadata checks that add latency to the critical path and often miss DMA-originated accesses. We show that metadata-access cost is not dominated solely by DRAM latency: a substantial portion of the delay comes from on-chip traversal and cache-related processing. Motivated by this result, we propose SerMC, a memory-controller-based tripwire mechanism that validates accesses when metadata arrives from Dynamic Random Access Memory (DRAM) and extends enforcement to DRAM-bound accesses issued by both processors and DMA-capable devices. SerMC keeps metadata checks off the CPU critical path while preserving compatibility with existing C/C++ programs. The design targets spatial and temporal violations that cross tripwire-protected DRAM regions; non-linear pointer corruption that avoids such regions and microarchitectural side channels remain outside the scope of the current design. Our evaluation on selected SPEC CPU workloads shows 9.25% average performance overhead, while the most memory-intensive workloads incur slowdowns of up to 50%. These results indicate that SerMC provides practical average-case overhead but still exposes a clear worst-case tradeoff when metadata traffic competes with demand memory requests.

KEYWORDS: Memory safety; cyber security; memory vulnerability exploitation; tripwire-based defense

1 Introduction

The growing complexity of modern software systems, combined with the widespread adoption of large-scale open-source frameworks, has significantly lowered the barrier to discovering and exploiting security vulnerabilities. Empirical evidence shows that memory safety violations constitute the dominant root cause of real-world exploits, accounting for the majority of reported security incidents [1]. A particularly concerning class of attacks occurs within a single process that serves multiple clients in the same address space, where corrupting pointer values can directly expose sensitive data belonging to other users. This risk is further amplified in widely deployed open-source libraries such as OpenSSL [2], where vulnerabilities, once disclosed, can be rapidly weaponized at scale.

Memory-safe programming languages, such as Rust, address many of these issues primarily through compile-time ownership and borrowing rules, complemented by runtime checks such as bounds checks for indexed accesses. These mechanisms prevent many classes of memory-safety violations without requiring the programmer to manually manage object lifetimes.

In this paper, we focus on spatial boundary violations and temporal dangling-pointer accesses that eventually manifest as DRAM-bound memory requests, including heap overflows, buffer underflows, out-of-bounds reads/writes, use-after-free accesses, and stack corruption when control-sensitive stack objects are protected with tripwires. The intended security property is therefore boundary-crossing detection rather than complete semantic pointer integrity. SerMC detects a violation when the resulting memory request reaches a tripwire-protected DRAM region, without requiring intrusive changes to the processor pipeline or application source code. Consequently, non-linear pointer corruption that redirects an access to a structurally valid but semantically unauthorized object may remain undetected if no tripwire boundary is crossed. Microarchitectural side channels that expose controller-resident state, such as TwLB entries or affine metadata parameters, require complementary defenses and are outside the scope of the present work.

To balance security and efficiency, recent research has explored architectural support for enforcing memory safety with lower overhead. Prior proposals encode bounds or object states directly in memory using auxiliary metadata, such as ECC-based encodings [3]. More recent designs employ tripwire-based mechanisms, where special sentinel values are inserted around allocated objects and any access to these values signals a violation [4,5].

However, the majority of existing defenses rely on additional metadata management to perform security checks [6–9]. To validate a memory access, the corresponding metadata must be retrieved and processed, which introduces additional memory requests. While previous work largely attributes this overhead to DRAM latency, a closer examination reveals a different bottleneck [9,10]. By decomposing metadata access latency into DRAM delay, on-chip traversal, and cache-related delay, we observe that a substantial fraction—nearly half—of the total latency originates from on-die processing rather than off-chip memory access. This insight suggests that optimizations focusing solely on reducing DRAM accesses are insufficient to achieve low-overhead memory safety.

Based on this observation, we propose SerMC, a memory-controller-based memory safety mechanism that enforces tripwire checks when metadata arrives from DRAM. Instead of routing metadata through CPU caches and on-chip interconnects, SerMC directly issues metadata requests to DRAM and performs tripwire detection within the memory controller. This design removes CPU-side on-chip traversal and cache interaction from the metadata critical path, reducing validation latency. Furthermore, placing enforcement at the memory controller extends checking to DRAM-bound access paths, including requests initiated by DMA-capable devices.

SerMC enforces spatial safety by placing tripwires at both boundaries of protected objects, ensuring that any buffer overflow or underflow triggers an exception. For temporal safety, freed memory regions are similarly marked as tripwires, preventing stale pointers from accessing reclaimed objects. This design leverages the memory controller's role as the gateway to physical memory, enabling early detection of violations while remaining agnostic to the source of the memory request.

We evaluate SerMC using twenty selected SPEC CPU workloads with reference inputs. Relative to CPU-centric approaches, SerMC offers a different tradeoff: it broadens coverage to DRAM-bound DMA accesses while incurring 9.25% average overhead across the evaluated workloads. The same results show that this average hides strong workload dependence: memory-intensive applications remain the hardest case, with worst-case slowdown reaching 50%.

Contributions

In summary, this work makes the following contributions:

1. We quantify the metadata-access path and show that on-chip traversal and cache-related processing account for a substantial portion of metadata latency, motivating validation closer to the DRAM interface rather than only optimizing DRAM access latency.
2. We propose SerMC, a memory-controller-based tripwire scheme that performs validation off the CPU critical path and extends enforcement to DRAM-bound requests from both CPU cores and DMA-capable devices.
3. We implement SerMC in gem5 and evaluate it on selected SPEC CPU workloads. The results show 9.25% average overhead, identify memory-intensive workloads as the main worst-case bottleneck with slowdowns of up to 50%, and clarify which attack classes are directly detected by the current tripwire-based design.

2 Background

2.1 Memory Corruption Attack

The root cause of memory corruption and information disclosure vulnerabilities lies in the widespread use of memory-unsafe and weakly typed programming languages such as C and C++. In these languages, memory accesses are largely unchecked, allowing subtle programming errors to directly translate into exploitable security flaws. For instance, the lack of strict bounds enforcement can lead to out-of-bounds memory accesses, while erroneous pointer arithmetic or unsafe dereferencing may inadvertently expose sensitive memory contents to attackers.

The exploitation of memory corruption and information disclosure vulnerabilities has consequently evolved into a persistent arms race between attackers and defenders, as Fig. 1 shows. Early exploitation techniques injected malicious data into a victim's address space and hijacked program execution to run attacker-controlled instructions [11]. To counter such threats, modern systems have widely deployed non-executable memory protections (e.g., NX [12]), which prevent code execution from data regions. In response, attackers shifted toward reusing existing executable code already present in the process memory. To further mitigate code-reuse attacks, systems subsequently introduced defenses such as address space layout randomization (ASLR) [13] and control-flow integrity (CFI) [14]. However, information disclosure vulnerabilities have repeatedly enabled attackers to bypass ASLR, while practical constraints and compatibility requirements have limited the effectiveness of deployed CFI mechanisms [15–17].

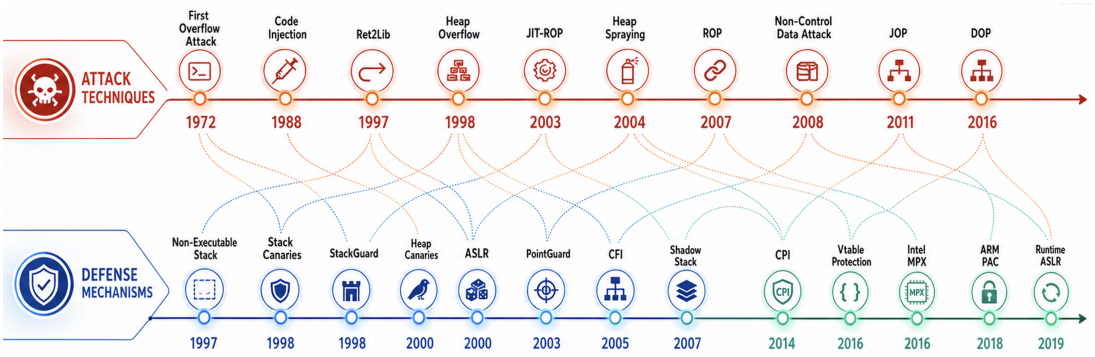

Figure 1: The timeline of memory vulnerability exploitation attack defense.

Code-reuse attacks construct malicious behaviors by chaining together short instruction sequences—commonly referred to as gadgets—that typically end with control-transfer instructions such as returns [18]. By carefully orchestrating these gadgets with crafted inputs, attackers can synthesize complex, and even

Turing-complete, payloads without injecting new code. Beyond control-flow manipulation, adversaries may also target non-control data to influence program behavior while preserving its legitimate execution paths [19]. These so-called data-only attacks repeatedly corrupt critical program variables to achieve malicious goals without violating control-flow integrity, making them particularly difficult to detect and defend against. Consequently, recent research has increasingly focused on practical memory-safety enforcement mechanisms that can provide strong protection for memory-unsafe languages while maintaining compatibility and low runtime overhead, such as software-based memory-safe execution frameworks [20].

2.2 Tripwire-Based Defense

To illustrate how tripwire-based mechanisms can prevent attackers from exploiting memory errors to tamper with application data, we begin with a representative buffer overflow vulnerability disclosed in a security-hardened operating system. Specifically, CVE-2022-27882 [21] in OpenBSD's slaacd daemon demonstrates how an integer signedness error can lead to a heap-based buffer overflow even in systems designed with strong security principles. A crafted IPv6 router advertisement can trigger incorrect length handling, allowing writes to extend beyond the allocated heap buffer. As a result, memory locations adjacent to the buffer—such as security-critical variables referenced by nearby pointers—can be overwritten, enabling malicious state manipulation.

A common strategy to defend against such exploitation is to isolate sensitive objects by surrounding them with deliberately invalid memory regions, commonly referred to as tripwires. These regions are marked such that any read or write access triggers a fault, immediately signaling a memory safety violation. Tripwires placed at both boundaries of the vulnerable buffer ensure that any overflow or underflow is detected as soon as it crosses the object's legitimate extent.

Recent work has adopted this principle to build more systematic memory safety defenses. For example, REST introduces tripwires by augmenting memory allocations with special sentinel values that enable both spatial and temporal safety guarantees. Subsequent refinements further reduce overhead by embedding tripwire-related metadata into otherwise unused gaps within program data layouts. This approach, exemplified by Caliform, achieves fine-grained memory protection while minimizing additional memory consumption.

3 Motivation

A large class of memory safety mechanisms enforce security policies by consulting auxiliary metadata during memory accesses. For example, bounds-checking schemes require metadata to describe the valid range associated with each pointer. While effective, these designs inevitably introduce additional memory requests, which can substantially degrade performance. Prior work has attempted to alleviate this overhead by co-locating metadata with program data, using fat pointers or embedding metadata into unused memory regions, thereby shortening metadata access paths. More recent work has continued to improve the efficiency of metadata-based memory safety through optimized metadata organizations and reduced metadata access overhead, as exemplified by GiantSan [22]. However, such approaches do not fundamentally eliminate metadata access latency at runtime.

An alternative strategy is to reduce the latency of metadata accesses themselves. To understand where this latency originates, we decompose a metadata request that misses in the last-level cache into two components: (1) the time from request issuance at the core to the DRAM interface, and (2) the DRAM access and data return latency. Fig. 2 presents this breakdown for selected SPEC CPU workloads with multiple reference inputs. The result shows that metadata latency is not dominated solely by off-chip DRAM service:

in many workloads, close to half of the total miss latency is spent on on-chip traversal before the request reaches DRAM.

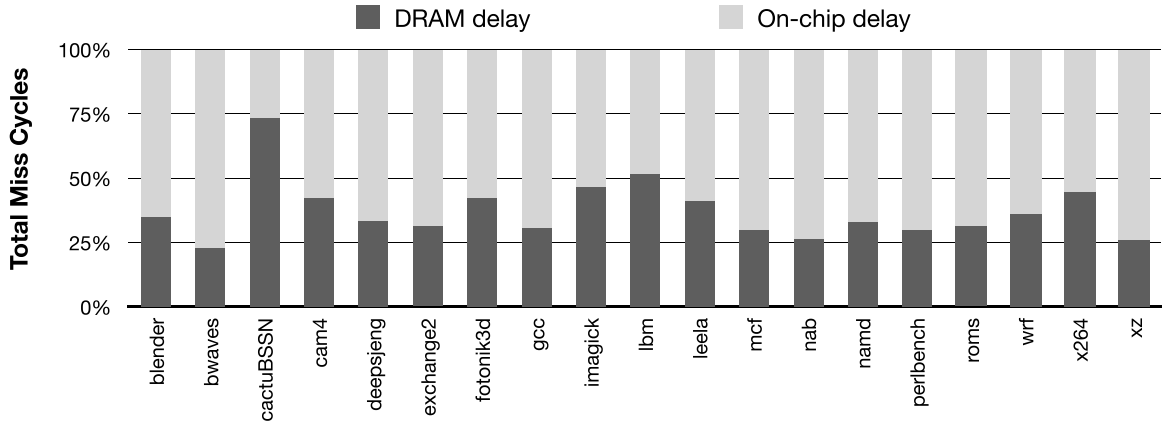

Figure 2: Breakdown of average metadata access latency into on-chip delay and DRAM delay.

This result motivates relocating tripwire validation away from the CPU and into the memory controller. By performing validation as soon as metadata arrives from DRAM, SerMC removes CPU-side traversal and cache interaction from the metadata critical path. This does not eliminate metadata bandwidth cost, but it directly addresses the latency component exposed by Fig. 2 and reduces cache pollution caused by metadata traffic.

Beyond performance concerns, existing tripwire-based memory safety mechanisms provide incomplete protection across some DRAM-bound access paths. Most prior solutions accelerate checks using tightly integrated CPU-side structures, such as cache or pipeline extensions. As a result, memory operations that bypass these structures—including non-temporal or uncacheable operations (e.g., MOVNTDQ on x86 or DCBZ on IBM POWER) and DMA-originated requests, may avoid CPU-side checking. This limitation motivates placing the tripwire monitor at the memory controller, where DRAM-bound traffic from multiple request sources converges.

Taken together, these findings lead to two design insights: (1) CPU-centric tripwire checks can miss DRAM-bound requests that bypass CPU-side structures, and (2) metadata-access overhead includes a substantial on-chip traversal component in addition to DRAM service time. SerMC addresses these two issues by moving validation to the memory controller, while still acknowledging that metadata bandwidth remains a cost for memory-intensive workloads.

4 Threat Model

Relative to prior work on memory-error-based attacks and defenses, we adopt a threat model that is both stronger and representative of real-world systems. We consider an adversary capable of causing arbitrary reads or writes within a victim process's address space through two access paths: conventional CPU load/store instructions and DMA-capable peripherals. The target application is assumed to contain one or more exploitable memory vulnerabilities that allow the attacker to repeatedly corrupt memory state without immediately terminating the program. The attacks of interest include spatial boundary violations, such as heap overflows, stack smashing, and out-of-bounds accesses, as well as temporal violations, such as use-after-free, when the resulting access reaches a tripwire-protected DRAM region.

A representative attack proceeds as follows. First, the adversary supplies crafted input to a vulnerable C/C++ program and triggers a memory error. In a spatial attack, the program issues a load or store that crosses the legitimate boundary of a protected object and reaches a tripwire-marked memory region. In a temporal attack, the program dereferences a stale pointer after the corresponding object has been freed and marked as protected. A similar threat can arise from a DMA-capable device: a compromised peripheral may issue writes to a victim buffer and then intentionally extend those writes beyond the legitimate boundary into adjacent protected memory. SerMC detects these attacks only when the resulting CPU or DMA request reaches a tripwire-protected DRAM region. If a corrupted pointer is redirected to another valid object without crossing a tripwire, the access is outside the direct detection capability of the current design.

We further assume a well-informed adversary who has access to the program's source code or binaries, as is common for widely distributed software. This enables the attacker to identify useful program artifacts, such as code-reuse gadgets, without relying on information leaks at runtime. the underlying hardware platform, operating system, kernel, and SerMC configuration interface are assumed to be part of the trusted computing base, and that kernel memory is not directly accessible to the attacker. SerMC focuses on protecting DRAM-resident data against active memory corruption and disclosure attempts that cross tripwire-protected regions. It does not claim complete protection against semantically valid but malicious pointer manipulations that never cross a tripwire boundary. Attacks exploiting microarchitectural side channels or other information leakage mechanisms are orthogonal to this work and remain outside the scope of the present design. In particular, TwLB state, affine metadata offsets, and other controller-resident bookkeeping structures are treated as trusted hardware state.

5 Design of SerMC

5.1 Design Challenges

Designing a practical tripwire-based memory safety mechanism requires reconciling strong security guarantees with stringent performance constraints. Unlike static protection schemes, the number and distribution of tripwires vary significantly across applications and execution phases. Some workloads require only a few hundred tripwires, while others dynamically deploy thousands. This variability imposes three fundamental design constraints on the protection mechanism.

First, tripwire membership queries must scale efficiently with the number of protected regions. A naive approach that compares a memory address against all registered tripwires incurs prohibitive overhead as such tables grows. Conventional lookup structures, such as flat look-aside tables or TLB-like caches, are ill-suited for this workload because security violations are rare events. Consequently, such structures suffer from poor locality and frequent misses, leading to expensive table walk-throughs on the critical path.

Second, address validation latency must be tightly bounded. Long-latency security checks not only degrade performance by stalling memory pipelines, but can also introduce security risks. In in-order pipelines, delayed validation reduces memory controller throughput, while in out-of-order pipelines, prolonged validation windows may allow transient exposure of sensitive data before a violation is detected.

Third, security enforcement must avoid perturbing the regular load/store execution path. Even small increases in load/store queue latency can have an outsized impact on overall system performance, particularly for memory-intensive workloads. Address-based checks that serialize or delay load/store operations therefore risk negating the benefits of hardware-assisted protection.

5.2 SerMC Design Principles

Guided by these constraints, we propose SerMC, a security-enhanced memory controller that performs tripwire-based enforcement off the CPU's critical path. SerMC is designed around three core principles: scalable tripwire lookup, low-latency validation, and minimal interference with normal memory traffic.

First, SerMC groups tripwire state at page granularity inside a Tripwire Look-aside Buffer (TwLB), so that lookup cost scales with realistic memory locality rather than with the total number of guarded objects.

Second, SerMC overlaps validation with ordinary memory service by clustering requests that target nearby physical regions, increasing effective TwLB reuse and reducing the visible cost of metadata lookup.

Third, SerMC uses a non-blocking controller pipeline so that one metadata miss does not serialize unrelated accesses. The detailed structures that realize these principles are described in Section 6.

5.3 Design Overview

Fig. 3 illustrates the overall system architecture incorporating SerMC. The proposed protection mechanism is integrated directly into the memory controller, which serves as the gateway to DRAM. SerMC extends the conventional controller with a high-throughput tripwire detection engine that allows trusted software to blacklist selected memory regions and block accesses to those regions. Building on this primitive, SerMC detects spatial and temporal violations when out-of-bounds or use-after-free accesses cross tripwire-protected DRAM regions before the targeted line is modified or returned. The detailed hardware mechanisms and security policies are described in Sections 6 and 7.

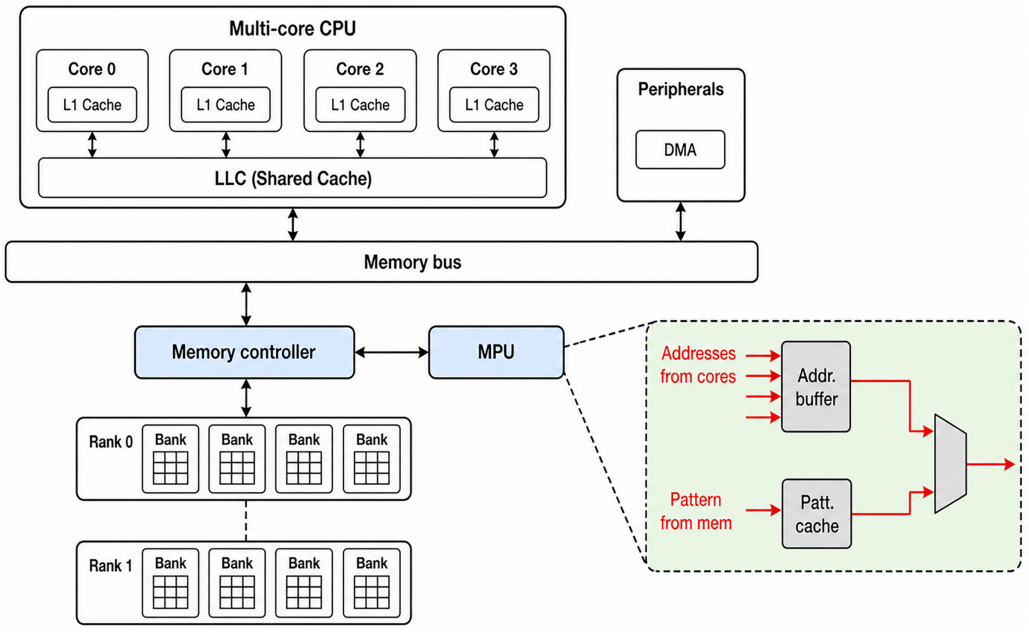

Figure 3: Design overview of SerMC.

At a high level, each memory request reaches SerMC after physical address translation. The request address is first inserted into the address buffer, optionally reordered to improve locality, and then checked against the TwLB. On a TwLB hit, SerMC immediately tests the bit corresponding to the addressed cache line. On a miss, the controller fetches the required tripwire vector from shadow metadata memory, fills the TwLB, replays the check, and then either forwards the request to DRAM or raises an exception if the targeted line is

marked as a tripwire. This explicit request flow separates software-managed configuration from per-access enforcement and clarifies how SerMC detects violations before data are committed to DRAM.

6 Implementation of SerMC

The Enhanced Memory Controller (SerMC) is designed as a lightweight architectural extension that provides a generic address validation primitive for enforcing tripwire-based memory safety policies. Rather than embedding policy-specific logic into the processor core, SerMC exposes a minimal yet expressive hardware interface that upper-layer software can leverage to implement spatial and temporal safety guarantees. Fig. 4 illustrates the high-level organization of SerMC within a conventional memory controller.

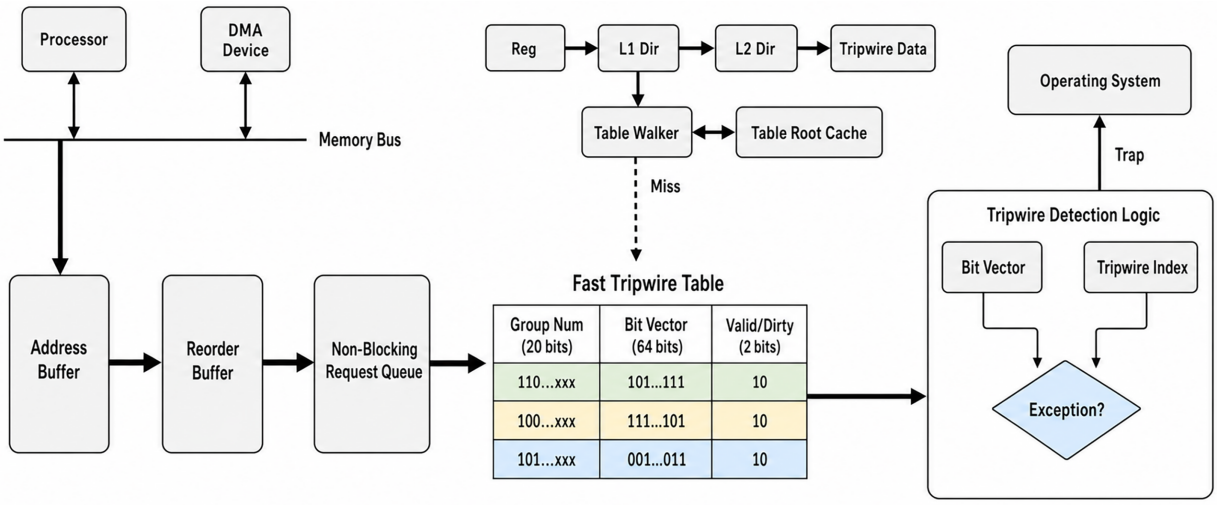

Figure 4: SerEMC hardware extension.

At a high level, SerMC operates asynchronously alongside normal memory servicing. Incoming memory requests are intercepted at the memory controller, where their physical addresses are validated against a set of registered tripwires before the requests are allowed to access DRAM. To achieve both high throughput and low latency, SerMC decomposes address validation into three cooperating subsystems: (1) a front-end address management pipeline, (2) a cached tripwire metadata lookup engine, and (3) a fast address-level violation detector, as Fig. 4 shows.

6.1 Hardware Extension

6.1.1 Fast Tripwire Table

Tripwire-based defenses protect programs by designating certain memory locations as invalid and triggering exceptions when these locations are accessed. Existing tripwire mechanisms generally fall into two categories. Content-based approaches embed special marker values directly into memory, whereas address-based approaches associate metadata with memory regions to indicate whether an access is permitted. While content-based schemes avoid metadata propagation, they are vulnerable to false positives when marker values collide with legitimate data.

SerMC adopts an address-based representation to guarantee precise enforcement with zero false positives—an essential requirement for real-world deployment. Physical memory is divided into fixed-size frames, each further partitioned at cache-line granularity. For each frame, SerMC maintains a compact bit vector that records whether individual cache lines are marked as tripwires. In the current design, a 1 KB

physical frame containing eight 128 B cache lines is associated with an 8-bit tripwire vector, resulting in negligible metadata overhead.

To balance metadata access latency and storage efficiency, SerMC stores tripwire metadata in a directly mapped region of physical memory rather than in a multi-level structure. Given a physical address, the corresponding tripwire metadata address is computed using a simple affine transformation based on a system-wide offset determined at boot time. This design avoids costly metadata page walks and ensures predictable access latency.

6.1.2 Tripwire Detection Logic

To further reduce metadata access overhead, SerMC integrates a Tripwire Look-aside Buffer (TwLB) within the memory controller. The TwLB caches recently accessed tripwire vectors, exploiting spatial and temporal locality in physical memory accesses. Each TwLB entry corresponds to a large physical region (a 128 KB "huge page") and contains both the frame identifier and a bit vector encoding the tripwire status of all cache lines within that region. The current implementation uses a 32-entry, fully associative circular buffer.

Tripwire detection is implemented as a simple combinational check. Upon receiving a memory request, SerMC extracts the frame identifier and cache-line offset from the physical address. If the frame identifier hits in the TwLB, the corresponding bit vector is retrieved immediately. The cache-line offset indexes into the vector, and if the selected bit is set, SerMC raises a hardware exception before the request reaches DRAM. On a TwLB miss, the corresponding tripwire vector is fetched from memory and inserted into the buffer.

By placing this logic within the memory controller, SerMC applies the same tripwire validation to DRAM-bound requests from CPU cores and DMA-capable devices, provided that those requests reach the protected memory-controller path.

Validation procedure. For clarity, the controller handles each access as follows:

1. Receive the physical address and request type from the memory-controller front end.
2. Derive the frame identifier and cache-line offset used for tripwire lookup.
3. Probe the TwLB for the cached tripwire vector corresponding to that frame or region.
4. On a miss, issue a metadata fetch, install the returned vector in the TwLB, and replay the validation step.
5. Test the bit selected by the cache-line offset. If the bit is clear, allow the request to proceed; if the bit is set, block the request and raise an exception before the protected line is modified or returned.

The bit-vector encoding also makes the metadata cost explicit. Because each 1 KB physical frame is represented by an 8-bit tripwire vector, the raw shadow metadata footprint is 1 byte per 1024 bytes of protected memory, or approximately 0.1%, excluding the small controller-resident TwLB and request-tracking structures.

6.1.3 Address Management and Latency Hiding

A naive implementation would perform tripwire lookup and validation strictly in arrival order. However, this approach leads to poor TwLB utilization and exposes metadata access latency directly on the critical path. SerMC therefore introduces a front-end address management pipeline that improves locality and overlaps long-latency operations.

Address Buffer. Incoming physical addresses are first placed into an address buffer capable of holding up to 128 pending requests. This buffer absorbs burst traffic and provides flexibility for downstream scheduling. For compute-intensive workloads with sparse memory accesses, buffering allows sufficient addresses to

accumulate for effective reordering. For memory-intensive workloads, the buffer provides back pressure when downstream resources are saturated.

Reorder Buffer. SerMC observes that tripwire checks for different memory requests are independent and do not impose ordering constraints. Leveraging this property, SerMC reorders buffered addresses to group requests targeting the same physical frame. Grouped requests are then issued together to the lookup engine, substantially increasing TwLB hit rates even under multi programmed workloads where interleaved access patterns would otherwise destroy locality.

Non-Blocking Request Queue. Although reordering improves cache efficiency, TwLB misses still incur DRAM latency. To prevent such misses from stalling the entire pipeline, SerMC employs a non-blocking request queue. Each outstanding tripwire lookup is tracked using a reservation-station-like structure and advanced through a small state machine. This design allows SerMC to overlap multiple tripwire metadata fetches and continue servicing independent requests while earlier misses are in flight, maximizing memory-level parallelism and effective bandwidth utilization.

6.2 Software Extension

SerMC enforces memory safety by treating tripwires as architectural guard regions that delimit security-sensitive memory objects. Any access to a tripwire is considered a policy violation and is detected directly by the memory controller, independent of the operating system's execution path. While enforcement is fully offloaded to hardware, the configuration and maintenance of tripwire metadata are managed by software, enabling flexible and application-specific protection policies.

To support this division of responsibility, SerMC introduces a minimal software interface that allows trusted system components to define, update, and revoke tripwire regions. Tripwire metadata are stored in kernel-resident shadow memory and accessed by SerMC through a direct-mapped addressing scheme. This design ensures that tripwire configuration remains protected from user-space adversaries while allowing the memory controller to perform fast lookup without invoking OS services on the critical path.

6.2.1 Tripwire Management Interface

SerMC extends the operating system with two dedicated system calls that allow controlled modification of the tripwire metadata. These system calls are responsible for inserting and removing tripwire entries in the shadow memory space that SerMC consults during address validation. Once the tripwire table is updated, SerMC invalidates or refreshes its internal lookup structures to ensure consistency between software configuration and hardware enforcement.

Tripwire metadata are laid out in a directly mapped shadow region, enabling SerMC to compute the metadata address associated with a physical memory location using a simple transformation. This avoids multi-level lookup and minimizes the latency of metadata retrieval. Importantly, after configuration, tripwire checks are performed entirely in hardware; no operating system intervention is required during normal memory accesses or when violations are detected.

6.2.2 Heap Protection Strategy

SerMC provides heap memory protection by integrating tripwire deployment into dynamic memory allocation. Rather than modifying application source code or compiler tool chains, SerMC relies on a customized memory allocator that transparently augments standard allocation routines. Before returning a heap object to the application, the allocator reserves additional space around the requested allocation and

marks the surrounding regions as tripwires. Alignment padding is inserted when necessary to ensure that tripwire boundaries coincide with cache-line granularity.

This approach enforces spatial boundary checking by isolating adjacent heap objects: a buffer overflow or underflow is detected once it crosses into a tripwire-marked boundary region. Temporal checking is achieved by marking freed heap regions as tripwires. Once an object is deallocated, stale or dangling-pointer accesses are blocked when they target the tripwire-marked reclaimed region. Together, these mechanisms mitigate heap-based buffer overflows and use-after-free vulnerabilities that manifest as accesses to protected DRAM regions.

To maximize deployability, SerMC implements this allocator as a shared library that can be dynamically preloaded at runtime. This allows existing binaries to benefit from heap protection without recompilation or source-level modifications.

6.2.3 Stack Protection Strategy

Stack-based attacks often target control-sensitive data such as return addresses, saved frame pointers, or function pointers. SerMC protects these critical stack elements by strategically placing tripwires around them. Under this model, even if a stack buffer overflow exists, an attacker must traverse one or more tripwire regions to reach a protected target.

Because tripwires are enforced at the memory controller, any unintended access to these guarded regions triggers an immediate exception before corrupted data can influence program execution. This strategy ensures that control-flow hijacking attempts—such as overwriting return addresses is detected early, regardless of how the overflow is triggered.

6.3 Multicore and DMA Scalability

Although the current evaluation uses a fixed simulated configuration, the SerMC enforcement point is naturally compatible with multicore and heterogeneous systems because all DRAM-bound requests converge at the memory controller. In such a configuration, each request entering SerMC can carry a requester identifier that distinguishes CPU cores, DMA engines, and accelerators. This identifier does not change the address-based tripwire policy, but it allows the controller to route exceptions, completion responses, and back-pressure signals to the correct source. Thus, a violation triggered by a DMA engine or by a particular CPU core can be attributed to the offending requestor rather than being treated as an ambiguous memory-system event.

Concurrent requestors mainly affect the controller-local metadata structures, especially the TwLB and reorder buffer. The TwLB can be managed as a shared fully associative metadata cache, allowing different requestors to benefit from common physical-address locality when they access nearby protected regions. Under high contention, the same TwLB can be logically partitioned, or a small number of entries can be reserved for different requestor classes, so that bursty DMA traffic or a memory-intensive core cannot evict all metadata used by other requestors. The reorder buffer only reorders independent tripwire checks and preserves per-requestor ordering constraints for requests that require ordered completion. To avoid starvation, a scalable SerMC design can combine priority-aware or age-aware round-robin arbitration with a bounded residence time for buffered requests before they are issued to the lookup engine. These mechanisms allow SerMC to support concurrent CPU and DMA traffic while preserving the same controller-side enforcement model.

7 Security Analysis

This section analyzes the security properties provided by SerMC's tripwire-based enforcement. By placing validation at the memory controller rather than inside the CPU pipeline, SerMC broadens coverage to DRAM-bound access paths that CPU-side defenses may not observe. This does not imply complete memory safety; the guarantee is limited to accesses that reach tripwire-protected DRAM regions. To overcome SerMC enforcement, an attacker can primarily adopt two strategies: exploiting weaknesses in the tripwire policy to avoid crossing protected regions, or targeting the SerMC infrastructure and metadata-management interface. Consequently, the security guarantees provided by SerMC depend on three key factors: (1) the strength of the tripwire enforcement policy, (2) the coverage of enforcement across DRAM-bound access paths, and (3) the robustness of the SerMC infrastructure. Heap overflows, use-after-free accesses, stack corruption against tripwire-protected control data, and DMA-originated writes that cross guarded boundaries are the primary scenarios directly addressed by SerMC.

In this section, we evaluate the security of SerMC with respect to each of these factors, while also making explicit which attack scenarios are directly covered and which remain outside the scope of the current design. Heap overflows, use-after-free accesses, stack corruption against tripwire-protected control data, and DMA-originated writes that cross guarded boundaries are the primary scenarios directly addressed by SerMC.

7.1 The Strength of the Tripwire Enforcement Policy

Non-Linear Memory Corruption. Tripwires primarily enforces linear spatial safety: it detects memory accesses that cross into explicitly protected regions. This is effective for classic heap overflows, stack-smashing attempts against guarded control data, and many use-after-free accesses once freed memory is marked as a tripwire. However, SerMC does not inherently prevent accesses to memory locations that are structurally valid yet semantically unauthorized. An attacker may exploit this gap by carefully manipulating pointers to target sensitive data that resides outside tripwire-marked regions. Techniques such as indirect pointer overwrites, where a corrupted pointer allows adversaries to selectively modify memory without ever touching a tripwire, may therefore evade detection.

Predictability of Tripwire Placement. SerMC adopts a tripwire-oriented protection model, in which invalid or forbidden memory regions are explicitly marked as tripwires. While this strategy enables efficient enforcement, it also inherits the fundamental limitations associated with tripwire approaches. If tripwire locations are statically determined, their layout may be inferred through binary analysis. With sufficient knowledge of tripwire boundaries, an attacker could construct memory accesses that intentionally bypass guarded regions via indirect pointer manipulation. Runtime randomization and diversified tripwire placement can increase attacker uncertainty, but they should be viewed as partial mitigations rather than as complete solutions to non-linear corruption.

7.2 The Enforcement Coverage on Memory Access Paths

Direct Memory Access Attacks. A defining advantage of SerMC is its ability to observe and validate all DRAM transactions traversing the memory bus. Consider a malicious DMA-capable device, such as a compromised GPU or network interface, that attempts to overwrite a victim buffer and then continue into an adjacent tripwire-protected line. The DMA request may bypass CPU-side cache or MMU-centric checks, but it still reaches the memory controller before DRAM is updated. SerMC therefore performs the same tripwire lookup used for CPU requests: it checks the target cache line against the corresponding tripwire metadata and blocks the transaction if the addressed line is marked as forbidden. Unlike IOMMU-based defenses, which operate at page granularity, SerMC validates accesses at cache-line granularity and can therefore catch fine-grained intra-page violations that page-level DMA isolation may miss.

Memory Corruption within Cache Hierarchy. Since SerMC monitors memory requests at the DRAM interface, a natural concern is whether memory corruption could succeed within the cache hierarchy before modified data are written back to memory. To address this issue, SerMC enforces explicit cache line write-back for any cache line associated with a tripwire-protected region. By ensuring that tripwire-marked cache lines are promptly flushed to DRAM, SerMC guarantees that all accesses to guarded regions are eventually validated by the memory controller, preventing attackers from exploiting transient cache-resident corruption.

Concrete DMA Attack Case Study. To better illustrate how SerMC protects against DMA-based attacks, we provide a concrete example involving a malicious DMA-capable device. Consider a compromised network interface card (NIC) that has obtained the ability to issue arbitrary DMA write requests to system memory. The attacker targets a victim buffer allocated in DRAM, which is followed by a tripwire-protected region inserted by the SerMC-aware allocator. The attacker first identifies the physical address range of the victim buffer, either through information leakage or predictable memory layout, and then the malicious device issues a sequence of DMA write requests to overwrite the buffer contents. These accesses remain within the valid buffer region and therefore do not trigger any protection, after which the attacker intentionally extends the DMA writes beyond the legitimate buffer boundary, attempting to overwrite adjacent sensitive data (e.g., function pointers or metadata).

In conventional CPU-centric defenses, such DMA accesses may bypass cache- or MMU-based checks, allowing the attack to succeed undetected. In contrast, under SerMC, all DMA requests must pass through the memory controller before reaching DRAM. When the malicious DMA write crosses the victim buffer boundary and targets a cache line marked as a tripwire, SerMC handles the request as follows. First, the controller receives the DMA write and extracts the physical address of the target cache line. Second, it probes the Tripwire Lookaside Buffer (TwLB) for the tripwire vector covering that physical region; on a miss, the controller fetches the corresponding metadata vector and installs it in the TwLB. Third, SerMC indexes the vector with the cache-line offset and checks whether the selected bit is marked as forbidden. Finally, if the bit is set, SerMC raises a hardware exception, identifies the offending request source, and terminates the DMA write before the protected DRAM line is modified. If the bit is clear, the request proceeds normally. This sequence detects the attack precisely at the boundary crossing point and prevents the DMA engine from corrupting adjacent protected memory. This example demonstrates that SerMC extends memory safety enforcement beyond CPU execution paths and provides fine-grained protection against DMA-originated attacks.

7.3 The Robustness of SerMC Infrastructure

Because SerMC enforcement is realized in hardware, attackers cannot directly disable or bypass the detector through conventional software techniques. Instead, attacks must focus on compromising the metadata or configuration interfaces upon which SerMC relies.

Metadata Integrity. Like many metadata-assisted defenses, SerMC must protect the integrity of its auxiliary data structures. If an attacker were able to corrupt the tripwire table, they could selectively suppress detection or entirely disable protection. To mitigate this risk, SerMC places all tripwire metadata in kernel memory, preventing direct access from user space. Furthermore, SerMC applies recursive protection by surrounding the tripwire table itself with tripwires. Even if a kernel vulnerability is exploited, unauthorized accesses to the metadata are detected and blocked by the memory controller.

Attacks on Configuration Interfaces. Another potential evasion strategy is to manipulate SerMC's configuration through legitimate system interfaces. For example, an attacker might attempt to invoke system calls—possibly via code-reuse techniques—to clear or disable the tripwire table. SerMC's tripwire mechanism

alone is insufficient to prevent such control-flow-driven attacks. However, this threat is orthogonal to SerMC's design goals and can be addressed by existing code-reuse defenses, which have been extensively studied and deployed in prior work.

Rationale for Monitor Placement. Placing the enforcement logic at the memory controller allows SerMC to cover DRAM-bound access paths that may bypass CPU-side checks, including non-cacheable accesses and DMA transactions issued by peripherals such as GPUs or network devices. This placement strengthens path coverage compared with defenses limited to the processor core or cache hierarchy. However, the guarantee remains tripwire-based: SerMC detects accesses that reach protected DRAM regions, but it does not provide complete protection against semantically malicious accesses that never cross a tripwire boundary.

Trusted Controller State and Side Channels. The current design assumes that controller-resident structures such as the TwLB, affine metadata mapping parameters, and exception-routing logic are protected by the trusted hardware platform. SerMC is not intended to defend against microarchitectural side channels that reveal or corrupt this internal state. Those attacks require complementary defenses and are outside the scope of the present work.

7.4 Scope, Limitations, and Tradeoffs

SerMC broadens enforcement coverage across DRAM-bound access paths, but its guarantees have clear limits. First, non-linear pointer corruption that never crosses a tripwire may remain undetected. Second, the current model assumes trusted hardware, operating-system support, and protected metadata management interfaces. Third, SerMC removes CPU-side metadata traversal from the critical path but does not eliminate metadata bandwidth cost; highly memory-intensive workloads can therefore experience substantial slowdown. Finally, our present evidence comes from cycle-level simulations rather than FPGA or silicon deployments. These tradeoffs are important when comparing SerMC with tagging-based, capability-based, or CPU-centric protection mechanisms.

8 Evaluation

The evaluation is designed to answer three questions. First, how much runtime overhead does SerMC introduce compared with a baseline system without memory-controller tripwire enforcement? Second, whether the overhead is primarily driven by DRAM-request intensity and metadata lookup frequency? Third, what storage cost is required to represent tripwire metadata? To answer these questions, all workloads are compiled with the same compiler and optimization flags, executed under the same gem5 configuration, and measured using the same warm-up and simulation interval. We report performance overhead, total DRAM request counts, and metadata storage overhead to connect the measured slowdown with memory-system behavior.

8.1 Methodology

To evaluate the performance and effectiveness of SerMC, we implemented it as a cycle-level extension to the gem5 memory-controller model in an Arm AArch64 full-system simulation environment. Our simulation platform captures microarchitectural details of the CPU cores, the complete cache hierarchy, and a detailed non-uniform latency DDR3 memory system. The configuration parameters used for simulation are summarized in Table 1.

Table 1: Out-of-order processor core configuration.

Parameter	Configurations
Core	ARM ISA, 2.5 GHz, 8-wide, out-of-order, 352-ROB entries, 128-LSQ entries
BTB	1024 × 4-way, cycles
LTLB/DTLB	64/64 entries
L1-I cache	32 KB, 4-way, 1-cycle, 64B line
L1-D cache	48 KB, 4-way, 1-cycle, 64B line
L2 cache	512 KB, 16-way, 8-cycle, 64B line
L3 cache	4 MB, 2-way, 32-cycle, 64B line
DRAM	50 ns access latency from L3 cache, 12.8 GB/s

We assess SerMC's runtime overhead using twenty selected SPEC CPU workloads that exercise a range of compute-intensive and memory-intensive behaviors. Each benchmark was compiled with Clang 5.0.0 using the optimization flags -O3 -fno-strict-aliasing to maintain aggressive performance optimizations while avoiding undefined behavior that could obscure memory safety measurements. Simulations are executed with reference input datasets, and statistics are collected after simulating one billion instructions following an initial warm-up phase to ensure stable cache and pipeline states.

Because the evaluation uses a fixed simulator configuration, the reported results correspond to deterministic workload-level measurements under the same architectural setup rather than to noisy repeated runs on physical hardware. These workloads provide stable reference executions for controlled architectural simulation, but they do not represent all modern server, graph, or heterogeneous workloads. Future evaluation should therefore extend this study to more diverse benchmark suites, including SPEC CPU 2017, PARSEC, GAP, and DMA-heavy heterogeneous workloads. Such experiments would better characterize SerMC under multicore contention, higher memory-level parallelism, and accelerator-driven memory traffic.

The metadata layout described in Section 6.1.2 also makes the storage overhead reproducible. One 8-bit tripwire vector protects each 1 KB frame, corresponding to approximately 0.1% raw shadow-metadata overhead for protected memory. The controller-local TwLB is also small: with 32 entries and each entry storing a 47-bit frame identifier plus a 1024-bit tripwire vector, the core TwLB state is approximately 4.2 KB before adding valid bits and minor control fields.

8.2 Performance Overhead

We report the overhead of the optimized SerMC design described in Sections 5 and 6. The baseline is the same simulated system without SerMC tripwire enforcement. This comparison isolates the additional cost introduced by controller-side metadata lookup, tripwire validation, and request-management logic.

Fig. 5 presents the performance overhead of SerMC across the evaluated workloads. The average overhead is 9.25%, but this single number hides strong workload dependence. Most workloads incur modest overhead, whereas a small group of memory-intensive workloads dominates the worst-case behavior. In particular, gcc, lbm, roms, and wrf incur approximately 40%, 35%, 50%, and 30% overhead, respectively. The gap between the 9.25% average and the 50% worst case shows that SerMC is not uniformly low-cost across all applications. Instead, its overhead increases when the workload already generates high DRAM traffic and the additional metadata requests compete with demand memory accesses for memory-system bandwidth. These

results support a more precise interpretation of SerMC's tradeoff: controller-side checking reduces CPU-side metadata-path latency and broadens DRAM-path coverage, including DMA visibility, but the current design remains sensitive to memory-bound workloads.

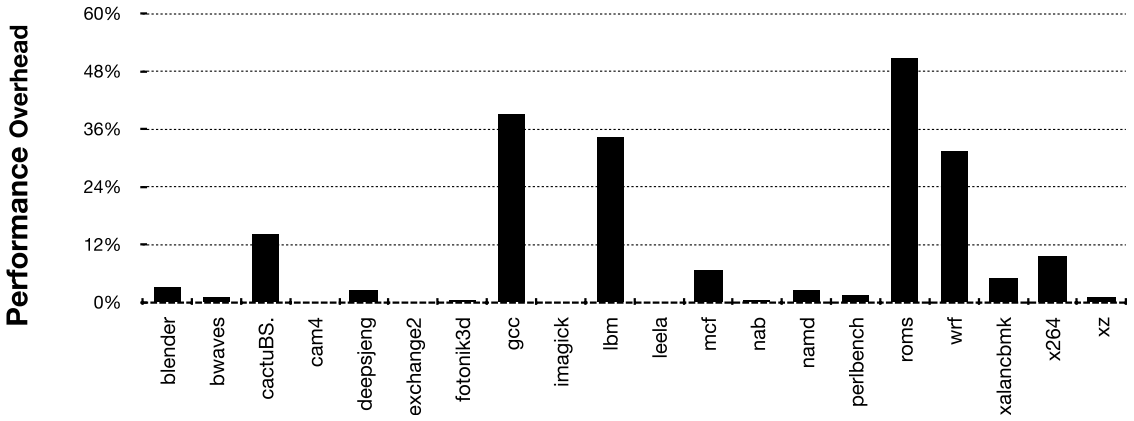

Figure 5: Performance overhead of SerMC.

Several future optimizations could reduce this worst-case penalty. First, metadata prefetching should be throttled according to controller pressure: prefetching tripwire vectors is useful when it increases TwLB hit rate, but it should be suppressed when demand requests already saturate DRAM bandwidth. Second, bit-vector compression or sparse encodings could reduce metadata traffic when tripwire density is low, which is common for many heap layouts where only boundary cache lines are marked. Third, the reorder buffer can use adaptive policies that coalesce nearby metadata lookups under moderate load but prioritize demand requests under saturation, avoiding excessive queuing delay. These mechanisms are orthogonal to the basic SerMC enforcement model and provide promising directions for reducing overhead on memory-intensive workloads such as gcc, lbm, roms, and wrf.

To better understand the source of the overhead in Fig. 5, we measure the total number of DRAM requests issued by each workload, as shown in Fig. 6. The high-overhead workloads in Fig. 5 are also among the workloads that generate heavier DRAM traffic in Fig. 6, indicating that SerMC slowdown is primarily tied to memory-system pressure rather than instruction count alone. Since SerMC performs tripwire validation for DRAM-bound accesses, higher DRAM-request intensity increases metadata lookup frequency and creates additional contention at the memory controller. This result explains why SerMC is attractive for workloads that need broader path coverage but do not saturate the memory system, while aggressively memory-bound workloads remain the limiting case for the current implementation.

9 Related Work

Memory safety has been extensively studied, with solutions differing in metadata usage, enforcement granularity, path coverage, and performance characteristics. Table 2 summarizes representative mechanisms, which we categorize into tripwire-based approaches, memory tagging techniques, bound-checking techniques, and recent capability-oriented designs. The discussion below emphasizes where SerMC sits in this design space rather than claiming an identical-condition head-to-head comparison with all prior systems.

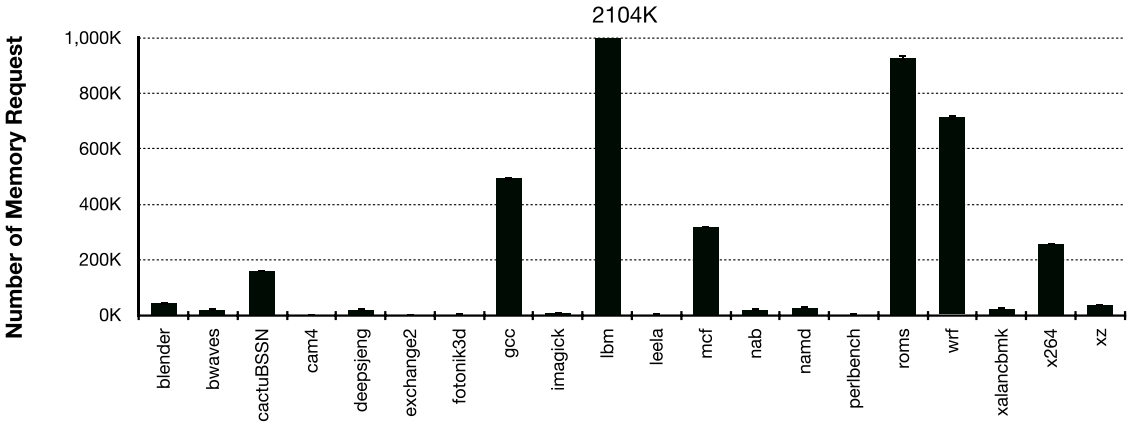

Figure 6: Total number of DRAM requests issued by each benchmark.

Table 2: Comparison of memory safety mechanisms.

Approach	Mechanism	Security Coverage	Data Path Coverage	Performance Overhead	Hardware Modification
Memory Tagging	ARM MTE [9]	Spatial & Temporal	Cache & MMU	N/A	Delicate tag cache
	AOS [10]	Spatial & Temporal	Cache & MMU	8.4%	Reused tag unit
	BOGO [6]	Spatial & Temporal	Cache & MMU	≤50%	N/A
Bound Checking	Intel MPX [8]	Spatial	Cache & MMU	≤50%	Delicate New instr.
	Hardbound [7]	Spatial	Cache & MMU	7%~9%	Delicate tag look-aside buffer
	SafeMem [3]	Spatial	Cache	1.6%~14.4%	Repurpose error correct code
Tripwire	Purity [23]	Spatial	Cache & MMU	4×~50×	Pure software solution
	REST [4]	Spatial & Temporal	Cache	2%~5%	Comparator in L1
	Caliform [5]	Spatial & Temporal	Cache	2%~16%	Reformate cache
	SerMC	Spatial & Temporal	Cache & MMU & DMA	9.25%	Comparator in MC

9.1 Tripwire-Based Approaches

Tripwire mechanisms enforce memory safety by inserting inaccessible guard regions around critical objects. Any access to these regions triggers a violation. Early work, such as Purity [23], demonstrated this

concept for software testing but suffered from high performance overhead. SafeMem extended the idea by encoding object bounds and freed memory locations with Error-Correcting Codes (ECC), enabling detection of illegal accesses. MemTracker [24] associates a few metadata bits with each memory word to indicate its state.

Recent approaches, including REST [4] and Caliform [5], optimize both memory and performance by embedding tripwire metadata in cache lines or repurposing cache line layouts. These methods significantly reduce overhead but still have inherent limitations: attackers can bypass tripwire regions, leaving certain spatial violations undetected.

9.2 Memory Tagging Approaches

Memory tagging techniques attach a small identifier, or "color," to memory regions and check tag consistency during access. For example, Arm Memory Tagging Extension (MTE) uses the top 4 bits of a 64-bit pointer as a tag, with dedicated hardware validating accesses. AOS achieves low overhead but suffers from high false positives in pointer-intensive applications [10], making it less attractive for zero-tolerance environments such as scientific workloads or high-reliability servers. The finite number of tag values also limits the achievable security guarantees and does not inherently extend coverage to every DRAM-bound access path.

9.3 Bound Checking Approaches

Bound checking enforces memory safety by verifying that accesses remain within the intended referent object. While this approach provides strong correctness guarantees, it typically suffers from high performance overhead and limited binary compatibility, reducing real-world adoption. Intel Memory Protection Extensions (MPX) is the first commercial implementation, but it also faces substantial runtime penalties. Techniques such as BOGO extend MPX by adding temporal safety, invalidating all pointers to freed memory regions, yet the performance cost remains high.

9.4 Capability-Based Approaches

Recent capability-oriented architectures provide another important comparison point. CHERIoT uses architectural capabilities to provide fine-grained memory safety and compartmentalization for embedded systems [25]. Cornucopia Reloaded extends CHERI-derived systems with improved heap temporal safety through capability revocation and load barriers [26]. These approaches can provide stronger pointer-integrity guarantees than tripwire-based mechanisms, but they also require ISA-level capability semantics and broader software-stack adaptation. SerMC instead preserves a conventional pointer model and moves enforcement to the memory controller, trading semantic strength for lower integration complexity and broader visibility over DRAM-bound accesses.

9.5 Comparison with SerMC

SerMC differs from prior systems primarily in enforcement placement. Rather than claiming same-platform superiority over MTE, AOS, or CHERI-derived mechanisms, we emphasize a different architectural tradeoff: SerMC processes metadata at the memory controller, keeps validation out of CPU caches and pipelines, and extends checking to DRAM-bound DMA accesses. This choice improves path coverage for accesses that CPU-centric mechanisms may miss, but it does not provide the semantic pointer-integrity guarantees of capability-based systems and remains subject to metadata bandwidth cost on highly memory-intensive workloads.

SerMC differs from previous mechanisms in three key aspects:

- **Memory Controller Enforcement:** SerMC implements tripwire validation at the memory controller, allowing CPU-originated and DMA-originated DRAM-bound requests to be checked by the same enforcement point when they reach protected memory regions.
- **Local Metadata Processing:** Unlike prior schemes that propagate metadata to caches or cores, SerMC processes metadata locally in the memory controller once it arrives from DRAM. This removes CPU-side on-chip traversal and cache interaction from the metadata critical path, although it does not eliminate metadata bandwidth cost.
- **Different Deployment Tradeoff:** By exploiting physical address locality and using the Tripwire Look-aside Buffer, SerMC achieves 9.25% average overhead on the evaluated workloads while extending path coverage to DRAM-bound DMA requests. The same design remains sensitive to memory-bound workloads, with worst-case slowdown reaching 50%, and therefore represents a different tradeoff from tagging- or capability-based systems.

10 Conclusions

In this work, we present SerMC, a memory-controller-based mechanism for detecting spatial and temporal memory-safety violations that cross protected DRAM regions. Our central observation is that metadata overhead is not dominated purely by DRAM latency; on-chip traversal and cache interaction also contribute significantly, motivating controller-side validation. Implemented in gem5, SerMC achieves 9.25% average slowdown on selected SPEC CPU workloads while extending enforcement to DRAM-bound accesses issued by both CPU cores and DMA-capable devices. The evaluation also exposes an important tradeoff: highly memory-intensive workloads can still incur slowdowns of up to 50% because metadata traffic continues to consume memory-system bandwidth. Future work will focus on improving protection against non-linear pointer corruption, reducing metadata overhead in the worst case through throttled metadata prefetching, compressed tripwire vectors, and adaptive reordering, and evaluating SerMC under larger multicore and heterogeneous configurations using more diverse benchmark suites such as SPEC CPU 2017, PARSEC, and GAP.

Acknowledgement: Not applicable.

Funding Statement: This research was funded by Ningbo "Science & Technology Innovation Yongjiang 2035" Grant 2024Z056.

Author Contributions: Conceptualization, Gen Xu and Li Lv; methodology, Li Lv and Jiayan Dong; design and implementation, Gen Xu and Li Lv; writing—original draft preparation, Gen Xu and Li Lv; supervision, Jiayan Dong. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. State of Security 2025. [cited 2026 Jan 8]. Available from: https://www.splunk.com/en_us/form/state-of-security.html.
2. OpenSSL vulnerabilities let attackers execute malicious code. [cited 2026 Jan 8]. Available from: https://cybersecuritynews.com/openssl-vulnerabilities/.
3. Qin F, Lu S, Zhou Y. SafeMem: exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In: Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA 2005); 2005 Feb 12–16; San Francisco, CA, USA. p. 291–302.
4. Sinha K, Sethumadhavan S. Practical memory safety with REST. In: Proceedings of the 45th International Symposium on Computer Architecture (ISCA 2018); 2018 Jun 4–6; Los Angeles, CA, USA. p. 600–11.
5. Sasaki H, Arroyo MA, Ibn Ziad MT, Bhat K, Sinha K, Sethumadhavan S. Practical byte-granular memory blacklisting using califorms. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2019); 2019 Oct 12–16; Columbus, OH, USA. p. 558–71.
6. Zhang T, Lee D, Jung C. BOGO: buy spatial memory safety, get temporal memory safety (almost) free. In: Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19); 2019 Apr 113–17; Providence, RI, USA. New York, NY, USA: ACM; 2019. p. 631–44.
7. Devietti J, Blundell C, Martin MMK, Zdancewic S. HardBound: architectural support for spatial safety of the C programming language. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08); 2008 Mar 1–5; Seattle, WA, USA. p. 103–14.
8. Oleksenko O, Kuvaiskii D, Bhatotia P, Felber P, Fetzer C. Intel MPX explained: an empirical study of intel MPX and software-based bounds checking approaches. arXiv:1702.00719. 2017.
9. Armv8.5-A memory tagging extension (MTE) white paper. [cited 2026 Jan 8]. Available from: https://developer.arm.com/community/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety.
10. Kim Y, Lee J, Kim H. Hardware-based always-on heap memory safety. In: Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2020); 2020 Dec 14–18; Athens, Greece. Piscataway, NJ, USA: IEEE; 2020. p. 1153–66.
11. Polychronakis M, Anagnostakis KG, Markatos EP. An empirical study of real-world polymorphic code injection attacks. In: Proceedings of the 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET 2009); 2009 Apr 21; Boston, MA, USA. Berkeley, CA, USA: USENIX Association; 2009. p. 1–9.
12. Riley RD, Petroski-Such T, Harrison C. An architectural approach to preventing code injection attacks. IEEE Trans Dependable Secure Comput. 2010;7(3):188–201. doi:10.1109/tdsc.2010.1.
13. PaX Team. Address space layout randomization. [cited 2026 Jan 8]. Available from: https://pax.grsecurity.net/docs/aslr.txt.
14. Abadi M, Budiu M, Erlingsson U, Ligatti J. Control-flow integrity: principles, implementations, and applications. In: Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 2005); 2005 Nov 7–11; Alexandria, VA, USA. New York, NY, USA: ACM; 2005. p. 340–53.
15. Shacham H, Page M, Pfaff B, Goh EJ, Modadugu N, Boneh D. On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS 2004); 2004 Oct 25–29; Washington, DC, USA. New York, NY, USA: ACM; 2004. p. 298–307.
16. Snow KZ, Monrose F, Davi L, Dmitrienko A, Liebchen C, Sadeghi AR. Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In: Proceedings of the 34th IEEE Symposium on Security and Privacy (SP 2013); 2013 May 19–22; San Francisco, CA, USA. Piscataway, NJ, USA: IEEE; 2013. p. 574–88.
17. Evans I, Long F, Otgonbaatar U, Shrobe H, Rinard M, Okhravi H et al. Control jujutsu: on the weaknesses of fine-grained control flow integrity. In: Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015); 2015 Oct 13–16; Denver, CO, USA. New York, NY, USA: ACM; 2015. p. 901–13.

18. Bletsch T, Jiang X, Freeh VW, Liang Z. Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2011); 2011 Mar 22–24; Hong Kong, China. New York, NY, USA: ACM; 2011. p. 30–40.
19. Hu H, Shinde S, Adrian S, Chua ZL, Saxena P, Liang Z. Data-oriented programming: on the expressiveness of non-control data attacks. In: Proceedings of the 37th IEEE Symposium on Security and Privacy (SP 2016); 2016 May 22–26; San Jose, CA, USA. Piscataway, NJ, USA: IEEE; 2016. p. 969–86.
20. Michael AE, Gollamudi A, Bosamiya J, Johnson E, Denlinger A, Disselkoen C, et al. MSWasm: soundly enforcing memory-safe execution of unsafe code. Proc ACM Program Languages. 2023;7(POPL):425–54.
21. NVD. CVE-2022-27882 detail. National vulnerability database, U.S. NIST, 2022. [cited 2026 Jan 8]. Available from: https://nvd.nist.gov/vuln/detail/cve-2022-27882.
22. Ling H, Huang H, Wang C, Cai Y, Zhang C. GIANTSAN: efficient memory sanitization with segment folding. In: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24); 2024 Apr 27–May 1; San Diego, CA, USA. New York, NY, USA: ACM; 2024. p. 433–49.
23. Hastings R, Joyce B. Purify: fast detection of memory leaks and access errors. In: Proceedings of the USENIX Winter 1992 Technical Conference; 1992 Jan 20–24; Berkeley, CA, USA. p.125–36.
24. Venkataramani G, Roemer B, Solihin Y, Prvulovic M. MemTracker: efficient and programmable support for memory access monitoring and debugging. In: Proceedings of the 13th IEEE International Symposium on High-Performance Computer Architecture (HPCA-13); 2007 Feb 10–14; Scottsdale, AZ, USA. p. 273–84.
25. Amar S, Chisnall D, Chen T, Filardo NW, Laurie B, Liu K, et al. CHERIoT: complete memory safety for embedded devices. In: Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2023); 2023 Oct 28–Nov 1; Toronto, ON, Canada. New York, NY, USA: ACM; 2023.
26. Filardo NW, Gutstein BF, Woodruff J, Clarke J, Rugg P, Davis B, et al. Cornucopia reloaded: load barriers for CHERI heap temporal safety. In: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2024); 2024 Apr 27–May 1; La Jolla, CA, USA. New York, NY, USA: ACM; 2024. p. 251–68.