BASTAG: Byte-level Access Control on Shared Memory using ARM Memory Tagging Extension

Junseung You
Department of ECE & ISRC
Seoul National University
Seoul, Republic of Korea
jsyou@sor.snu.ac.kr

Jiwon Seo
Department of Cybersecurity
Dankook University
Yongin, Republic of Korea
jwseo@dankook.ac.kr

Kyeongryong Lee
Department of ECE & ISRC
Seoul National University
Seoul, Republic of Korea
krlee@sor.snu.ac.kr

Yeongpil Cho*
Department of Computer Science
Hanyang University
Seoul, Republic of Korea
ypcho@hanyang.ac.kr

Yunheung Paek*
Department of ECE & ISRC
Seoul National University
Seoul, Republic of Korea
Core Trust Link
Seoul, Republic of Korea
ypaek@snu.ac.kr

Abstract

As software grows in size and complexity, modular designs are increasingly adopted, leading to frequent interactions via shared memory between components. This design however increases the risk of vulnerabilities from uncontrolled memory access to shared memory. Enforcing *byte-level access control* can mitigate these risks by enabling byte-level permissions on complex shared objects and their sub-elements. However, existing approaches face performance limitations as they increase the granularity of control to byte level. In this paper, we present BASTAG, a novel system that leverages ARM's Memory Tagging Extension (MTE) to tack this challenge. Although MTE enforces tag-matching between pointers and memory, its hardware-defined granularity is too coarse to support byte-level control on its own. To address the inherent limitations of applying MTE for nuanced access control, BASTAG incorporates a technique known as *shadow memory tagging* that places separate, but associated MTE tags for the actual memory targets, allowing for more flexible and finer access control with efficiency. We implemented a BASTAG prototype on AArch64 hardware with MTE support and evaluated it on three real-world use cases. Our results demonstrate that BASTAG significantly outperforms existing byte-level access control mechanisms.

CCS Concepts

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

Keywords

Memory Tagging Extension; software hardening; shadow memory

*Corresponding authors

ACM Reference Format:
Junseung You, Jiwon Seo, Kyeongryong Lee, Yeongpil Cho, and Yunheung Paek. 2025. BASTAG: Byte-level Access Control on Shared Memory using ARM Memory Tagging Extension. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3744849

1 Introduction

Modularization is a principal design methodology in modern software that involves splitting the software into several distinct components, henceforth referred to as *domains*. These domains, while separate, are interconnected through numerous interactions. According to the methodology, memory within these domains is classified as either *private* or *shared*, each requiring different levels of access control. In a modular design, implementing access controls on memory involves assigning domain-specific access permissions and complying with these permissions. Controlling access to private memory is relatively straightforward, achieved by enforcing domain-exclusive access permissions. However, controlling access to shared memory is much more complex because shared memory requires the application of non-exclusive, adequate access permissions that vary depending on the domain types and the nature of their interactions. Blunt access control on shared memory, such as granting unrestricted access permissions (*e.g.*, read-writable) to shared objects across all domains—contrary to the programmer's intention to limit access to a subset of domains or to enforce asymmetric permissions—has led to numerous vulnerabilities in software systems. These vulnerabilities [19, 20, 24–26, 33] are prevalent across diverse applications, including kernel drivers and multithreaded applications, as further elaborated in Section 3. Due to the complex interactions between domains and the intricate data structures and their small-sized sub-elements, controlling access to shared memory calls for the delicate, exact management of access permissions with *byte-level* granularity.

To date, byte-level access control mechanisms have been studied in different respects. One line of work maintains a copy of the shared object in each domain's private memory [55, 56]. They synchronize the private copies of shared objects by incorporating additional code that transmits and receives messages between the domains to appropriately reflect the changes in sub-elements of the object. Unfortunately, this layer of synchronization management incurs non-negligible overheads as it must be invoked for every occurrence of domain interaction, such as cross-domain function calls. Another line of work uses the inline-reference monitor (IRM) [17, 50, 53]. They adopt a data structure, known as the access control list (ACL), that stores the metadata such as bounds information and access permissions. The code is instrumented to consult the ACL during memory access, using IRMs to enforce control. This IRM-based approach provides a robust framework for managing and controlling access, ensuring that operations on memory are checked against predefined security rules. However, it has a critical downside in that it suffers from too much performance overhead for practical use even when the monitors are applied only to the subset of accesses (*e.g.*, writes). In response, some IRM-based mechanisms sought assistance from the hardware to reduce the overhead of permission checks [54, 84]. However, implementing these solutions requires modifying the existing architecture by adding hardware components, such as registers to hold metadata and comparison/lookup logic, specifically tailored for these purposes. This necessitates customized architectures, which must have been realized either as an extension to open architectures like RISC-V or through simulation using tools like Gem5. As a result, despite their superior performance, it is practically hard to immediately integrate these solutions into commodity products for real-world deployment.

To address this deployment issue in hardware-assisted solutions, this paper presents Bastag, our efficient mechanism for byte-level access control in shared memory. Bastag leverages the Memory Tagging Extension (MTE) [49], a hardware feature equipped in recent commodity ARM processors [5, 6, 34]. Memory tagging, which involves associating tags with pointers and physical memory, enables quick comparisons of tags during each memory access. Access is granted only when the tags on the pointers and their referents match (*i.e.*, carry identical tags). Typically, MTE would tag shared memory directly, requiring domains to access the memory through pointers with a valid tag. Unfortunately, such straightforward use of MTE for access control introduces drawbacks in terms of granularity and the number of supported access permissions. For instance, the minimum granularity of access control is restricted to the predefined granularity of memory blocks for tag comparison, which is 16 bytes for MTE. Aligning shared objects and even their sub-elements to 16 bytes may alleviate this drawback, but such alignment inhibits optimizations within complex data structures (*e.g.*, unions) and urges code modifications across all domains accessing the shared object, inducing major retrofitting efforts. Furthermore, binary decision (*i.e.*, tag match or mismatch) resulting from tag comparisons restricts articulation to just two access permissions (*i.e.*, accessible and non-accessible), which is insufficient for supporting multiple policies such as read-only access.

To cope with all these complications associated with MTE, we incorporate into Bastag a new technique, called *shadow memory tagging* (SMT), which places associated but separate memory tags, called *shadow tags*, that represent the domain's access permissions for the target shared memory. To associate shadow tags with shared memory, SMT reserves *tag regions* that are assigned per domain to attach shadow tags. When accessing shared memory, an additional memory operation is executed to consult the corresponding tag regions, effectively initiating an efficient MTE tag comparison that verifies the validity of the access against the shadow tags. Capitalizing on SMT allows Bastag to circumvent the inherent limitations of MTE by configuring shadow tags in tag regions for fine-grained access control. By applying shadow tags per byte of the shared memory, Bastag can exert precise byte-level access control. Furthermore, Bastag is designed to assign a pair of tag regions for each domain, one for managing read accesses and another for write accesses, thus supporting multi-permission (*i.e.*, not-accessible, read-only, read-writable) access control for shared memory across different domains.

To ease integration into existing systems, Bastag provides a comprehensive framework that includes a library and a compiler. The library offers a suite of APIs that developers can utilize to manage access control tasks, such as configuring access policies and granting or revoking access permissions. With the source code equipped with these APIs, the compiler plays a crucial role in identifying shared memory access operations and augmenting them by inserting instructions that perform tag comparisons against the shadow tags within the appropriate tag regions. This integration ensures that Bastag can be seamlessly integrated to enhance security and access control in systems making use of shared memory. We implement a prototype of Bastag on an off-the-shelf machine equipped with ARMv9 CPUs and evaluate Bastag on three realistic use cases of memory sharing - between core kernel and extensions, between the tasks on top of middleware, and between threads in multithreaded applications. Our evaluation demonstrates the versatility and practicality of Bastag, incurring 3.2% overheads for kernel extensions, 7.1% for inter-task communication messages, and 5.75% for multithreaded application (*e.g.*, Memcached), which outperforms state-of-the-art IRM-based mechanism that supports byte-level access control.

2 ARM Memory Tagging Extension

ARM Memory Tagging Extension (MTE) was first introduced in ARMv8.5-A architecture, and is being built into the ARMv9 family of processors as well [2]. It realizes the tagged architecture by introducing two types of tags - pointer tags and memory tags. Pointer tags are implemented by utilizing the Top Byte Ignore (TBI) feature introduced in ARMv8.1 [4]. TBI allows ARM processors to ignore the top byte of pointers during address translation, thus enabling the top byte to be used to store additional metadata. MTE uses four bits (*i.e.*, [59:56] bits of the address) to store the pointer tags. Memory tags consist of four bits as well and are stored separately from physical memory, where each is associated with a 16-byte aligned memory block. On memory access, MTE checks whether the pointer tag matches the memory block's tag and raises an exception in case of a mismatch. In MTE's synchronous mode, the exception is raised immediately at the memory access. MTE introduces additional instructions (*e.g.*, `LDG`, `STG`) to load and store memory tags.

```
int ccci_ringbuf_read(int md_id,
  struct ccci_ringbuf *ringbuf
  unsigned char *buf, int read_size) {
  unsigned int read, write, length;
  // Offsets are retrieved from shared memory
  read = (unsigned int)(ringbuf->rx_control.read);
  // Msg is read outside of the shared memory buffer
  CCIF_RBF_READ(ringbuf->buffer,
                  buf, read_size, read, length);
  return read_size;
}
```

Listing 1: Simplified vulnerable function in kernel driver from CVE-2022-21769 [25] caused by sub-field `read`.

```
// Main thread
void dispatch_conn_new(...) {
    CQ_ITEM *item = cqi_new(thread->ev_queue);
    notify_worker(thread, item);
}
// Worker thread
static void thread_libevent_process(...) {
    item = cq_pop(me->ev_queue);
}
```

Listing 2: Simplified main and worker thread from Memcached 1.6.22. Object `item` is intended to be shared only between the main and selected worker threads.

3 Motivation

Shared memory generally refers to memory space that is allocated for simultaneous access from separate domains. Each domain is associated with an access policy (*i.e.*, read-only (ro), read-write (rw), not-accessible (na)) that governs its interaction with the shared memory. In this context, the concept of shared memory within BASTAG is defined as memory space accessible by two or more domains, each assigned with access policy other than na. This definition distinguishes shared memory from private memory, which exclusively grants rw permission to a single domain while all other domains are assigned with na permissions. We illustrate the need of enforcing sophisticated access control, such as assigning byte-granular per-domain permissions, on shared data with three examples from widely-used software.

Core Kernel and Extensions. The modern kernel interacts with various drivers and extensions through complex shared objects. While the sharing of these data structures is essential for enabling the intended functionalities of the drivers, providing unrestricted access to all fields within the shared structure can pose a potential risk of compromising the core kernel. Listing 1 shows an example of such a vulnerability, where the driver is granted with rw permissions for the shared buffer (ringbuf) that includes the metadata (read) as its sub-element. This setup allows the driver to read beyond the original shared memory region by modifying the metadata. To mitigate this vulnerability, it is crucial to impose read-only access control on the driver regarding metadata sub-fields. Similar vulnerabilities [19, 20, 22, 24, 27, 33] are reported as well, highlighting the necessity of access control over byte-granular sub-elements of the structures shared between the core kernel and its extensions.

Multithreaded Applications. In many multithreaded programs, it is often necessary for shared objects to be accessible only to specific threads, rather than all threads, for security reasons. These shared objects can vary in granularity, sometimes even at the byte level, and their sizes can differ based on the attributes of the applications. Listing 2 demonstrates the use case of exclusive sharing in

```
int main(int argc, char *argv[]) {
  ros::NodeHandle nh("~");
  ros::Subscriber time_ref_sub;
  // Subscribe to topic stored in
  // time_ref_topic parameter
  time_ref_sub = nh.subscribe(time_ref_topic, 10,
    time_ref_cb, ros::TransportHints().unreliable()
    .maxDatagramSize(1024).reliable()
    .tcpNoDelay(true));
  return 0;
}
```

Listing 3: Simplified vulnerable function in shared memory driver in ROS middleware from CVE-2022-48198 [26].

Memcached [52], wherein the data object CQ_ITEM is intended to be read-writable solely by the main thread and a designated worker thread. Furthermore, it is necessary to enforce asymmetric access permissions (*e.g.*, rw for the main thread and ro for the worker thread) at the byte-level to mitigate vulnerabilities which exploit configuration variables that are accessible for modification by all threads (*e.g.*, CVE-2021-21309 [23]).

Inter-Task Communication. In widely-used middlewares such as Robot Operating System (ROS) [60] and PX4 autopilot [63], task management relies on communication via shared messages and parameters [59, 65]. However, unregulated access to these shared memory components poses security risks, including the leakage of sensitive information from other tasks and the potential for attacker-controlled behavior [28]. Listing 3 shows an example of such a vulnerability, wherein all nodes (*i.e.*, tasks) have access to the shared time_ref_topic parameter. This parameter is utilized by the middleware to determine the appropriate topic (*i.e.*, message type) to subscribe to for its operation. An adversarial node could maliciously modify this parameter to subscribe to an attacker-controlled topic, thereby triggering unintended operations that can compromise the robot's security. Additionally, our investigation revealed that different tasks access only specific subsets of sub-fields within the shared message. Granting unrestricted access to all sub-fields poses a potential vulnerability, as a compromised node could overwrite a sub-field not used in normal operation, thereby affecting the behavior of other nodes. For example, the VehicleRatesSetpoint message in PX4 is utilized by four tasks, each accessing different subsets of sub-fields within the message structure (*e.g.*, bool reset_integral, float32 roll). Access control, such as enforcing ro permission to shared parameters after initialization, and applying rw permission to specific sub-fields of the message, is necessary to mitigate these vulnerabilities.

3.1 Existing Access Control Mechanisms

Page Table based Access Control. Various approaches have been pursued to enforce access control by configuring permission flags in PTs that map shared memory. The first generation of PT-based access control mechanisms [13, 15, 62, 73] instantiated domains as separate processes, assigning each domain its own PTs with different permission flags set for shared memory. However, this approach slowed down domain cooperation significantly as domains had to rely on costly Inter-Process Communication (IPC) to communicate. To address this limitation, the second generation of PT-based access control mechanisms, such as Arbiter [79], SMV [36], and lwC [46], introduced per-domain PT to enable access control within a single process. However, supporting per-domain PTs necessitates extensive kernel-level modifications and is limited to a specific domain type: thread. The third generation of PT-based access control mechanisms integrated hardware support [18, 31, 35, 76, 92], leveraging primitives like Intel Memory Protection Keys (MPK) [21], ARM Memory Domains (MD) [10], and ARM Memory Protection Unit (MPU) [11] to accelerate PT permission management. Unfortunately, PT-based approaches inherently employ page granularity (*i.e.*, 4KB) for access control. While suitable for regulating access to larger memory regions (*e.g.*, private), this approach lacks the capability to support intra-object access control. This limitation arises because an object is allocated in a page, resulting in equal access permissions being applied to all sub-fields of an object.

IRM/Shadow Memory based Access Control. Another line of research, such as BGI [17] and LXFI [50], addresses access control through the adoption of inline reference monitors (IRM) for memory access management. These mechanisms typically utilize distinct access control lists (ACLs) to store the access policies corresponding to memory spaces of each domain. Upon memory access, ACLs are consulted by IRMs to ensure adherence to the specific policies. IRM-based mechanisms build upon shadow memory schemes [57, 70], which associate each byte of program memory with metadata stored in shadow memory. In IRMs, the shadow memory specifically holds ACL entries, mapping each byte of program memory to a corresponding ACL index. This mapping enables the IRM to dynamically retrieve and enforce domain-specific permissions during runtime. However, this design introduces substantial performance overhead due to additional operations required before each memory access—specifically, computing the ACL entry address, retrieving access permissions, and validating these permissions. For instance, Listing 4 illustrates an IRM instruction sequence employed by [17]. Initially, three instructions calculate the ACL (*i.e.*, shadow memory) address that stores the permissions for the memory address contained in register `X1` (Lines 2-4). The permissions are then retrieved (Line 5) and checked (Line 6) before executing the original memory access (Line 10). Depending on the specific ACL implementation, additional instructions may be necessary to handle exceptional situations, such as ACL conflicts (Line 8).

Message based Access Control. Message-based access control, such as LXD [55] and LVD [56], maintains a copy of a shared object in each domain's private memory. To facilitate synchronization of objects across domains, message-based access control employs an Interface Definition Language (IDL) to specify the subset of an object's fields that will be marshaled across domains. Subsequently, the IDL generates glue code, which acts as an interface between domains interacting with shared objects. Functioning on the basis of remote procedure call (RPC) implementation, this glue code transmits and receives messages to seamlessly manage synchronization across multiple domains. This mechanism enables intra-object access control but introduces notable performance overhead due to additional glue code function calls, `memcpy` operations for marshaling sub-fields, and frequent RPC-based synchronization during domain interactions.

MTE-only Access Control. MTE-only access control is a mechanism that straightforwardly utilizes MTE by tagging shared memory and accessing it exclusively with pointers bearing a specific tag. However, employing MTE in a one-dimensional manner encounters limitations inherent to the MTE architecture. Tag comparison

```
1  // X1: target addr.
2  MOV     X0, X1
3  ASR     X0, X0, #0x2
4  TST     X0, X0, #0x1C
5  LDR     X0, [X0]
6  CMP     X0, #0x2
7  BEQ     L1
8  BL      _slowCheck
9  L1:
10 LDR     X0, [X1]
```

Listing 4: Instruction sequence of IRM used by [17].

Mechanism	BG	PAP	MP	SA	LPO
PT-based	✘	✔	✔	✔	✔
IRM-based	✔	✔	✔	✔	✘
Msg-based	✔	✔	✔	✘	✘
MTE-only	✘	✘	✘	✘	✔
BASTAG	✔	✔	✔	✔	✔

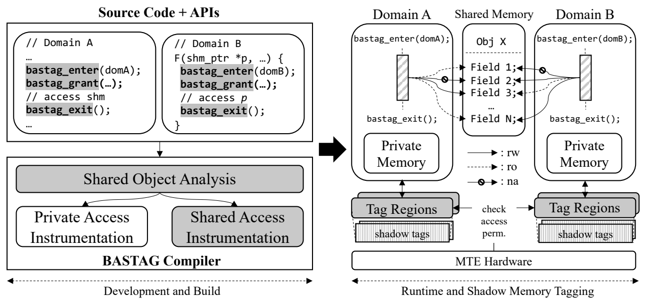
Table 1: Comparison between BASTAG and other access control mechanisms based on the requirements (Section 5).

results in a binary decision (*i.e.*, tag match or mismatch), limiting MTE-only access control to providing only two access policies. Additionally, the mechanism fails to provide asymmetric access permissions, as memory tags must be modified to express different access permissions on different domains. Although one may re-tag shared memory to change permissions, this approach hinders concurrent memory access by different domains, as a domain must wait for the tag update before performing access. Moreover, tagging directly applied to shared memory sets access control granularity determined by MTE's tag granularity, falling short of providing access control on object sub-fields smaller than 16 bytes, as acknowledged by other works on MTE as well [69, 71].

4 Threat Model

Assumptions. BASTAG is designed to provide access control on memory that is *shared* by the domains composing complex software, such as user programs and kernel extensions. As explained previously, the memory accessible by domains is classified as either *shared* or *private*. We assume that access control on *private* memory is achieved through domain isolation techniques such as software fault isolation (SFI) [85, 86] or in-process isolation [38, 83, 88]. BASTAG is designed to complement existing private memory isolation techniques, where the integration will be later explained in detail at Section 6.4. To this end, we assume that domain isolation techniques provide per-domain stack isolation and that the integrity of BASTAG APIs (Section 5.6) are guaranteed by SFI trampolines or in-process isolation call gates. We assume that the core kernel (*e.g.*, interrupt handlers) is trusted and that the OS enforces standard DEP (*i.e.*, W⊕X). We assume the hardware is trusted, and thus exclude exploits targeting hardware vulnerabilities, such as microarchitectural attacks [41, 42, 45], from the scope of this work.

Attacker Capabilities. We consider an adversary capable of compromising the domain through vulnerabilities such as buffer overflows and control flow subversion. Even so, BASTAG must prevent domains from reaching shared memory not permitted to them.



```
// Domain A
…
bastag_enter(domA);
bastag_grant(…);
// access shm
bastag_exit();
…
```

```
// Domain B
F(shm_ptr *p, …) {
 bastag_enter(domB);
 bastag_grant(…);
 // access p
 bastag_exit();
}
```

```
bastag_enter(domA);
bastag_exit();
```

```
Obj X
Field 1;
Field 2;
Field 3;
…
Field N;
```

```
bastag_enter(domB);
bastag_exit();
```

Figure 1: Overview of BASTAG.

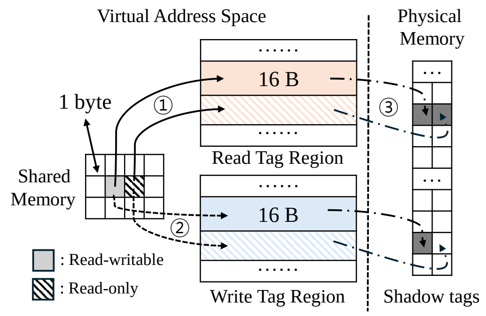


Figure 2: Shadow memory tagging. BASTAG maps each byte of shared memory to 16B of read tag region (①) and write tag region (②). Shadow memory tags corresponding to the tag region blocks are colored if the operation is allowed (③).

5 Design

Goal and Requirements. BASTAG aims to support byte-level access control for shared memory by fulfilling the following requirements that existing mechanisms (Section 3.1) fall short of providing, as summarized in Table 1.

- *Byte-level Granularity (BG).* Memory sharing at a fine-grained, sub-object level requires byte-level access control.
- *Per-domain Access Policy (PAP).* Access control on shared memory requires different access permissions per domain.
- *Multiple Policies (MP).* Access control on shared memory requires more than two policies (*i.e.*, `rw`, `ro`, and `na`).
- *Simultaneous Access (SA).* Access control needs to support concurrent memory accesses from multiple domains.
- *Low Performance Overhead (LPO).* The above requirements must be realized with reasonable runtime overheads.

5.1 BASTAG Overview

BASTAG offers an efficient solution for implementing byte-level multi-policy access control over memory objects shared between distinct domains, leveraging MTE as depicted in Figure 1. As explained in Section 3.1, relying solely on MTE in a conventional manner, such as directly attaching tags to target memory subject to access control (*i.e.*, shared memory), is insufficient due to MTE's limited tag granularity of 16 bytes and its binary permission granting system (either read-writable or not-accessible). To overcome these limitations, BASTAG incorporates SMT, a technique that adds a layer of indirection to manage memory tags containing access permissions in a separate region. In detail, *shadow tags*, which are MTE memory tags, represent the access permissions of domains to shared memory in SMT (*i.e.*, hold non-zero tag if accessible, and hold tag zero if inaccessible). SMT reserves *tag regions* for each domain, where a block of tag region is associated with a byte of shared memory. The shadow tags are then attached to the corresponding virtual addresses of tag regions, thereby establishing a relationship between the shadow tag and the shared memory. Subsequently, whenever there is an access to shared memory in domains, BASTAG performs an additional memory operation to consult the shadow tag in the corresponding tag region. This operation is performed by accessing a tag region address that corresponds to the target shared memory, alongside a pointer tag identical to the one employed for the shadow (memory) tags. Naturally, accessing the tag regions triggers a MTE tag comparison using the shadow tags, determining whether the access to the shared memory is permissible or not. It is worth noting here that BASTAG enhances access control by configuring shadow tags and tag regions. Specifically, BASTAG assigns a shadow tag to each byte of the shared memory through the tag region, facilitating byte-level access control. Additionally, BASTAG allocates a pair of tag regions to each domain, each designated to control read and write access to the shared memory, respectively, thus supporting multi-policy access control for the shared memory across domains.

To facilitate the implementation of SMT, BASTAG incorporates a framework consisting of a compiler, a library, and a verifier. At a high level, developers are provided with a set of APIs (Section 5.6) to manage access control, including tasks such as granting or revoking access permissions. Upon receiving the source code annotated with these APIs, BASTAG compiler (Section 6.1) initiates an analysis phase to identify shared memory access operations (*i.e.*, loads and stores). Subsequently, the compiler augments the identified operations with instructions tailored to invoke tag comparisons against shadow memory tags of the corresponding tag regions. The verifier then verifies whether the additional instructions are instrumented correctly to enforce access control. During runtime, when BASTAG library configures tag regions and shadow tags based on the API calls made by developers, the instrumented code executes with access control enabled.

5.2 Shadow Memory Tagging (SMT)

SMT, depicted in Figure 2, is the core technique of BASTAG, enabling efficient byte-level multi-policy access control on shared memory across domains. In this technique, access permissions of domains to shared memory are represented as shadow tags, which are MTE memory tags corresponding to virtual addresses of domain-specific tag regions. Each domain is assigned two tag regions: a read tag region to manage read permissions, and a write tag region to manage write permissions. To express permissions, shadow memory tags are colored (*i.e.*, hold non-zero tag) if the operation is permitted and uncolored (*i.e.*, hold tag 0) if the operation is not permitted. These shadow tags are referenced in MTE's tag comparisons by accessing the tag region associated with the shared memory with a valid pointer tag (*i.e.*, the same tag number used in the shadow memory tag) prior to the actual access to shared memory. As shadow tags are uncolored if the shared memory is inaccessible, the tag comparison will automatically fail if an illegal access is made, since the pointer tag holds a non-zero value when accessing tag regions to check the permissions. Hence, BASTAG effectively enforces access control on shared memory by assigning shadow tags to per-domain tag regions according to the access policies.

Enabling Byte-level Multi-policy Access Control. Under shadow memory tagging design, each byte of shared memory must be associated with a separate shadow tag to support byte-level access control. To establish this relationship, BASTAG maps each byte of shared memory to 16-byte tag region blocks, as depicted in Figure 2-① and ②. Subsequently, BASTAG attaches shadow memory tags representing access permissions to tag region blocks (Figure 2-③), leveraging MTE's capability to assign memory tags at a 16-byte granularity, thereby enabling access control at a finer granularity of a byte. BASTAG provides multi-policy access control (i.e., `rw`, `ro`, `na`) on shared memory by allocating each domain a pair of tag regions — one for controlling read operations and the other for controlling write operations — and attaching shadow tags to each tag region pair of domains. This allows BASTAG to express multiple policies by appropriately setting the shadow tags attached to the read/write tag regions according to the desired policy. For example, both the shadow tags for the read and write tag region blocks corresponding to the gray shared memory byte in Figure 2 are colored to express `rw`. To disallow read or write operations, the shadow tags for the respective tag regions are left uncolored, as depicted in Figure 2, where the shadow tag matching the write tag region block for the dashed shared memory byte is uncolored to express `ro` permission.

Shared Memory-to-Tag Region Mapping. To establish the mapping between shared memory and the tag region pairs of domains, BASTAG manages fixed base addresses of the tag regions, which are determined during the initial creation of tag regions for each domain. The address of the tag region ($addr_{tr}$) is calculated by adding the base address ($addr_{base}$) to a shifted address of the shared memory ($addr_{shm}$) - *i.e.*, $addr\_tr = addr\_base + addr\_shm << 4$. Specifically, BASTAG shifts the shared memory address by 4 bits to map each byte of the shared memory to a 16-byte block of a tag region. This shifted address is then added to the base address to compute the corresponding tag region address. The mapping between shared memory and tag regions serves two main purposes. First, when API calls are made in a domain to set access permissions for the shared memory, BASTAG populates the associated tag regions with shadow tags using this mapping. Second, when a domain accesses the shared memory, BASTAG utilizes the mapping to enforce access control by performing shadow tag-based comparisons on the relevant tag regions.

5.3 Access Control Enforcement

Following the reservation of tag regions and the configuration of shadow tags, BASTAG effectively enforces access control using MTE, wherein tag comparisons against shadow tags are automatically executed upon enabling MTE. However, the tag regions are distinct from shared memory, implying that accessing shared memory does not inherently trigger tag comparisons based on shadow tags. To address this issue, BASTAG instruments each access to shared memory by adding supplementary instructions that execute tag comparisons using shadow tags. To be specific, BASTAG inserts two instructions before memory operations (*i.e.*, `LDR, STR`) targeting

```
// Shared load instrumentation
BFI    X21, X1, #4, #32    ; ❶
STR    XZR, [X21]          ; ❷
LDR    X0, [X1]            ; ❸
// Shared store instrumentation
BFI    X21, X1, #4, #32
STR    XZR, [X21]
STR    X0, [X1]
```

Listing 5: Instruction sequences instrumented for BASTAG's access control enforcement. BASTAG computes associated tag region address (❶), checks access permission with dummy store (❷), and accesses shared memory if ❷ is passed (❸).

the shared memory as depicted in Listing 5-❶ and ❷. One instruction is employed to calculate the tag region address corresponding to the target shared memory address (❶), while the other serves to invoke tag comparisons by accessing the tag region address (❷). In this instrumentation, two registers (*i.e.*, `X21` and `X22`) are reserved to store the base addresses of read and write tag regions of a domain, respectively. Additionally, to ensure correct access control checks through tag comparisons against shadow tags, the reserved registers are tagged with a pointer tag whose tag number matches that of the shadow tag (tag M). Reserved registers are set through BASTAG APIs, as will be explained in detail at Section 5.6.

Computing Tag Region Address. Given a target shared memory address (`X1` in Listing 5, where shared memory load before the instrumentation is `LDR X0,[X1]`), corresponding tag memory address is computed using a bit-field insert instruction (`BFI`, Listing 5-❶). The instruction transfers a segment of the source register to a specified location within the destination register while leaving unrelated bits unchanged. Specifically, with a given syntax of `BFI` $X_d, X_n, \#lsb, \#w$, the instruction copies low-order w bits from the source register X_n into the same number of adjacent bits in the destination register X_d, starting from the lsb-th bit. Utilizing the `BFI` instruction, the calculation of the tag region address consolidates two arithmetic operations (addition and left shift) into one instruction. For example, when integrated with SFI mechanism that configures the domain's address space size to $4GB = 2^{32}$, BFI is instrumented with parameters $w = 32$ and $lsb = 4$ for the tag region address calculation. Destination register in the `BFI` instruction depends on the type of shared memory access: `X21` is used for loads, holding the base address of the read tag region, while `X22` is used for stores, holding the base address of the write tag region.

Checking Access Permissions. After calculating and storing the tag region address corresponding to the target shared memory to the reserved register using `BFI`, BASTAG executes an additional `STR` instruction with that register as a memory operand (Listing 5-❷), thereby triggering tag comparisons to enforce access control. Recall that the tag regions of each domain have been marked with shadow tags according to the access policies for the shared memory, and the reserved registers have been assigned pointer tags identical to those of the shadow memory tags (tag M). Therefore, tag comparisons during execution determine accessibility, resulting in either a match for accessible memory or a mismatch for inaccessible memory.

Advantages of MTE-based Instrumentation. BASTAG's design leverages SMT with shadow tag regions and enforces access control

```
// Shared load instrumentation
BFI     X21, X1, #4, #32  ; ❶
BFI     X23, X1, #0, #48  ; ❷
STR     XZR, [X21]        ; ❸
LDR     X0, [X23]         ; ❹
BFI     X21, XZR, #0, #36 ; ❺
// Shared store instrumentation
BFI     X22, X1, #4, #32
BFI     X23, X1, #0, #48
STR     XZR, [X22]
STR     X0, [X23]
BFI     X22, XZR, #0, #36
```

Listing 6: BASTAG's access control enforcement hardened to prevent access control bypass. Added instructions compared to the baseline instrumentation from Listing 5 are highlighted. Target shared memory address is moved to the reserved register X23 (❷) which is used as an address operand for the memory access (❹). Tag region address is cleared while retaining the pointer tag and base address of tag region after the shared memory access (❺).

by inserting instructions, resembling the IRM/shadow memory-based mechanisms described in Section 3.1. The critical advantage of BASTAG over purely software-based IRM/shadow memory approaches lies in its use of MTE to validate access permissions efficiently and transparently, without explicit retrieval or software-based comparison. Specifically, referring back to Listing 4, which shows the instruction sequence for IRM, BASTAG replaces both the LDR instruction (Line 5)—used to retrieve access permissions from shadow memory—and the subsequent CMP instruction (Line 6)—used to compare the permissions, which must wait until LDR completes—with a single MTE-enabled dummy STR instruction targeting the tag region address corresponding to the shared memory byte. This instruction triggers MTE's hardware-based tag check, which implicitly validates the access against the tag representing the permission, thereby eliminating software overhead associated with permission retrieval and comparison. BASTAG uses the STR instruction, rather than LDR, to trigger tag checks because STR typically incurs lower performance penalties, benefiting from micro-architectural store queues and buffers that allow writes to be handled lazily without stalling the pipeline.

5.4 Preventing Access Control Bypass

In addition to enforcing access control, BASTAG prevents malicious attempts aimed to perform illegal accesses to shared memory.

Preventing Instrumentation Circumvention. BASTAG prevents malicious attempts aimed at circumventing the instrumentation to perform illegal memory accesses. Consider Listing 5, where Line 2 computes the tag region address corresponding to X1 and Line 3 checks the access permissions. In this scenario, non-intended control flow to LDR with attacker-controlled value in X1 leads to illegal memory access as LDR is performed using X1 as a memory operand. BASTAG thwarts this attempt by conducting shared memory access through an additional reserved register, X23, as depicted in Listing 6-❹. In order to replace the original target address register with the reserved register, BASTAG instruments the shared memory accesses with additional BFI instruction to copy the valid virtual address space bits (*i.e.*, 48 bits) from the original memory operand register (X1) to the reserved register (X23), as shown in Listing 6-❷. As an adversary cannot control the value of reserved register, X23, landing on Listing 6-❸ or ❹ that bypass the preceding instructions does not lead up to a practical attack.

```
1 // Private load instrumentation
2 BFI     X24, X1, #0, #48
3 LDR     X0, [X24]
```

Listing 7: Instrumentation on private memory accesses to prevent them from illegally accessing shared memory.

In case of redirecting the control flow to Listing 6-❷, where the address in X1 is controlled by the attacker, the adversary can pass the permission check conducted at ❸ if the valid tag region address in X21 is maintained from the previous shared memory access. BASTAG mitigates such attacks by resetting the reserved register utilized for tag region address following the shared memory access, while preserving its pointer tag and base address of the tag region. This process is accomplished through the BFI instruction with XZR as the source register. For example, given that the tag region address is computed by ❶, Listing 6-❺ clears the bottom 36-bits of X21. Through this approach, an attacker is prevented from reusing the valid tag region address, resulting in a fault at ❸ due to cleared X21 (or X22) should the control flow circumvent ❶ and reach ❷.

Preventing Misuse of Private Memory Accesses. In addition to mitigating malicious attempts aimed at bypassing the access control enforced by the instruction sequence, BASTAG must also prevent the exploitation of private memory accesses that aims to gain unauthorized access to shared memory. Consider instruction, LDR X0, [X1], where the private memory is accessed with X1 register value as an address. Note that, private memory accesses are not instrumented as shared memory accesses. This enables an attacker to manipulate the value stored in X1 to the address of shared memory. Consequently, attacker can directly access shared memory using memory instructions originally intended for accessing private memory. BASTAG thwarts this attempt by conducting private memory access through an additional reserved register, X24, as depicted in Listing 7. Similar to the instrumentation employed to mitigate access control bypass attempts, private memory accesses undergo augmentation through the addition of BFI instruction. This instruction serves to transfer the valid virtual address space bits from the original memory operand register (X1) to the reserved register (X24).

The key difference between reserved registers X24 and X23, the latter serving to hold the shared memory address, lies in their respective assignment of distinct pointer tags. Specifically, X24 retains pointer tag zero for accessing private memory, while X23 retains non-zero pointer tag (*e.g.*, tag N), distinct from the one utilized for shadow tags (tag M), for accessing shared memory. As will be explained in Section 5.6, the memory is tagged with tag N when it is registered as shared memory through the BASTAG API. Since an adversary lacks control over the reserved register, X24, unintended control flow landing on Line 3 in Listing 7 does not lead up to a practical attack. Additionally, attacker is unable to access shared memory by manipulating the original memory operand register (X1), as private memory access occurs via X24 with a pointer tag zero, resulting a tag mismatch if X1 contains the address of shared memory tagged with non-zero value N.

```
1  /* ccci_ringbuf.c */
2  int ccci_ringbuf_read(
3    int md_id, struct ccci_ringbuf *ringbuf, ...){
4    ...
5  + bastag_enter(DRIVER_DOMAIN_ID);
6  + bastag_register(ringbuf, BUF_SIZE);
7    if (ringbuf == NULL)
8      return -CCCI_RINGBUF_PARAM_ERR;
9  + bastag_set(ringbuf, // NA: 0, RO: 1, RW: 2
10 +             sizeof(ringbuf->rx_control), 1);
11   // SIZE_OF_BUFFER: size of ringbuf->buffer
12 + bastag_set(ringbuf+offset,SIZE_OF_BUFFER,2);
13   // Access shared memory
14   read = (int)(ringbuf->rx_control.read);
15   CCIF_RBF_READ(ringbuf->buffer, buf, read);
16 + bastag_exit();
17   ...
18 }
```

Listing 8: Example of using Bastag APIs for the simplified vulnerable kernel driver described in Section 3. Access control is enforced on shared memory object `ringbuf`, imposing read-only permission to byte-level sub-field (`rx_control`).

In summary, Bastag reserves four general-purpose registers for the system:

- `X21`:contains read tag region address and pointer tag M.
- `X22`:contains write tag region address and pointer tag M.
- `X23`:contains shared memory address and pointer tag N.
- `X24`:contains private memory address and pointer tag 0.

We find the impact of reserving up to four registers to be minimal (Section 7), which is reported in recent work that reserves up to five registers for their system as well [85].

5.5 Optimizations

Tag regions would incur memory overheads linear to the size of shared memory due to shadow tag allocations. Bastag supports two optimizations to address the problem.

Tag Region Sharing. Bastag supports a tag region sharing that allows different domains to share the tag regions by mapping them to the same physical memory to use the same memory tags. This enables the domains that have the same access permissions to shared memory (*i.e.*, shadow tags are attached to respective tag regions in the same pattern) to use the same tag regions, thereby reducing the memory necessary for the shadow tags. Tag region sharing operates at a tag region page. Bastag identifies the tag region pages with identical access permissions by managing a 2-bit bitmap for each tag region page, where each 2 bits contains the information about a 16B tag region block (*i.e.*, 64B bitmap per page). These two bits represent whether the 16B tag region block is tagged, untagged, not permitted for tag region sharing, or not used for access control. Upon a request to set the access permissions and enable tag region sharing through the APIs (Section 5.6), Bastag consults the bitmaps and maps two tag region pages with identical pattern to the same physical page. Note that, Bastag does not manage the bitmaps for every tag region page but creates them on-demand when tag region sharing is enabled for specific range of shared memory. Tag region sharing shows its strength when multiple domains are assigned to have the identical access permissions to the same shared memory objects. For example, in case of inter-task communication where the tasks have read permissions to shared messages but differ in write permissions, tag region sharing allows all read tag regions to be shared to use the same physical memory.

Lazy Tag Region Mapping. Instead of mapping every tag region page to physical memory, Bastag initially maps only the tag region page that needs to be tagged. Not mapping such pages does not harm Bastag's access control as accessing unmapped tag region will naturally fault, showing the same behavior as one will expect from a invalid shared memory access check through the mapped tag region with memory tag 0. Bastag lazily maps such tag region pages on-demand when the access permissions are updated (*i.e.*, require the shadow tags to have non-zero value).

5.6 Bastag APIs

Bastag provides the following APIs available for developers to manage shared memory and access permissions:

```
void bastag_enter(int domain_id);
void bastag_exit();
bool bastag_register(void *ptr, size_t size);
bool bastag_set(void *ptr, size_t size, int p);
void bastag_enable(void *ptr, size_t size);
void bastag_destroy(void *ptr, size_t size);
```

To explain the API usage, we use the vulnerable kernel driver from Section 3 as an example, where we employ Bastag to enforce access control on the memory object `ringbuf`, shared between the driver and the main kernel. By registering the target object as shared and granting appropriate access permission on its sub-field, the hardened driver is restricted from modifying the shared object to access the kernel's private memory. Listing 8 shows the simplified code snippet that contain the modifications (marked with "+").

The first two Bastag enter and exit APIs are used at the start and the end of shared memory access, registration, and access permission setting as depicted in Line 5 and 16 in Listing 8. The `bastag_enter` API updates the reserved registers, `X21` and `X22`, to hold the base addresses of tag regions with pointer tag M. Additionally, pointer tags of reserved registers, `X23` and `X24`, are configured to N and zero, respectively. On its initial invocation, `bastag_enter` reserves 4KB tag regions for the domain identified by `domain_id` and records the associated (*domain_id*, *base*, *size*) tuple. This information is later consulted to prevent overlaps when reserving tag regions for additional domains. The `bastag_exit` API clears the reserved registers to zero. `Enter` and `exit` APIs must be used in pairs without nesting and the pair must be called inside a same function. The `bastag_register` API registers the memory range from *ptr* to *ptr+size* as shared. It maps the corresponding tag region pages, as they may be tagged to grant access to the associated memory. If the new registration overlaps an existing region, it allocates a new tag region base and synchronizes its contents with the current one. `bastag_set` serves to grant and revoke access permissions p from *ptr* to *ptr+size* at a byte granularity. This is achieved by appropriately tagging (with tag M) or untagging the shadow tags. For example, given a shared memory object `ringbuf`, `ro` permission is assigned to `rx_control` sub-field (Line 9), whereas `rw` permission is granted to `buffer` sub-field. `bastag_enable` API enables the tag region sharing (Section 5.5) to tag region for *ptr* to *ptr+size*. This API is to be used after `bastag_register` and `bastag_set` to correctly find the identical access permission patterns in the tag regions.

```
1 // X1: holds original branch target address
2 BFI     X24, X1, #0, #48
3 LDR     XZR, [X24]
4 BLR     X24
```

Listing 9: Instrumentation on indirect jumps outside of enter-exit API pairs to prevent illegal jumps to APIs.

`bastag_destroy` untags the domain's tag region corresponding to the [*ptr*, *ptr+size*]. The API also checks whether the corresponding tag region page has tagged memory blocks, unmapping the page if all blocks are untagged. In order to reflect the initialization and modifications to tag regions correctly, `register`, `set`, `enable`, and `destroy` APIs must be used inside the enter-exit pairs. The APIs configure the tag region for appropriate domain through `X21`.

Preventing Malicious API Calls. Malicious API calls by manipulating the target addresses of indirect jump/calls to alter the access permissions can lead up to unauthorized shared memory accesses. To cope with this threat, BASTAG first assigns a non-zero memory tag to code blocks that calls the APIs according to programmer's intention and code blocks of APIs in BASTAG library. Then, BASTAG instruments the indirect jump/call instructions outside the enter-exit API pairs as depicted in Listing 9. In the example, `X1` holds the original branch target address. BASTAG first moves the target address to `X24` using `BFI` instruction (Line 2). This preserves the valid address and fixes the pointer tag to zero by using the reserved register `X24`. Next, `LDR` is inserted with `X24` as the memory address operand (Line 3), which triggers a tag comparison between pointer tag zero and memory tag of the code block at branch target address. As the code blocks for APIs are tagged with non-zero memory tag, any malicious attempts to call the APIs will cause a tag mismatch. The indirect jumps will be conducted through `X24`, thus preventing `BFI` or `LDR` bypassing as the register is reserved, only after the tag checking is passed (Line 4).

6 Implementation

BASTAG consists of a compiler to enforce access control and a verifier to ensure that the machine code generated with the compiler behaves as intended. Minor tweaks are made to the Linux kernel to provide BASTAG for kernel extensions as well. As the design mandates register reservation for the proper functioning, BASTAG supports instrumented versions of the LLVM/musl C/C++ toolchain and libraries (musl-libc [66] v1.2.4, compiler-rt, libc++, libc++abi, and libunwind [48]).

6.1 BASTAG Compiler

BASTAG compiler is implemented on top of SVF [74], a static value flow analysis tool based on Anderson's analysis, using LLVM 10.0.0. BASTAG implements two passes for access control enforcement - one at LLVM intermediate representation (IR) level to locate loads and stores onto shared objects, and another at AArch64 LLVM backend to insert necessary instructions. IR pass first locates initialization sites of shared objects specified by API (*i.e.*, bastag_register). The pass then traverses the value flow graph (VFG) starting from the initialization site to find accesses to the object. The VFG consists of def-use chains of the values (*i.e.*, pointers/objects), thereby having load and store IRs related to the tracked value as its graph nodes. As a result, VFG traversal allows BASTAG to locate the accesses to the objects of interest - shared objects in this case. To inspect shared object accesses across modules (*i.e.*, files), BASTAG takes a similar approach as in KSplit [37] that analyzes shared data between core kernel and device drivers. BASTAG first collects the types of objects (*i.e.*, struct types) that are initialized as shared objects from all modules. Next, struct types that are accessible transitively through function parameters are collected. Objects of the types that appear in both steps above are considered as shared and go through VFG traversal when they are found as function arguments. Additionally, SVF is modified to insert a virtual edge between `PtrToInt` and `IntToPtr` instruction since the original SVF terminates the value flow when the pointer is converted to `int` data type and recognizes conversion back to a pointer as a new allocation. The load and store IR instructions found during the traversal are marked with metadata to pass the information to the compiler backend. The metadata is propagated by modifying the instruction selection (ISel) pass to convey IR metadata to emitted backend instructions. Lastly, BASTAG backend pass locates instructions with metadata and inserts access control instructions accordingly. Additionally, BASTAG compiler checks whether all `enter` and `exit` API pairs are properly closed by constructing a a control flow graph and exploring the code path with `enter` API as an entry point. This check is sound, as the enter-exit pair must be called inside a same function.

6.2 BASTAG Verifier

BASTAG adapts the verifier offered by the latest SFI framework for ARM [85] which utilizes the Binary Ninja disassembler [12], an ELF reader, and ARM's Machine Readable Specification (MRS) [9] to verify the following properties:

- Memory accesses that use `X23` as a memory operand register have correct instrumentation as described in Listing 6.
- Memory accesses that use `X24` as a memory operand register is conducted after correct `BFI` instrumentation.
- `X21-X24` are only modified through the BASTAG APIs.
- Instructions for manipulating MTE memory tags, such as `LDG` and `STG`, are solely employed by the BASTAG APIs.
- Indirect calls and returns that uses `X23` is conducted after correct `BFI/LDR` instrumentation.

6.3 Tag Regions for Kernel Use Case

BASTAG reserves the address space for tag regions by resizing the `vmalloc` region that is used to allocate virtually contiguous memory. The prototype implementation resizes the region to reserve 1TB for BASTAG, by adjusting `VMALLOC_END` constant in the kernel source. Additionally, BASTAG makes use of a custom page mapping function to allocate tag regions within the reserved address space, thereby avoiding collision with any other normal kernel allocations.

6.4 Integration with Isolation Mechanisms

We describe the integration of BASTAG into two prominent isolation techniques — Software Fault Isolation (SFI) and Page Table (PT)-based mechanisms. Later in Section 7.5, we demonstrate the integration of BASTAG into the latest state-of-the-art SFI [85].

Integration with SFI. SFI techniques revolve around instrumenting every memory access within the domain to confine them within a predefined region, commonly referred to as the sandbox [43, 61, 67, 78, 85, 87]. This is typically accomplished by inserting additional instruction(s) preceding the memory access. Upon integration, BASTAG supersedes such instrumentation for shared memory, replacing it with the instruction sequence described in Listing 6. For private memory accesses, BASTAG leverages `X24` (Section 5.3) in addition to the original instrumentation for SFI. As a result, the isolation model is modified to allow access to shared memory under the control of BASTAG, while maintaining isolation for private memory accesses.

Integration with PT-based Isolation. PT-based techniques allocate a group of pages exclusively to a specific domain [18, 83, 92]. In this scheme, shared memory functionality is achieved by designating a page to be accessible from multiple domains. In this context, the integration of BASTAG is straightforward, requiring instrumentation to be applied to private and shared memory accesses as described in Section 5. Consequently, BASTAG refines the level of access control from page-level to byte-level while simultaneously enforcing different permissions across multiple domains.

7 Evaluation

We evaluate BASTAG using both micro and macro benchmarks. For macro benchmarks, we present three case studies that represent the real world usage scenarios of BASTAG. Moreover, we assess the performance of BASTAG when integrated with the latest SFI technique using SPEC2017rate. All experiments are built with `-O2`.

Experimental Setup. We conduct our experiments on an Pixel 8 [34] with 2.45GHz ARMv9.0-A Cortex-A715 quad-core processor and 8GB RAM. We use Linux kernel version 5.15.110 as an OS. All analysis and instrumentations are applied using the LLVM 10.0.0 compiler framework [47] and SVF commit version @a03400e.

Comparison Systems. In addition to the native baseline (i.e., no access control), we compare BASTAG with a message-based scheme and a software-only IRM-based solution described in Section 3.1. The message-based approach maintains per-domain private copies of shared objects, synchronizing them via `memcpy` on domain crossings, and requires RPC for multi-domain coordination. For the IRM-based solution, we emulate BGI [17] on ARM by porting its x86 permission-checking sequence (*CheckRight*) to equivalent ARM instructions (Listing 4). Our emulation captures BGI's optimal overheads, excluding failure-handling paths.

7.1 Micro-benchmarks

Access Control Overheads. We measure the cost of enforcing access control on shared memory by timing 100 per-byte reads of a 256-byte block across different mechanisms. For BASTAG and IRM-based schemes, each read is instrumented with the corresponding instruction sequence. For message-based schemes, the block is first copied to a private buffer using `memcpy`. To isolate overheads, we flush the cache on each iteration and use the `cntvct_el0` register [7] to record timing. The benchmark runs 100 times (10,000 reads total), and we report average cycles in Table 2. BASTAG incurs 8.1% overhead compared to a baseline with no access control, outperforming IRM-based enforcement (45.9%) due to reduced instruction count and lack of ACL lookup latency. Message-based

Mechanism	Baseline	IRM-based	Msg-based	**BASTAG**
$\Delta Counter$	37	54	78	40

Table 2: Access control overheads.

	32B	64B	128B	256B	512B
IRM-based	4ms	8ms	12ms	21ms	45ms
Msg-based	82ms	82ms	81ms	82ms	81ms
BASTAG	3ms	6ms	10ms	17ms	32ms

Table 3: Access permission update overheads.

schemes show over 2× slowdown from `memcpy` overhead, even without marshalling glue code—making this a best-case for them.

Overheads of Changing Access Permissions. We evaluate the cost of changing byte-granular access permissions from `rw` to `ro` over varying shared memory sizes. BASTAG performs this by untagging memory in the write tag region, while the IRM-based scheme emulates ACL updates. For message-based schemes, we use `mprotect`, as their glue code cannot be dynamically modified. Each result is averaged over 100,000 runs. As shown in Table 3, BASTAG achieves the lowest latency for per-byte permission updates. Its overhead is comparable to IRM-based schemes, as both rely on memory stores. While BASTAG 's cost scales with memory size, it outperforms alternatives up to 512 bytes. For BASTAG, the evaluation untags the memory tags in the write tag region while for IRM-based scheme, we emulate the code sequence that updates the access control list. Table 3 shows the results.

Comparison vs. SW-only Shadow Memory-based Schemes. Additionally, we evaluate the performance benefits of MTE-based instrumentation over software-only shadow memory-based access control (as described in Section 5.3), by measuring the execution cycles of three instruction sequences used to enforce access control under both cache-hit and cache-miss scenarios:

- `LDR`: An idealized software-only shadow memory implementation, where the shadow memory holds the target memory address if access is permitted and `NULL` otherwise. A subsequent access using the loaded address will trigger a fault if the access is invalid.
- `LDR+CMP`: A typical implementation where access permissions are explicitly loaded from shadow memory and compared against a predefined value to determine whether the access is valid.
- `STR` *with MTE enabled*: BASTAG, where the `STR` instruction triggers a hardware-enforced tag check via MTE, implicitly validating the access based on the tag associated with the memory location.

Figure 3-(a) and (b) show the `cntvct_el0` cycles required to execute each instruction sequence 1,000 times under cache-hit and cache-miss conditions, respectively. Under a cache hit, `LDR` incurs 1.6× more cycles than `STR`, and 1.68× more under a cache miss. In both scenarios, `LDR+CMP` exhibits even higher overhead due to the additional comparison, taking over 1.7× more cycles. These results indicate that BASTAG's MTE-based instrumentation outperforms software-only shadow memory schemes, benefiting from implicit hardware tag checks and microarchitectural store buffers that defer stores without stalling the pipeline. We further compare the relative overhead of `LDR+CMP` plus N additional instructions ($N = 0, 1, 2, 3$), modeling the ACL lookup cost typical in shadow memory schemes (*e.g.*, [17]), against BASTAG 's `BFI+STR`, as shown in Figure 3-(c). As

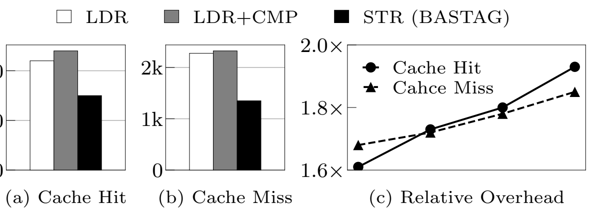


Figure 3: BASTAG compared to software-only shadow memory-based schemes in terms of cycles taken for different instruction sequences under (a) cache hit and (b) cache miss.

	STR	STG (BASTAG)	Relative Overhead
Cache Hit	14	14	< 1%
Cache Miss	1457	1484	1.85%

Table 4: Cost of access control updates in terms of cycles.

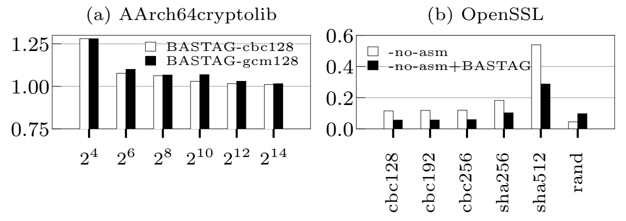


Figure 4: Cost of register reservation on cryptographic code.

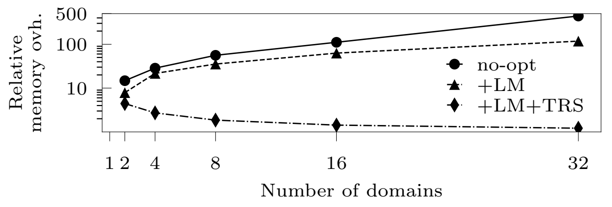


Figure 5: Relative memory overhead with respect to the baseline. (LM-*Lazy Mapping* and TRS-*Tag Region Sharing*).

expected, the overhead increases with each additional instruction, reaching up to 1.93× under cache miss.

We also evaluate the cost of BASTAG's access control updates via memory tag assignment (*i.e.*, STG) compared to updating shadow memory via STR, under both cache-hit and cache-miss conditions, as shown in Table 4. The results show that assigning a memory tag incurs a similar number of cycles as a regular memory store, with less than 1% overhead under a cache hit and only 1.85% under a cache miss—indicating that the difference is negligible in practice.

Impact of Register Reservation. We evaluate the impact of register reservation using AArch64cryptolib [8] and OpenSSL [75]. For AArch64cryptolib, we reserve registers not used by its inline assembly to avoid interference, and run AES-CBC-128 and AES-GCM-128 over varying input sizes. As shown in Figure 4-(a), which reports normalized execution time (lower is better), the overhead is modest—5.3% for CBC and 6.8% for GCM. To examine a extreme-register-pressure scenario, we benchmark OpenSSL compiled with `-no-asm` (sw-only) with and without register reservation. Figure 4-(b) shows normalized throughput (higher is better): removing assembly reduces performance to 0.18×, and reserving registers lowers it further to 0.11×. This represents a worst-case scenario. In practice, such high register pressure is rare and localized, while BASTAG targets general-purpose code with significantly lower pressure.

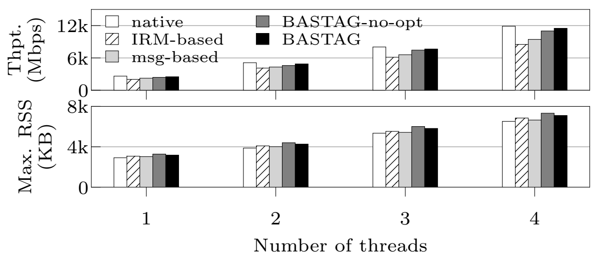


Figure 6: Throughput and memory utilization of `nullnet`.

Impact of Optimizations. We evaluate the impact of optimizations using a microbenchmark in which messages are read by varying numbers of domains. Each thread represents a domain with its own read and write tag regions. We allocate one million 256B messages, each writable by a single domain and read-only for the others, evenly distributing writable messages across domains. Threads with write access read messages byte-by-byte. We measure the maximum RSS after processing (Figure 5). Without optimizations, memory overhead grows with the number of domains due to per-domain tag region mappings. Lazy tag region mapping reduces this by avoiding write mappings for read-only messages. Tag region sharing further reduces overhead by mapping identical read/write tag regions to the same physical page when access permissions are uniform across domains, yielding a 1.21× reduction at 32 domains. This benchmark assumes all memory is shared, excluding private allocations, to stress tag region overhead.

7.2 Case Study 1: Kernel and Extensions

We evaluate the cost of enforcing access control between the kernel and extensions using two device drivers, `nullblk` and `nullnet`, both commonly used to assess driver isolation mechanisms [55, 56]. These allow us to stress-test BASTAG without hardware constraints. Shared structure accesses are instrumented for both BASTAG and an IRM-based scheme. For comparison, we implement a message-based synchronization mechanism by extending the marshalling layer from LVD [56]. Leveraging Ksplit [37] to identify shared struct fields, we apply `rw` permissions to 406 fields in `nullblk` and 156 in `nullnet`; remaining fields are marked `ro`.

Dummy Network Driver (nullnet). We evaluate `nullnet` using `iperf3` [32] with 1–4 threads, reporting TCP transmit bandwidth and maximum RSS averaged over 10 runs (Figure 6). BASTAG achieves 94.3% of native throughput (2478 Mbps for a single thread), outperforming IRM-based (77.1%) and message-based (85.3%). The latter incurs increasing overhead with more threads (20.7% degradation with 4 threads) due to increased marshalling costs when managing more threads. BASTAG's memory overhead ranges from 10.4–11.8%[2], which is reduced to 4.3–7.9% with optimizations.

Multi-Queue Block Device Driver. We evaluate `nullblk` using `fio` [39] with a 512-byte block size. BASTAG achieves 96.8% of native performance (55.3 kops for a single thread), outperforming the IRM-based (47.9 kops, 83.9%) and message-based (43.4 kops, 76%) mechanisms, as shown in Figure 7. While BASTAG and IRM-based approaches maintain stable overheads across thread counts, the

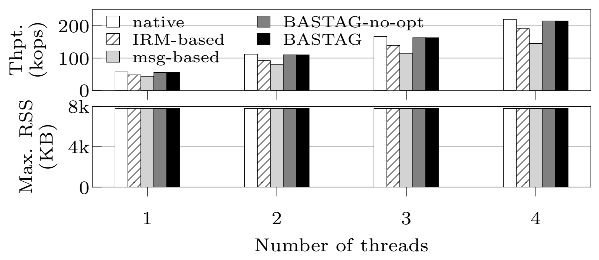


Figure 7: Throughput and memory utilization of `nullblk`.

	vehicle	sensor-gyro	adc-report	esc-status
IRM	1.2	1.18	1.22	1.24
Bastag	1.07	1.05	1.09	1.07

Table 5: Normalized latency to read uORB messages.

message-based scheme degrades significantly due to increased domain crossings to perform the I/O as shared objects need to be marshaled and synchronized in the occurrence of domain crossing. Memory overheads are negligible across all mechanisms, as the benchmark saturates available physical memory.

7.3 Case Study 2: Inter-task Communication

In the second case study, we consider memory sharing scenario where multiple tasks in PX4 [63] middleware that communicate through messages via uORB [65] protocol. We investigate four type of messages from PX4 uORB publication/subscription graph [64] that are accessed by multiple tasks where each task uses different set of subfields of the message struct. We compare Bastag's performance with IRM-based mechanism, in terms of latency to read 10,000 messages, when the access control is enforced at a byte-level. We omit the comparison with message-based mechanism as uORB already copies the message (without access control over its subfields) via publication and subscription intrinsics. Table 5 shows the normalized performance. Bastag demonstrates 7.1% overheads on average across four message types, outperforming IRM-based (21.1% overheads on average).

7.4 Case Study 3: Multi-threaded Application

To understand end-to-end overheads of Bastag on multi-threaded applications, we conduct evaluations using Memcached v1.6.28. We enforce access control on thread-specific metadata and globals, similar to shared objects such as `CQ_ITEM` discussed in Section 3. We use the memaslap [14] load generator to send random TCP requests to the server (10% set and 90% get) running with two service threads while varying the number of concurrent connections. Figure 8-(a) shows the throughput in operations per second (ops). Bastag incurs 5.75% overheads on average with peak throughput of 123kops compared to the native performance, outperforming IRM-based mechansism that shows 16.98% overheads on average. The memory overhead comes from managing the tag regions for dispatch queues where `CQ_ITEM` is pushed from the main thread and popped from the worker thread, showing 15.3%[2] on average as the maximum size of the queue is determined. Figure 8-(b) shows the memory overheads after the optimizations with respect to the overheads without them. While the effect of optimizations is dependent on the workload characteristics, general trend shows that the optimizations suppresses the memory overheads from increasing linearly due to the additional tag regions for the new domains.

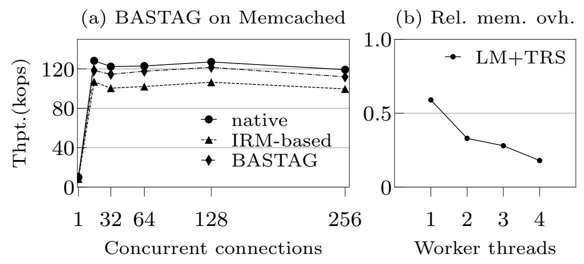


Figure 8: (a) Performance of Bastag on Memcached and (b) memory overheads relative to those without optimizations.

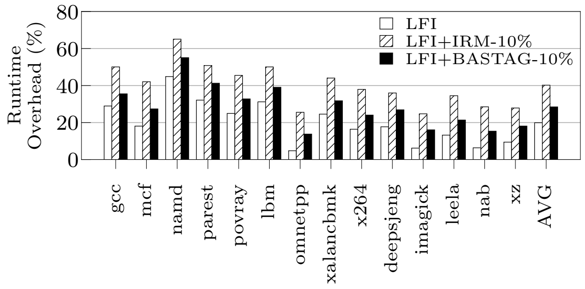


Figure 9: Performance overheads of Bastag integrated with [85] on SPEC2017 with varying ratio of shared memory.

7.5 Integration with Isolation Technique

To evaluate Bastag in conjunction with cooperative isolation on private memory, we integrate it into the latest SFI framework [85] by replacing private memory instrumentation with Bastag 's shared memory instrumentation (Section 6.4), and run it on SPEC2017rate. We set the proportion of shared objects to 10%, granting `rw` permissions at allocation time. Byte-level access control is applied, and results are averaged over ten runs (Figure 9). With 10% shared objects, Bastag incurs 8.55% overhead on top of SFI (19.9%), outperforming the IRM-based scheme, which incurs 20.27%. Bastag shows 2.77× memory overhead at 10% due to the accumulation of tag regions, which are not released on `free` due to lack of object size tracking. This prevents optimization and causes tag regions to persist, potentially increasing memory usage up to 32×. We expect significantly lower overheads with proper untagging, as seen in prior evaluations.

7.6 Comparison with SW-only Shadow Memory

To evaluate Bastag's performance advantage from MTE-based instrumentation over software-only shadow memory mechanisms, we compare it against AddressSanitizer [70] on SPEC2006. For a fair comparison, we (1) disable logging and syscall interception, (2) remove stack protection from ASan LLVM IR pass, and (3) set quarantine_size_mb=0 to disable heap quarantine for use-after-free detection, retaining only sanitizer checks [91]. ASan redzones are configured to 32 bytes and updated on `malloc` and `free`. We

[2]Memory overhead includes the cost of memory tags. On Pixel 8, the physical memory used for MTE tags is reserved by the bootloader [3] and is not reflected in maximum RSS. To account for this, we report the memory tag overhead used in the tag regions separately and add it to the measured RSS for each benchmark.

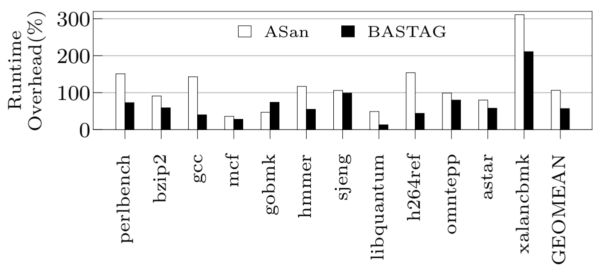


Figure 10: Overheads of ASan and Bastag on SPEC2006.

preserve ASan's memory access checks and instrument the same LLVM IR locations for Bastag. Analogous to ASan's redzone updates, Bastag tags memory on allocation and clears tags on deallocation. The results are shown in Figure 10. As shown in Figure 10, Bastag incurs 57% overhead on average, compared to 106% for ASan. This comparison highlights the performance benefit of MTE-based instrumentation for access control, though it is not a direct feature-for-feature comparison: ASan provides broader memory safety coverage, while Bastag is tailored for supporting multi-domain access control on shared memory.

7.7 Security Analysis

An adversary interested in compromising the access control provided by Bastag may attempt to manipulate either the pointer tags or memory tags. However, Bastag 's instrumentation of compelling memory accesses to be conducted through reserved registers (Section 5.3) prevents any arbitrary pointer tag manipulations from circumventing Bastag. An adversary may also try to directly change the memory tags of the tag regions. Such attack vector is not applicable to Bastag since the memory tags can only be altered by specific MTE instruction (*i.e.*, `STG`) which is prevented from being emitted during the compile time and is verified by the verifier (Section 6.2). An adversary interested in compromising the access control provided by Bastag may also attempt to conduct the attack by leveraging software vulnerabilities such as control flow hijacking or buffer overflows. However, an adversary does not have the capability to neither control the target address of the memory access nor leverage private accesses as Bastag binds the memory accesses to reserved registers with preset tags (Section 5.3).

8 Related Works

Shadow Memory. Bastag relates to tools like AddressSanitizer (ASan)[70], Valgrind[58], and their optimized variants [16, 44], which use shadow memory to enforce memory safety. Unlike these, Bastag uses shadow memory only to associate tags with memory via lightweight indirection. Prior approaches load metadata from shadow memory for policy enforcement, whereas Bastag attaches shadow tags and triggers hardware-based MTE tag comparisons through dummy stores, eliminating costly loads (Section 5.3).

Techniques for Spatial Safety. Mechanisms like SoftBound [53], Baggy Bounds [1], and BOGO [90] enforce spatial safety using fat pointers or Intel MPX. However, they lack support for multi-domain policies like `rw`, `ro`, and `na`, which are essential for fine-grained shared memory control across domains, as provided by Bastag.

Tagged Architectures. Tagged architectures [29, 72, 77, 80–82, 89] offer hardware-enforced isolation. Examples include Loki [89], HDFI [72], CODOMs [77], TIMBER-V [81], and CHERI [80]. Most remain prototypes or target private memory isolation. In contrast, Bastag targets fine-grained, multi-policy access control over shared memory on commodity ARM hardware.

Applications of ARM MTE. Several works apply ARM MTE to enhance memory safety. HAKC [51] combines MTE and Pointer Authentication for kernel compartmentalization, and SFITAG [68] isolates kernel extensions. Capacity [30] and PeTAL [40] enforce intra-process isolation using MTE. Unlike these, Bastag extends MTE beyond its native granularity to support byte-level, policy-rich access control between domains.

9 Conclusion

Bastag is an efficient, byte-level access control system for shared memory that addresses the limitations of previous approaches by leveraging MTE. While a direct usage of MTE for access control faces limitations such as insufficient granularity and limited types of access permission, Bastag introduces a concept of shadow memory tagging. By placing MTE tags associated with shared memory in separate tag regions, Bastag achieves byte-level access control and supports multiple access permissions across domains. Evaluation results on realistic use cases demonstrate the versatility and practicality of Bastag, outperforming existing mechanisms in terms of overhead. Bastag presents a promising solution for efficient access control in shared memory scenarios.

Acknowledgments

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2024-00337414, Binary Micro-Security Patch Technology Applicable with Limited Reverse Engineering Capability under SW Supply Chain Environments, No.RS-2024-00438729, Development of Full Lifecycle Privacy-Preserving Techniques using Anonymized Confidential Computing, No.2021-0-00528, Development of Hardware-centric Trusted Computing Base and Standard Protocol for Distributed Secure Data Box, and IITP-2023-RS-2023-00256081, under the artificial intelligence semiconductor support program to nurture the best talents); in part by Korea Planning & Evaluation Institute of Industrial Technology (KEIT) grant funded by the Korea government (MOTIE) (No.RS-2024-00406121, Development of an Automotive Security Vulnerability-based Threat Analysis System (R&D)); in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2023-00277326); in part by Inter-University Semiconductor Research Center (ISRC); and in part by the BK21 FOUR program of the Eduction and Research Program for Future ICT Pioneers, Department of Electrical and Computer Engineering, Seoul National University in 2025.

References

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Conference on USENIX Security Symposium* (Montreal, Canada) *(SSYM'09)*. USENIX Association, USA, 51–66.

[2] Android. 2023. Arm Memory Tagging Extension. https://source.android.com/docs/security/test/memory-safety/arm-mte. [Online; accessed May-2023].

[3] Android Open Source Project. 2023. Bootloader Support for Memory Safety. https://source.android.com/docs/security/test/memory-safety/bootloader-support. Accessed: 2025-05-05.
[4] Android Open Source Project. 2023. Tagged Pointers. https://source.android.com/docs/security/test/tagged-pointers. Accessed: May-2023.
[5] arm. 2023. Cortex-A510. https://developer.arm.com/Processors/Cortex-A510. [Online; accessed Nov-2023].
[6] arm. 2023. Cortex-A715. https://developer.arm.com/Processors/Cortex-A715. [Online; accessed Nov-2023].
[7] Arm Ltd. 2022. CNTVCT_EL0 – Counter-timer Virtual Count Register. https://developer.arm.com/documentation/ddi0601/2022-12/AArch64-Registers/CNTVCT-EL0--Counter-timer-Virtual-Count-register. Accessed: 2025-06-18. From: ARMv8-A Architecture Reference Manual, DDI 0601 (2022-12).
[8] Arm Ltd. 2023. AArch64 Crypto Library. https://github.com/ARM-software/AArch64cryptolib. Accessed: 2025-05-05.
[9] Arm Ltd. 2024. Arm Development Tools for Architecture Exploration. https://developer.arm.com/downloads/-/exploration-tools. Accessed: May-2024.
[10] Arm Ltd. 2024. The Memory Management Unit: Memory Attributes – Domains. https://developer.arm.com/documentation/den0013/d/The-Memory-Management-Unit/Memory-attributes/Domains. Accessed: May-2023. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition.
[11] Ying Bai. 2016. *ARM® Memory Protection Unit (MPU)*. Wiley-IEEE Press, Hoboken, New Jersey, USA, 951–974. doi:10.1002/9781119058397.ch12
[12] Binary Ninja Team. 2021. Ground Up: AArch64. https://binary.ninja/2021/04/05/groundup-aarch64.html. Accessed: May-2023.
[13] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, San Francisco, CA, USA, 309–322.
[14] Brian Aker and contributors. 2024. libmemcached: C Client Library for memcached. https://libmemcached.org/libMemcached.html. Accessed: May-2024.
[15] David Brumley and Dawn Song. 2004. Privtrans: automatically partitioning programs for privilege separation. In *Proceedings of the 13th Conference on USENIX Security Symposium* (San Diego, CA). USENIX Association, USA, 5.
[16] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 985–999.
[17] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. 2009. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 45–58.
[18] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-Grained Execution Units with Private Memory . In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 56–71.
[19] Inc. Cisco Systems. 2021. CVE-2020-3347 Detail. https://nvd.nist.gov/vuln/detail/CVE-2020-3347. [Online; accessed May-2023].
[20] Inc. Cisco Systems. 2021. CVE-2021-34758 Detail. https://nvd.nist.gov/vuln/detail/CVE-2021-34758. [Online; accessed May-2023].
[21] Jonathan Corbet. 2015. Memory protection keys. https://lwn.net/Articles/643797/. [Online; accessed May-2023].
[22] National Vulnerability Database. 2016. CVE-2016-8633 Detail. https://nvd.nist.gov/vuln/detail/CVE-2016-8633.
[23] National Vulnerability Database. 2021. CVE-2021-21309 Detail. https://nvd.nist.gov/vuln/detail/CVE-2021-21309.
[24] National Vulnerability Database. 2022. CVE-2022-21765 Detail. https://nvd.nist.gov/vuln/detail/CVE-2022-21765.
[25] National Vulnerability Database. 2022. CVE-2022-21769 Detail. https://nvd.nist.gov/vuln/detail/CVE-2022-21769.
[26] National Vulnerability Database. 2022. CVE-2022-48198 Detail. https://nvd.nist.gov/vuln/detail/CVE-2022-48198.
[27] National Vulnerability Database. 2023. CVE-2023-2008 Detail. https://nvd.nist.gov/vuln/detail/CVE-2023-2008.
[28] Gelei Deng, Guowen Xu, Yuan Zhou, Tianwei Zhang, and Yang Liu. 2022. On the (In)Security of Secure ROS2. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 739–753.
[29] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Benjamin C. Pierce, and André DeHon. 2014. PUMP: a programmable unit for metadata processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy* (Minneapolis, Minnesota, USA) *(HASP '14)*. Association for Computing Machinery, New York, NY, USA, Article 8, 8 pages.
[30] Kha Dinh Duy, Kyuwon Cho, Taehyun Noh, and Hojoon Lee. 2023. Capacity: Cryptographically-Enforced In-Process Capabilities for Modern ARM Architectures. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 874–888. doi:10.1145/3576915.3623079
[31] Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J. Walls, and John Criswell. 2022. Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2281–2298.
[32] ESnet. 2024. iPerf3: A TCP, UDP, and SCTP Network Bandwidth Measurement Tool. https://github.com/esnet/iperf. Accessed: May-2024.
[33] FreeBSD. 2019. CVE-2017-1087 Detail. https://nvd.nist.gov/vuln/detail/CVE-2017-1087. [Online; accessed May-2023].
[34] Google. 2023. Meet Pixel 8 and Pixel 8 Pro, our newest phones. https://blog.google/products/pixel/google-pixel-8-pro/. [Online; accessed Nov-2023].
[35] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 489–504.
[36] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 393–405. doi:10.1145/2976749.2978327
[37] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2022. KSplit: Automating Device Driver Isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 613–631.
[38] Jinsoo Jang and Brent Byunghoon Kang. 2019. In-process Memory Isolation Using Hardware Watchpoint. In *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA) *(DAC '19)*. Association for Computing Machinery, New York, NY, USA, Article 32, 6 pages. doi:10.1145/3316781.3317843
[39] Jens Axboe and fio contributors. 2024. fio: Flexible I/O Tester. https://git.kernel.dk/cgit/fio/. Accessed: May-2024.
[40] Juhee Kim, Jinbum Park, Yoochan Lee, Chengyu Song, Taesoo Kim, and Byoungyoung Lee. 2024. PeTAL: Ensuring Access Control Integrity against Data-only Attacks on Linux. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 2919–2933.
[41] Juhee Kim, Jinbum Park, Sihyeon Roh, Jaeyoung Chung, Youngjoo Lee, Taesoo Kim, and Byoungyoung Lee. 2025. Tiktag: Breaking ARM's Memory Tagging Extension with Speculative Execution . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 4063–4081.
[42] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.
[43] Joshua A. Kroll, Gordon Stewart, and Andrew W. Appel. 2014. Portable Software Fault Isolation. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium (CSF '14)*. IEEE Computer Society, USA, 18–32.
[44] Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. 2024. GIANTSAN: Efficient Memory Sanitization with Segment Folding. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 433–449.
[45] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, et al. 2020. Meltdown: Reading kernel memory from user space. *Commun. ACM* 63, 6 (2020), 46–56.
[46] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 49–64.
[47] LLVM Project. 2020. LLVM 10.0.0 Release Notes. https://releases.llvm.org/10.0.0/docs/ReleaseNotes.html. Accessed: May-2023.
[48] LLVM Project. 2024. Clang Compiler User's Manual: Toolchain. https://clang.llvm.org/docs/Toolchain.html. Accessed: May-2024.
[49] Arm Ltd. 2019. *ARMv8.5-A Memory Tagging Extension*. Technical Report. Arm Architecture Design Group. https://developer.arm.com/documentation/102925/latest/ White Paper.
[50] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 115–128. doi:10.1145/2043556.2043568
[51] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. 2022. Preventing kernel hacks with HAKC. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, Vol. 22. NDSS, San Diego, CA, USA, 1–17.
[52] Memcached Development Team. 2024. Memcached: A Distributed Memory Object Caching System. https://memcached.org/. Accessed: May-2023.

[53] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 245–258. doi:10.1145/1542476.1542504

[54] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean Tullsen, and Deian Stefan. 2023. Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 266–281. doi:10.1145/3582016.3582023

[55] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. 2019. LXDs: Towards Isolation of Kernel Subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 269–284.

[56] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2020. Lightweight kernel isolation with virtualization and VM functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 157–171. doi:10.1145/3381052.3381328

[57] Nicholas Nethercote and Julian Seward. 2007. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (San Diego, California, USA) *(VEE '07)*. Association for Computing Machinery, New York, NY, USA, 65–74. doi:10.1145/1254810.1254820

[58] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.

[59] Open Source Robotics Foundation. 2024. ROS 2 Humble: DDS Implementations. https://docs.ros.org/en/humble/Installation/DDS-Implementations.html. Accessed: May-2023.

[60] Open Source Robotics Foundation. 2024. ROS 2: The Robot Operating System. https://github.com/ros2. Accessed: May-2023.

[61] Gregor Peach, Runyu Pan, Zhuoyi Wu, Gabriel Parmer, Christopher Haster, and Ludmila Cherkasova. 2020. ewasm: Practical software fault isolation for reliable embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3492–3505.

[62] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing privilege escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12* (Washington, DC) *(SSYM'03)*. USENIX Association, USA, 16.

[63] PX4 Autopilot Contributors. 2024. PX4 Middleware Architecture. https://docs.px4.io/main/en/middleware/. Accessed: May-2023.

[64] PX4 Autopilot Contributors. 2024. uORB Graph Visualization. https://docs.px4.io/main/en/middleware/uorb_graph.html. Accessed: May-2024.

[65] PX4 Autopilot Contributors. 2024. uORB Messaging Architecture. https://docs.px4.io/main/en/middleware/uorb.html. Accessed: May-2024.

[66] Rich Felker and musl contributors. 2024. musl libc: Lightweight, fast, simple, free-as-in-libc. https://musl.libc.org/. Accessed: May-2024.

[67] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting software fault isolation to contemporary CPU architectures. In *Proceedings of the 19th USENIX Conference on Security* (Washington, DC) *(USENIX Security'10)*. USENIX Association, USA, 1.

[68] Jiwon Seo, Junseung You, Yungi Cho, Yeongpil Cho, Donghyun Kwon, and Yunheung Paek. 2023. Sfitag: Efficient Software Fault Isolation with Memory Tagging for ARM Kernel Extensions. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security* (Melbourne, VIC, Australia) *(ASIA CCS '23)*. Association for Computing Machinery, New York, NY, USA, 469–480.

[69] Kostya Serebryany. 2019. ARM memory tagging extension and how it improves C/C++ memory safety. *The Usenix Magazine* 44, 2 (2019), 12–16.

[70] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

[71] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. arXiv:1802.09517 [cs.CR] https://arxiv.org/abs/1802.09517

[72] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. 2016. HDFI: Hardware-assisted data-flow isolation . In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–17. doi:10.1109/SP.2016.9

[73] Raoul Strackx, Pieter Agten, Niels Avonds, and Frank Piessens. 2015. Salus: Kernel Support for Secure Process Compartments. *EAI Endorsed Transactions on Security and Safety* 2, 3 (01 2015).

[74] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC '16)*. Association for Computing Machinery, New York, NY, USA, 265–266. doi:10.1145/2892208.2892235

[75] The OpenSSL Project. 2023. OpenSSL: Cryptography and SSL/TLS Toolkit. https://github.com/openssl/openssl. Accessed: 2025-05-05.

[76] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1221–1238.

[77] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting software with code-centric memory domains. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 469–480.

[78] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, USA) *(SOSP '93)*. Association for Computing Machinery, New York, NY, USA, 203–216.

[79] Jun Wang, Xi Xiong, and Peng Liu. 2015. Between mutual trust and mutual distrust: practical fine-grained privilege separation in multithreaded applications. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA) *(USENIX ATC '15)*. USENIX Association, USA, 361–373.

[80] Robert NM Watson, Peter G Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, et al. 2019. *Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (version 7)*. Technical Report. University of Cambridge, Computer Laboratory.

[81] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. 2019. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v.. In *NDSS*. NDSS, San Diego, CA, USA, –.

[82] Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 304–316.

[83] Jiali Xu, Mengyao Xie, Chenggang Wu, Yinqian Zhang, Qijing Li, Xuan Huang, Yuanming Lai, Yan Kang, Wei Wang, Qiang Wei, and Zhe Wang. 2023. PANIC: PAN-assisted Intra-process Memory Isolation on ARM. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 919–933. doi:10.1145/3576915.3623206

[84] Shengjie Xu, Wei Huang, and David Lie. 2021. In-fat pointer: hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 224–240.

[85] Zachary Yedidia. 2024. Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 649–665. doi:10.1145/3620665.3640408

[86] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP '09)*. IEEE Computer Society, USA, 79–93. doi:10.1109/SP.2009.25

[87] 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.

[88] Ziqi Yuan, Siyu Hong, Ruorong Guo, Rui Chang, Mingyu Gao, Wenbo Shen, and Yajin Zhou. 2024. LightZone: Lightweight Hardware-Assisted In-Process Isolation for ARM64. In *Proceedings of the 25th International Middleware Conference* (Hong Kong, Hong Kong) *(Middleware '24)*. Association for Computing Machinery, New York, NY, USA, 467–480. doi:10.1145/3652892.3700786

[89] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. 2008. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, CA) *(OSDI'08)*. USENIX Association, USA, 225–240.

[90] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2019. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 631–644.

[91] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. 2022. Debloating Address Sanitizer. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4345–4363.

[92] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. 2014. ARMlock: Hardware-based Fault Isolation for ARM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 558–569.