ZOMETAG: Zone-Based Memory Tagging for Fast, Deterministic Detection of Spatial Memory Violations on ARM

Jiwon Seo, Junseung You, Donghyun Kwon, Yeongpil Cho, and Yunheung Paek, *Member, IEEE*

***Abstract*— Against spatial memory violations threatening a vast amount of legacy software, various safety solutions have been suggested for decades. However, their practical uses have been impeded by diverse reasons, such as significant overheads and mandatory modifications of existing architectures. Accordingly, there has been a clear need for a practical safety solution that is fast enough and yet runs on commodity systems for its wide applicability in the field. As an effort to meet this need, a major processor vendor, ARM, recently announced a hardware extension, called *Memory Tagging Extension* (MTE), that helps engineers to implement efficient safety solutions. However, due to lack of hardware tags to isolate all data objects, MTE either resorts to a probabilistic memory safety guarantee, which is susceptible to a security loophole, or suffers from severe performance degradation to guarantee deterministic security. The aim of our work is to develop a MTE-based deterministic spatial safety solution, called ZOMETAG, with high efficiency by capitalizing on salient architectural features. Our key idea for fast, deterministic safety is to somehow assign permanently all objects unique tags throughout program execution. For this, ZOMETAG first divides the data memory into a number of small regions, called *zones*, and distributes data objects over the zones subject to certain constraints (to be discussed later). Then, we extend the notion of a tag in a way that each object stored with MTE tag t in zone z is uniquely assigned the zone-tag pair <z,t> as a new tag. To work with this new tag assignment, we devise a novel mechanism, called *two-layer* isolation, that is basically a combination of MTE-based tagging (for one-layer of isolation) with zone-based tagging (for the other) both of which collaborate together to ensure spatial safety for all objects by preventing a pointer currently assigned one zone-tag pair from erroneously referring to objects assigned different pairs. Our experimental results are quite encouraging. ZOMETAG enforces deterministic spatial safety with overheads of 35% in SPEC CPU2006 and merely of 6% in real world applications like nginx.**

***Index Terms*— Spatial memory violations, memory safety, Memory Tagging Extension (MTE), bounds checking (BC).**

I. Introduction

SPATIAL memory violations, such as buffer and stack overflows, are wrongful accesses that break legitimate bounds of memory objects. These errors, considered as a major threat to programs written in unsafe languages like C and C++, often occur when dereferencing a pointer that refers to any memory address outside its referent object. As a prominent approach to protecting programs from spatial memory violations, *bounds checking* (BC) has been actively studied by researchers. BC is able to detect any occurrence of these errors by checking at every memory access whether pointers are referring to within valid bounds of their referents. Unfortunately, despite its verified security, BC is not broadly adopted in real-world applications mainly because of its excessively large performance overhead for practical use. Most of the time-consuming operations severely and adversely contributing to performance are those conducted to accomplish two major tasks of BC: *boundary comparison* and *metadata maintenance*. To be specific, for sanitizing memory accesses by pointers, BC must load the corresponding lower/upper bounds information and compare the loaded bounds with the addresses referred to by pointers on every pointer dereference and/or pointer arithmetic during program execution. In addition, in order to maintain the metadata for these bounds associated with pointers up-to-date, it must keep track exhaustively of every pointer creation/propagation.

As an alternative to BC for spatial memory safety, *memory tagging* (MT) [1] has gained traction among researchers thanks to the performance merit of MT over BC that a relatively low-cost integer comparison can replace each round of bounds comparison for the sanity check. In the MT scheme for

Manuscript received 23 December 2021; revised 5 June 2023; accepted 11 July 2023. Date of publication 27 July 2023; date of current version 11 August 2023. This work was supported in part by the BK21 FOUR Program of the Education and Research Program for Future Information and Communications Technology (ICT) Pioneers, Seoul National University, in 2023; in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) funded by the Korean Government [Ministry of Science and ICT (MSIT)], South Korea, through the Analysis on Technique of Accessing and Acquiring User Data in Smartphone under Grant 2020-0-01840; in part by IITP funded by the Korean Government (MSIT) through the Reduced Instruction Set Computer-Version 5 (RISC-V) Based Secure Central Processing Unit (CPU) Architecture Design for Embedded System Malware Detection and Response under Grant 2021-0-00724; in part by the MSIT under the Information Technology Research Center (ITRC) Support Program Supervised by the IITP under Grant IITP-2023-2020-0-01797; and in part by the National Research Foundation of Korea (NRF) funded by the Korean Government (MSIT) under Grant NRF-2022R1A4A1032361. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Andrew Clark. *(Jiwon Seo and Junseung You contributed equally to this work.) (Corresponding authors: Yeongpil Cho; Yunheung Paek.)*

Jiwon Seo, Junseung You, and Yunheung Paek are with the Department of Electrical and Computer Engineering (ECE) and the Inter-University Semiconductor Research Center (ISRC), Seoul National University, Gwanak-gu, Seoul 08826, South Korea (e-mail: jwseo@sor.snu.ac.kr; jsyou@sor.snu.ac.kr; ypaek@snu.ac.kr).

Donghyun Kwon is with the School of Computer Science and Engineering, Pusan National University, Pusan 46241, South Korea (e-mail: kwondh@pusan.ac.kr).

Yeongpil Cho is with the Department of Computer Science, Hanyang University, Seoul 04763, South Korea (e-mail: ypcho@hanyang.ac.kr).

Digital Object Identifier 10.1109/TIFS.2023.3299454

memory safety, pointers and memory objects are assigned unique identification integers, called *tags*. On each memory access via a pointer, MT performs sanity checks by comparing the tags of the pointer and its referent. MT regulates that all pointers and their legitimate referents have the same tags. Therefore, if the tags do not match, the memory access will be deemed illegal and prohibited immediately to ensure memory safety. In the MT scheme, there are two elemental tag operations on pointers and objects: *tag coloring* to assign tags to pointers/objects and *tag matching* to compare the tags of pointers and objects. A notable strength of MT is its structural suitability for realization in hardware, known as the *tagged memory architecture* (TMA), which is to accelerate MT operations that would otherwise be too expensive to use in general applications. TMA has a memory whose word is extended with extra bits for tag as metadata. It also comes with special instructions to work with tags, and offers architectural extensions to enable simultaneous execution of tag matching and pointer deferencencing. The MT scheme with such strong hardware support from TMA has earned great potential to mitigate spatial memory errors with more ease and efficiency than the BC counterpart.

For the past decades, as the effectiveness of MT has been evinced, a variety of TMAs have been introduced to processor designs. Among them, a version, named as the *Memory Tagging Extension* (MTE) or *Application Data Integrity* (ADI), is built in mainstream processors, such as ARMv8.5-A [1] and SPARC M7/8 [2]. With the advent of MTE/ADI, the MT scheme seemingly becomes more practical and attractive in detecting memory safety violations for a broad range of real-world programs running on commodity systems. However, when reality kicks in, there is always a trade-off. Ideally, to guarantee complete spatial memory safety, MT requires tens of thousands or even more tags that can be assigned distinctively to pointers and objects in a program. In reality, however, MTE/ADI only offer 16 tag values because they set aside just four unused bits for tags to avoid substantial modifications and increasing logic complexities added to the existing architecture [3]. Inevitably, different objects might be assigned/colored the same tag value over time, or at the same time throughout program execution. This suggests that violation detection with MTE/ADI is probabilistic as the random tag/color will be equal to another random one with a probability of about 1/16. Unfortunately, this probabilistic guarantee of spatial safety leads to a security loophole where some wild accesses are statistically likely to be uncaught. Therefore, to achieve the same level of deterministic security that BC does for sanity checking, we must somehow mend this innate loophole that originated from architectural design constraints. One remedy would be to carefully assign tags in a way to ensure that multiple objects are never colored with the same MTE/ADI tag at any moment during execution. For example, we may instrument the program code with dynamic tag management that always keeps each tag exclusively assigned to one object by uncoloring a colored object before coloring another object with the same tag. The downside of this exclusive tag assignment is that it incurs quite frequent coloring/uncoloring operations at runtime, thus significantly raising the runtime cost of MT and offsetting the performance advantage of MT over BC. In conclusion, the MT scheme with ideal hardware support can be a viable alternative to BC, which suffers from serious performance problems, in fighting against memory safety violations. But owing to physical limitations of real hardware, MTE/ADI cannot afford enough support for complete and efficient memory safety enforcement, and hence demands the actual MT scheme to trade off either performance for security or vice versa.

In this paper, we present ZOMETAG, our MT scheme engineered for fast and deterministic detection of spatial memory violations. ZOMETAG is designed to specifically target ARM MTE by taking full advantage of its architectural support for violation detection while taking into account its hardware limitations. In principle, like SPARC ADI, ARM MTE offers spatial isolation between objects for memory safety by assigning different tag colors (or equivalently, memory regions) to them. But in practice, it does not supply enough colors for total isolation among all live objects created and accessed at runtime. To overcome the color shortage problem, ZOMETAG adopts a *two-layer isolation* mechanism by adding another layer of isolation based on *zones*, which are literally divided regions with boundaries in the memory space. To put two-layer isolation into effect, ZOMETAG prohibits different objects allocated in one zone from being assigned the same tag color, while allowing one tag to be used to color objects in different zones. With this tag assignment regulation enforced, note that ZOMETAG is technically offering two layers of isolation by assigning mutually different pair combinations of zones and tags, denoted by (z,t), to each object allocated with tag t inside zone z. In ZOMETAG, when a pointer is created or modified to point newly to a referent associated with (z,t), it is assumed to be virtually colored with the same zone-tag pair (z,t) although the pointer is physically colored by MTE only with tag t in the pair. Therefore to enforce spatial memory safety with our two-layer isolation, ZOMETAG applies the rule that permits a pointer colored with one zone-tag pair to access only the object with the same pair.

Applying this safety rule to pointers/objects colored with the zone-tag pairs having the same zone is rather trivial; that is, ZOMETAG simply relies on MTE for tag matching as objects in the same zone are all assured to have different tags in our two-layer isolation mechanism. The real challenge arises when we need to enforce the rule among those colored with the pairs having different zones. Obviously, we can no longer use MTE for safety violation detection because by our tag assignment regulation, objects are allowed to have the same tag as long as they are allocated in different zones. In fact, addressing this challenge is the reason why we have introduced to our isolation mechanism the *zone-based isolation* that prevents a pointer from accessing beyond the boundaries of the native zone of its referent. The basic principle of zone-based isolation is to disallow any address generated by pointer arithmetic from referring to objects in other zones than the native one referred to by input pointers of the arithmetic. If this principle is compromised, a pointer can be maliciously used to access objects in a foreign zone that are colored with different zone-tag pairs, which is clearly a violation of the aforementioned

spatial safety rule. To establish zone-based isolation for spatial safety, ZOMETAG imposes a certain restriction on pointer arithmetic. The way of holding this restriction or limitation may vary depending on the actual implementation for zone-based isolation. In our work, we place a limit on the range of offset values added to or subtracted from pointers in arithmetic operations; that is, their maximum values are restricted not to be more than the zone size. This restriction has to do with our unique design for layout of zones in memory, where we partition the entire address space of a program evenly into a sequence of zones, and classify them into two classes: *blue* and *red*. In the sequence, we arrange the blue and red zones alternately in a row. Objects are allocated only in the blue zones, while no access is allowed to the red ones which are basically gaps or no man's lands reserved for location-based violation checking.

As will be detailed in Section IV, ZOMETAG is able to efficiently enforce our restriction on pointer arithmetic at runtime by utilizing special hardware registers and instructions on commodity processors. Let us note here that due to the restriction on offset values, a single addition/subtraction operation on a pointer results in the output address just pointing to either the same blue zone the pointer originally refers to or the red one right next to it. There is no safety violation in the former case and no extra action is required. The latter surely constitutes a breach of spatial safety as red zones are inaccessible. Therefore, ZOMETAG must be equipped with a safeguard to alert for wrongful access to red zones. Fortunately, as also discussed in Section VI, ZOMETAG is able to detect trespass from a blue zone into its neighboring red zone with low runtime cost by checking merely a single bit that indicates the overflow of an address beyond the current blue zone. All in all, through our experiments, by introducing two-layer isolation, ZOMETAG has been evidenced to successfully perform deterministic detection of spatial memory violations on real-world applications. Moreover, it has been empirically demonstrated that ZOMETAG ensures such spatial memory safety with relatively low performance overhead by capitalizing on existing architectural support for our *zone-based memory tagging* (*i.e.*, MTE coupled with zone-based isolation).

II. Background

MTE, which has been introduced since ARMv8.5 architectures [1], facilitates efficient sanity checks on memory access. In the MTE, pointers and memory objects are assigned tags, which are specifically called pointer (or logical) tags and memory (or allocation) tags, respectively. In MTE, 4-bit memory tags, each of which respectively corresponds to a memory block of 16 bytes (64 bytes in SPARC's ADI), are stored in separate tag memory through special instructions, such as `STG` and `ST2G`. On the other hand, 4-bit pointer tags are located at [56:59] bits of pointers. Interestingly, the pointer tags are not taken in memory addressing by the Top Byte Ignore feature of ARM architecture which renders the top byte of a virtual address ignored during address translation. A 4-bit pointer tag can be numbered from 0 to 15. Among them, a tag 0, which untagged pointers will have by default, is specially treated by MTE not to restrict memory access. Also, tag 15 is not recommended in use due to a performance issue. When enabled, the MTE verifies memory accesses by comparing each corresponding pointer tag and memory tag. Note that memory accesses are permitted only if tags match, and if not, MTE raises *tag check faults*. Here, MTE provides precise and imprecise modes, which respond to the faults, respectively, by raising exceptions synchronously or by reporting asynchronously. Comparing the two modes, the imprecise mode cannot exactly identify faulty memory access but incurs much smaller overheads than the precise mode. Furthermore, MTE offers the imprecise mode that includes an additional feature to ensure fault reporting is completed prior to context switching to the OS kernel. This helps prevent faults from causing significant damage, such as information leaks. Therefore, the imprecise mode is suitable for enabling protection in practical use [4], and we also capitalize on this mode in ZOMETAG.

III. Threat Model and Assumptions

We postulate security assumptions that are consistent with related research on spatial memory safety as described in Section VIII. Target programs written in C/C++ languages are benign but have vulnerabilities potentially exploitable by attackers who intend spatial memory errors. Against such spatial errors, ZOMETAG provides spatial safety at a granularity of objects, as in recent previous studies [5], [6], [7]. It implies that, as in the previous studies, ZOMETAG does not respond to certain spatial memory errors occurring between sub-objects in an object of a composite data type, but which is helpful in increasing compatibility with C/C++ programming practices, such as initializing/copying/comparing objects using a single pointer [8], [9]. In this work, other types of threats, such as code injection [10], [11], use-after-free [12], [13], [14], [15], type confusion [16], [17], and hammering [18], [19], except for inter-object spatial errors are not considered. That is, we assume that all the metadata of ZOMETAG and control flow of target programs are intact as long as ZOMETAG is enabled.

IV. Design

ZOMETAG is aimed to provide efficient spatial memory safety. In this section, we elaborate on how ZOMETAG achieves its goal on commodity systems featured with MTE.

A. Two-Layer Isolation at a Glance

The two-layer isolation mechanism is devised to effectively and efficiently enforce spatial memory safety. With our two-layer isolation put into effect, every object in a program is associated or colored with a distinct zone-tag pair (z,t) whenever each of them is allocated in one designated (blue) zone z and assigned an MTE tag t different from those already assigned to other objects in z. Now, the mission of ZOMETAG is to uphold two-layer isolation for spatial memory safety by efficiently forcing pointers to access their legitimate referent objects according to their assigned zone-tag pairs. For efficiency, ZOMETAG takes advantage of existing hardware support for zone-based memory tagging on top of the MTE-supported spatial isolation. The simultaneous application of

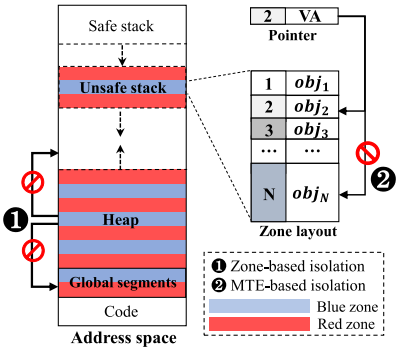


Fig. 1. Overview of ZOMETAG design.

these two hardware-supported spatial isolations is geared to confine the memory access capability of a pointer into memory objects associated with exactly the same zone and tag assigned to the pointer. For example, assume that a pointer whose legitimate referent was already defined is about to have a new address value set by an assignment instruction. The zone-based isolation first reduces the accessible memory space of the pointer from the entire program data space to one zone where its referent is located. Next, MTE further tightens the pointer's memory access capability within the zone by limiting its accessible region to the memory objects with the same MTE tag value as its own tag. In this way, the two-layer isolation mechanism of ZOMETAG ensures that pointers are only allowed to access their legitimate referents, thwarting any spatial memory violations that attempt to access other objects with different tag colors or residing in different zones.

Figure 1 illustrates the design of our two-layer isolation mechanism at a glance. To implement the mechanism, we first construct zones by evenly splitting the data memory space of a program. We then allocate any types of objects that need sanity checks, such as heap, stack and global variables, into these zones. In our mechanism, we actualize MTE-based spatial isolation within a zone by restricting the number of objects allocated in one zone not to exceed the total number of MTE tags (*i.e.*, 16 on ARM) and assigning every object mutually distinct tags. To enable the zone-based isolation, the other pillar of our two-layer isolation, we instrument all pointer arithmetic operations. The purpose of the instrumentation is to impose a certain restriction on the size of offset values in pointer arithmetic operations. Importantly, we here match the size of offset values with the size of each zone, and arrange two types of zones (*i.e.*, blue ones for object allocations and red ones for violation detection) alternately in a row along the data memory space. Note that in our instrumented code, any change in a pointer value resulting from pointer arithmetic can never be larger than the zone size. This implies that no single arithmetic operation performed on a pointer referring to a blue zone can induce the pointer to access objects in zones other than the same blue zone and the immediate next red zone. Obviously, in this design of our zone-based isolation, detecting spatial violation is quite straightforward because red zones act as tripwires for erroneous memory accesses crossing zone boundaries, and thus any address pointing to a red zone as a result of pointer arithmetic is solid evidence of the violation. As blue and red zones are of identical size (normally, a power of two) and arranged alternately, it is relatively easy to determine whether an address falls into a red zone or a blue zone by testing a specific bit in the address field, as will be detailed later.

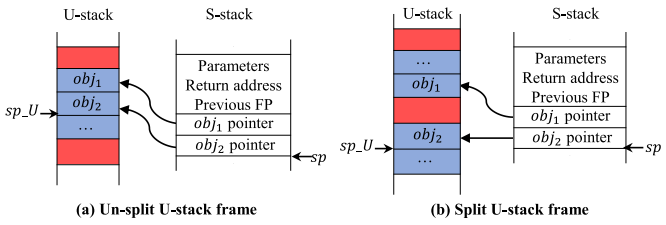


Fig. 2. Example of dynamic offsets of U-stack objects.

B. Zone Constructions and Object Allocations

The provision of spatial safety by ZOMETAG begins with allocating objects into zones. To achieve this, ZOMETAG logically splits a program's data memory space into blue and red zones, each of which is of identical size, specifically 4 GB, as illustrated in Figure 1. Objects are assigned to blue zones, and pointers are only permitted to access blue zones. Conversely, red zones serve as inaccessible regions, enabling the detection of spatial violations resulting from erroneous or malicious attempts to access objects beyond the boundaries of blue zones. When allocating an object to blue zones, ZOMETAG complies with these two constraints:

- **C1**: The number of objects allocated in a blue zone must not be more than that of available MTE tags (16 in this work)
- **C2**: The total combined size of all objects allocated in the same blue zone must not exceed the zone size (*i.e.*, 4 GB).

In a program, data objects are typically categorized into three types: heap, stack, and global. According to their types, ZOMETAG allocates objects differently as shown below, subject to **C1** and **C2**.

1) Heap Objects: As will be described in Section V, we have implemented a custom memory allocator that observes **C1** and **C2** while allocating objects into blue zones. When `malloc()` is invoked to allocate an object m, our allocator first rounds the requested size for m to a multiple of the tag granule (*i.e.*, 16 B). It then fetches a blue zone satisfying **C1** and **C2** for the object m. If the allocator finds no such zone among the candidates, it exits with an error code. Our allocator separately manages its internal metadata in a separate zone so that the metadata can be protected against malicious heap accesses by our zone-based isolation mechanism.

2) Stack Objects: One challenge in our stack object allocation is that to meet **C1** and **C2**, the stack tends to consume more memory space as it is allowed only to contain blue zones and each zone is allowed only to store a limited number of objects. This memory consumption problem will become grave particularly if a program has many deep function calls or recursive calls. To alleviate this problem, we have modified the stack layout as well as the stack management policy by adopting the idea behind SafeStack [20] whose

security goal (*i.e.*, ensuring control-flow integrity) is different from ZOMETAG. SafeStack separates objects into two groups considering their exploitability. To be specific, exploitable objects like arrays are stored in the *Unsafe stack* (U-stack), and non-exploitable ones like simple integer variables in the *Safe stack* (S-stack). Such a classification is just for SafeStack to leave unsafe objects unprotected that are not of interest in its security goal. In the perspective of ZOMETAG aiming at ensuring memory safety, however, such a classification inspired that we only need to spend extra memory and time in applying our two-layer isolation scheme to obey **C1** and **C2** for exploitable, unsafe objects stored in the U-stack. In our implementation, whenever unsafe objects are allocated into or deallocated from the U-stack, ZOMETAG keeps track of the total count and size of all objects occupying the current blue zone in the U-stack, and checks **C1** and **C2**. If ZOMETAG cannot find in the U-stack any blue zone that satisfies **C1** and **C2** for a newly allocated object, it adds a new blue zone to the U-stack and places the object in the zone. Likewise, as the U-stack continues to grow or shrink during program execution, a new blue zone is included or an existing zone is excluded from the U-stack, respectively.

By default, SafeStack defines two stack pointers: one legacy stack pointer (*i.e.*, `sp`) for the S-stack and an additional one (*i.e.*, `sp_U`) for the U-stack. Let us note here that objects in a legacy stack are typically referenced using offset addressing based on the stack pointer (SP). However, in our runtime environment where ZOMETAG is enabled, the SP-based offset addressing is not available for U-stack objects, but only for S-stack ones. Like ordinary objects stored in the legacy stack, S-stack objects can be accessed by instructions with offset addressing mode because their offsets (or distances) from the SP can be statically determined during compilation. In contrast, U-stack objects have offsets varying with different call stack sequences dynamically generated during runtime. To explain this, consider an example in Figure 2 where multiple U-stack objects are allocated in a function. We can see that their locations (equivalently, offsets from the `sp_U`) in the U-stack may vary depending on the decision of whether or not they are allocated in the same zone. Unfortunately, this decision is only made at runtime by the ZOMETAG allocator that checks **C1** and **C2** when those objects are actually allocated. Thus, to always refer properly to U-stack objects with such dynamic offsets, ZOMETAG provides each U-stack object with an individual pointer that stores the actual address in a blue zone where the corresponding object is allocated during execution. As those pointers referring to U-stack objects are basically non-exploitable, they are all allocated in the S-stack and referenced via the SP-based offset addressing.

3) Global Objects: While heap and stack objects are allocated dynamically at runtime, global objects are permanently allocated at fixed locations when a program is loaded and freed when it terminates. This means that ZOMETAG does not have to pour its energy into zone-aware allocations for global objects at runtime. Instead, ZOMETAG predetermines the positions of global objects at compile time and distributes them over several blue zones that, of course, satisfy **C1** and **C2**.

C. Zone-Based Isolation

By constructing zones and coloring objects in each blue zone subject to two constraints **C1** and **C2**, we can establish one layer (*i.e.*, MTE-based) of spatial isolation between every heap, stack, and global object within each individual zone. Now, to obtain a complete mechanism for our two-layer isolation, we must build in another layer (*i.e.*, zone-based) of spatial isolation, briefly described in Section IV-A. The purpose of zone-based isolation is to limit the memory accessibility of a pointer to the specific zone where its referent belongs. A naive but costly solution to realize this isolation would be performing BC on every memory access in a way to prevent any address generated by pointer arithmetic from being used to access objects in other zones than the native one initially referred to by input pointers of the arithmetic. Being compared to such inter-zone BC, our solution requires fairly low overhead for performing extra operations to enforce isolation between zones, and furthermore, being coupled with the MTE-based isolation, it constitutes complete inter- as well as intra-zone isolation throughout the entire program execution.

As discussed above, in our design for the zone-based isolation, all pointer arithmetic instructions are instrumented to have only 32-bit registers as the operands for offset values. Also, in the design, each zone is set to have a 4GB address space. The design that (1) forces every pointer operation to have 32-bit register operands and (2) sets every zone to 4GB is not decided by chance but rather intended to facilitate memory violation detection in our mechanism. Specifically, our design guarantees that the maximum range of address values on every pointer arithmetic is equal to 2^{32}, which matches 4G, the size of a blue/red zone. To explain this in more detail, suppose that as stated in Section IV-B, there is a sequence of blue zones and red zones alternately ordered in the memory where b_i and r_i are the i-th zones in the sequence. Then, it is clear from our design that any pointer arithmetic operation cannot make a pointer currently having its referent object in zone b_i refer to an object in a different zone b_j. The only address that the pointer will have as a result is an address either in b_i or in r_i. As has been said earlier, any operation that results in a pointer accessing the red zone r_i is immediately considered a spatial safety violation. From this fact, we can see that our zone-based isolation always successfully deters or detects any malicious attempt to violate inter-zone isolation. In our design, it is noteworthy that detecting illegal access to a red zone can be done with low cost by examining a single bit. Similar to identifying even numbers by checking the 0th bit, we can distinguish between adjacent b_i and r_i by reading the 32nd bit of the pointer value:*e.g.*, $0 \rightarrow$ blue zone and $1 \rightarrow$ red zone. To be specific, after each pointer arithmetic operation, ZOMETAG extracts and tests the 32nd bit of the output pointer value, and aborts if the bit value indicates a red zone. Luckily, this procedure can be efficiently implemented on ARM by executing a single test-and-branch [1] instruction that takes a branch depending on the value of a designated bit of its operand.

For complete isolation, we basically need to instrument the code for testing the output pointer value after every

pointer arithmetic operation. Although inserting just one test instruction after each pointer operation is sufficient to deter trespassing in a red zone, the total overhead for testing all pointer operations is still not negligible. Fortunately, we have discovered that much of the overhead can be optimized as tests can be omitted for a certain case where pointer operations are immediately followed by loads/stores. This scenario is commonly encountered in programs because pointer operations are typically used to calculate addresses for subsequent memory operations in the execution flow. In this case, even if the output pointer becomes invalid (referring to a red zone) as a result of a pointer operation, and manages to bypass the test for accessing a red zone during a load/store operation, ZOMETAG has an additional safety measure to deter erroneous loads/stores from escaping blue zones to approach red ones. Our empirical observation reveals that a majority of tests are redundant and eliminated in our benchmarks, consequently helping ZOMETAG save a significant amount of time and space overhead that would otherwise be added to our performance numbers.

D. Multithreading

ZOMETAG supports multithreaded programming because its two-layer isolation mechanism operates in a thread-safe manner. We can abstract the operations of the mechanism into two classes: (1) a zone-tag pair allocation per object and (2) object isolation on the basis of zone-tag pairs. Of the two operation classes, the latter one is naturally free from any concurrency issue because once allocated to objects, zone-tag pairs are continuously read for object isolation but never changed. In ZOMETAG, the operations in the former class do not suffer from any concurrency issues, even though there may be simultaneous allocations of zone-tag pairs in multiple threads. Note that allocating zone-tag pairs to objects involves two types of tasks: placing each object into one of the zones and coloring the object with a unique tag. The tag coloring task is independent of concurrency issues since this task only affects the tag memory locations exclusively occupied by each object. On the other hand, we must pay attention to the object placement task because the constraints **C1** and **C2** should be considered globally by multiple threads that simultaneously run and allocate objects in the same zone. Fortunately, among the three kinds of memory objects, heap, stack, and global, that are isolated by ZOMETAG, global and stack objects are worry-free as the former ones are allocated statically, and the latter ones are allocated thread-locally. As for heap objects, we can avoid concurrency issues by checking the constraints within a critical section which is typically protected in a heap allocator by mutex locks. For the sake of more efficient multithreading, we may introduce to ZOMETAG a concept of *thread-local* zones that assigns zones dedicated to each thread. This way will allow us to check **C1** and **C2** without concurrency issues because threads are no longer sharing zones for object allocations.

V. Implementation

We implemented the prototype of ZOMETAG using LLVM 4.0.0. Considering the constraints **C1** and **C2**, we developed a memory allocator for heap objects and an LLVM pass/runtime library for global and stack objects. We also developed another LLVM pass to realize the zone-based isolation by instrumenting all pointer arithmetic operations. The following subsections present the implementation details.

A. Tag Coloring

Every time objects are allocated, ZOMETAG colors them with unique random tags in each zone. Heap objects are colored by the memory allocator before returning. Stack and global objects are colored by the runtime library before they are accessed. For example, stack objects are colored at each function prologue, and global objects are colored by a separate initialization function before `main()` runs. In tag coloring, ZOMETAG does not use the entire 16 tags but only 14 tags (from tag 1 to 14), excluding two exceptional tags: tag 0 and 15, as explained in Section II. This means that each zone can allocate up to 14 objects. Tag 0 is used specifically to mark red zones as tripwires. Since in Linux, tag 0 is the initial tag of memory regions when mapped with MTE enabled, we can naturally render red zones inaccessible by keeping their tags intact (*i.e.*, tag 0) and using only non-zero tags in blue zones for all pointers and objects.

B. Zone-Based Heap Memory Allocator

To implement our heap memory allocator, we modified the existing allocator [21] that uses size-segregated freelists, in which unallocated (or free) regions of the same size are linked together. To avoid being too fragmented, as many freelists as predefined size classes (refer to the appendix in [21]) are prepared during initialization. Whenever `malloc()` is called with a request size, our allocator first rounds the request size up to the closest predefined size class. The allocator then takes a freelist corresponding to the size and performs the requested allocation. If the corresponding freelist is empty, the allocator acquires free regions matching the request size from a blue zone and adds them to the freelist so that it can deal with the allocation. Note here that such region acquisitions should be performed under constraints **C1** and **C2**. In order to meet the constraints without requiring explicit checks, the allocator efficiently acquires the maximum number of available free regions from a blue zone, considering both the request size and the size of the blue zone. For instance, if the request size is 64 B, it takes advantage of the maximum number of free regions available (14) since $14 \times 64B < 4GB$. However, if the request size is 500 MB, it only takes 8 free regions, as $8 \times 500MB < 4GB$.

To manage blue zones, the allocator defines three states, `UNUSED`, `ALLOCATED`, and `UNALLOCATED`. The `UNUSED` state implies that the blue zone is not yet used, so the allocator can acquire free regions from it when a freelist is empty. The `ALLOCATED` state indicates that free regions of the blue zone have already been linked to a freelist and part or all of them are allocated for objects. On the other hand, the `UNALLOCATED` state means that all the linked free regions are currently not allocated for objects. Initially, blue zones are in the `UNUSED` state, and are used preferentially for

region acquisitions. However, if no UNUSED zones remain, the allocator reclaims some UNALLOCATED zones as UNUSED ones by removing all the links between their free regions and freelists. If all UNUSED/UNALLOCATED zones are exhausted, the allocator cannot perform region acquisitions anymore, resulting in allocation failure.

C. Zone-Based Unsafe Stack

To protect stack objects, ZOMETAG relies on SafeStack, which classifies stack objects into S-stack and U-stack ones. Of the two types of objects, the primary focus of ZOMETAG is to provide isolation specifically for objects within the U-stack that have the potential for exploitation. By concentrating on the U-stack, which is particularly vulnerable to attacks, ZOMETAG aims to enhance the security of the application. It ensures that these potentially exploitable U-stack objects are isolated and protected, reducing the risk of compromise and potential exploitation. Basically, it can be achieved by allocating U-stack objects into the zone-based U-stack. However, as discussed in Section IV-B, U-stack objects allocated in the U-stack cannot be referenced using the offset addressing because their offsets will vary at runtime. For this reason, ZOMETAG makes all accesses to U-stack objects be performed through separate pointers generated at runtime. To do this, we modified the original implementation of SafeStack. More specifically, we inserted calls to runtime functions, `_Ustack_growth_()` and `_Ustack_shrink_()`, at every function prologue and epilogue, respectively. The two runtime functions are responsible for growing and shrinking the U-stack by including and excluding blue zones dynamically considering **C1** and **C2**. The `_Ustack_growth_()` function also plays an important role in creating pointers for U-stack objects within the S-stack. This allows the U-stack objects to be accessed using these pointers instead of relying on offset addressing.

D. Zone-Based Global Object Allocations

To allocate global objects in blue zones, we utilize LLVM and a linker. Specifically, our LLVM pass identifies all global objects and determines their respective sizes. Taking into account the constraints **C1** and **C2**, each object is assigned to a separate section, such as `global_1`, `global_2`, ..., `global_n`. After compilation, the section information is referred to generate a linker script that adjusts the position of the global object sections along blue zones. Now, the linker script is used for linking, and the global objects are placed into blue zones when the program is loaded.

E. Instrumentations for Zone-Based Isolation

To enable zone-based isolation, we built an additional LLVM pass that instruments all instructions, including pointer arithmetic operations: *e.g.*, arithmetic/bitwise instructions (addition, subtraction, shift, xor, etc) that use a pointer as an operand, and load/store instructions using pre- or post-indexed addressing modes. As stated in Section IV-C, two types of instrumentations are required: (1) replacing 64-bit operands

```
<foo>:                          <foo>:
  movz  x0, #0x0, lsl #48         movz  x0, #0x0, lsl #48
  ......                          ......
  ldr   x0, [x1, x2]              ldr   x0, [x1, w2]      ①
  mov   w9, w1                    mov   w9, w1
  ......                          ......
  ldr   x4, [sp, #16]             ldr   x4, [sp, #16]
  str   x3, [x4, x5]              str   x3, [x4, w5]      ②
  ......                          ......
  ldr   x7, [sp, #32]             ldr   x7, [sp, #32]
  add   x7, x7, #100              add   x7, x7, #100
                                  tbz   x7, #32, <overflow> ③
```

(a) Before **(b) After**

Fig. 3. Assembly code instrumented for ZOMETAG zone-based isolation. Changed or inserted instructions are highlighted. (1) and (2) convert offset operands to 32-bit registers, and (3) checks if a pointer points to a red zone after pointer arithmetic.

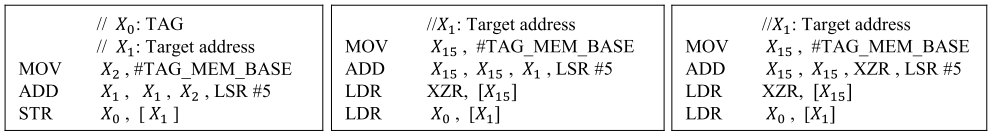
register into 32-bit ones to limit the modification range of pointer values to the zone size and (2) inserting a test-and-branch instruction to test whether a modified pointer points to a red zone. Figure 3 shows an example with both types of instrumentations applied. As seen in the example, the instrumentations of the first type rarely increase the number of instructions, as many ARM instructions have a variant that takes 32-bit registers with the prefix 'w' as the second operand. For the instrumentations of the other type, TBZ or TBNZ instructions can be used. The TBZ tests a specific bit of the given operand register and performs a branch if the bit is zero. The TBNZ is similar but performs a branch if the bit is non-zero. So, inserting either a single TBZ or TBNZ after each pointer arithmetic operator is enough to detect erroneous pointer values pointing to a red zone. Note here that as TBZ and TBNZ can perform a branch only in ±32 KB range, we inserted multiple identical abort routines for violation detection throughout the program code. This ensures that all instructions have a path to at least one of the abort routines, allowing for effective detection of violations.

VI. Evaluation

In this section, we evaluate ZOMETAG it in terms of efficiency and security. To do this, we answered the following question:

- How do we measure the performance overhead of MTE without a real hardware implementation? (Section VI-A)
- How are the performance and memory overheads of ZOMETAG in artificial benchmarks? (Section VI-B)
- How are the performance and memory overheads of ZOMETAG in real-world applications? (Section VI-C)
- Is ZOMETAG effective in detecting spatial errors? (Section VI-D)
- Is ZOMETAG secure against threats to neutralize it? (Section VI-E)

Experimental Setup: At the time of researching this work, no ARM cores (including M1 [22]) had implemented MTE. For this reason, we used two different platforms to perform evaluations. We first confirmed the functional correctness of ZOMETAG by implementing its prototype on a software emulator, ARM Fast Models [23], that supports the ARMv8.5 architecture, including MTE. Unfortunately, the emulator does not provide cycle-accurate execution. To measure performance numbers, therefore, we ported ZOMETAG



(a) Tag Store **(b) Tag Load (method-1)** **(c) Tag Load (method-2)**

Fig. 4. Instructions for estimation of MTE overheads.

to a software development board, ODROID-C4 [24] with Cortex A-55 quad-core CPU @ 2.0 GHz and 4 GB RAM. Note here that since MTE is absent in the Cortex-A55 cores, we estimated the performance impact of MTE on the system as in Section VI-A.

A. Estimation of MTE Overheads

In MTE, most performance overheads come from (1) memory tag coloring and (2) tag matching operations. To estimate MTE overheads, therefore, we mimicked these two types of operations in software. Simply put, we reserved a large memory region, and substituted all memory tag accesses with ordinary memory accesses to the reserved memory region (dummy tag memory, hereafter).

1) Memory Tag Coloring: These operations are performed during object creation to fill corresponding tag memory regions with 4-bit tags. As stated in Section II, ARM originally provides special instructions to perform these operations, but unfortunately, these special instructions were not available in our experimental environment. Therefore, we estimated their overheads by executing substitutive memory instructions that write single bytes to the dummy tag memory (Figure 4.(a)).

2) Tag Matching: These operations to compare pointer tags with memory tags are executed upon every memory access. These operations are clearly divided into two sub-operations: (1) tag loading and (2) tag comparison. Of these two sub-operators, the latter operations that compare pointer tags with memory tags incur negligible overhead because they are performed by a separate hardware logic running simultaneously with CPU cores [4]. On the other hand, the former operations entail non-negligible overhead because they cause a program to perform additional memory loads that fetch memory tags to a cache proportionally to the number of memory accesses. To estimate the tag loading overhead, we performed code instrumentations in two methods. In method-1, instructions are inserted to load a corresponding memory tag from the dummy tag memory prior to each memory access (Figure 4.(b)). However, considering the fact that, in a real implementation of MTE, tag loading will be performed transparently by an underlying hardware logic, *i.e.*, tag cache, the method-1 exaggerates the tag loading overhead due to the inserted instructions that increase pressure in the CPU's functional components, such as a fetcher, a decoder, an ALU, and a memory unit. In this regard, to grasp the extra overhead caused by the inserted instructions, we additionally carried out the method-2 instrumentation (Figure 4.(c)). In the method-2, the same instructions are inserted as in the method-1. However, unlike in method-1, these instructions always load the same memory tag from the constant location for each memory access. After all, in method-2, since the memory tag will always reside in the cache, we can measure the specific overhead incurred in the CPU by the execution of the inserted instructions, excluding the overhead of fetching memory tags into the cache. As a result, by subtracting the overhead of method-2 from that of method-1, we can determine the tag loading overhead that occurs when fetching the memory tag from the cache.

The estimation process described above does not consider two cases: (1) reordering between a tag load and subsequent memory access to be checked, and (2) thrashing issues in TLB and cache between tag and ordinary memory accesses. The first case does not harm the accuracy in our estimation because, as stated earlier in Section II, ZOMETAG utilizes MTE's imprecise mode that performs tag comparisons and mismatch detections asynchronously. Next, the second case makes our estimation for the tag loading overhead conservative and at least not optimistic because the case would not happen or be minimized if dedicated TLB and cache are included in a real implementation [1], [4].

B. Overheads on Artificial Benchmarks

1) Performance Overhead: First, to measure the performance impact on single-threaded applications, we tested ZOMETAG using SPEC CPU2006 benchmark suite [25]. Note that, as will be discussed in Section VII, ZOMETAG has a threshold on the maximum counts of live objects. It means that ZOMETAG cannot support programs whose maximum live object counts exceed the threshold, and in the benchmark suite, `perlbench`, `omnet++`, and `xalancbmk` correspond to such programs in question. For these benchmarks, therefore, we used a `train` workset with reduced input size to run with ZOMETAG. For the other benchmarks, we used the native `ref` workset. The results are shown in Figure 5. In total, ZOMETAG causes 35% geometric mean (geomean) runtime overhead. We also compared the performance overhead of ZOMETAG with SGXBounds [26] and LowFat [27], which are the latest object-based spatial safety solutions applicable on ARM, and AddressSanitizer, a popular solution for detecting memory bugs shipped in commodity compilers (gcc and clang) [28]. For a fair comparison, we ported these competing solutions to ARM and reproduced their performance numbers in our experimental environment. To be specific, we first disabled stack protections for both SGXBounds and LowFat due to their architecture dependent operations for stack switching and pivoting. We also turned off the optimizations based on x86_64 specific instructions such as `BMI` since there were no appropriate substitutions on ARM. The two changes mentioned will differently affect performance, but since the former

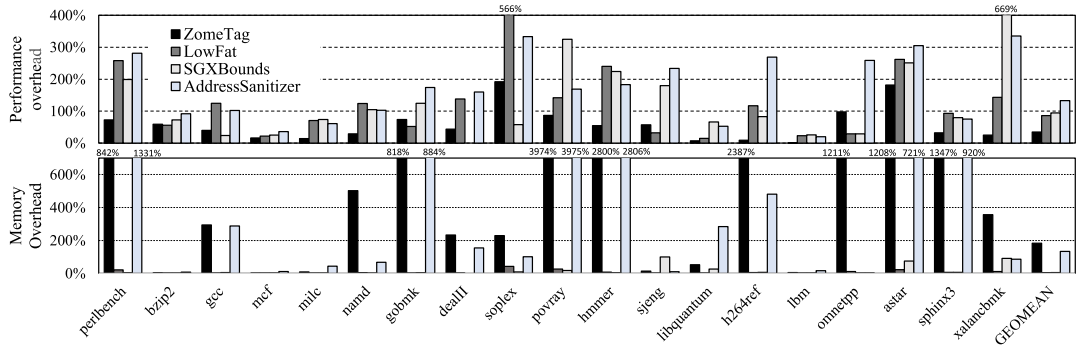


Fig. 5. SPEC CPU2006 for ZOMETAG: Performance (top) and memory (bottom) overheads over native execution.

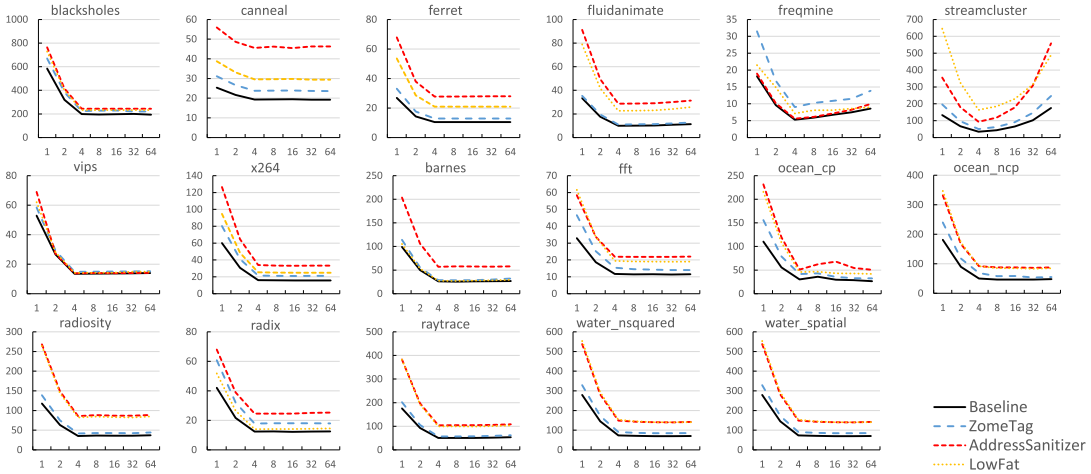


Fig. 6. Comparison of the execution time on PARSEC.

one regarding stack is more deeply involved in the execution of a program, the competing solutions will show *less* overhead than their full-fledged implementations. When we ran SGXBounds, we faced unexpected aborts/crashes in 6 out of 19 benchmarks due to its limited compatibility [29]. To obtain performance numbers, therefore, we modified SGXBounds to perform dummy sanity checks with unlimited bounds covering the whole address space. Additionally, we applied a small patch [30] on the benchmarks to prevent abrupt crashes when running with the AddressSanitizer.

Overall, we can see that our design is (far) more efficient than other solutions. ZOMETAG incurs less than half of the overhead compared to SGXBounds, LowFat, and AddressSanitizer whose overheads are 94%, 86%, and 133%, respectively. Overall, the overheads measured for our competing solutions are higher than the results reported in their papers. This is attributed to significant differences in experimental environments of theirs and ours, such as architecture (x86_64 vs ARM), cache/memory size, and the number of CPU's functional components (load unit and ALU). ZOMETAG shows a general increase in overhead as the number of live objects in a program. This is expected because the objects are distributed over sparsely placed zones in the program's address space, resulting in a large number of TLB misses. For example, ZOMETAG showsalmost native runtime performance on the `mcf` benchmark, which uses only 4 maximum live objects during its execution, but shows 98% overhead on `omnet++`, which has 436 K live objects.

For a more comprehensive understanding, we separately measured the overheads by the four main sources. The results are in Figure 8. We observe that tag-related overhead is a predominant element with geomean 16%. This is expected because tag operations involve a considerable number of memory accesses to tags. The overhead of the allocator is highly dependent on the number of allocations of each program. This is due to the fact that our allocator is not TLB-friendly by allocating objects sparsely over a large address space. The overhead of the zone-based isolation and that of applying ZOMETAG to stack/global objects dependent on a program's characteristics. The former overhead is non-negligible on pointer-intensive benchmarks as more bit-checking instructions should be inserted. The latter overhead is noticeable when programs allocate many stack and global variables.

We also conducted a set of experiments with PARSEC benchmark [31] to evaluate the scalability of ZOMETAG in multithreaded programs. Figure 6 shows the performance results of LowFat, AddressSanitizer and ZOMETAG in comparison to baseline. The geometric means of ZOMETAG, Lowfat, and AddressSanitizer overheads to run concurrent threads respectively range from 24.6% to 30%, 67.1% to 82.8%, and 93.7% to 101%. While most benchmarks have fairly low overhead, `freqmine` suffers from a relatively

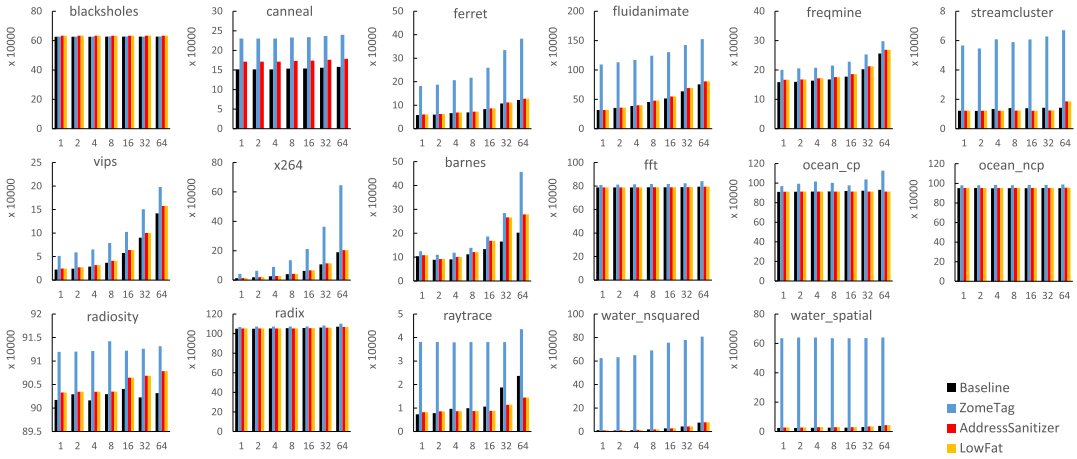


Fig. 7. Memory usage on PARSEC.

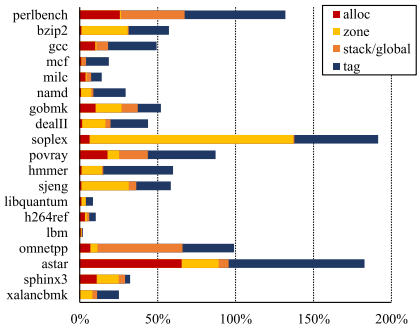


Fig. 8. Anatomy of Performance Overhead on SPEC CPU2006. The legends indicate overhead sources. **alloc**: allocator, **zone**: zone-based isolation, **stack/global**: applying ZOMETAG to stack/global objects, **tag**: memory tag coloring and tag matching.

high overhead ranging from 53.7% to 75.2%. Our analysis reveals that this overhead is attributed to the allocator side. By default, `freqmine` uses a custom allocator that internally allocates large objects in a reserved memory chunk without going through expensive heap management functions. Therefore, substituting ZOMETAG's allocator for the default one has adversely affected the overall performance. Nevertheless, in PARSEC, ZOMETAG certainly incurs overhead less than half compared to LowFat and AddressSanitizer. It demonstrates the efficiency of ZOMETAG on multithreaded programs.

2) Memory Overhead: To evaluate the overall memory consumption, we measured the maximum resident size reported by `time -v` during the execution of the benchmarks. According to Figure 5, our geometric mean across all benchmarks is 183% which is comparable to 133% of AddressSanitizer. Figure 7 also shows the maximum resident set size (RSS) values of the four experiment settings (baseline, ZOMETAG, Lowfat, and AddressSanitizer) for PARSEC benchmarks. The geometric mean of the overhead spans from 77.5% to 103% with ZOMETAG as we increase the number of threads from 1 to 64. Note that it spans from 14.3% to 16.7% with Lowfat and from 47.4% to 62.7% with AddressSanitizer. The geometric mean of SGXBounds and LowFat over all benchmarks is 5% and 4%, respectively, which is minimal, as they require only a small amount of additional memory for their metadata.

The degree of memory overheads caused by ZOMETAG depends on the size of objects allocated in programs. Note that when only small objects are allocated, ZOMETAG cannot fully utilize pages in a zone due to the constraint **C1**. For example, ZOMETAG entails relatively large memory overhead in `xalancbmk` where the majority of the allocations are less than 512 bytes in size. On the other hand, the memory overhead is only a few percent for programs, such as `sjeng` and `lbm`. This is because ZOMETAG can avoid wasting space in pages despite the constraint **C1**. It indicates that ZOMETAG has the most efficiency in both performance and memory aspects when it runs with memory-intensive programs that tend to deal with large objects like arrays. We believe that this fact does not undermine the value of ZOMETAG considering that these programs have been difficult to harden their security using the existing techniques because of considerable performance overheads, as shown in our experimental results stated earlier.

C. Overheads on Real World Applications

For the purpose of the performance evaluation and compatibility demonstration, we conducted additional evaluations using real-world applications, including two web server applications (Nginx and lighttpd) and an in-memory key-value store application (Memcached).

1) Memcached: We evaluated Memcached v1.4.15 [32] using the memaslp benchmark shipped with libmemcached v1.0.18 client. The benchmark was performed requesting `SET` and `GET` operations in a proportion of 1 to 9 over a 1Gbits/s link. The results are shown in Figure 9. On average, we observed a 13% decrease in throughput, which decreased to 4% when saturation was reached. In comparison, LowFat and AddressSanitizer showed a 22% and 23% decrease, respectively.

2) Web Server Applications: Nginx, Lighttpd: We also conducted tests on web server applications, specifically Nginx v1.4.0 [33] and Lighttpd v1.4.59 [34], using ApacheBench [35]. During the tests, we gradually increased the concurrency of the benchmark to simulate an increasing number of concurrent clients, as shown in Figure 9. For Nginx, ZOMETAG achieved a throughput of 94%, outperforming

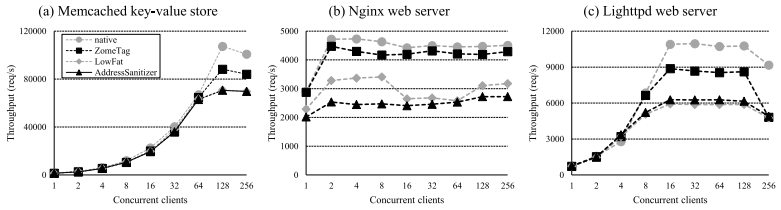


Fig. 9. Performance overhead on real-world applications.

Application	Source code	Bug type	Detected
bc-1.06	id->a_name=next_array++; a_names[id->a_name]=name;	heap	Zone-based
	sprintf(genstr, "F%d,%s.%s[",...);	global	Tag-based
gzip-1.2.4	strcpy(ifname, iname);	global	Tag-based
man-1.5h1	tmp_section_list[i++] = my_strdup (p);	stack	Tag-based
ncompress	strcpy(tempname,*fileptr);	stack	Tag-based
polymorph-0.4.0	strcpy(target, optarg);	global	Tag-based
	strcpy(newname, "");	stack	Tag-based

Fig. 10. The effectiveness of ZOMETAG in detecting spatial errors.

AddressSanitizer and LowFat, which achieved 57% and 67% throughput, respectively. Lighttpd showed a similar pattern to Nginx. In the case of Lighttpd, LowFat had the highest overhead, reaching only 60% throughput on average compared to the native build with gcc, while AddressSanitizer achieved 63%. ZOMETAG achieved an average throughput of 80%, with a negligible drop of 2% when servicing up to 8 concurrent clients.

D. Effectiveness in Detecting Spatial Errors

To evaluate the effectiveness of ZOMETAG in detecting spatial errors, we selected real programs with spatial errors from BugBench [36]. We compiled the BugBench benchmark suite with ZOMETAG and executed it using various input sets. As shown in Figure 10, itBug type indicates the location of the overflow, whether it occurred in the heap, stack, or global memory. In `Detected`, *Tag-based* and *Zone-based* indicate the techniques used by ZOMETAG to detect the errors. *Tag-based* detection involves matching tags within the same zone, while *Zone-based* detection identifies violations of inter-zone isolation. For example, in the case of `bc-1.06`, it contains a heap/global buffer overflow. Specifically, in the case of a heap overflow, an element (`id->a_name`) may exceed the boundaries of heap objects (`a_names`). We detected a potential crash by identifying inconsistent tag values when accessing beyond the boundary of an object. With the zone-based isolation technique, we can detect spatial safety violations even if pointer arithmetic operations lead to access outside the designated zone. In the case of global overflow, we detected that the source of `sprintf`, originating from the input file, may exceed the length of 80 characters. Furthermore, we also identified access to other objects beyond the boundary of `genstr`. Similarly, we discovered global overflow vulnerabilities in `gzip-1.2.4` and `polymorph-0.4.0` that arise when inputting a long file name. By constraining memory access to objects with matching tag values, we were able to successfully detect these vulnerabilities. In the cases of `ncompress` and `polymorph-0.4.0`, the excessively long input file name would overflow the array (`tempname`, `newname`), ultimately overwriting the stack return address with the overflowed stack array. As mentioned in Section V-C, we allocate stack objects in the zone-based US (U-stack). This ensures that the return address cannot be overwritten and the value of stack variables cannot be manipulated. In the case of `man-1.5h1`, there is a stack array overflow bug within a for-loop caused by an incorrect loop exit condition. This can result in overflow when long inputs are provided. However, we can easily detect this scenario through tag matching, thereby preventing any potential exploitation.

E. Security Analysis

In the threat model described in Section III, we assume that the metadata of ZOMETAG and the control flow of a program remain intact, ensuring spatial safety enforced by ZOMETAG. This implies that attackers are unable to disable or bypass ZOMETAG's two-layer isolation mechanism, which relies on its own memory allocation strategies and code instrumentations. Therefore, the only way attackers can undermine ZOMETAG's security is by manipulating pointers, specifically their address values and tags. To do this, attackers may attempt to abuse pointer arithmetic operations to manipulate pointers. However, ZOMETAG's zone-based isolation, which instruments all pointer arithmetic operations, effectively prevents such attacks. Instead, attackers may try to manipulate pointers by causing sub-object spatial errors within the same objects. While ZOMETAG, like other studies focusing on spatial safety at object boundaries for higher compatibility with C/C++ programming practices, does not explicitly address sub-object spatial errors, it still provides a probabilistic defense against them due to its reliance on MTE. Unlike other studies that may be powerless against sub-object spatial errors, ZOMETAG has the advantage of probabilistically thwarting such attacks through its integration with MTE.

VII. Discussion

A. Limitations on Object Size and Count

We admit that our two-layer isolation mechanism inevitably limits the maximum size of each object as ZOMETAG cannot allocate an object of size larger than the zone size (4 GB). But we argue that this limitation is acceptable in practice since a single object of such a huge size is not common in real-world programs.

As another concern about our mechanism, ZOMETAG limits the number of objects that can be allocated in a program. To be specific, ZOMETAG imposes restrictions on object counts per blue zone, and the finite virtual address space of a program

Application	Maximum allocation	Total allocation	Test method
pbzip2	57	435,389	Compress linux source files
mysql	4306	57,340	Inserting 2,000,000 records using mysqlslap
wget	3487	177,653	Download Ubuntu 20.04 LTS
pfscan	7379	776,700	Search 1.45 GB file

Fig. 11. The maximum and total numbers of object allocations in real-world applications.

can accommodate only a limited number of blue zones. A blue zone can contain up to 14 objects, and pairs of blue and red zones of $2 \times 2^{32}B$ can exist in a program's virtual address space, which is at most $2^{48}B$ on ARM architectures. In an optimistic calculation, $14 \times 2^{15} = 458,752$ objects, including heap, stack, and global ones, can be allocated in a program. It is important to note that, as ZOMETAG allows dynamic object (de-)allocations, the calculated number indicates the largest possible number of live objects that can exist simultaneously during program execution. As ZOMETAG could not support some benchmarks of SPEC CPU2006 (3 out of 19 benchmarks) and PARSEC (1 out of 18 benchmarks), we admit that this number is not good enough to run all types of applications. However, we still have confidence in the usefulness of ZOMETAG since it can be applied to a broad spectrum of applications. In our experiments, we observed that three real-world applications (Nginx, lighttpd and memcached) could run with ZOMETAG. For other real-world applications, we measured the maximum numbers of live objects, and the results reported in Figure 11 show that ZOMETAG can support all these applications, given their maximum possible live object counts, which is far less than the number of objects in which ZOMETAG can accommodate.

To run applications with a lot of live objects, we can employ SafeStack-like techniques classifying objects into (unexploitable) safe and (exploitable) unsafe ones for heap and global objects as well as stack objects as done in Section IV. In many studies [37], [38], these techniques have been used to reduce runtime overhead by eliminating sanity checks for safe objects. In ZOMETAG, however, these techniques can help protect as many more unsafe objects through its two-layer mechanism as the number of objects classified as safe.

B. Compatibility

A tagged pointer scheme that stores metadata inside pointers has been widely used in many studies [26], [29]. The scheme enables efficient metadata management. However, it can sometimes reveal compatibility issues [29] that lead to program crashes. These compatibility issues arise due to programming and compiler optimizations that do not consider in-pointer metadata. For example, issues can occur when in-pointer metadata is unintentionally modified by pointer arithmetic operators or when pointers generated by uninstrumented external libraries do not include the necessary metadata.

In that MTE takes advantage of the tagged pointer scheme, ZOMETAG is potentially subjected to such compatibility issues as well, but the design of ZOMETAG inherently mitigates them. For example, in ZOMETAG, its in-pointer metadata, pointer tags, are always preserved in pointer arithmetic thanks to the zone-based isolation that limits the range of address values in pointer arithmetic. Also, when untagged pointers are produced and used by uninstrumented libraries, the program to which ZOMETAG is applied still can run without crashes. This is because such untagged pointers (*i.e.*, tagged with 0) have access to the whole program memory, as explained in Section II. In response to other cases that may cause compatibility issues, we can refer to the previous work [29] that already discussed various scenarios and proposed feasible solutions.

VIII. Related Work

A. Low-Fat Pointer Approach

LowFat [21], [27], SGXbounds [26] and Delta Pointers [29] have proposed a low-fat pointer scheme that embeds bounds metadata within the 64-bit pointer representation. LowFat divides the virtual address space into evenly distributed regions and performs bounds checks by utilizing the bounds information derived from the region index in the upper bits of the pointer. Similarly, ZOMETAG also partitions the virtual address space into zones. However, unlike LowFat, ZOMETAG partitions the virtual address space into zones but does not need to perform explicit bounds checks due to its two-layer isolation mechanism, resulting in improved efficiency. SGXBounds alters the pointer representation to store its upper bounds in the upper 32 bits of the pointer. Due to such a change, it can use only 4GB of the virtual address space in a program. In comparison, ZOMETAG also encodes an MTE tag to the upper part of the pointer, but in ZOMETAG, the total usable address space is not limited to 4 GB. Furthermore, SGXBounds requires additional memory operations to retrieve lower bounds as they are stored separately, while ZOMETAG avoids this overhead. Deltapointers also modifies the pointer representation to include the distance from the current pointer to the end of the object, known as the delta tag, in higher bits. It detects spatial errors by observing overflow in the delta tag when the pointer points to an out-of-bounds address. However, Deltapointers cannot detect spatial memory violations beyond the lower bound, whereas ZOMETAG is capable of detecting such violations.

BIMA [39], No-FAT [40], and AOS [41] have proposed hardware-supported BC scheme. Specifically, these approaches involve add special hardware components (*e.g.*, bounds checking module) to the microarchitecture. They also introduce dedicated instructions for performing bounds checks efficiently. As a result of this hardware support, they have achieved efficient spatial memory safety, with AOS achieving an 8.6% overhead and No-FAT achieving an 8% overhead. However, the aggressive changes required in the hardware design limit the immediate practicality and adoption of these approaches. On the other hand, ZOMETAG does not require any modifications to the hardware and can be applied to systems that provide MTE of the ARM architecture. Since ARM is a widely used processor architecture in mobile and embedded systems, ZOMETAG can be more easily utilized without the need for extensive hardware changes.

B. Red Zone Approach

There are existing approaches, such as AddressSanitizer [12] and LBC [42], that use red zones around memory objects to detect spatial memory errors. Specifically, this approach consists of the following steps. First, they install red zones between memory objects. Then, whenever the pointer is dereferenced, they check whether the accessed memory address is in the red zone or not. Our zone-based isolation is also inspired by the red zone approach. However, there are several differences between existing studies and ZOMETAG in terms of how the red zone is used. First, in ZOMETAG, the red zone is placed between the blue zones, not between each memory object. In other words, the purpose of the red zone in ZOMETAG is to detect erroneous memory accesses that cross zone boundaries. Additionally, ZOMETAG is able to detect red zone violations by instrumenting pointer arithmetic operations.

C. Memory Tagging Approach

There have been various studies, such as Loki [43] and CHERI [44], that enforce spatial memory safety using Tagged Memory Architectures (TMA). These approaches assign tags to pointers and memory objects, allowing memory access only when the tag values of the pointer and target memory address match. The security guarantees of these approaches are determined by the size of the tags. For this reason, existing TMA studies utilize a considerable size of storage for tags. For example, CHERI [44] uses 128-bit for fat pointer, and Loki [43] PUMP [45] support 64-bit tags. However, such an extensive size of tags has not been utilized in commodity processors due to hardware cost and performance issues. Instead, in real-world scenarios, processors typically use a small number of tags, *e.g.*, ARM MTE provides 4-bit tags. ZOMETAG is the first work that demonstrates deterministic spatial memory safety with such a limited number of tags.

D. Software Fault Isolation (SFI)

CHANCEL [46] and TDI [47] have proposed Software Fault Isolation (SFI) schemes to limit unauthorized memory accesses by modifying the memory layout. First, while servicing requests, CHANCEL reserves a per-thread private memory region for storing sensitive and a shared read-only memory region. Then, it applies SFI to isolate a thread from each other by instrumenting code for spatial safety. TDI enforces type-based data isolation, which allocates an area of memory, called *arena*, based on pointer types. Conceptually, their arena corresponds to our zone. Therefore, TDI is related to ZOMETAG in that both employ the zone/arena-based isolation mechanism. However, there are clear differences because TDI requires an additional instruction for masking to restrict the arena boundaries, whereas pointer arithmetic in ZOMETAG does not require additional instructions by forcing the use of a 32-bit register as an operand to the offset value. Furthermore, TDI that enforces type-safety cannot defend against overflows that occur within the same memory region, whereas ZOMETAG can detect using different tags between objects that can cause spatial errors in the same region. This gives ZOMETAG the advantage of being more efficient than TDI and providing complete memory safety.

IX. Conclusion

ARM MTE is expected to offer architectural support for the development of efficient MT solutions to ensure memory safety. Unfortunately, in reality, however, being subject to hardware limitations, it cannot afford enough physical tags to isolate objects for a fast, deterministic safety guarantee. To overcome the shortage of tags, ZOMETAG extends the notion of a tag to form pairs of a zone and an MTE tag as the new tags used for object isolation. The maximum count of these extended tags (*i.e.*, zone-tag pairs) is technically equal to the number of zones multiplied by that of MTE tags. Since there is an abundant number of zones available on modern 64-bit machines, ZOMETAG is able to supply enough tags to uniquely assign to virtually all program objects at runtime, as empirically demonstrated in this paper. We have discussed how ZOMETAG deterministically enforces spatial safety with the two-layer isolation mechanism that prevents a pointer from referring to the object assigned a zone-tag pair different from what it is currently assigned. Not only that, our experiments prove that by taking full advantage of ARM's support for our zone-based memory tagging, ZOMETAG manages to achieve spatial safety with relatively lower overhead than existing solutions.

References

[1] A. Holdings, "ARM architecture reference manual, ARMv8, for ARMv8—A architecture profile," ARM, Cambridge, U.K., Tech. Rep. ARM DDI 0487F.b (ID040120), 2021.

[2] K. Aingaran et al., "M7: Oracle's next-generation Sparc processor," *IEEE Micro*, vol. 35, no. 2, pp. 36–45, Mar. 2015.

[3] P. Nasahl, R. Schilling, M. Werner, J. Hoogerbrugge, M. Medwed, and S. Mangard, "CrypTag: Thwarting physical and logical memory vulnerabilities using cryptographically colored memory," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, May 2021, pp. 200–212.

[4] A. Holdings. *ARMv8.5-A Memory Tagging Extension*. Accessed: May 2023. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf

[5] N. Burow, D. McKee, S. A. Carr, and M. Payer, "CUP: Comprehensive user-space protection for C/C++," in *Proc. Asia Conf. Comput. Commun. Secur.*, May 2018, pp. 381–392.

[6] G. J. Duck and R. H. C. Yap, "EffectiveSan: Type and memory error detection using dynamically typed C/C++," in *Proc. 39th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2018, pp. 181–195.

[7] T. Kroes, K. Koning, C. Giuffrida, H. Bos, and E. van der Kouwe, "Fast and generic metadata management with mid-fat pointers," in *Proc. 10th Eur. Workshop Syst. Secur.*, Apr. 2017, pp. 1–6.

[8] D. Chisnall et al., "Beyond the PDP-11: Architectural support for a memory-safe C abstract machine," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 117–130, 2015.

[9] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: An empirical study of Intel MPX and software-based bounds checking approaches," 2017, *arXiv:1702.00719*.

[10] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proc. 10th ACM Conf. Comput. Commun. Secur.*, Oct. 2003, pp. 272–280.

[11] A. Francillon and C. Castelluccia, "Code injection attacks on Harvard-architecture devices," in *Proc. 15th ACM Conf. Comput. Commun. Secur.*, Oct. 2008, pp. 15–26.

[12] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Tech. Conf. (USENIX)*, 2012, pp. 309–318.

[13] B. Lee et al., "Preventing use-after-free with dangling pointers nullification," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.

[14] E. van der Kouwe, V. Nigade, and C. Giuffrida, "DangSan: Scalable use-after-free detection," in *Proc. 12th Eur. Conf. Comput. Syst.*, Apr. 2017, pp. 405–419.

[15] J. Shin, D. Kwon, J. Seo, Y. Cho, and Y. Paek, "CRCount: Pointer invalidation with reference counting to mitigate use-after-free in legacy C/C++," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–15.

[16] I. Haller et al., "TypeSan: Practical type confusion detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 517–528.

[17] C. Meadows, "A procedure for verifying security against type confusion attacks," in *Proc. 16th IEEE Comput. Secur. Found. Workshop*, Jun. 2003, pp. 62–72.

[18] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, p. 71, Mar. 2015.

[19] V. van der Veen et al., "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 1675–1689.

[20] G. Chen et al., "SafeStack: Automatically patching stack-based buffer overflow vulnerabilities," *IEEE Trans. Depend. Secure Comput.*, vol. 10, no. 6, pp. 368–379, Nov. 2013.

[21] G. J. Duck, R. H. C. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.

[22] A. Holdings. *Cortex-m1*. Accessed: May 2023. [Online]. Available: https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m1

[23] A. Holdings. *Fast Models*. Accessed: May 2023. [Online]. Available: https://developer.arm.com/tools-and-software/simulation-models/fast-models

[24] Odroid. *Odroid-c4*. Accessed: May 2023. [Online]. Available: https://wiki.odroid.com/odroid-c4/odroid-c4

[25] Spec2006. *Spec2006*. Accessed: May 2023. [Online]. Available: https://www.spec.org/cpu2006/

[26] D. Kuvaiskii et al., "SGXBOUNDS: Memory safety for shielded execution," in *Proc. 12th Eur. Conf. Comput. Syst.*, Apr. 2017, pp. 205–221.

[27] G. J. Duck and R. H. C. Yap, "Heap bounds protection with low fat pointers," in *Proc. 25th Int. Conf. Compiler Construction*, Mar. 2016, pp. 132–142.

[28] AddressSanitizer. *AddressSanitizer*. Accessed: May 2023. [Online]. Available: https://clang.llvm.org/docs/AddressSanitizer.html

[29] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, "Delta pointers: Buffer overflow checks without the checks," in *Proc. 13th EuroSys Conf.*, 2018, pp. 1–14.

[30] AddressSanitizer. *AddressSanitizer*. Accessed: May 2023. [Online]. Available: https://github.com/vusec/instrumentation-infra/blob/master/infra/targets/spec2006/asan.patch

[31] PARSEC. *The Parsec Benchmark Suite*. Accessed: May 2023. [Online]. Available: https://parsec.cs.princeton.edu

[32] Memcached. *Memcached*. Accessed: May 2023. [Online]. Available: https://rpmfind.net/linux/RPM/centos/7.9.2009/x86_64/Packages/memcached-1.4.15-10.el7_3.1.x86_64.html

[33] Nginx. *Ningx-1.4.0*. Accessed: May 2023. [Online]. Available: http://nginx.org/en/download.html

[34] Lighttpd. *Lighttpd*. Accessed: May 2023. [Online]. Available: https://www.lighttpd.net/2016/7/16/1.4.40/

[35] Apache. *Apache Http Server Benchmarking Tool*. Accessed: May 2023. [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html

[36] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "BugBench: Benchmarks for evaluating bug detection tools," in *Proc. Workshop Eval. Softw. Defect Detection Tools*, vol. 5. Chicago, IL, USA, 2005, pp. 1–5.

[37] V. Kuznetzov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *The Continuing Arms Race: Code-Reuse Attacks Defenses*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 81–116.

[38] A. Khan, H. Kim, B. Lee, D. Xu, A. Bianchi, and D. J. Tian, "M2MON: Building an MMIO-based security reference monitor for unmanned vehicles," in *Proc. 30th {USENIX} Secur. Symp. ({USENIX} Secur.)*, 2021, pp. 285–302.

[39] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, and A. DeHon, "Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2013, pp. 721–732.

[40] M. T. I. Ziad, M. A. Arroyo, E. Manzhosov, R. Piersma, and S. Sethumadhavan, "No-FAT: Architectural support for low overhead memory safety checks," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2021, pp. 916–929.

[41] Y. Kim, J. Lee, and H. Kim, "Hardware-based always-on heap memory safety," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2020, pp. 1153–1166.

[42] N. Hasabnis, A. Misra, and R. Sekar, "Light-weight bounds checking," in *Proc. 10th Int. Symp. Code Gener. Optim.*, Mar. 2012, pp. 135–144.

[43] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware enforcement of application security policies using tagged memory," in *Proc. OSDI*, vol. 8, 2008, pp. 225–240.

[44] J. Woodruff et al., "CHERI concentrate: Practical compressed capabilities," *IEEE Trans. Comput.*, vol. 68, no. 10, pp. 1455–1469, Oct. 2019.

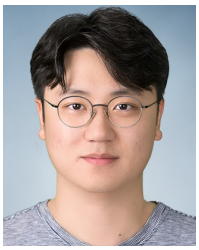
[45] U. Dhawan et al., "Architectural support for software-defined metadata processing," in *Proc. 20th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2015, pp. 487–502.

[46] A. Ahmad, J. Kim, J. Seo, I. Shin, P. Fonseca, and B. Lee, "CHANCEL: Efficient multi-client isolation under adversarial programs," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–18.

[47] A. Milburn, E. Van Der Kouwe, and C. Giuffrida, "Mitigating information leakage vulnerabilities with type-based data isolation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 1049–1065.

Jiwon Seo received the B.S. degree in electrical and computer engineering from Seoul Women's University, South Korea, in 2016, and the Ph.D. degree in electrical and computer engineering from Seoul National University, South Korea, in 2023. Her research interests include the system security against various types of threats.

Junseung You received the B.S. degree in electrical and computer engineering from Seoul National University, South Korea, in 2019, where he is currently pursuing the Ph.D. degree in electrical and computing engineering. His research interests include system security against various types of threats.



Donghyun Kwon received the B.S. and Ph.D. degrees in electrical and computer engineering from Seoul National University, South Korea, in 2012 and 2019, respectively. He is currently a Professor with the School of Computer Science and Engineering, Pusan National University, South Korea. His research interests include the system security against various types of threats.

Yeongpil Cho received the B.S. degree in electrical engineering from POSTECH, South Korea, in 2010, and the Ph.D. degree in electrical and computer engineering from Seoul National University, South Korea, in 2018. Currently, he is a Professor with the Department of Computer Science, Hanyang University. His research interests include the system security against various types of threats.

Yunheung Paek (Member, IEEE) received the B.S. and M.S. degrees in computer engineering from Seoul National University, South Korea, in 1988 and 1990, respectively, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 1997. Currently, he is a Professor with the Department of Electrical and Computer Engineering, Seoul National University. His research interests include system security with hardware, the secure processor design against various types of threats, and machine learning-based security solution.