2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)

# Hardware-based Always-On Heap Memory Safety

Yonghae Kim
*Georgia Institute of Technology*
yonghae@gatech.edu

Jaekyu Lee
*Arm Research*
jaekyu.lee@arm.com

Hyesoon Kim
*Georgia Institute of Technology*
hyesoon@cc.gatech.edu

***Abstract*—Memory safety violations, caused by illegal use of pointers in unsafe programming languages such as C and C++, have been a major threat to modern computer systems. However, implementing a low-overhead yet robust runtime memory safety solution is still challenging. Various hardware-based mechanisms have been proposed, but their significant hardware requirements have limited their feasibility, and their performance overhead is too high to be an always-on solution.**

**In this paper, we propose AOS, a low-overhead always-on heap memory safety solution that implements a novel bounds-checking mechanism. We identify that the major challenges of existing bounds-checking approaches are 1) the extra instruction overhead for memory checking and metadata propagation and 2) the complex metadata addressing. To address these challenges, using Arm PA primitives, we leverage unused upper bits of a pointer to store a key and have it propagated along with the pointer address, eliminating propagation overhead. Then, we use the embedded key to index a hashed bounds table to achieve efficient metadata management. We also introduce a micro-architectural unit to remove the need for memory checking instructions. We show that AOS overcomes all the aforementioned challenges and demonstrate its feasibility as an efficient runtime memory safety solution. Our evaluation for SPEC 2006 workloads shows an 8.4% performance overhead on average.**

***Index Terms*—Security; software and system safety; pointer authentication;**

## I. INTRODUCTION

Memory safety violations have been a conventional but persistent problem in computer systems. Memory safety issues have inherently existed in unsafe programming languages such as C and C++ because of the illicit use of pointers. Recent industry reports [1], [2] revealed that memory safety errors addressed in their products accounted for more than 70% of all security issues. This demonstrates that memory safety errors are still prevalent and exploitable by attackers.

Researchers have proposed extensive amounts of software- and hardware-based work to prevent such vulnerabilities. Software techniques [3]–[7] provide strong security guarantees, but they are not suitable runtime solutions because of their significant performance overhead. Instead, their primary purpose is for testing and debugging. For example, AddressSanitizer (ASan), one of the most popular memory error detectors, showed a 73% slowdown [3].

Hardware-based mechanisms typically achieve less performance overhead, but they do not attain desired properties altogether, such as broad security coverage, high performance, and low hardware overhead. Instead, they trade off one property for another. For example, hardware-based blacklisting mechanisms [8], [9] set the surrounding regions of memory objects as redzones and prohibit their access to prevent over/underflow attacks. Such an approach is efficient since monitoring could be performed in parallel with normal operations, as in REST [8]. However, they cannot prevent non-adjacent illegal accesses that jump over the redzones. Given the upward trend of non-adjacent spatial safety violations (over 60% since 2014) [2], we expect that their effectiveness will be increasingly limited.

Whitelisting mechanisms enforce memory operations to only access allowed memory locations, providing stronger security capabilities. For example, bounds-checking mechanisms [10]–[13] associate bounds metadata with pointers to protect and perform address range checking. Despite more powerful security guarantees, they incur significant hardware changes and design complexity. Their performance overhead is also too high to be an always-on solution.

We observe that the major challenges of existing bounds-checking approaches are 1) the extra instruction overhead for memory checking and metadata propagation and 2) the complex metadata addressing. For instance, Watchdog [11], a prior hardware-based bounds-checking mechanism, showed 44% more dynamic instruction counts, causing significant performance degradation. It also requires register extensions (up to 256-bit) to propagate the metadata and use it inside a CPU core, which significantly increases power consumption. Moreover, prior approaches often require a complex addressing scheme for metadata accesses. For example, Intel Memory Protection Extensions (MPX) [12] requires approximately three register-to-register moves, three shifts, and two memory loads to access its hierarchical bounds table.

To tackle these challenges, we propose **AOS**, a low-overhead **A**lways-**O**n memory **S**afety solution for heap protection that implements a novel bounds-checking mechanism. We utilize Arm pointer authentication (PA) primitives [14] to store a pointer authentication code (PAC) into the unused high-order bits of a pointer for memory safety. By doing so, we allow the embedded PAC to be passed along with the pointer address, removing extra instructions for metadata propagation. Furthermore, we sign all data pointers returned by dynamic memory allocation, i.e., placing a PAC into the pointer, and use the PAC to index a hashed bounds table that stores bounds metadata. This scheme enables efficient metadata management since the addressing becomes simplified using the base address of the table and the PAC as an offset.

To remove additional instructions required by prior work [10]–[12] for bounds checking, we introduce a new

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00095

micro-architectural structure, a memory check unit (MCU). In AOS, every memory instruction is enqueued in the MCU when it is issued to the load-store unit (LSU). If a pointer address is signed, i.e., has an embedded PAC, we perform bounds checking to validate the access, which enables an efficient *selective* memory safety checking mechanism.

AOS achieves efficient yet complete spatial and temporal memory safety for a heap region. AOS prevents spatial safety violations (e.g., out-of-bounds access) by checking bounds for all signed memory accesses. Moreover, AOS can detect temporal errors, such as the use of a dangling pointer, use-after-free (UAF), and double free. When a signed data pointer is freed, AOS clears the associated bounds information while leaving the pointer as being signed. Because of the absence of its bounds metadata, subsequent use of the pointer will fail in bounds checking. With the prevalence of heap memory vulnerabilities, AOS provides robust protection against the most prevailing attack vectors.

We also discuss how AOS can be extended to support pointer integrity by utilizing Arm PA primitives and achieve practical defenses against runtime control-flow attacks, such as return-oriented programming (ROP) [15] and jump-oriented programming (JOP) [16], and data-oriented attacks by corrupted data pointers. AOS also provides precise exception handling by delaying architectural state updates until an instruction retires with a successful bounds checking. This enables AOS to prevent leakage of secret data by an illegal read and memory corruption by an illegal write.

Given the limited PAC size (11 to 32 bits) under typical virtual address schemes in a processor, some memory objects may have the same PAC value, causing PAC collisions. To address this issue, we develop a multi-way bounds-table structure with gradual resizing to accommodate multiple bounds metadata for each PAC. A process begins its execution with a modest-size table and increases the associativity of the table upon an insertion failure due to insufficient capacity. This approach enables efficient and scalable bounds-table management.

This paper claims the following contributions:

- We propose AOS, which overcomes the main challenges of existing bounds-checking approaches and realizes a practical bounds-checking mechanism for heap protection.
- We implement the AOS design and present performance evaluation for SPEC 2006 workloads. Our results show a marginal 8.4% performance overhead on average.
- We describe how AOS can cooperate with pointer integrity solutions, which demonstrates that the security capabilities of AOS can be extended with little overhead.
- We present a security analysis and demonstrate the effectiveness of the AOS protection mechanism.

## II. Background

### A. Memory Safety Violations

Memory safety violations are a substantial threat to modern computer systems since they can lead to system crashes and security vulnerabilities. Fig. 1 shows an example of heap exploitation, *House of Spirit*, which is a data-oriented attack on *glibc*. The attack crafts a data pointer controlled by an attacker so it can bypass the security tests of `free()`. Once freed, the pointer is inserted into a fastbin, which is one of the linked lists holding free chunks. Then, the next `malloc()` returns the address 16 bytes ahead of the crafted data pointer and ends up allowing subsequent malicious operations on the attacker-controlled memory locations.

```
1  struct fast_chunk {
2    size_t prev_size, size;
3    struct fast_chunk *fd, *bk;
4    char buf[0x20];
5  };
6
7  struct fast_chunk fchunk[2];
8  void *ptr, *victim;
9
10 // Craft chunks to pass security tests
11 fchunk[0].size = sizeof(struct fast_chunk);
12 fchunk[1].size = sizeof(struct fast_chunk);
13
14 // Attacker overwrites a pointer
15 ptr = (void *) &fchunk[0].fd;
16
17 // fchunk[0] gets inserted into fastbin
18 free(ptr);
19
20 // Returns 16 bytes ahead of fchunk[0]
21 victim = malloc(0x30);
```

Fig. 1. Heap exploitation example: House of Spirit.

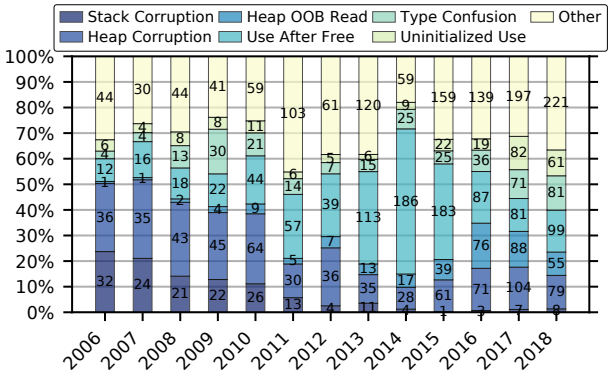



Fig. 2. The root cause trend of memory safety issues.

Fig. 2 shows a root cause trend of memory safety vulnerabilities reported by Microsoft [2]. We observe that stack corruption errors show a downward trend thanks to software protection methods, including canaries, address space layout randomization (ASLR), and pointer authentication. It also shows that heap vulnerabilities, such as heap corruption, out-of-bounds (OOB) read, and use-after-free, have been dominant in recent years. Since other vulnerabilities, such as type confusion and uninitialized use, can be detected and mitigated by static analysis tools [17]–[19], we recognize that heap memory vulnerabilities are the most problematic and challenging.

```
1 pacia lr, sp                  // sign lr
2 stp fp, lr, [sp, #-FS]!
3 mov fp, sp
4 // function body
5 ldp fp, lr, [sp], #FS
6 autia lr, sp                  // authenticate lr
7 ret lr
```

Fig. 3. Return address signing using Arm PA.

### B. Arm Pointer Authentication

Arm PA is a security primitive added in the Armv8.3-A extension [14]. It is designed to prevent illicit pointer modification by adding new instructions for signing and authenticating pointer authentication codes (PACs). The PAC is calculated using a cryptographic algorithm such as the QARMA block cipher [20]. QARMA takes two inputs, a pointer, and a context. The truncated output from QARMA becomes the PAC. The effective virtual address space in 64-bit architectures is typically less than 64 bits. By leveraging this idea, Arm PA places a PAC into the unused high-order bits of a pointer to protect and verifies its integrity by authenticating the PAC before use. Depending on the virtual address scheme in a processor, the PAC size ranges from 11 to 32 bits.

Fig. 3 shows a return address signing scheme using Arm PA. In the prologue, `pacia` (line 1) computes and embeds a PAC into the return address (`lr`) using a stack pointer (`sp`) as a modifier and `key A`. In the epilogue, `autia` (line 6) authenticates the return address using the same modifier and key pair. If a pointer is corrupted, the authentication fails, and any further use of the pointer leads to a translation fault. Prior work [21] has reported a negligible performance overhead (∼0.5%) by return address signing.

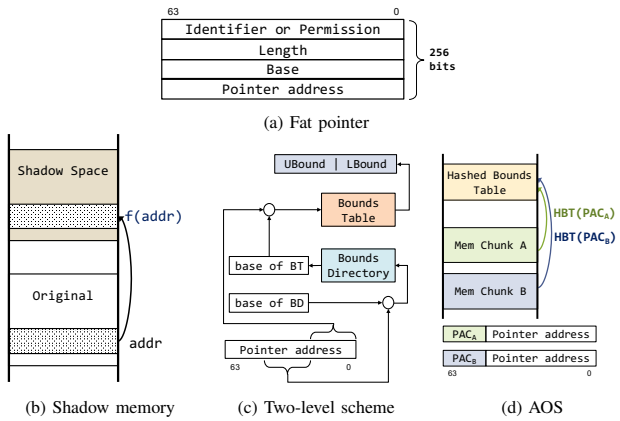
Despite its low overhead, Arm PA does not provide spatial and temporal safety. It can detect neither out-of-bounds accesses nor temporal errors. The prevalence of such spatial (e.g., heap OOB read) and temporal (e.g., use-after-free) violations, shown in Fig. 2, demonstrates that the PA mechanism cannot be a sole security solution, and cooperation with a stricter defense is essential.

## III. AOS: Always-On Memory Safety

In this section, we describe challenges in implementing an efficient bounds-checking mechanism and how AOS overcomes such challenges. Then, we outline AOS and its operations.

### A. Challenges

*Challenge 1. Register extension.* Many prior studies [10], [11], [22], [23] implement a fat pointer to hold metadata needed for memory checking by extending every register. For example, Watchdog [11] extends registers up to 256 bits for bounds and use-after-free checking. A fat pointer itself contains the metadata, as shown in Fig. 4a, and every pointer dereferencing is checked with its associated metadata. However, such an approach not only requires changes to almost the entire pipeline stages but also increases power consumption because of the increased bit width of register read/write operations.




(a) Fat pointer

(b) Shadow memory (c) Two-level scheme (d) AOS

Fig. 4. Comparison of metadata management methods.

```
1  // ① Heap Allocation
2  ptr = malloc(size)
3  key = unique_id++
4  lock = new_lock()
5  *(lock) = key
6  id = (key, lock)
7  q = setid(ptr, id)
8  // ② Heap Deallocation
9  id = getid(ptr)
10 *(id.lock) = INVALID
11 add_free_list(id.lock)
12 free(ptr)
13 // ③ Load
14 check R2.id
15 ld R1.id <- ShadowMem[R2].id
16 ld R1 <- Mem[R2]
17 // ④ Store
18 check R2.id
19 ShadowMem[R2].id <- R1.id
20 Mem[R2].val <- R1
21 // ⑤ Add Immediate
22 R1.id <- R2.id
23 add R1 <- R2 + imm
24 // ⑥ Add
25 if (R2.id != NULL)
26   R1.id <- R2.id
27 else
28   R1.id <- R3.id
29 add R1 <- R2 + R3
```

(a) Watchdog

```
1  // ① Heap Allocation
2  ptr = malloc(size)
3  pacma ptr, sp, size
4  bndstr ptr, size
5
6
7
8  // ② Heap Deallocation
9  bndclr ptr
10 xpacm ptr
11 free(ptr)
12 pacma ptr, sp
13 // ③ Load
14 ld R1 <- Mem[R2]
15
16
17 // ④ Store
18 Mem[R2].val <- R1
19
20
21 // ⑤ Add Immediate
22 add R1 <- R2 + imm
23
24 // ⑥ Add
25 add R1 <- R2 + R3
26
27
28
29
```

(b) AOS

Fig. 5. Comparison of memory safety operations between Watchdog and AOS. Extra instructions are blue colored.

*Challenge 2. Bounds-checking operation.* Bounds checking is typically performed by extra instructions [10]–[12]. For instance, Watchdog inserts a `check` $\mu$op before every memory access (③ and ④ in Fig. 5a). The `check` $\mu$op uses its identifier (`id`) for use-after-free detection and bounds checking. Adding `check` $\mu$ops resulted in 29% more dynamic instructions in their evaluation.

*Challenge 3. Metadata propagation.* Since the destination register does not automatically inherit the metadata from the source register, additional instructions are necessary to propagate the metadata throughout program execution. For example, Watchdog requires the extra instructions for the

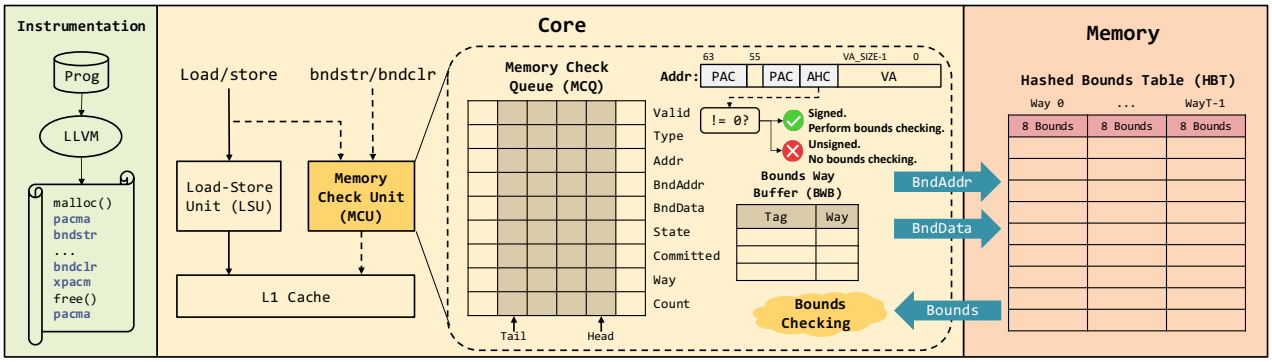



Fig. 6. AOS overview.

pointer arithmetic (⑤ and ⑥ in Fig. 5a).

*Challenge 4. High memory overhead.* Bounds metadata often incurs significant memory overhead. In particular, shadow memory implementations mirror virtual address space to the shadow space [3], [10], as shown in Fig. 4b, which results in severe memory fragmentation caused by page-granularity allocation. For instance, ASan reserves one-eighth of the virtual address space for shadow space [3].

*Challenge 5. Complex metadata addressing.* To perform bounds checking, the associated metadata of each pointer needs to be loaded from memory. Intel MPX [12] implements a two-level address translation scheme to access bounds metadata, as shown in Fig. 4c. This multi-stage translation requires approximately three register-to-register moves, three shifts, and two memory loads, causing significant performance degradation for memory-intensive applications.

### B. How We Overcome Challenges

To tackle the above challenges, we use Arm PA as a vehicle. If we sign a data pointer as Arm PA does, which places a PAC into its upper bits, the PAC is passed through along with the pointer address without any hardware support or extra operations. This scheme allows us to avoid register extension (Challenge 1) and additional instructions for metadata propagation (Challenge 3).

Furthermore, data-pointer signing leaves a mark in the pointer, indicating that the memory access by the pointer needs to be validated by bounds checking. Once we see a signed data pointer being dereferenced, we perform bounds checking and validate the access. Since the need for bounds checking is determined only by examining the pointer address, this approach removes the necessity of explicit extra instructions for bounds checking (Challenge 2).

Last but not least, using a pointer's PAC to index a bounds table simplifies the addressing scheme, as shown in Fig. 4d. We can calculate the metadata address by adding a PAC to the base address of a bounds table, enabling efficient bounds metadata accesses. Moreover, it alleviates the memory overhead, as it neither mirrors memory pages into shadow space nor causes significant memory fragmentation. As a result, we tackle challenges 4 and 5.

### C. AOS Operations

In this section, we briefly describe the operations of AOS. Fig. 6 shows an overview of AOS.

*1) Pointer Signing:* We sign data pointers returned by `malloc()` using extended Arm PA instruction set architectures (ISAs) (§IV-A). We instrument all `malloc` and `free` calls to generate a PAC and bounds-table management instructions (§IV-C) using our custom LLVM passes (§IV-B).

*2) Bounds Checking:* To enable bounds checking without explicit instructions, we add a memory check unit (MCU) (§V-A) in the core pipeline. When a memory instruction is issued to the load-store unit (LSU), it is also sent to the MCU. Since we sign every data pointer returned by `malloc()`, the MCU can identify memory accesses that require validation and perform selective bounds checking. For bounds-checking failures, we develop a new class of exceptions handled by the operating system (OS) (§IV-D).

*3) Metadata Management:* We store the bounds information of all heap memory chunks in a *hashed bounds table* (HBT) (§V-B), which is similar to a *hash table*. A key of the HBT is a PAC value embedded in a pointer. To handle hash collisions, i.e., too many bounds to store in a table row, we dynamically resize the HBT. For an efficient bounds-searching process, we keep track of recently used bounds in a tag buffer (§V-C). To reduce memory overhead and traffic, we develop a bounds-compression scheme (§V-D).

*4) Precise Exception:* We provide precise exception handling by preventing architectural state updates until an instruction requiring validation retires with successful bounds-checking. This prevents illegal memory accesses from reading sensitive data or writing malicious data.

### D. Threat Model

To demonstrate the effectiveness of our defense mechanism, we follow the typical attacker capabilities consistent with prior work. Our threat model assumes a powerful attacker who can exploit one or more memory vulnerabilities that may exist in a user space process, reading or writing arbitrary memory locations. We do not limit the location of vulnerabilities and possible attack vectors enabled by the attacker capabilities. However, we assume that the attacker cannot infer PA keys,

which are stored in special registers and must be invisible to user space. We also assume that hardware components are trustworthy and therefore leave out of scope any attacks that exploit hardware vulnerabilities, such as row-hammer attacks [24], side-channel attacks [25], [26], and speculative execution attacks [27], [28].

With the prevalence of heap vulnerabilities, our work focuses on heap protection. We believe that our approach can be applied to other data-pointer types (e.g., stack pointers) in a similar manner but leave this as future work.

## IV. Software Extensions

### A. ISA Extension

We introduce the following new instructions in AOS:

- `pacma/pacmb <Xd>,<Xn|SP>,<Xm>`: These are variants of `pacda/pacdb` in Arm PA that compute a PAC and insert it into a pointer `<Xd>` using a modifier `<Xn|SP>` and key A or B. `pacma/pacmb` take the additional third operand `<Xm>`, the size of a new memory chunk. This is used to calculate a 2-bit address hashing code (AHC), which is used to 1) indicate a signed (protected) pointer when the value is nonzero and 2) indicate a type of invariant bits in a pointer associated with a memory object. We detail how the AHC is created and used in Section V-C.
- `xpacm <Xd>`: This instruction is a variant of `xpacd`. It strips both PAC and AHC from a pointer.
- `autm <Xd>`: This is a variant of `autda/autdb`, Arm PA authentication instructions. `autm` authenticates a pointer `<Xd>` by checking whether the pointer was signed by AOS, i.e., has a nonzero AHC. However, it does not strip the AHC after authentication. If the AHC is zero, the pointer is considered corrupted, and the authentication fails.
- `bndstr <Xn>,<Xm>`: This instruction encodes 8-byte bounds metadata using a pointer address `<Xn>` and a size `<Xm>`, and stores them in an entry of the HBT.
- `bndclr <Xn>`: This instruction clears the bounds metadata associated with a pointer address `<Xn>` by writing an 8-byte zero value into its corresponding bounds in the HBT.

### B. Compiler Support

To instrument new instructions for bounds checking, we implement new passes to the optimizer and the AArch64 backend in LLVM [29]. The `AOS-opt-pass` optimizer pass is designed to detect memory allocation and deallocation calls and insert new intrinsic functions. For the allocation call, the `aos_malloc` intrinsic function is inserted with two operands after the call, the pointer address returned by `malloc()` and the size of a new memory chunk. Also, after the deallocation call, the `aos_free` intrinsic function is inserted with one operand, the signed pointer address to free. These intrinsics are detected by the new pass, `AOS-backend-pass`, in the AArch64 backend and replaced with new instructions, as shown in Fig. 7.

```
1 ptr = malloc (size);
2 pacma ptr, sp, size
3 bndstr ptr, size
4
```

(a) AOS-malloc

```
1 bndclr ptr
2 xpacm ptr
3 free (ptr);
4 pacma ptr, sp, xzr
```

(b) AOS-free

Fig. 7. Data-pointer instrumentation.

### C. Program Instrumentation

**Assign bounds upon memory allocation.** Fig. 7a shows data-pointer signing and bounds storing. AOS inserts two new instructions following `malloc()`. `pacma` (line 2) signs the data pointer returned by `malloc()` using a stack pointer (`sp`) as a modifier, and `bndstr` (line 3) computes bounds and stores them in the HBT.

**Release bounds upon memory deallocation.** Fig. 7b shows data-pointer stripping and bounds clearing. First, we insert `bndclr` before `free()` (line 1). The `bndclr` instruction clears the bounds metadata associated with the data pointer to free. The initialized entry will be reused later by a newly allocated memory object that has the same PAC. We also place `xpacm` (line 2) because freeing a memory chunk inherently invokes out-of-bounds accesses by the data pointer to free. In many standard C libraries, such as *glibc* (GNU C library), the memory allocator accesses the metadata of the prior and next memory chunks and performs block coalescing to reduce memory fragmentation. Since these accesses are legitimate, we use `xpacm` to strip the pointer temporarily and avoid unnecessary bounds checking during `free()`. Finally, we place `pacma` after `free()` (line 4) to re-sign the pointer. Since the size does not matter, we pass the zero register, `xzr`, as a size operand. Re-signing a pointer here, i.e., leaving a freed pointer as being signed, effectively *locks* the pointer. Any subsequent use of the pointer would fail in bounds checking because of the absence of its corresponding bounds in the HBT. In this way, AOS detects temporal violations.

Unlike prior studies [3], [8], our approach does not demand a quarantine pool to provide temporal safety. Given that the REST [8] software framework's use of a quarantine pool mostly contributed to its performance overhead, avoiding the use of a quarantine pool will be beneficial in terms of performance.

### D. OS Support

The OS handles bounds-table management (allocation and resizing) and bounds-checking failures. The OS creates an HBT when a process is initiated. AOS introduces a new class of exceptions, an AOS exception, to handle a new type of failure. If a CPU core detects an instruction faulted by a bounds operation, it triggers an AOS exception, and the OS handles it depending on the instruction type:

- `bndstr`: This indicates a bounds-store failure due to insufficient capacity in a row of the HBT. We allocate a new, larger bounds table and copy entries of the old table into the new one.

- `bndclr`: This indicates a bounds-clear failure due to double free or `free()` by an invalid address.
- Load/store: This indicates a bounds-checking failure, which could occur by one of the memory safety violations.

Upon a failure, the information will be signaled to a user. Developers can implement the exception handler to either 1) terminate the process or 2) report an error and resume.

## V. Hardware Extensions

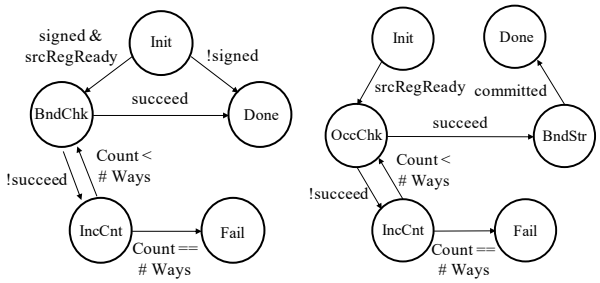
### A. Memory Check Unit

To perform memory safety checking without extra instructions, AOS adds a functional unit, called an MCU, inside a core. The MCU comprises a memory check queue (MCQ) and a bounds way buffer (BWB). It is located next to the LSU and performs bounds-table management and bounds checking. As shown in Fig. 6, we issue the bounds-table management instructions, `bndstr` and `bndclr`, directly to the MCU. To validate memory accesses when a memory instruction is issued to the LSU, we also send the instruction to the MCU. Thus, an instruction can be issued when both the LSU and the MCU are not full.

Then, we check the instruction enqueued in the MCU to see whether its pointer address has been signed by checking the AHC value. For signed pointers, AOS performs bounds checking, which may iterate over multiple entries in an HBT row until it finds valid bounds. To reduce the overhead of iterative bounds searching, we utilize 64-byte (a typical cache-line size in a core) load requests and perform *parallel* bounds checking using multiple bounds in the same cache line. Since each bounds metadata is compressed to 8 bytes (§V-D), up to eight bounds can be stored in one way of an HBT row. Note that all the bounds-access operations in AOS conform to existing cache coherence protocols and memory consistency models.

*1) Memory Check Queue:* The MCQ stores in-flight bounds store, clear, and checking operations. Each MCQ entry has the following fields:

- *Valid* indicates the validity of an entry.
- *Type* differentiates the instruction type between bounds-table management (`bndstr`/`bndclr`) and load/store.
- *Addr* stores a pointer address. During bounds checking, we compare the address against the lower and upper bounds.
- *BndData* holds 8-byte bounds from a `bndstr` or `bndclr` instruction to store them in the HBT.
- *BndAddr* stores the address of an HBT way.
- *Way* indicates which way to access in a row of the HBT.
- *Count* keeps the number of ways accessed so far for the given bounds-checking operation. *Count* will be incremented if valid bounds are not found at an attempt.
- *Committed* indicates whether the instruction has been committed from the ROB. To maintain the store-store ordering enforced by memory consistency, bounds-store requests should be sent in *program order*. Hence, they can be sent to memory only after *Committed* is set.
- Finally, *State* stores the operation state of the current finite state machine (FSM) in the MCQ.




(a) Load/store (b) `bndstr`/`bndclr`

Fig. 8. Finite state machines in the MCQ.

*2) Finite State Machines:* Depending on the instruction type, each MCQ entry operates in one of the FSM modes: one for `bndstr`/`bndclr` (Fig. 8a) or another for load/store (Fig. 8b). We describe the states and the transitions among them as follows.

**Init.** In this state, *BndAddr* and *BndData* are calculated for a `bndstr` or `bndclr`, and the current state transitions to the *OccChk* state. In the case of a signed load/store, *BndAddr* is calculated, and this state moves to the *BndChk* state. If the pointer is not signed, it moves to the *Done* state.

**OccChk.** The multiple bounds addressed by *BndAddr* are loaded from memory, and occupancy checking is performed. `bndstr` looks for an empty (zero) bounds, and `bndclr` checks if the loaded lower bound is the same as its pointer address. If it succeeds, this state transitions to *BndStr* and *IncCnt* otherwise.

**BndChk.** Parallel bounds checking is performed using the multiple bounds addressed by *BndAddr* in this state. If the valid bounds are found, this state moves to *Done*. Otherwise, it goes to the *IncCnt* state.

**BndStr.** The instruction waits until *Committed* is set, i.e., waits until the instruction is retired from the ROB. Once set, it sends a bounds-store request and moves to the *Done* state.

**IncCnt.** Upon entering this state, the instruction increases its *Count* by 1 and recalculates *BndAddr* to access the next way. If *Count* reaches the associativity of the HBT, a bounds failure occurs, and the state transitions to *Fail*. Otherwise, this state transitions back to the previous state, *OccChk* in the case of `bndstr`/`bndclr` or *BndChk* for a load or store.

**Fail.** This state indicates that the instruction has faulted because of a bounds failure. When an entry retires from the MCQ after becoming the head, an AOS exception is raised and handled by the OS. Before becoming the head, this entry can be replayed by the store-load replay mechanism, which is explained in Section V-E.

**Done.** The instruction in this state has completed its bounds operation and is ready to commit. The head entry in the *Done* state with *Committed* set, i.e., the instruction is retired from the ROB, can be deallocated from the MCQ.

### B. Bounds Table with Gradual Resizing

We store bounds metadata in a per-process bounds table, called an HBT, the design of which is inspired by Cuckoo

hashing [30], which specializes in hash collision resolution. Our HBT design shares some similarities with the Cuckoo hash table, but there are notable differences. Whereas a hash value in Cuckoo hashing is newly calculated whenever a hash table is accessed, AOS calculates a PAC with the base address of a newly allocated memory chunk and keeps the same PAC for a pointer. Furthermore, as opposed to Cuckoo hashing, which maintains multiple hash tables indexed with different hash functions, AOS maintains one hash table indexed with PACs that are embedded in pointers.

The table has a fixed number of rows, which is the same as the range of PAC values. For example, we have 64K rows for the 16-bit PAC size. An HBT is a multi-way structure to accommodate multiple bounds of different heap memory chunks that have the same PAC. To handle overflow, AOS develops a *gradual resizing* scheme. The process begins with a modest-size bounds table and dynamically resizes the table when an insertion failure occurs. Upon resizing, we create a new table that is twice the size of the original table by doubling the associativity. To compute a bounds address, we add two registers: 1) `BND_BASE` to store the base address of the HBT and 2) `BND_ASSOC`[1] to store the current associativity of the HBT. The row offset is computed using a PAC and `BND_ASSOC`, as shown in Eq. 1. Then, `BndAddr` is calculated using Eq. 2, where $W$ is the way to access. Note that `BndAddr` is always 64B aligned to bring a single cache line.

$$RowOffset = PAC << (log_2 BND\_ASSOC + 6) \quad (1)$$

$$BndAddr = BND\_BASE + RowOffset + (W << 6) \quad (2)$$

### C. Bounds Way Buffer

Although we perform parallel bounds checking, bounds searching that incurs multiple cache line accesses may degrade performance since we prevent an instruction from being retired until it finishes its validation. Hence, minimizing the number of iterations is important to avoid performance loss. To optimize the searching process, we introduce a tag buffer, BWB, which keeps track of recent bounds-table accesses and gives the correct location of valid bounds for subsequent bounds checking. Each BWB entry stores a 32-bit tag and the last used way in the HBT. Using only PACs as tags is not enough, since all bounds in the same row have the same PAC. Thus, we concatenate PACs with partial memory addresses and AHCs and use as tags. However, native concatenation can result in different tags for pointer addresses that belong to a same memory chunk. This is because even the upper bits of pointer addresses can change depending on the memory sizes. To resolve this issue, we utilize an embedded AHC that indicates the invariant bits within the memory region.

Algorithm 1 describes how we determine the invariant bits and calculate the AHC. We categorize memory objects into three groups based on the size: 1) small: about 64B size, 2) medium: about 256B size, and 3) a large group. Note that these are the bin sizes used by typical memory allocators.

[1]We consider only power-of-two associativities.

As explained in Section IV-A, `pacma`/`pacmb` embed an AHC along with a PAC in a pointer upon pointer signing.

**Algorithm 1** AHC calculation (*Addr*, *Size*)

```
1: tAddr := Addr ⊕ (Addr+Size-1)
2: if tAddr[VA-1:7] = 0 then // VA: virtual address size
3:     AHC := 1 // ≈ 64-byte chunk
4: else if tAddr[VA-1:10] = 0 then
5:     AHC := 2 // ≈ 256-byte chunk
6: else
7:     AHC := 3 // >≈ 256-byte chunk
8: return AHC
```

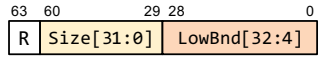
Algorithm 2 describes how a tag is calculated. It takes three inputs: 1) a pointer address, 2) an AHC, and 3) a PAC. Based on the AHC value, we derive different bits from the pointer. When checking bounds, we compute the tag and look up the BWB to find a matching entry. If found, the corresponding way is used to calculate the bounds address to access. Otherwise, a searching process starts from way-0. `bndstr` always retrieves way-0. When an instruction retires from the MCQ, the BWB is updated with its tag and the last accessed HBT way.

**Algorithm 2** BWB tag calculation (*Addr*, *AHC*, *PAC*)

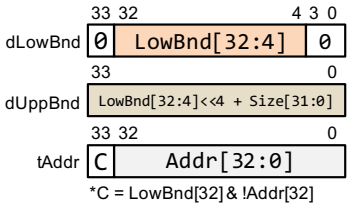
```
1: if AHC = 1 then
2:     return concat(PAC[15:0], Addr[20:7], AHC[1:0])
3: else if AHC = 2 then
4:     return concat(PAC[15:0], Addr[23:10], AHC[1:0])
5: else
6:     return concat(PAC[15:0], Addr[25:12], AHC[1:0])
```

### D. Bounds Compression

Malloc-intensive applications require a larger HBT. Since bounds are stored in the cache hierarchy, AOS incurs more cache insertions, causing cache pollution. To alleviate such side effects, AOS develops a bounds-compression scheme that takes advantages of the following two observations: 1) `malloc()` returns a 16-byte aligned address, and 2) `malloc()` takes a 32-bit size argument. Compared to 16-byte (lower and upper bounds) and 12-byte (base and size) bounds, we encode 8-byte bounds using a 29-bit partial address (bits 4 to 32) from the base address (lower bound) and a 32-bit memory size. Fig. 9a shows the bounds-compression format.




(a) 8-byte compressed bounds (R: reserved bits)




(b) Decompressed bounds and truncated address to be compared.

Fig. 9. Bounds compression and decompression format.

For bounds checking, we first calculate the truncated address (tAddr) from the current pointer address (Addr), as shown in Fig. 9b. Then, tAddr is compared against the lower bound (dLowBnd) and the upper bound (dUppBnd) decoded from the compressed bounds. Note that C-bit in tAddr is used to compensate for losing a carry bit, caused by partial address encoding (LowBnd[32:4] in Fig. 9a).

While we preserve the lower 33 bits of the lower bound, losing upper bits may cause false positives across pointer addresses that share the same lower bits. However, we believe that this is hardly exploitable because colliding addresses should be at least 8GB apart in the virtual address layout. Moreover, they should have the same PAC to pass bounds checking, which is rare considering the high PAC entropy. We further discuss false positives in Section VII-E.

### E. Store-Load Replay

Modern micro-architectures are equipped with out-of-order execution to improve performance, and AOS implements a similar out-of-order MCQ execution. To conform to underlying cache coherence protocols and memory consistency models, we develop a store-load replay mechanism. To preserve store-load ordering, when sending a bounds store request to the cache, bndstr and bndclr check newer MCQ entries. All newer entries with the same PAC need to be replayed with an initialized Count unless an entry is in the *Done* state because it has completed its execution with valid bounds. Since an HBT is private to each process, AOS does not consider load-load replay or other inter-process memory consistency issues within a core or across multiple cores.

### F. Optimization

*1) Bounds Cache:* Even with our novel bounds-compression scheme, bounds metadata may occupy a non-trivial portion of the L1 cache. To reduce the cache pollution, AOS optionally adds a bounds cache, *L1 B-cache*, which is similar to a lock location cache in Watchdog [11]. While the rest of the cache hierarchy remains the same, we store all bounds metadata in the L1 B-cache, instead of in the L1 D-cache. Since the size of bounds information is small and constant (8-byte), regardless of a memory chunk size, a modest-size cache (e.g., a few tens of KB) can be sufficient.

*2) Bounds Forwarding:* AOS allows bounds forwarding from a store to a load. When sending a bounds-load request, if an older bounds store with the same PAC is found, its bounds are forwarded to the load, and bounds checking is performed using the forwarded bounds. This forwarding can improve performance by 1) reducing memory accesses for loading bounds and 2) examining bounds early without waiting for the bounds to arrive. Bounds forwarding can be effective, especially when a pointer is dereferenced right after the memory allocation.

*3) Bounds Table Access During Resizing:* Resizing an HBT is expensive since we need to copy all the entries from the old table into a new one. Stalling a process until resizing is done would incur a non-trivial performance overhead. Inspired by a parallel hash table look-up in the Elastic Cuckoo Hash Table [31], we develop a similar non-blocking bounds-table access scheme and introduce a micro-architecture-based table manager in charge of row-by-row bounds migration.

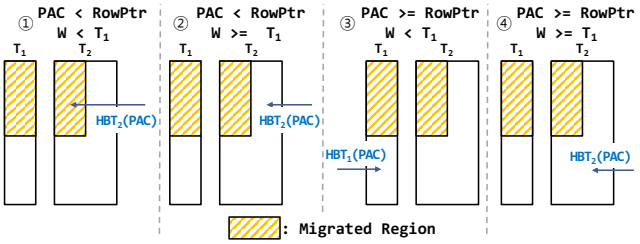



Fig. 10. Bounds-table access during HBT resizing.

TABLE I
HARDWARE OVERHEAD

| Metric | MCQ | BWB | L1-B Cache | L1-D Cache (for reference) |
|---|---|---|---|---|
| Size | 1.3KB | 384B | 32KB | 64KB |
| Area (mm2) | 0.0096 | 0.00285 | 0.1573 | 0.2628 |
| Access time (ns) | 0.1383 | 0.12755 | 0.2984 | 0.3217 |
| Dynamic access energy (pJ) | 0.0014 | 0.00077 | 0.0347 | 0.0436 |
| Leakage power (mW) | 3.2269 | 1.10712 | 58.295 | 122.69 |

Fig. 10 depicts the bounds-table access during the HBT resizing. $HBT_1$ and $HBT_2$ indicate the old and new tables, each of which has $T_1$ and $T_2$ ($=T_1*2$) ways, respectively. $W$ indicates the way of a row to access. During resizing, the manager maintains two base addresses of old and new tables. It also maintains a row pointer (RowPtr) pointing to a row in the old table, splitting the table into two regions: the migrated region and the live region. If accesses fall into out-of-way in the old table ($W \geq T_1$) or the migrated region (PAC $<$ RowPtr), the new table will be accessed using the base address of the new table. Otherwise, it accesses the old table. After resizing is finished, the base address of the old HBT is cleared, and the old HBT is deallocated.

### G. Hardware Overhead

Table I summarizes the hardware overhead of AOS. To estimate the area, access time, dynamic access energy, and leakage power, we use CACTI-6.0 [32] at 45nm technology. Overall, the AOS structures incur modest overhead.

## VI. Discussion on PAC Collisions

In AOS, PAC collisions affect both performance and security. With a high PAC collision rate, the average number of bounds-table accesses taken to find valid bounds could increase. It may also introduce false positives by data pointers with the same PAC. In this section, we study PAC collisions using QARMA [20]. We base our approach on two assumptions: 1) the block cipher algorithm used to calculate PACs does not suffer from severe hash collisions and 2) most applications rarely maintain a large number of *active*, i.e., allocated but not yet deallocated, memory chunks.

To study the first assumption, we run a microbenchmark that continuously calls malloc() 1 million times and

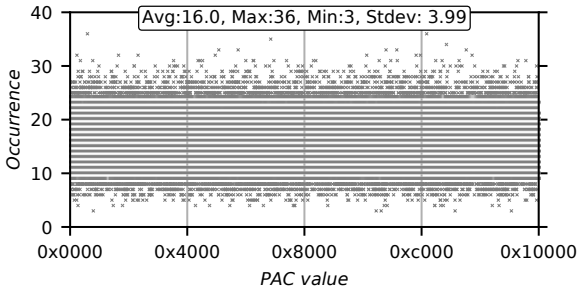



Fig. 11. PAC distributions by QARMA.

TABLE II
MEMORY USAGE PROFILES FOR SPEC 2006 WORKLOADS

| Name | Max Active # | Allocation | Deallocation |
|---|---|---|---|
| bzip2 | 10 | 29 | 25 |
| gcc | 81825 | 1846825 | 1829255 |
| mcf | 6 | 8 | 8 |
| milc | 61 | 6523 | 6474 |
| namd | 1316 | 1328 | 1326 |
| gobmk | 1021 | 137369 | 137358 |
| soplex | 140 | 98955 | 34025 |
| povray | 11667 | 2461247 | 2461107 |
| hmmer | 1450 | 1474128 | 1474128 |
| sjeng | 6 | 6 | 2 |
| libquantum | 5 | 180 | 180 |
| h264ref | 13857 | 38275 | 38273 |
| lbm | 5 | 7 | 7 |
| omnetpp | 1993737 | 21244416 | 21244416 |
| astar | 190984 | 1116621 | 1116621 |
| sphinx3 | 200686 | 14224690 | 14024020 |

generates 16-bit PAC values. To compute a PAC, we use a 64-bit context value (0x477d469dec0b8762), a 128-bit key (0x84be85ce9804e94bec2802d4e0a488e9), and the pointer address returned by `malloc()`. Fig. 11 shows that the PAC values are well distributed, demonstrating QARMA's usability as a hash function.

For the second assumption, we analyze memory management characteristics. Table II presents the memory usage profiles from full program execution of SPEC 2006 workloads [35] with reference input. To gather the information, we use Valgrind [6] with the `--trace-malloc` option enabled. Interestingly, most of the workloads are not malloc-intensive, i.e., memory allocations are not frequent. For example, `mcf`, `sjeng`, and `lbm` have fewer than 10 `malloc` calls. More importantly, for all the applications, we observe that the average number of active memory chunks is significantly smaller than the number of total allocation calls. `povray` and `hmmer` have more than 1 million allocation calls but have only a few thousand active chunks at most. This characteristic matches our expectation because typical applications allocate new memory chunks on demand and free them when they are no longer needed. The analysis of real-world benchmarks shown in Table III indicates a similar tendency. The number of allocation and deallocation calls is proportional to the size of the input (in `pbzip2`, `pigz`, `axel`) or the number of requests (in `apache`, `mysql`), but all of them have a modest number of maximum active memory chunks.

TABLE III
MEMORY USAGE PROFILES FOR REAL-WORLD BENCHMARKS

| Name | Description | Max # | Alloc. | Dealloc. |
|---|---|---|---|---|
| pbzip2 | Compress 1.4GB file, 8 threads | 110 | 12425 | 12423 |
| pigz | Compress 1.4GB file, 8 threads | 110 | 24511 | 24511 |
| axel | Download 1.4GB file, 8 threads | 172 | 473 | 473 |
| md5sum | Calculate MD5 hash, 1.4GB file | 32 | 34 | 34 |
| apache | Apache bench [33], 10K req. | 7592 | 13.36M | 13.36M |
| mysql | Sysbench [34], 100K req. | 5380 | 28622 | 28621 |

```
1  // Heap allocation, T: typename, N: # elements
2  T *ptr = (T *)malloc(sizeof(T)*N);
3  pacma ptr, sp, size; //size=sizeof(T)*N
4  bndstr ptr, size
5  // Heap OOB access
6  T varA = ptr[N+1];// Bounds-checking failure
7  ptr[N+1] = 0;     // Bounds-checking failure
8  // Valid free()
9  bndclr ptr;
10 xpacm ptr;
11 free(ptr);
12 pacma ptr, sp, xzr;
13 // Dangling pointer or UAF
14 T varB = ptr[0];  // Cannot find valid bounds
15 // Double free
16 bndclr ptr;       // Cannot find bounds to clear
17 xpacm ptr;
18 free(ptr);
19 pacma ptr, sp, xzr;
```

Fig. 12. Memory safety violations detected by AOS.

## VII. SECURITY ANALYSIS

### A. How AOS Detects Memory Safety Errors

In Fig. 12, we provide an example of memory safety errors and describe how AOS can detect them. First, AOS signs the pointer returned by `malloc()` and stores its bounds in the HBT (lines 2-4). Any subsequent memory access by the pointer triggers bounds checking. Hence, spatial safety violations (heap OOB accesses in lines 6-7) are detected by bounds checking. Since a valid `free()` (lines 9-12) deallocates the memory chunk and clears the associated bounds, any ensuing use of the freed pointer will fail in bounds checking because of the absence of the valid bounds. This prevents temporal safety errors such as the use of a dangling pointer or use-after-free (line 14) and double free (lines 16-19).

AOS can also prevent other various heap exploitation attacks. In the example shown in Fig. 1, `free()` with the crafted address (line 18) triggers the error. AOS prevents this attack by ensuring that only valid, signed data pointers can be freed. This is because any `free()` with a crafted or unsigned address will be prevented by `bndclr` inserted before `free()`.

### B. Pointer Integrity

In many attack scenarios, corrupting pointers becomes a preferred attack vector. For instance, control-flow attacks, such as ROP and JOP, corrupt the return address of a function to hijack the control flow of a program. In addition, Chen et al. [36] demonstrated that non-control-data attacks can even forge user credentials or change security-critical configurations without compromising code pointers. This kind of data-

```
ldr ptr, [SP, #0]    ; load data pointer
autm ptr             ; authenticate
```

Fig. 13. On-load authentication for AOS data pointers.

oriented attack exploits data pointers to corrupt variables that influence program behavior.

By inheriting Arm PA's primitives, AOS can be extended to ensure pointer integrity, enabling effective protection against pointer corruption attacks. For instance, Liljestrand et al. [21] proposed a PA-based code- and data-pointer integrity, which signs and authenticates PACs to prevent illicit pointer modification. While their return address and code-pointer signing schemes can be easily deployed to AOS, we especially care about how to cooperate with data-pointer signing. For data-pointer integrity, the authors implement *on-load* authentication, in which data pointers are signed immediately before they are stored in memory and authenticated immediately after being loaded from memory.

Since the AOS data pointers would already have been signed, AOS does not re-sign them when they are stored in memory. Also, AOS does not authenticate them as in [21]. Because their PACs were calculated using the base address of each memory chunk, the authentication will fail if the current address differs from the base address. Instead, AOS authenticates them using `autm` (§IV-A) by checking whether the AHC value of a pointer is nonzero, as shown in Fig. 13. To further enhance the security capabilities, we integrate AOS with this pointer integrity mechanism and evaluate performance overhead in Section IX.

### C. PAC/AHC Forging

Since we store PAC and AHC in the upper bits of a pointer, an attacker can try to manipulate the values via stack corruption or integer overflow. Deploying the `autm` authentication (§VII-B) enhances protection against AHC forging attacks that try to bypass the bounds-checking process. Regarding PAC forging attacks, our authentication method cannot detect arbitrary modification on a PAC. However, the attack surface exposed by such attacks would be limited for the following reasons. First, even if one can change the PAC of a pointer, knowing the PAC of the target memory region is difficult. Second, as long as we enforce bounds checking for memory accesses, all data pointers signed by AOS cannot be used to exploit common attack vectors outside the heap region, such as procedure linkage table (PLT), global offset table (GOT), virtual table (VTable), heap metadata, and kernel data.

### D. Heap Protection

Heap vulnerabilities are typically exploited by data-pointer manipulations. For example, an attacker can inject a malicious offset to a data pointer and make it point to a target memory location. Then, subsequent read/write operations using the tainted pointer lead to illegal memory corruption errors or leakage of sensitive information. Furthermore, heap safety issues correlate with the continuous development of new libraries. For instance, in *glibc* 2.26, the new thread local-caching mechanism, `tcache`, exposed a new heap exploit, double free. AOS provides an effective defense against those attacks by ensuring inter-object isolation, heap metadata protection, and temporal safety.

TABLE IV
SIMULATION PARAMETERS FOR PERFORMANCE EVALUATION

| Parameter | Value |
|---|---|
| Core | 2GHz, 8-wide, out-of-order, L-TAGE [38], 32-entry load and store queues, 192 ROB entries, 48 MCQ entries |
| Private L1-I cache | 32KB, 4-way, 1-cycle, 64B line |
| Private L1-D cache | 64KB, 8-way, 1-cycle, 64B line |
| Private L1-B cache | 32KB, 4-way, 1-cycle, 8B bounds |
| Shared L2 cache | 8MB, 16-way, 8-cycle local, 16-cycle remote, 64B line |
| DRAM | 50ns access latency from L2, 12.8 GB/s |
| Arm PA | 16-bit PAC size, signing/authentication: 4-cycle, stripping: 1-cycle |
| HBT | Initial 1 way, 4MB size |
| BWB | 64 entries, 1-cycle, eviction policy: LRU |

### E. False Positives

As described in Section VI, PAC collisions may introduce false positives across data pointers with the same PAC. However, exploiting PAC collisions is difficult given the PAC entropy. For instance, with a 16-bit PAC under typical AArch64 Linux systems, an attacker would require 45425 attempts to achieve a 50% likelihood for a correct guess [21]. Since the OS exception handler can terminate a process upon a bounds-checking failure or a PAC authentication failure, a brute-force attack would be infeasible.

### F. Bounds Narrowing

Several bounds-checking mechanisms narrow bounds to detect intra-object overflows (i.e., overflowing one field into another field in the same struct), and they reported such type of errors in SPEC CPU 2006 workloads [7], [12]. For example, `gcc` and `soplex` use custom memory management that causes intentional intra-object overruns. In `h264ref` and `perlbench`, benign intra-object buffer overruns are reported. The current AOS implementation does not support the bounds narrowing. We leave this for future work.

## VIII. METHODOLOGY

We evaluate SPEC CPU 2006 workloads [35] on Arm AArch64 using the gem5 simulator [37]. We use reference input sets and run the first 3 billion instructions. We do not count instrumented instructions (§IV-C) to run the same number of instructions. Table IV shows the simulation parameters. Note that we choose an initial 1-way HBT from an empirical study. A compiler-based profiling method can be used to find optimal associativity, but we leave this for future work.

We evaluate the following four system configurations:

- *Baseline:* the baseline without security features.
- *Watchdog*: prior work that features user-after-free and bounds checking [11].
- *PA*: a PA-based solution for code- and data-pointer integrity. We use the implementation of Liljestrand et al. [21], [39].
- *AOS*: the AOS bounds-checking mechanism.
- *PA+AOS*: *AOS* integrated with *PA* (§VII-B).

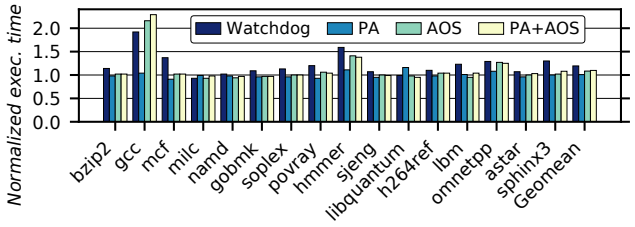



Fig. 14. Normalized execution time.

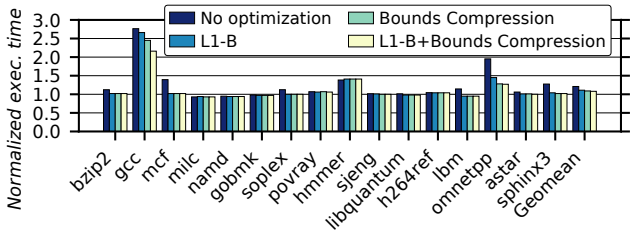



Fig. 15. L1-B cache and bounds-compression results.

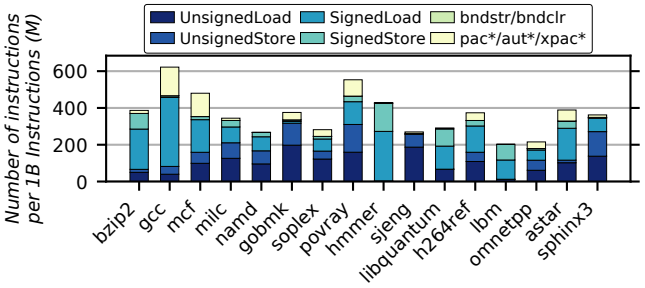



Fig. 16. Statistics of instructions of interest.

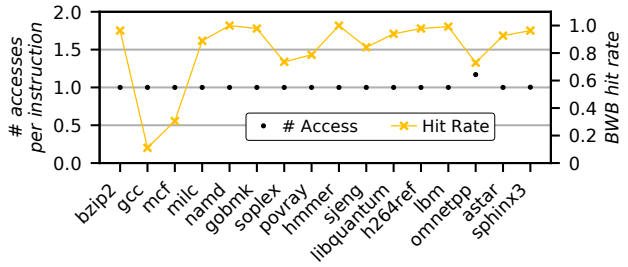



Fig. 17. Analysis on bounds-table accesses.

## IX. Performance Evaluation

### A. Execution Time

Fig. 14 shows the performance evaluation of various mechanisms. First, *Watchdog* incurs a 19.4% performance overhead on average. Even with its in-core memory checking, many applications show non-trivial performance overhead (>10%). We identify that the overhead is due to the extra instructions for memory checking and metadata propagation. Also, *Watchdog* has larger metadata of 24 bytes, compared to 8 bytes in AOS, thus causing more cache pollution.

*PA* incurs negligible performance overhead, as reported in [21]. Most applications show less than 1% overhead, except for `hmmer` and `omnetpp` (∼10%). We discover that their overhead is caused by frequent function calls that lead to a large number of stack frame creations. Since the signing and authentication instructions are inserted for every stack frame, the overhead is proportional to the number of function calls.

*AOS* shows an 8.4% performance overhead, and supporting pointer integrity (*PA+AOS*) imposes a 1.5% additional overhead. *PA+AOS* incurs negligible overhead for most applications, showing its feasibility as an always-on memory safety solution. Applications such as `milc`, `namd`, `gobmk`, and `astar` show slightly better performance than the baseline. We identify that the back-pressure on the issue pipeline stage due to the MCQ being full prevented aggressive branch predictions, resulting in fewer branch mispredictions and thus better performance. To find the root cause of the performance overhead of other applications, we consider the following three factors: 1) cache pollution, 2) delayed retirement, and 3) heap memory de-/allocation.

**Cache pollution.** Cache pollution caused by bounds metadata can degrade performance, in particular for memory-intensive applications. For instance, `gcc` has a large memory footprint and exhibits the worst slowdown, 2.16x, because of the increased cache misses. To see the impact of cache pollution, we study the performance of AOS with and without the L1-B cache (§V-F1) and bounds compression (§V-F3). As shown in Fig. 15, both optimizations are effective, but bounds compression brings a higher performance gain since it reduces the L2 cache pollution as well. Compared to when no optimization is used, the L1-B cache reduces the overhead by 10%, and bounds compression further reduces it by 3% on average. In particular, `gcc` and `omnetpp` show 60% and 68% lower overhead, respectively, when using both optimizations.

**Delayed retirement.** In AOS, the validation process may delay instruction retirement. Hence, the more memory accesses requiring bounds checking we have, the more likely the degradation caused by delayed retirement increases. Fig. 16 shows the statistics of instructions of interest. In `bzip2`, `gcc`, `hmmer`, and `lbm`, memory accesses by signed data pointers account for more than 80% of the total accesses. In particular, `hmmer` requires bounds checking for over 99% of all memory accesses, resulting in a 41% overhead by *AOS*. In contrast, although `lbm` requires bounds checking for most of its memory accesses, it does not suffer from a high overhead because it is not memory-intensive.

**Frequent heap memory de-/allocations.** This negatively affects performance for two reasons: 1) Overhead from extra instructions and 2) the high PAC collision rate, which could increase the number of bounds-table accesses. For instance, `gcc` and `omnetpp` invoke more than 20 million `malloc` calls during program execution, as shown in Table II. In this case, the slowdown imposed by extra instructions may not be trivial. Fig. 17 shows the average number of bounds-table accesses per instruction and the BWB hit rate. `omnetpp` shows the highest average accesses, 1.17, and others show close to one access

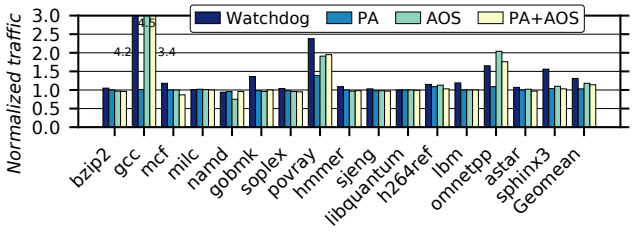



Fig. 18. Normalized network traffic.

thanks to the high BWB hit rate. Most applications show a BWB hit rate higher than 80%. This demonstrates that the performance overhead caused by the way iteration in AOS is not significant.

*1) HBT Resizing:* As explained in Section V-F3, resizing an HBT can be costly. During our simulation, no resizing occurred except for `sphinx3` (1) and `omnetpp` (2) since the initial 1-way HBT could cover up to 512K bounds. Also, we found that the performance impact from resizing was amortized at runtime because of non-blocking accesses to the old table.

### B. Network Traffic

To evaluate memory overhead from bounds-table accesses, we measure the number of bytes transferred between caches and between the last-level cache and DRAM. Fig. 18 shows 31% and 18% additional network traffic on average by *Watchdog* and *PA+AOS*, respectively. Although *Watchdog* keeps bounds in extended registers in a core, it requires more traffic because of its larger metadata size than that of AOS. While most applications have modest traffic overhead (<10%) in AOS, we observe relatively high overhead in `gcc`, `povray`, and `omnetpp` because of frequent bounds-table accesses.

## X. Related Work

**Trip-wire.** This class places blacklisted regions around all the objects to be protected. Any access to these regions is prohibited. REST [8] stores randomized secret tokens around sensitive data and detects any access to them using a value-matching unit in the cache hierarchy. Califorms [9] achieves memory safety at the granularity of individual objects. By leveraging unused padding bytes, they mitigate the performance overhead for fine-grained memory safety. Although such trip-wire-based approaches are low cost, they cannot prevent accesses from jumping over the blacklisted regions.

**Memory tagging.** These mechanisms tag memory regions to enable the tracking of illegal memory operations. For instance, HDFI [40] tags 1 bit with sensitive data and enforces data flow isolation at the word granularity. SPARC Application Data Integrity (ADI) [41] and Arm Memory Tagging Extension (MTE) [14] place a 4-bit tag into the upper bits of a pointer and associate the tag with the corresponding memory region. These approaches impose moderate performance overhead, but their implementations require significant hardware changes to the entire memory hierarchy and interfaces, such as tag cache, tag checking logic, and additional cache metadata extensions. Also, the limited size of tags reduces security guarantees. Given the probability of bug detection, specifically 94% with 4-bit tags, an attacker may bypass the protection with a sufficient number of attempts. Increasing the tag size is not trivial because of the memory overhead and hardware cost. Compared to prior work, AOS utilizes more bits and has a high PAC entropy, which makes the attacks less feasible.

**Bounds checking.** This class associates bounds metadata with pointers and enforces every memory operation to access memory regions within the bounds [10]–[13]. BOGO [13] extended Intel MPX to guarantee temporal safety on top of spatial safety supported by MPX. Despite its lightweight scheme, it has a high performance overhead (∼60%) caused by its underlying mechanism, Intel MPX.

**Capability model.** Capability models extend existing ISAs and provide security guarantees at the entire system level. In particular, CHERI architectures [22], [23], [42], [43] use a fat pointer to fit metadata, such as bounds information and permission bits. Every memory access is checked with its embedded metadata. However, the implementation requires changes to the entire system, including the pipeline stages, compiler, language runtime, and the OS. Although explorations of system designs remain open to the research community, the performance overhead and design complexity are high.

## XI. Conclusion

This paper proposed AOS, a defense mechanism against memory safety violations, to ensure heap spatial and temporal safeties. By utilizing the Arm PA's primitives, AOS signed data pointers to protect and used PACs embedded in the pointers to index a hashed bounds table. AOS also introduced a micro-architectural unit to eliminate extra bounds-checking instructions. Furthermore, this paper discussed how AOS cooperates with pointer integrity solutions to enhance security capabilities further. Our evaluations showed that AOS overcame the major challenges of existing bounds-checking mechanisms and achieved marginal performance overhead while providing strong security guarantees. We believe that AOS can serve as an effective runtime safety solution.

## Acknowledgment

We would like to thank Sharjeel Khan, Taesoo Kim, Moinuddin Qureshi, Euna Kim, Hyojong Kim, other HPArch group members, Dam Sunwoo, Krishnendra Nathella, Derek Miller, Chris Reed, Mathias Brossard, Prakash Ramrakhyani, and the anonymous reviewers for their feedback to improve the paper.

## References

[1] Google, "Google queue hardening," https://security.googleblog.com/2019/05/queue-hardening-enhancements.html, 2017.

[2] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf, 2019.

[3] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (ATC)*," USENIX, 2012, pp. 309–318.

[4] N. Hasabnis, A. Misra, and R. Sekar, "Light-weight bounds checking," in *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO)*," Association for Computing Machinery, 2012, pp. 135–144.

[5] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Proceedings of the Winter 1992 USENIX Conference*, 1992, pp. 125–138.

[6] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*," Association for Computing Machinery, 2007, pp. 89–100.

[7] G. J. Duck and R. H. C. Yap, "Effectivesan: Type and memory error detection using dynamically typed c/c++," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*," Association for Computing Machinery, 2018, pp. 181–195.

[8] K. Sinha and S. Sethumadhavan, "Practical memory safety with REST," in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 600–611.

[9] H. Sasaki, M. A. Arroyo, M. T. I. Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan, "Practical byte-granular memory blacklisting using califorms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 558–571.

[10] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the c programming language," in *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008, pp. 103–114.

[11] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *Proceedings of the 39st Annual International Symposium on Computer Architecture (ISCA)*, 2012, pp. 189–200.

[12] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: A cross-layer analysis of the Intel MPX system stack," *Proceedings of the ACM Measurement and Analysis of Computing Systems (POMACS)*, vol. 2, no. 2, Jun. 2018.

[13] T. Zhang, D. Lee, and C. Jung, "BOGO: Buy spatial memory safety, get temporal memory safety (almost) free," in *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*," Association for Computing Machinery, 2019, pp. 631–644.

[14] Arm, "Arm®architecture reference manual Armv8, for Armv8-A architecture profile," https://developer.arm.com/docs/ddi0487/fb/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile, 2020.

[15] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2007, pp. 552–561.

[16] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010, pp. 559–572.

[17] K. Lu, C. Song, T. Kim, and W. Lee, "Unisan: Proactive kernel memory initialization to eliminate data leakages," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*," Association for Computing Machinery, 2016, pp. 920–932.

[18] A. Milburn, H. Bos, and C. Giuffrida, "SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities," in *The Network and Distributed System Security Symposium (NDSS)*, Feb. 2017.

[19] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "Typesan: Practical type confusion detection," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*," Association for Computing Machinery, 2016, pp. 517–528.

[20] R. Avanzi, "The QARMA block cipher family. Almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes," *IACR Transactions on Symmetric Cryptology (ToSC)*, vol. 2017, no. 1, pp. 4–44, Mar. 2017.

[21] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "PAC it up: Towards pointer integrity using ARM pointer authentication," in *Proceedings of the 27th USENIX Security Symposium (Security)*," USENIX Association, 2019, pp. 177–194.

[22] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA)*," IEEE Press, 2014, pp. 457–468.

[23] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. M. Norton, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, A. T. Markettos, M. Roe, P. G. Neumann, R. N. M. Watson, and S. W. Moore, "CHERI concentrate: Practical compressed capabilities," *IEEE Transactions on Computers (TC)*, vol. 68, no. 10, pp. 1455–1469, 2019.

[24] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA)*," IEEE Press, 2014, pp. 361–372.

[25] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Proceedings of the 2006 Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*," Springer-Verlag, 2006, pp. 1–20.

[26] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX Security Symposium (Security)*," USENIX Association, 2014, pp. 719–732.

[27] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, 2019, pp. 1–19.

[28] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proceedings of the 27th USENIX Security Symposium (Security)*, 2018, pp. 973–990.

[29] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO): Feedback-Directed and Runtime Optimization*," IEEE Computer Society, 2004, pp. 75–86.

[30] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

[31] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism," in *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*," Association for Computing Machinery, 2020, pp. 1093–1108.

[32] N. P. J. Naveen Muralimanohar, Rajeev Balasubramonian, "CACTI 6.0: A tool to model large caches," https://www.hpl.hp.com/techreports/2009/HPL-2009-85.pdf.

[33] "ab- apache http server benchmarking tool." https://httpd.apache.org/docs/2.4/programs/ab.html.

[34] "Scriptable database and system performance benchmark." https://github.com/akopytov/sysbench.

[35] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[36] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of the 14th USENIX Security Symposium (Security)*," USENIX Association, 2005, pp. 177–191.

[37] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[38] A. Seznec, "The L-TAGE branch predictor," *Journal of Instruction-Level Parallelism (JILP)*, vol. 9, pp. 1–13, 2007.

[39] "Parts-llvm," https://github.com/pointer-authentication.

[40] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-assisted data-flow isolation," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016, pp. 1–17.

[41] Oracle, "Hardware-assisted checking using Silicon Secured Memory (SSM)," 2015. [Online]. Available: https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html

[42] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. M. Watson, and T. M. Jones, "CHERIvoke: Characterising pointer revocation using CHERI capabilities for temporal memory safety," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*," Association for Computing Machinery, 2019, pp. 545–557.

[43] R. Sharifi and A. Venkat, "CHEx86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 762–775.