PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication

Yuan Li
Tsinghua University
li-y18@mails.tsinghua.edu.cn

Wende Tan
Tsinghua University
twd2.me@gmail.com

Zhizheng Lv
Tsinghua University
lvzz20@mails.tsinghua.edu.cn

Songtao Yang
Tsinghua University
yst18@mails.tsinghua.edu.cn

Mathias Payer
EPFL
mathias.payer@nebelwelt.net

Ying Liu*
Tsinghua University,
Zhongguancun Lab
liuying@cernet.edu.cn

Chao Zhang*
Tsinghua University, BNRist,
Zhongguancun Lab
chaoz@tsinghua.edu.cn

Abstract

Memory safety is a key security property that stops memory corruption vulnerabilities. Different types of memory safety enforcement solutions have been proposed and adopted by sanitizers or mitigations to catch and stop such bugs, at the development or deployment phase. However, existing solutions either provide partial memory safety or have overwhelmingly high performance overheads.

In this paper, we present a novel sanitizer PACMem to efficiently catch spatial and temporal memory safety bugs. PACMem removes the majority of the overheads by sealing metadata in pointers through the COTS hardware feature – ARM PA (Pointer Authentication) and saving the overhead of pointer metadata tracking. We have developed a prototype of PACMem and systematically evaluated its security and performance on the `Magma`, `Juliet`, `Nginx`, and `SPEC CPU2017` test suites. In our evaluation, PACMem shows no false positives together with negligible false negatives, while introducing stronger bug detection capabilities and lower performance overheads than state-of-the-art sanitizers, including HWASan, ASan, SoftBound+CETS, Memcheck, LowFat, and PTAuth. Compared to the widely deployed ASan, PACMem has no false positives and much fewer false negatives, and reduces the runtime overheads by 15.80% and the memory overheads by 71.58%.

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

sanitizer; spatial memory safety; temporal memory safety

ACM Reference Format:
Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. 2022. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560598

1 Introduction

Memory safety violations are still the most common root cause of modern exploits [2, 32]. In practice, there are two types of memory safety violations [45]: (1) spatial safety violations where a program performs an out-of-bound memory access, e.g., buffer overflow and buffer under-read, and (2) temporal safety violations where a program accesses memory of invalid state (i.e., unallocated or freed), e.g., use-after-free and double free. Such memory safety violations break the integrity of memory and program states, which can further lead to denial of service, sensitive data leakage and corruption, privilege escalation, or even control-flow hijacking.

Many sanitizers have been proposed to catch memory safety violation bugs during development or testing. For instance, fuzzers that aim at discovering bugs in kernels, drivers, blockchains/smart contracts, or network services often come with sanitizers. AddressSanitizer (ASan) [42] is the most widely deployed sanitizer used in fuzz testing, e.g., by AFL [52]. It detects buffer overflow and use-after-free bugs, by quarantining freed objects and padding active objects with non-accessible redzones and tracking/verifying the accessibility of each memory byte at runtime. However, these sanitizers in general have many limitations.

On the one hand, most sanitizers only provide partial memory safety guarantees. A full memory safety solution should guarantee that, each memory object can only be accessed within its bounds (spatial safety) and must be in a valid state when accessed (temporal safety). But existing solutions fail to enforce this property. For instance, SoftBound [34] and LowFat [17, 18] only provide spatial memory safety. CETS [35] and the recently proposed ARM PA-based solution PTAuth [19] only provide temporal memory safety, while PTAuth exclusively protects heap objects (ignoring the stack

*Chao Zhang and Ying Liu are corresponding authors.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560598

and global variables). Even for the most widely deployed sanitizer – ASan, it also has many false negatives, i.e., it fails to detect certain memory safety violations as demonstrated by MEDS [23].

On the other hand, sanitizers that aim at providing full memory safety guarantees, in general, have overwhelmingly high performance overheads. ASan has 374.29% memory overhead on average. HWASan [43] utilizes hardware features to lower the memory overhead, but suffers from 108.39% runtime overhead. Memcheck [36] yields 1649.97% runtime overhead and 241.09% memory overhead on average. The overwhelmingly high performance overheads renders sanitizers inefficient, greatly slowing down the process of debugging, fuzzing, and other applications.

To enforce full memory safety, as discussed by previous studies [42, 45], we must track properties (metadata) of each object, including (1) its base address and size (spatial property) and (2) its birthmark (temporal property), and perform property checks when objects are accessed via pointers. The key to the success of such solutions is how to *efficiently* track properties (or metadata) and perform property checks *without lowering* security guarantees. In general, overheads of sanitizers come from four types of operations: creating metadata (at the time of object creation, i.e., *alloc*), propagating metadata (at the time of pointer operation, i.e., *ptr_x=ptr_y*), checking metadata (at the time of object access, i.e., **ptr_x*), and cleaning up metadata (at the time of object deallocation, i.e., *free*). The metadata propagation operation is the most time-consuming one (as shown in LowFat [18]), which could be optimized to improve the performance. For instance, ASan [42] only tracks metadata per object instead of per pointer, which shows better performance than its precedent solutions. But its overhead is still high, and it cannot provide full memory safety guarantees.

In this paper, we propose a novel sanitizer PACMem to catch spatial and temporal memory safety bugs at low overheads with no false positives and negligible false negatives. Since PACMem is a sanitizer that aims to catch memory safety bugs in programs during development or testing, it is acceptable to have (very) few false negatives. PACMem improves performance by eliminating metadata propagation in a clever way, i.e., encoding the pointers' metadata into pointers using the COTS hardware feature – ARM PA (Pointer Authentication [38]).

Specifically, we first create and place objects' metadata in a linear table, and utilize ARM PA to generate a PAC (Pointer Authentication Code) signature for each object, which represents the metadata as well. The PAC is then stored in the high-order bits of pointers associated with the object, i.e., the pointers' metadata are embedded in pointers. Such pointer metadata will be implicitly propagated no matter how the pointer is used (e.g., pointer assignment, pointer arithmetic operation, and function argument passing), thus saving the overhead of metadata propagation. Furthermore, the PAC signature in each pointer serves as an index to the metadata table, which enables efficient metadata lookup and greatly reduces the overhead of metadata checking, together with the signature verification support provided by ARM PA.

Compared to pure-software implementations, PACMem utilizes the functionalities of hardware PA instructions to efficiently generate, track, and verify such well-distributed metadata in pointers, and thus effectively reduces the performance overheads.

On the other hand, PACMem provides a strong memory safety guarantee. It has no false positives, since it precisely tracks the full memory safety properties and performs precise spatial and temporal safety checks at each memory access. In addition, PACMem has negligible false negatives, much lower than existing solutions. The only source of false negatives comes from sub-object overflow, since no metadata is tracked for fields within objects due to performance reason (as most sanitizers did). Specifically, it provides stronger bug detection capabilities than vanilla solutions built upon ARM PA and its successor ARM MTE (memory tagging extension).

We have implemented a prototype of PACMem on the ARM64 architecture in a real device. Then, we systematically evaluated its security on the `Magma` [24] and `Juliet` [1] benchmark, and evaluated its performance on `SPECspeed 2017` and `Nginx`. Evaluation results show that, PACMem provides stronger security guarantees than state-of-the-art sanitizers, including HWASan, ASan, SoftBound+CETS, Memcheck, LowFat, and PTAuth, while introducing lower performance overheads. Specifically, PACMem has 68.73% runtime overhead and 106.39% memory overhead on average. Compared with ASan, PACMem has much lower false negatives and exhibits 15.80% lower runtime overheads and 71.58% lower memory overheads. Furthermore, we evaluate the actual performance of ARM PA instructions on real devices.

In summary, we make the following contributions:

(1) We propose a novel sanitizer PACMem which utilizes the COTS hardware feature – ARM PA to provide stronger bug detection capabilities than vanilla ARM PA and ARM MTE.
(2) We propose to use ARM PA instructions to eliminate the overhead of metadata propagation, and to enable efficient runtime metadata lookup and safety checks, therefore greatly reduces the runtime performance overheads.
(3) We implement a prototype in a real hardware device, estimate the performance overheads of PA-related instructions on Apple M1 mini, and study the overheads of PACMem.
(4) We evaluate PACMem systematically in terms of security, runtime and memory overhead, and show that it outperforms state-of-the-art sanitizers.

2 Background

2.1 Memory Safety

Memory safety violations fall into two categories [46]: (1) *spatial violation*s happen when a pointer accesses out of its referent object's bound, e.g., buffer overflow, and (2) *temporal violation*s happen when a pointer accesses an invalid object (unallocated or freed), e.g., double free and use after free. In addition to memory safety bugs, programs may have several other types of bugs, including uninitialized variables, type cast bugs, misuse of functions with variable arguments (e.g., format string), or integer overflow, command injection etc., which are out of the scope of this paper.

Spatial Memory Safety. Spatial corruption refers to out-of-bounds accesses. Buffer overflow is the most common type of spatial corruption vulnerabilities. It has a long story of being studied by adversaries and exploited to launch attacks [40]. Bound checking is the most effective solution to detect (and prohibit) spatial corruptions. ASan [42] uses redzones around objects to stop out-of-bound violations by checking if redzones are accessed. HWASan [43] tags each

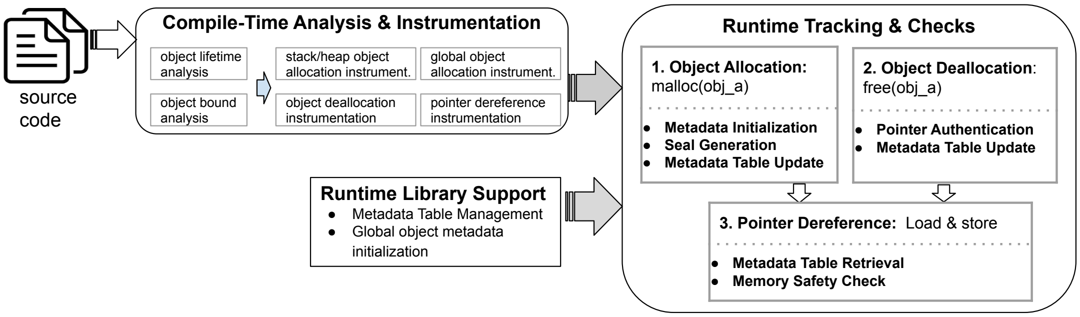
source code
Compile-Time Analysis & Instrumentation
object lifetime analysis
object bound analysis
stack/heap object allocation instrument.
global object allocation instrument.
object deallocation instrumentation
pointer dereference instrumentation
Runtime Library Support
Metadata Table Management
Global object metadata initialization
Runtime Tracking & Checks
1. Object Allocation: malloc(obj_a)
Metadata Initialization
Seal Generation
Metadata Table Update
2. Object Deallocation: free(obj_a)
Pointer Authentication
Metadata Table Update
3. Pointer Dereference: Load & store
Metadata Table Retrieval
Memory Safety Check

Figure 1: Overview of PACMem. It updates metadata of objects at object allocation and deallocation sites, seals the metadata into pointers, and checks metadata at each pointer dereference using the seal in pointers to retrieve object metadata.

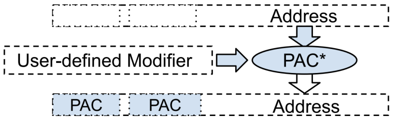

Figure 2: Illustration of the ARM PA mechanism. The PAC (pointer authentication code) is generated by PAC*-family instructions, using the target pointer, a user-defined modifier (i.e., the execution context), and a key (stored in kernel).

allocated memory block and its pointer, and utilizes the AArch64 hardware feature to store the pointer tag in the pointer. If the tag of the pointer mismatches the refered object's, an access violation is caught. LowFat [17, 18] also encodes the bound information into the pointer, with a special encoding scheme. So LowFat can utilize the pointer to retrieve the bound information before each pointer dereference, to check if that memory access is within the bound.

Temporal Memory Safety. Generally, temporal memory safety violations are caused by programs accessing unallocated or deallocated objects. When programs explicitly deallocate an object, the object becomes invalid, and all pointers to this object become *dangling pointers*. When a dangling pointer is freed or used again, a double-free or use-after-free (UAF) bug is yielded. Tag matching is a common method used by mitigations [9, 35, 37] to catch temporal safety violations. In general, they assign tags to objects and pointers, and compare them at pointer dereferences. Another common method is detecting dangling pointers before they are used. Specifically, such methods will monitor and track pointers passed to the `free` function. If a marked pointer is used later, a temporal memory violation is reported. However, this method cannot handle copies of dangling pointers. Thus, some tools not only mark pointers to be freed but also maintain an object-to-pointer map to invalidate copies of dangling pointers when `free` is called, such as Undangle [12], DangNull [30], FreeSentry [51], and DangSan [47].

2.2 ARM Pointer Authentication

Armv8.3-A introduces the PA (Pointer Authentication) [38] security extension, which has been applied in recent iOS devices [25]. This extension enforces pointer integrity by signing the pointer at definition points and verifying the signature at dereference points.

As shown in Figure 2, `PAC*` instructions sign a target pointer and compute its Pointer Authentication Code (PAC), which is stored in unused high-order bits of the target pointer. Further, `AUT*` instructions authenticate a signed pointer. If the authentication succeeds, the PAC embedded in the pointer is stripped, and the pointer can be dereferenced as normal. Otherwise, the pointer will be modified to an invalid pointer in Armv8.3-A or will trigger an exception in ARMv8.6-A. We assume PA has the latter behavior in this paper.

Users of ARM PA could choose modifiers as wish to tune signatures, so that a same pointer can yield different signatures in different execution contexts. The modifier also serves as a bound between the pointer definition point and verification point, since it has to be the same in order to pass the check.

3 Methodology

3.1 System Overview

PACMem is designed for efficiently catching memory safety bugs in target programs at runtime, with negligible false negatives. Following common practice, it also tracks metadata of objects and pointers, and performs metadata checks before memory accesses. To enforce full memory safety, PACMem tracks *all* necessary spatial and temporal metadata precisely, and checks *all* memory accesses.

In general, metadata is created when objects are allocated, is propagated to other pointers during pointer operations, is checked when pointers are used to access objects, and is cleaned up when objects are deallocated. Each of these four steps would cause runtime overheads, among which the metadata propagation and checking are the most time consuming. PACMem utilizes ARM PA to eliminate metadata propagation and accelerate metadata checking.

Figure 1 demonstrates the overview of PACMem. At compile time, PACMem analyzes the bound and liveness properties of objects (including global objects), and instruments at proper locations to track such metadata or perform memory safety checks. Specifically, PACMem generates metadata when an object is created, utilizes ARM PA to generate a PAC signature (denoted as a *seal*) of this metadata and embeds it into pointers associated with this object, and places the metadata in a table indexed by this seal. The seal will implicitly propagate along with pointers no matter how they are used, saving overheads of the metadata propagation. Furthermore, before a pointer is dereferenced, the seal serves as an index to efficiently lookup the metadata table, enabling efficient metadata checks with the support of signature verification provided by ARM PA. Lastly, when an object is deallocated, the integrity and validity

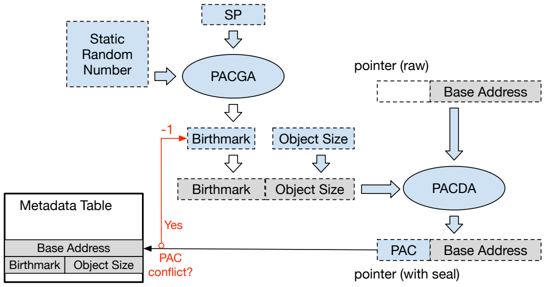

Figure 3: Metadata creation: when a memory object is allocated, a random birthmark is yielded to compose the object's metadata, which will be (1) placed in the metadata table and (2) further signed to generate the pointer's seal. The pointer's seal will be used as the index to the metadata table, thus reflects the pointer's metadata too.

of the base address of the object will be checked, and the metadata will be removed from the metadata table once the check succeeds.

3.2 Metadata Creation

To enforce full memory safety, we have to track all necessary memory safety metadata. In PACMem, each object's metadata consists of (1) its base address, (2) its object size, and (3) its birthmark, while the former two are used for spatial memory safety checks and the last one for temporal memory safety checks. Specifically, the metadata takes 128 bits, whereas the birthmark and object size take 32 bits respectively.

As shown in Figure 3, the metadata is stored in a metadata table, i.e., a linear array for performance consideration. To retrieve object metadata, it is common for sanitizers to directly use the associated pointer as the index, as ASan does. However, such solutions in general will yield very high memory overheads, since the table size is proportional to the pointers' value space. We propose to seal metadata into the high-order bits of pointers (via the hardware feature ARM PA), and use the seal as the index to retrieve objects' metadata, which greatly reduces the memory size of the metadata table and also enables efficient runtime metadata retrieval.

As shown in Figure 3, the seal is a PAC signature of the metadata, and thus has a special bond with the pointer's metadata, which can be verified at runtime. Specifically, for each newly allocated object, we assign it with a pseudo-random birthmark. However, as there are no lightweight randomness sources on commercial ARM devices to our knowledge , we again utilize ARM PA to generate a pseudo-random birthmark, by taking the current dynamic stack pointer as the pointer and taking a static random number generated at compile time as the modifier. Since the birthmark is pseudo-random, adversaries may bypass PACMem with a low probability (see Section 6), but it is unlikely to be bypassed during program testing which involves no active adversaries. Then, we utilize ARM PA to yield the seal for the metadata (i.e., base address and size, and the birthmark).[1] At runtime, metadata indexed by this seal should be consistent with it.

Ideally, two different objects should have different metadata and seals. Although QARMA [10] can make the PACode well-distributed, due to the limitation of digital signature, two different metadata could yield a same seal. As a result, two metadata will be stored at the same slot in the metadata table, and at least one runtime metadata check will fail and cause false positives when these two objects are both accessed. To ensure the uniqueness of each object's seal, the metadata initialization code instrumented by PACMem will repeatedly change the modifier to yield different seals until a non-conflicting one is found. In other words, we implement a re-hashing strategy upon collisions when new objects are created. Specifically, if the seal of a newly created object collides with an existing object's (i.e., the slot in the metadata table is taken), the modifier (i.e., birthmark) decreases by one and a new seal will be yielded. Note that, this collision mitigation overhead only exists when a new object with a conflicted seal is created. After creating the object, all seals are conflict-free and can be used (e.g., to retrieve metadata from the table) without extra overheads. Whenever the seal is used as the index to retrieve metadata, there are no conflicts and will not introduce extra overheads. Since the number of seals equals the number of live memory objects, the length of PA code should be large enough. On the other hand, the PA code shares the same space with the pointer, such that a larger PA code size will shrink the address space available for the program. In general, a 39-bit address space is sufficient for programs (see Section 5.2 for the microbenchmark evaluation). Thus, PACMem sets the PA code size to 24 (i.e., 63-39).

Compared to existing works, PACMem's metadata management scheme has several advantages. First, it tracks all temporal and spatial properties, and enables full memory safety enforcement. Second, it avoids the metadata propagation overheads. The indices of the metadata table (i.e., seals) are shipped together with pointers, and are not affected by pointer operations (e.g., pointer arithmetic operations). Thus, we do not need to perform extra metadata propagation like Watchdog [33] did. Third, the metadata retrieval is simple and lightweight, i.e., the seal can be used as an index to retrieve data from a linear array (i.e., the metadata table). In addition, PACMem has significantly lower memory overheads than other methods (e.g., ASan and HWASan) due to the compact design of metadata and seals.

3.3 Metadata Tracking and Checking

Once the metadata (or seal) is created for an object (or pointer), the seal will propagate along with the pointer automatically. When the pointer is dereferenced (i.e., memory access or object deallocation), its metadata will be checked.

3.3.1 Memory Access Checking

In this phase, each pointer dereference will be checked to enforce full memory safety. Figure 4 outlines what the instrumentation of PACMem does.

Whenever a memory address is loaded through a pointer, a full memory safety check must be enforced. As shown in Figure 4, PACMem adopts a bound check to enforce spatial memory safety. Specifically, it uses the pointer's seal as the index to retrieve base address and size of the intended object from the metadata table. With this base address, we can calculate the offset of the pointer being dereferenced, and verify that the access is in valid bounds.

[1] The seal and metadata initialization should be done when objects are allocated. For global objects without explicit allocation sites, we utilize custom initializer functions to generate seal and metadata for them at program startup.

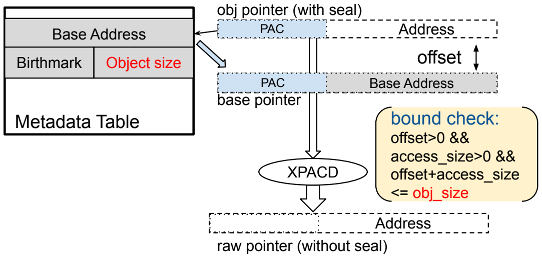

Figure 4: Metadata checking: when a pointer is dereferenced, PACMem retrieves metadata from the metadata table, calculates the pointer's offset to its base address, and performs a delicate bound checks to enforce full memory safety.

Note that this bound check also provides temporal safety guarantees. If the pointer being dereferenced is a dangling pointer, the aforementioned bound check will (likely) fail. Since the dangling pointer's intended object has been freed, its metadata entry in the metadata table is cleared. In the first case, the cleared entry has not been taken by other objects yet, then the retrieved base address and object size are all 0, and the bound check will fail. Furthermore, the probability of a hash collision for use-after-free is approximately 5.96×10^{-8}, i.e., violations are detected with over 99.9999999% probability. Even though the corresponding entry is taken by another object which accidentally has the same seal, the retrieved base address and object size are unlikely the same as the freed object's, and the bound check will fail too. As a result, this bound check is sufficient to catch temporal safety bugs, including use-after-free (UAF) bugs.

3.3.2 Object Deallocation Checking

When a heap or stack object is freed (explicitly deallocated or implicitly purged), we need to remove its outdated metadata from the metadata table, i.e., set the table entry to zeros.

If the object to be freed is a heap object, then the pointer used to deallocate an object should be the base address of a valid object, i.e., the pointer is not a dangling pointer and does not point inside the object. Therefore, we need to do an extra check before clearing the metadata table of the heap object.

Specifically, PACMem retrieves the birthmark and object size (i.e., the modifier) from the metadata, and uses it to authenticate the seal of the pointer using ARM PA instruction `AUTDA`. If the authentication succeeds, PACMem sends the raw pointer to the heap allocator to free.

The `AUTDA` can successfully authenticate the signed pointer and strip the seal of the pointer if and only if the pointer is the base address of the object signed by `PACDA` and the modifier is the correct one used to sign the object.

In this way, it stops the heap allocator from freeing illegal pointers including the dangling pointers and pointers that are not the base address of an object. Thus, PACMem can catch double-free and invalid-free bugs.

Figure 5 outlines what the instrumentation of PACMem does when deallocating objects.

3.4 Compatibility with Unprotected Modules

Like most sanitizers, e.g., ASan, PACMem only catches bugs in the modules which are hardened by PACMem. Since PACMem utilizes upper bits of pointers to store seals, it would cause compatibility issues if pointers are used across protected and unprotected modules. Therefore, PACMem takes extra steps to deal with such compatibility issues.

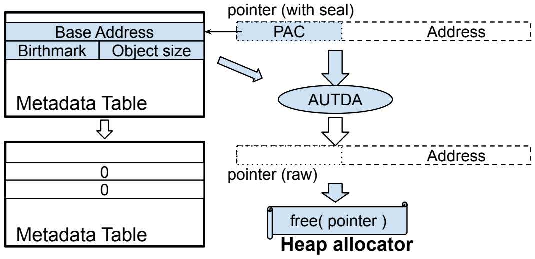

Figure 5: Object deallocation phase: clear the metadata table, and authenticate the pointer if it is a heap object.

First, when a signed pointer is passed to a unprotected module, PACMem will do a safety check for it (i.e., verify the pointer and remove PAC). This will eliminate compatibility issues, and by the way stops potential dangling pointers from being used in unprotected modules.

Second, when an non-signed pointer is yielded by unportected modules and returned to protected modules,[2] PACMem will generate a conservative metadata (i.e., minimum base address and maximum object size) for it, since it does not know how the object is created. This addresses the compatibility issue but leaves an attack surface for adversaries. However, this is inevitable for unprotected modules. To provide full protection, developers are strongly encouraged to protect all modules with PACMem.

4 System Implementation

4.1 Implementation Details

The PACMem system prototype includes (i) a custom compiler extension for analysis and instrumentation and (ii) a runtime support library for creating the metadata table and initializing metadata for global objects. The current prototype supports C and C++ programs. It supports all common memory allocation APIs such as `malloc`, `free`, `new`, and `delete`.

Our compiler extension is based on LLVM 14.0.0, which already supports ARM PA instructions. We first analyze the target program using a new LLVM pass, and insert the security checks mentioned in Section 3.3 at proper locations. It has to work with the `-fPIC` compiler option to make target programs position-independent, so that global objects are all referenced via the `GOT` (global offset table) and easy to recognize. Our runtime library mainly allocates a metadata table in shadow memory before the protected program starts. We use a constructor function which will run at program startup to initialize the metadata of all global variables and put seals of the pointers in the `GOT`. Then, we make the `GOT` read-only to prevent the `GOT` from being corrupted by attackers.

Each capability corresponds to a total of 16 bytes of metadata, including 8-bytes signed base address and 8-bytes modifier (including 4-bytes birthmark and 4-bytes object size).

[2] If there are recursive pointers in the objects, PACMem could follow the API definition to get all those pointers and sign them recursively.

4.2 Performance Optimizations

To lower the overheads of PACMem, we have made the following optimizations without lowering security guarantees.

Loop-Independent Memory Operations. If a memory operation in a loop accesses a same memory range no matter how many times the loop has iterated, PACMem just checks once for it before the loop entry. In that case, there is no need to repeat the same check in all loop iterations.

Loop Bound Pointers. If a pointer only increases or decreases linearly within a loop, and we can statically determine its upper bound and lower bound at compile time, then PACMem will skip checks for dereferences to the pointer within this loop and safely performs checks for the upper bound and lower bound before the loop entry.

Redundant Check Elimination. If one memory access instruction dominates or post-dominates another one, and their accessed address ranges are the same, the checks for the second instruction are considered redundant and could be removed. For those accesses that can be statically verified, the runtime security check will also be eliminated.

Write-only Check. By default, PACMem checks all pointer dereferences, no matter read or write access. In practice, a read access violation in general has much lower security implications, while a write access violation is the foundation of launching further exploits. Thus, we provide an optional working mode of PACMem that focuses on integrity (but not confidentiality) and only enforces memory safety checks for memory write accesses, similar to existing sanitizers.

5 Evaluation

This section evaluates the PACMem prototype in terms of security and performance, and answers the following questions:

- What types of vulnerabilities can PACMem detect?
- Are there any types of vulnerabilities that PACMem can detect but other sanitizers cannot?
- How much runtime performance and memory overheads does PACMem introduce?

5.1 Environment and Comparison Targets

Environment. There are very few environments supporting ARM PA yet. In addition to the official emulator ARM FVP [3], the new Apple M1 machines have support for ARM PA. But macOS does not provide ARM PA support for third-party applications yet. Therefore, we build and run a Linux kernel v5.14.0 on an Apple M1 Mac mini, and apply some custom patches to make it support more memory architectures. On the other hand, we follow ARM's documentation [7] to build the FVP environment for testing ARM PA, and use the default memory configuration of FVP and apply it to the Linux kernel too, i.e., 39-bit for pointers and 24-bits for ARM PAC code.

Besides, Apple claimed that M1 includes 4 high-performance cores and 4 high-efficiency cores. But we found that there are 3 types of cores in M1 mini by measuring the execution time of the ADD instruction. Specifically, on core 0, an ADD instruction requires 0.714 ns; on cores 1-3, an ADD instruction requires 0.485 ns; on cores 4-7, an ADD instruction requires 0.335 ns. All security tests and performance tests were run on the Apple M1 Mac mini. To be fair, we used the numactl to ensure that the target programs only run on cores 4-7.

Table 1: Clock cycles of ARM PA instructions on Apple M1 and AWS Graviton3

			Apple M1	Graviton3
ARM PA* Instructions	Signature	PACDA	2.344 ns	1.926 ns
		PACDZA	2.344 ns	1.926 ns
		PACGA	2.344 ns	1.926 ns
	Authentication	AUTDA	2.344 ns	1.926 ns
		AUTDZA	2.344 ns	1.926 ns
	Stripping	XPACD	2.344 ns	0.770 ns
ALU Operations		ADD	0.335 ns	0.385 ns
		AND	0.335 ns	0.385 ns
PAC* Instructions : ALU Operations			7:1	5:1
AUT* Instructions : ALU Operations			7:1	5:1
XPACD Instructions : ALU Operations			7:1	2:1

Comparison Targets. Moreover, we carefully choose several state-of-the-art (SOTA) open sourced solutions to conduct comparison experiments. Among SOTA solutions, we only found HWASan and ASan integrated in the clang compiler support the ARM architecture[3], while most other sanitizers, e.g., LowFat [18], Memcheck [36, 44], and SoftBound [34] and CETS [35], only support the x86 architecture. So, we evaluate the security and performance of PACMem, ASan, and HWASan on the ARM architecture, and evaluate that of ASan, LowFat, Memcheck, and SoftBound+CETS on the x86 architecture, then use the evaluation results of ASan as a bridge to compare PACMem with other x86-based sanitizers.

5.2 Microbenchmark Testing

Clock cycles of ARM PA. We have evaluated the performance of PA instructions on the Apple M1 Mac mini. *To the best of our knowledge, we are the first to evaluate the overheads of ARM PA-based solutions on real hardware.* The results are listed in Table 1. A PA instruction roughly takes 7x time as an ALU operation on real devices. A software implementation of similar algorithms would take much longer. SipHash [8], a MAC algorithm, takes about 123 cycles to process 8 bytes. The evaluation demonstrates the superior performance of the ARM PA.

To compare against servers, we also evaluated the performance of PA instructions on an AWS Graviton3 instance[4]. According to Table 1, the PA instructions run faster on AWS Graviton3 than Apple M1. All PA instructions take the same amount of time on Apple M1 mini, but the XPACD instruction is much faster than other PA instructions on AWS Graviton3.

Whether the PAC length is sufficient for use? Since the PAC is 24-bits in our setting, then the metadata table has at most $2^{24} = 16,777,216$ slots. If a program has more objects than this number, then the metadata table will overfill and PACMem will have false positives. Therefore, we have evaluated the maximum number of simultaneous live objects (including heap variables, global variables, and stack variables) in several long-running and large real world programs. It is worth noting that the reported object count is the peak instead of the cumulative number. As shown in Table 2, they have less than 100 thousand live objects, which is far

[3] Another solution PARTS [31] provides pointer integrity guarantee rather than memory safety guarantee. So it is not a proper target to compare with.

[4] Through an anonymous reviewer, we learned that AWS Graviton3 also supports ARM PA.

Table 2: Number of live objects in long-running programs. For Nginx, Apache, and Node.js, we used ApacheBench for stress testing, making 10,000 requests with 2,000 concurrent threads. For Redis, we used Redis-benchmark to make 1,000,000 requests with 3,000 concurrent threads.

	Programs	Maximum number of objects
	Nginx	5,635
	Apache	7,886
	Node.js	89,873
Redis	Redis-server	7,544
	Redis-benchmark	17,731

from the aforementioned maximum threshold. *Thus, we believe the PAC length is sufficient for use in practice.*

5.3 Security Evaluation

5.3.1 Test Suites of Vulnerabilities

`Magma` [24] provides a benchmark of ground truth security-critical ground truth bugs in commonly-used open-source libraries. We used the `Magma` test suite to evaluate the detection capability of PACMem and ASan on real-world programs.

To further systematically evaluate the security and functionality correctness of PACMem, we utilize test sets from the `Juliet` test suite consisting of many memory safety vulnerabilities to conduct experiments. Among those test sets, we have selected the test sets pertaining to spatial and temporal memory safety. These selected test sets are listed in Table 3.

Both the `Magma` and `Juliet` test suites have C benchmarks and C++ benchmarks. `Poppler` and `php` in Magma are C++ benchmarks, `sqlite3` and `libtiff` also contain C++ files, and the rest are all C benchmarks. The `Juliet` test suite has more C++ benchmarks. As shown in Table 5, there are 889 C++ benchmarks that Softbound+CETS fails to compile.

5.3.2 Selected Cases

As listed in Table 3, a small number of test cases are not selected to test, because they cannot trigger the intended vulnerabilities, in order to measure the false-negative rates accurately. We analyzed the test cases and found that some test cases can cause fake false negatives, which fall into the following categories:

First, some type-confusion test cases do not trigger buffer overflows, such as `Heap_Based_Buffer_Overflow__sizeof_double*`. Specifically, these test cases allocate memory to an object with the pointer size rather than the object size, while the object's size should be 64 bits (such as a `double` type). And then, it accesses the target object using the object size rather than the allocated size. On a 64-bit system like ARM64, the pointer size and the object size are both 64 bits, and there is no buffer overflow. Second, some test cases rely on a random number to trigger the vulnerability and may not trigger out-of-bounds access during testing.

Table 3: The test sets selected in the `Juliet` test suite.

Test Sets	Vulnerability Type	All Cases	Selected Cases
CWE121	Stack-based Buffer Overflow	6200	6104
CWE122	Heap-based Buffer Overflow	7740	7260
CWE124	Buffer Underwrite	2336	2240
CWE126	Buffer Overread	1740	1644
CWE127	Buffer Underread	2336	2240
CWE415	Double Free	1636	1636
CWE416	Use After Free	786	786
CWE476	NULL Pointer Dereference	612	576
CWE761	Invalid Free	576	576

Table 4: Security evaluation results on Magma. The second column lists the number of proof-of-concept (poc) samples provided by Magma (from the AFL++ directory), the last two columns show the number of PoCs that can be caught by PACMem and ASan respectively.

open-source libraries	PoCs	PACMem	ASan
libpng	634	0	0
LibTIFF	3716	115	115
Libxml2	19614	0	0
Poppler	7343	1329	1329
OpenSSL	655	194	194
SQLite	1777	0	0
PHP	1443	**1128**	**1083**
Lua	0	0	0
libsndfile	0	0	0

Moreover, there are 18 test cases in the `CWE476` class called `NULL_Pointer_Dereference__null_check_after_deref*`.

The only difference between the `BAD` and `GOOD` programs of these test cases is that each `BAD` program has an extra null-pointer check after the target pointer has been dereferenced. There are annotations of the source code of these test cases, which said: "This NULL check is unnecessary." We believe that these 18 test cases are used to evaluate the accuracy of static analysis tools and cannot trigger the vulnerability in practice. Therefore, we omit the results of these 18 `BAD` programs when calculating false-negative rates.

Since all these cases do not have out-of-bound access on the testing platform, we do not count them as false negatives for memory safety sanitizers.

5.3.3 Metrics and Configuration

To evaluate the security performance, we select five classic and commonly-used open-source sanitizers, i.e., HWASan, ASan, LowFat, Memcheck, SoftBound+CETS, and compare their performance with PACMem.

Metric. For each test case, the `Juliet` test suite generates two test programs, `GOOD` and `BAD`. Each `GOOD` program is not vulnerable, and each `BAD` program has an exploited memory corruption. Therefore, it is a false positive for a sanitizer to report a `GOOD` program as anomalous. Also, it is a false negative for a sanitizer not to report a `BAD` as anomalous.

Sanitizer configuration. To provide a fair comparison, we made every effort to reduce the false negatives and false positives of other sanitizers caused by implementations. For example, in some test cases of the `Juliet`, some memory objects are used without proper deallocation and therefore detected by the LeakSanitizer embedded in ASan, which causes ASan to report a false positive. The same situation exists in Memcheck. So we disable the memory leak detection mechanism of both ASan and Memcheck.

In addition, LowFat may put objects in aligned but larger memory blocks (e.g., a 10-bytes object takes 16 bytes memory). As a result, an illegal input that causes out-of-bound access (e.g., accessing 11 to 16 bytes) may get silently mitigated, and LowFat will not report an alarm. To avoid this type of unintended false negatives, we increase the input values of `BAD` programs in the `Juliet` test suite, enabling LowFat to report an alarm when such objects are overflowed.

5.3.4 Security Evaluation Results on `Magma`

We evaluated the security performance of both PACMem and ASan on the real-world vulnerability benchmark Magama, to check whether they can catch bugs in the benchmark. Results showed that, *PACMem has better detection capability than ASan on the real-world bugs.* Proof of concept (PoC) exploits are used to reveal the security weaknesses within software. Magma [24] collects some open-source libraries with real-world vulnerabilities and provides sufficient PoCs. We used PoCs provided by Magma to evaluate target programs instrumented by PACMem or ASan, respectively.

As we can see from Table 4, PACMem and Asan detect the same number of PoCs for most programs. For PHP, PACMem found 45 more PoCs than ASan, all of which were PoCs from CVE-2018-14883, a heap-based buffer over-read caused by an integer overflow. ASan fails to detect 45 PoCs because the illegal memory access crosses the RedZone instrumented by ASan and access another valid object. However, since PACMem can accurately obtain the base address and the object size of the memory object to which the pointer corresponds, PACMem can effectively detect these PoCs.

Specifically, these 45 PoCs had a total of three cases of out-of-bounds access, i.e., memory violation was acted by three different instructions. Two situations of the out-of-bounds access were in the `php_ifd_get32u` function, which used the vulnerable pointers for array accesses. The other situation is a call instruction of the `memcpy` function, which uses the vulnerable pointer to copy 6 bytes of the targeted object. Note that, Magma contains not only just memory corruption vulnerabilities, and the PoCs may not trigger all the vulnerabilities. So it is expected that some target programs do not detect an exception once. The evaluation focuses on the comparison with the common-used ASan.

5.3.5 Security Evaluation Results on `Juliet`

PACMem has no false positives. The results of the Juliet test suite are shown in Table 5. `FP` represents the False-Positive rate, and `FN` represents the False-Negative rate. As mentioned before, PACMem has no false positives because it uses rehashing to avoid the only source of positives – collision of two different active metadata (i.e., taking a same table slot).

PACMem provides better memory safety guarantees. While ASan has the lowest false-negative rates among existing open-source sterilizers, PACMem has even fewer false negatives than ASan, thus provides better memory safety guarantees. PACMem cannot detect sub-object overflow, which is a common problem of all these sanitizers, including ASan, HWASan, LowFat, and Memcheck, due to the trade-off with metadata tracking overhead. SoftBound claims to detect sub-object overflow, but it does not detect any of the sub-object overflow test cases in the `Juliet` test, as we will discuss the root cause later. Additionally, PACMem covers more types of vulnerabilities than HWASan and LowFat.

Summary. Overall, PACMem has better detection ability than ASan, HWASan, LowFat, Memcheck, and SoftBound+CETS. PACMem performs well in detecting spatial and temporal memory vulnerabilities. Specifically, PACMem performs the authentication for the pointer passed to the `free` function before deallocating a memory object, so PACMem can effectively detect double free and invalid free bugs. Since the memory safety check is required for each memory operation, PACMem can also detect out-of-bounds access, use-after-free, and null pointer dereference.

5.3.6 Root Cause Analysis of False Negatives

In general, as shown in Table 5, existing sanitizers have much more false negatives, i.e., they would miss real vulnerabilities at runtime. The reasons for the false negatives of these sanitizers are analyzed and discussed in detail in the following paragraphs, both in terms of implementation and design.

PACMem. The false negatives come from test cases with sub-object overflow. Since the metadata of PACMem just contains the size of the entire object and has no information about the sub-objects in the target object, PACMem cannot detect such violations.

HWASan. In terms of design, on the one hand, HWASan only matches tags on memory accesses and therefore cannot detect dangling pointers passed to third-party libraries; on the other hand, if a pointer passed to the `free` function does not point to the beginning of an object, HWASan cannot detect it because its tag does match the tag of the object. In other words, HWASan cannot detect invalid free bugs. Besides, HWASan also cannot detect sub-object overflow. Moreover, HWASan relies on 8-bits tags and, in practice, many tags conflicted. We ran `CWE415` three times and found that the conflicting tags caused false negatives for some cases. Therefore, false negatives for HWASan are the average of the three results. In terms of implementation, HWASan cannot handle stack variables allocated by the `ALLOCA` function, so no buffer overflows of such memory objects can be detected by HWASan. In addition, HWASan does not handle some common memory access functions such as the `snprintf` and `strncat` functions.

ASan. In terms of design, ASan cannot detect cross-object overflow that skips redzones, or access an object that has been freed for a while, as shown in MEDS [23]. But PACMem can effectively detect such cases. However, PACMem and ASan both cannot detect sub-object overflow. In terms of implementation, we found that the function `__asan_alloca_poison` used by ASan to set the redzones does not work in some cases, causing false negatives.

LowFat. LowFat, a sanitizer for detecting spatial memory vulnerabilities, cannot detect any temporal memory vulnerabilities. In terms of design, a reason for the high false-negative rates in `CWE121` is that some test cases use indexes to access arrays. LowFat uses the pointer itself to find the corresponding metadata, so it ignores such cases that a memory operation using pointer offsets accesses another memory object. In such cases, the access length does not exceed the boundary of the victim object, LowFat considers the operation as legitimate. Array access is the most common cause of such cases, and we provide cases for further analysis and discussion in the Section 5.3.7. Moreover, LowFat also cannot detect sub-object overflow. In terms of implementation, since LowFat rounds up the allocated size of an object to a proper number to facilitate metadata (i.e., the bounds) retrieval, it cannot detect memory access that is beyond the object size but within the allocated size, which is a reason for high false-negative rates in some test sets. Besides, similar to HWASan, LowFat does not handle the `snprintf` functions.

Memcheck. In terms of design, Memcheck has a high false-negative rate in some test sets due to its inability to detect memory overflows in global or local objects. Apparently, Memcheck also cannot detect sub-object overflow.

SoftBound+CETS. SoftBound+CETS [34, 35] is believed to be a good combination providing full memory safety. However, it has a much higher false positive rate and false negative rate. Regarding

Table 5: Security evaluation results on the Juliet test suite. The evaluation runs for 3 times. In each test set (of a specific vulnerability type), there are multiple GOOD and BAD test cases, as presented in the brackets. Since SoftBound+CETS [34, 35] does not support C++ programs completely, they are evaluated on fewer test cases, as shown in the third to last column.

	PACMem		HWASan		ASan (without LSan)		LowFat		Memcheck		SoftBound+CETS		
Vulnerability	FP	FN	FP	FN	FP	FN	FP	FN	FP	FN	#CASES	FP	FN
CWE121 (3052+3052)	0	1.180%	0	16.972%	0	8.159%	0	46.888%	0	74.935%	2817+2817	2.520%	25.240%
CWE122 (3630+3630)	0	0.992%	0	0.992%	0	2.314%	0	42.369%	0	27.053%	3331+3331	46.142%	13.990%
CWE124 (1120+1120)	0	0	0	7.202%	0	8.75%	0	0	0	34.375%	1030+1030	23.301%	10.971%
CWE126 (822 + 822)	0	0	0	6.650%	0	14.842%	0	12.409%	0	57.664%	760 + 760	19.079%	7.105%
CWE127 (1120+1120)	0	0	0	14.435%	0	11.25%	0	14.911%	0	31.161%	1030+1030	23.301%	11.165%
CWE415 (818 + 818)	0	0	0	2.445%	0	0	0	1	0	0	748 + 748	2.273%	0
CWE416 (393 + 393)	0	0	0	17.303%	0	0	0	1	0	0	392 + 392	67.347%	10.204%
CWE476 (288 + 288)	0	0	0	0	0	0	0	0	0	0	270 + 270	16.296%	0
CWE761 (288 + 288)	0	0	0	1	0	0	0	1	0	0	264 + 264	2.273%	0

false negatives, likely HWASan, SoftBound+CETS does not handle some common memory access functions such as the strncat functions and it also cannot detect the dangling pointers passed to third-party libraries. Regarding false positives, there are some flaws in the prototype implementation of SoftBound+CETS, and thus it has a high false positive rate. Besides, it currently cannot support C++ programs, reducing the number of supported test cases. Moreover, even though SoftBound claims it can detect sub-object overflow in theory, the security tests have shown that the implementation cannot. We extensively analyze the current prototype using the intermediate LLVM bitcode. If the overflowing sub-object is the first sub-object of a structure, then it uses the boundary information of this structure for the bound check. The SoftBound paper does not elaborate on distinguishing boundary information when the program uses an object pointer to load the bound information, and the implementation falls back to the first sub-object to access the memory. Specifically, SoftBound uses the addresses in pointers to retrieve their bound information and cannot distinguish an object pointer and the first sub-object pointer of this object, who have the same addresses but different bounds. While the approach works for some cases, it is not generalizable. Take *char_type_overrun_memcpy_01 in CWE121 for example, where the memcpy function is called to assign value to a 16-byte array, i.e. the first sub-object of the global value ‘charVoid‘. In the bad case, the memcpy function uses the size of global value ‘charVoid‘ instead of 16 bytes. SoftBound calls __softboundcets_memcpy_check before calling the memcpy function, which uses the bounds information of the global value ‘charVoid‘ rather than that of the first sub-object, so SoftBound cannot detect the sub-object overflow.

5.3.7 Comparison between Sanitizers

PACMem outperforms other sanitizers in detecting out-of-bounds access, UAF, and invalid free vulnerabilities.

- For out-of-bounds access, if attackers can control the index used to access an array object to access another valid object out-of-bounds, ASan, Memcheck, and LowFat cannot detect such out-of-bounds access.
- For UAF, if the quarantine used by Asan is exhausted, Asan cannot detect UAF effectively; Memcheck cannot detect such UAF cases that the freed object is reallocated.
- For invalid free, since the pointer used to be freed still points to the middle of the object, its tag still matches the object's tag and can pass the HWASan security checks. So HWASan cannot detect invalid free vulnerabilities.

The above vulnerabilities can be detected by PACMem effectively. Next, we will use specific examples to demonstrate the advantages of PACMem.

Out-of-Bounds Access. As Listing 1 shows, if attackers can control the index used to access the buf1 array, they can access the memory of buf2 and cause a security violation. Many sanitizers, including ASan, Memcheck, and LowFat, cannot detect such out-of-bounds access.

Although ASan uses redzones to detect out-of-bounds access, it can only detect a memory operation accessing illegal memory areas, but cannot detect a memory operation jumping over the redzones and accessing another object. Memcheck only checks whether the accessed address is valid and whether there is an initialized object at the address. Therefore, it cannot catch out-of-bounds access to a valid and initialized object. Besides, LowFat utilizes the pointer itself to retrieve the bounds, i.e., LowFat retrieves the bounds using (buf1 + ptr_offset). It is clear that the bounds retrieved by LowFat is that of buf2, when ptr_offset is larger than the offset between buf2 and buf1 (i.e., buf2 - buf1). Thus, such out-of-bounds access can also pass the bound check of LowFat.

Since PACMem utilizes base addresses of objects as parts of seals and the seals can propagate along with pointers in a program, PACMem can calculate the accessed offset of the actual referent object for a pointer. When ptr_offset is larger than buf2 - buf1, the accessed offset is larger than the size of buf1, thus PACMem can detect such spatial safety violations.

Use After Free. As Listing 2 shows, after deallocating data1, a 256MB object is allocated and freed. Then, data2 is allocated. data2 has the same size as data1 and is located at the same address. Immediately afterward, the program uses data1_copy, which is a copy of data1, to store 'A' to the memory. The sanitizers ASan, Memcheck, and LowFat fail to detect such UAF vulnerabilities. ASan

```
char *buf1 = new char[100];
char *buf2 = new char[1000];
int ptr_offset = 0;
// Leak the address of buf1.
printf("%p\n", (void *)buf1);
// Leak the address of buf2.
printf("%p\n", (void *)buf2);
scanf("%d", &ptr_offset);
buf1[ptr_offset] = 'A';
// 'A' may end up somewhere in buf2.
delete [] buf1;
delete [] buf2;
```

Listing 1: An example of out-of-bounds access.

```
char *data1 = (char *)malloc(10 * sizeof(char));
char *data1_copy = data1;
free(data1);
char *a = (char *)malloc(sizeof(char) << 28);  // 256MB
free(a);
char *data2 = (char *)malloc(10 * sizeof(char));
data1_copy[0] = 'A';  // Use after free.
free(data2);
```

Listing 2: An example of use-after-free.

utilizes a quarantine to prevent a just freed object from being immediately reallocated. However, the quarantine size is limited, which is 256MB in the default configuration. Thus, ASan cannot detect such UAF vulnerabilities if the quarantine is exhausted and the memory of a previously freed object is reallocated. Since Memcheck only checks whether the accessed address is valid and whether there is an initialized object at the address, it is also unable to detect such vulnerabilities. LowFat also fails to detect UAF violations since it focuses only on spatial memory safety.

PACMem clears the metadata of the object pointed by `data1_copy` after the deallocation of the object. Therefore, the seal in `data1_copy` has no corresponding metadata in the metadata table and `data1_copy` cannot be used to access the referent object (i.e., `data2`). Besides, thanks to the unique birthmark for each allocation, both the seals of `data1_copy` and `data2` and the metadata of referent objects are different, though the base addresses and the size of them are the same. Thus, `data1_copy` cannot reuse the metadata of `data2`. In a word, the birthmark check of `data1_copy` will fail, and the attacker cannot use `data1_copy` to store `'A'` to the target memory.

Invalid Free. In Listing 3, `buff` firstly points to a newly allocated object, and then it is increased and does not point to the beginning of that object. Finally, `buff` is used to deallocate some object. Such vulnerabilities are called Invalid Free.

For HWASan, since `buff` still points to the middle of the object, its tag still matches the tag of the object and can pass the HWASan security checks. Similar issues may exist in other tag-based mechanisms including ARM Memory Tagging Extension (MTE). LowFat cannot detect Invalid Free vulnerabilities as it is designed for detecting spatial safety violations. PACMem checks whether a pointer used to deallocate some object points is a valid object pointer. In this case, `buff` does not point to the beginning of the object, so the check will fail. In this way, PACMem can detect such vulnerabilities.

5.4 Performance Evaluation

Test Suite and Inputs. Regarding the performance evaluation, we use all C benchmarks from the SPEC CPU2017 (of the SPECspeed 2017 suite) and Nginx to evaluate the overheads. Four test cases do not work correctly with the reference input on some sanitizers we compare with. In order to conduct a fair comparison, only these targets are changed to utilize the training input (which is more complex than the test input). For example, gcc_s compiled with LowFat cannot run through the reference input. Therefore we use the training input to test all gcc_s binaries with different sanitizers. Specifically, the reference input of 602 causes a segmentation fault in LowFat; the reference input of 605 causes HWASan to report the error: "Unable to allocate memory"; the reference inputs of 638 and 644 cause MemCheck to enter an endless loop (running for 1~3 days without stopping). Except for these special cases, all other benchmarks use the reference input.

```
char *buff;
buff = (char *)malloc(10);
buff++;
free(buff);  // Invalid free.
```

Listing 3: An example of invalid free.

Besides, since the compiler used by Softbound+CETS is too old to compile SPEC CPU2017, we use SPEC CPU2006 to compare the performance of PACMem with that of Softbound+CETS. For SPEC CPU2006, SoftBound+CETS fails to compile `400.perlbench`, `403.gcc`, `429.mcf`, and `462.libquantum`. In addition, the `445.gobmk` and `456.hmmer` benchmarks crash at runtime. Therefore, only 6 SPEC CPU2006 C benchmarks can be used to measure performance overhead. Note that reference inputs were used.

Experimental Setup. PACMem, ASan, and HWASan all detect global variable overflow for the `ldecoder` of `x264_s` benchmark and `gcc_s` benchmark. We manually confirmed that these are not false alarms. But, LowFat indeed has some false positives. However, we need to run the program completely to measure the performance overheads. Thus, we modify the default configuration to allow the program to run normally until completion, even if the sanitizer detects some violation.

Runtime Overhead. Table 6 shows the runtime overhead for PACMem, HWASan, and ASan on the ARM architecture. The average runtime overheads for ASan and HWASan are 81.63% and 108.39%, respectively. And the average runtime overhead of PACMem is 68.73%, which is lower than ASan and HWASan (while providing stronger security guarantees). We also tested the runtime overhead of the weaker form of PACMem, which only enforces memory safety checks for memory write accesses. The results are shown in Table 6 as well (denoted as PACMem w.o.). The runtime overhead of PACMem w.o. is 31.83%, which is much lower than others. To explore the PA Code collision's impact on runtime overhead, we counted the total number of memory objects allocated and the number of objects with collisions, thus calculating the collision rate. Combining the collision rate and runtime overhead results, we can find that the overhead is lower for the cases with no collisions. If the number of objects with collisions is high (such as `perlbench` and `gcc`), the runtime overheads of PACMem are more than `ASan` in such cases.

Other open-source sanitizers do not support the ARM architecture, so we tested their performance on the x86 architecture, as shown in Table 6 and Table 8. Compared to other open-source sanitizers on x86 architecture, ASan has the lowest runtime overhead. LowFat has an outlier when running `perlbench_s`, but the average runtime overhead after subtracting this outlier is still higher than ASan. Through the above comparison, we believe that PACMem will have a significant runtime overhead advantage if these sanitizers can be evaluated on the same experimental platform.

Memory Overhead. The memory usage of all sanitizers is shown in Table 7 and Table 9. The memory consumption of PACMem is much lower than that of ASan and Memcheck. Although the memory consumption of PACMem is higher than HWASan and LowFat, LowFat only detect spatial safety violations and the runtime overhead of HWASan is higher than that of PACMem.

Table 6: Runtime overheads of PACMem and PACMem write-only, HWASan, ASan, LowFat, and Memchck. Since LowFat has an outlier, the average overhead when including the outlier and excluding the outlier are calculated separately.

Test Case	AArch64						X86		
	Collision Ratio	Collision Num	PACMem	PACMem (+w.o)	ASAN	HWASAN	ASAN	LOWFAT	Memcheck
600.perlbench_s	0.96%	54930652	200.83%	101.63%	159.10%	360.79%	163.62%	20647.38%	3526.00%
602.gcc_s	1.48%	920485	201.27%	120.25%	135.44%	270.31%	114.14%	188.70%	2334.16%
605.mcf_s	0.02%	36	107.20%	57.69%	52.82%	36.13%	128.53%	62.83%	1840.636%
619.lbm_s	0	0	2.90%	1.69%	10.97%	14.45%	75.02%	-3.67%	311.56%
625.x264_s	0	0	5.55%	-11.43%	88.18%	105.80%	151.73%	370.46%	3673.84%
638.imagick_s	0	0	95.78%	30.08%	154.80%	80.62%	69.53%	79.94%	2124.29%
644.nab_s	0.23%	883	112.93%	28.10%	131.51%	116.94%	4.44%	21.65%	683.93%
657.xz_s	0	0	15.75%	10.28%	9.93%	18.87%	62.06%	72.78%	1796.04%
nginx	0	0	12.65%	3.74%	70.43%	190.84%	65.41%	∗	∗
Geometric Means			**68.73%**	**31.83%**	81.63%	108.39%	86.26%	237.21% / 87.20%	1649.97%

∅: Compilation failed. ∗: Crashed at runtime

Table 7: Memory usage of PACMem, HWASan, ASan, LowFat, and Memchck. Since LowFat has an outlier, the average overhead when including the outlier and excluding the outlier are calculated separately.

Test Case	AArch64					X86			
	Origin	PACMem	PACMem (+w.o)	ASAN	HWASAN	Origin	ASAN	LOWFAT	Memcheck
600.perlbench_s	158296 KB	167.94%	166.73%	608.13%	69.72%	158976 KB	435.50%	117422.86%	181.50%
602.gcc_s	95464 KB	331.20%	323.27%	455.41%	83.74%	96924 KB	409.06%	26.90%	183.16%
605.mcf_s	558784 KB	9.01%	8.95%	52.88%	7.23%	559024 KB	52.54%	22.83%	39.69%
619.lbm_s	3302496 KB	-0.01%	-0.01%	6.92%	6.291%	3305228 KB	12.71%	0.04%	34.06%
625.x264_s	120038 KB	11.45%	10.94%	36.21%	15.07%	120107 KB	24.98%	647.48%	95.60%
638.imagick_s	22500 KB	37.10%	34.29%	146.01%	0.30%	16436 KB	204.02%	-1.87%	1753.78%
644.nab_s	11540 KB	927.59%	925.61%	7060.09%	372.66%	16658 KB	4081.68%	7.25%	2570.13%
657.xz_s	4529600 KB	0.03%	0.03%	4.59%	6.66%	17291512 KB	1.58%	-0.09%	26.77%
nginx	3256 KB	243.61%	219.16%	7428.87%	404.18%	1884 KB	380.89%	∗	∗
Geometric Means		**106.39%**	**103.55%**	374.29%	67.59%		221.19%	230.99% / 43.05%	241.09%

∅: Compilation failed. ∗: Crashed at runtime

Table 8: Runtime overheads of PACMem, compared with ASan and SoftBound+CETS on the SPEC CPU2006.

Benchmarks	AArch64		X86	
	ASan	PACMem	ASan	Softbound+CETS
401.bzip2	63.92%	116.94%	57.78%	274.15%
433.milc	102.16%	48.71%	48.58%	282.78%
458.sjeng	76.60%	97.73%	107.24%	212.30%
464.h264ref	276.66%	228.94%	153.28%	1192.95%
470.lbm	28.25%	2.73%	115.47%	139.05%
482.sphinx3	51.19%	69.56%	76.84%	654.47%
Geometric Means	86.99%	82.17%	89.89%	367.43%

Table 9: Memory overheads of PACMem, compared with ASan and SoftBound+CETS on the SPEC CPU2006.

Benchmarks	AArch64		X86	
	ASan	PACMem	ASan	Softbound+CETS
401.bzip2	12.35%	0.38%	8.76%	104.79%
433.milc	41.38%	0.23%	40.86%	99.17%
458.sjeng	3.97%	15.81%	2.22%	5.29%
464.h264ref	804.48%	367.16%	691.17%	53.91%
470.lbm	13.94%	0.09%	13.20%	0.11%
482.sphinx3	928.15%	588.46%	801.88%	479.93%
Geometric Means	136.50%	82.96%	124.05%	83.66%

Since PA Code is well-distributed, for cases with a small heap (e.g. nab), the memory overhead of PACMem is not optimistic. However, due to the limited size of metadata tables, the memory overhead of PACMem is optimistic for the test cases with large heap memory.

Summary. Overall, PACMem has lower runtime overhead than HWASan and ASan. Its memory overhead is significantly lower than ASan's but higher than HWASan's. However, PACMem provides stronger security guarantees than HWASan. Specifically, PACMem has reduced the runtime overheads by 15.80% and the memory overheads by 71.58% than ASan. *Considering all performance overheads and security detection capabilities, PACMem is a better choice for balancing performance and security.*

6 Limitation

Limited PACs. Table 2 demonstrates that the 24-bit PA length is sufficient for regular programs. For instance, a long-running Apache server with 10,000 requests and 2,000 threads will consume at most 7,886 slots in the metadata table. But the metadata table of PACMem has 16,777,216 slots, which is way more than necessary.

However, in theory, there may be extreme cases that the metadata table is exhausted. In that case, the metadata table needs to grow, otherwise the PAC collision issue cannot be resolved. PACMem can use techniques like linked lists to address this problem. Specifically, when the metadata table is full, we can use a linked list to store conflicted metadata (i.e., with the same PAC). At the time of the security check, the linked list will be retrieved and iterated to find the matching metadata. It brings higher performance overheads.

False Negatives Caused by Sub-object Overflow. Same as other common sanitizers (such as ASan, LowFat and BaggyBounds), PACMem cannot detect overflows within objects, i.e., sub-object overflow. SoftBound [34] provides a best-effort approach, which uses static type information (i.e., from the source code) to narrow bounds to detect sub-object overflow in nearby location. It requires the object fields' metadata to be associated with pointers. In its current implementation, SoftBound cannot detect sub-object overflows if pointer casting is involved or some LLVM optimizations are applied. Besides, SoftBound cannot detect the cases discussed in Section 5.3.6. In summary, it is hard to track object fields' metadata efficiently. We therefore leave it as future work.

Birthmark and Temporal Safety. Commercial ARM devices lack a straight-forward lightweight source of randomness. Instead, we used a static random number generated at compile time and the dynamic stack pointer (i.e., the SP) to represent the context of the allocation site and stack, respectively. Even if the SP is the same, different allocation sites will generate different birthmarks. However, PACMem still can be bypassed with a low probability, when the spraying object is allocated at the same allocation site and SP as the freed object and they have the same object size and base address. However, its probability is so low that we did not observe a clash in any of our experiments. When future ARM devices introduce a lightweight source of randomness, we can change the static random number generator to the new randomness source, which will provide better supports to UAF detection and perhaps reduce the overheads.

7 Related Work

7.1 Memory Safety Sanitizer

Security researchers use sanitizers to detect security vulnerabilities dynamically. Different sanitizers target different classes of vulnerabilities, e.g., spatial safety violations [17, 18, 29, 34], temporal safety violations [12, 30, 47, 51], type confusion [21, 26], or undefined behavior [14]. Some sanitizers [16, 27, 30, 41, 47] only focus on spatial safety violations or temporal safety violations. We further discuss some classical and commonly-used sanitizers.

Memcheck [36, 44] is a memory corruption detector. It maintains a valid-address table to determine whether a pointer being dereferenced is valid and maintains a valid-value list to check whether the accessed object has been initialized. However, Memcheck cannot detect memory overflow in global variables and stack variables and cannot detect UAF since the freed object could be taken by another valid and initialized object. Besides, Memcheck introduces excessive-performance overheads.

SoftBound+CETS [34, 35] provides a full memory safety solution. SoftBound+CETS utilizes the pointer-based bounds-checking and identifier metadata associated with pointers. However, SoftBound+CETS introduces high runtime overheads.

AddressSanitizer (ASan) [42] has a better performance which utilizes the redzones to detect buffer overflows and uses a quarantine to catch Use-After-Free vulnerabilities. Utilizing the redzones can quickly detect memory vulnerabilities, but the redzones introduce high memory overheads. Besides, ASan cannot effectively detect UAF vulnerabilities after the quarantine has been exhausted.

LowFat [17, 18] further reduces overhead, which utilizes the bound check to detect out-of-bounds errors. It uses a special encoding scheme to encode the bounds into the pointers themselves for fast retrieval. However, LowFat just can detect spatial memory safety vulnerabilities. And if a malicious pointer accesses another object beyond the bounds and the access length does not exceed the bounds of the victim object, LowFat cannot effectively detect such spatial safety violations.

7.2 Metadata Management

Sanitizers, in general, will use metadata (such as bound information or tags) to track memory safety states and catch memory access violations at runtime. Metadata management is strongly related to sanitizers' runtime overheads and memory overheads.

ASan [42] and HWAsan [43] use the direct-mapped shadow to store the metadata for a block of 8 bytes. It is very efficient, but it wastes the memory occupied by metadata.

TypeSan [21], Intel MPX [39], and METAlloc [22] use a multilevel shadow to index metadata. Compared with the direct-mapped shadow, this method can effectively reduce the memory of metadata, but significantly degrades the runtime performance of programs with frequent memory accesses.

Oscar [13] and PTAuth [19] increase the allocation size of each object and append the metadata to the data of the objects. This metadata indexing approach called Embedded Metadata can effectively reduce the memory overhead caused by metadata. However, when the object's size is large enough, this approach may have false positives or excessive performance overheads.

Many efforts [15, 33, 48, 49] propagate a pointer with the corresponding metadata, i.e., a fat pointer, and implement different forms of fat pointers by extending registers or language implementations. However, the approach needs to change the processor hardware and increases runtime overheads. CHERI [48] is a new architecture extension in which pointers become capabilities. This extension is used to store information such as the capability to support memory safety checks at runtime. The metadata of CHERI is propagated at the same time as the pointer, and the overhead of accessing metadata during safety checks is low. However, CHERI requires modifications to the hardware and overall system architecture, which makes it difficult to deploy on commercial devices in a short time. ARM Morello [6] is the only physical implementation of CHERI, which implements the temporal memory safety mitigation derived from Cornucopia [20]. However, Morello is an experimental CHERI-extended implementation. Apple Firebloom [5] utilizes a structure to represent a pointer with the corresponding metadata [4], such as the lower and upper bounds. Firebloom can use the boundary information to check for out-of-bounds access. Compared to PACMem, Firebloom requires additional instructions to manage the metadata, and each new pointer or new structure requires 0x20 bytes, which requires more memory space.

LowFat [17, 18] encodes the object-bound information into the pointer itself via a special encoding scheme and has good compatibility. However, it modifies the heap allocator to round up the allocated sizes of objects to facilitate metadata (i.e., the bounds) retrieval, and stores objects with different sizes into different memory pages, causing significant memory fragmentation problems.

Some mechanisms [12, 35] use disjoint metadata to improves the compatibility. The disjoint metadata often corresponds to each pointer, so during the pointer propagation process, it is necessary to maintain the propagation and copy of its metadata.

In contrast to the above schemes, PACMem leverages unused bits of the original pointer as labels, and the tags for indexing the metadata. This type of technique has better compatibility than fat pointers and does not introduce additional cache pressure. Also, the method does not need to maintain metadata propagation during pointer propagation, has faster metadata queries relative to multilevel tables and embedded metadata, and does not waste excessive memory overhead relative to direct-mapped shadow.

7.3 Hardware Expansions

HWASan [43] and ARM MTE [11] use the higher-order bits of the pointer to store the tag, and memory access is only possible if the tag of the pointer and the memory are identical. The memory tag in HWASan is 8 bits, and ARM MTE is a 4-bit integer associated with each 16-byte aligned memory region. Due to the length of the tag, the probability of tag collision is 6.25%. So MTE and HWASan can only probabilistically detect memory corruption.

AOS [28] is a heap memory safety guard. This scheme implements a set of variant instructions based on PA and some additionally needed hardware extensions. PACMem and AOS also use tags to index metadata, but there are critical differences between these two schemes. On the one hand, AOS uses one tag for multiple metadata sets to resolve hash collisions, which perhaps increases the number of queries at the time of visit. When the metadata table is full, AOS needs to increase the number of sets for each tag to extend the metadata table, which is expensive because of the copy of all the entries. PACMem ensures the uniqueness of tags during tag generation, thus avoiding hash collisions. On the other hand, AOS uses SP to generate the pointer signature, uses the object size to generate the 2-bits address hashing code (AHC), and may fail to detect use-after-realloc vulnerabilities (with the same base address, object size, and SP). PACMem uses SP and a static random number to yield a random and conflict-free birthmark, which further contributes to the pointer authentication code, and thus avoids such FNs. In addition, this scheme only strips pointers when memory objects are deallocated and does not verify the integrity of pointers, so it cannot detect invalid free vulnerabilities. Moreover, AOS requires changes to the processor, ISA, and the OS kernel. These extra required hardware extensions increase hardware costs and make its adoption challenging. However, PACMem takes advantage of existing hardware features without any hardware modifications. Compared with AOS, PACMem is more practical (compared to AOS and does not require hardware modifications), secure, and efficient.

BOGO [53] utilizes Intel MPX to implement the spatial and temporal security. Specifically, the scheme finds dangling pointers by scanning the MPX table and invalidating their bound information stored in the MPX Table. However, Intel claims that MPX has been deprecated and both GCC and Linux kernel no longer support Intel MPX. The method of No-FAT [54] indexing metadata is similar to LowFat in that it uses the pointer itself to compute the object's base address and thus check if memory accesses are out of bounds. The difference with LowFat is that No-Fat's base address calculation and memory access checking functions are implemented in hardware rather than software. The difference with LowFat is that No-Fat's base address calculation and memory access checking are implemented in hardware. Furthermore, No-Fat also uses tags to catch temporal safety violations.

In-Fat [50] indexes metadata for heap, stack, and global variables, using embedded metadata, additional registers, and direct-mapped shadow, respectively. However, In-Fat only implements detection for spatial memory safety bugs, not for temporal security.

7.4 Memory Safety based on ARM PA

PARTS [31] provides pointer integrity for programs using the ARM PA feature. It utilizes type information of the target pointers and other auxiliary information as modifiers to sign and authenticate pointers. Since PARTS uses static type information as the modifier to sign pointers, attackers can bypass it by reusing signed pointers with the same static modifier. Besides, PARTS is a mitigation against control flow hijacking attacks. PACMem can detect memory corruption bugs to fix, thus stopping such potential attacks.

PTAuth [19] proposes an effective runtime protection scheme for heap-based temporal safety using the ARM PA extension. PTAuth assigns a unique ID for each object to sign the object's base address. PTAuth can use the unique ID to do a pointer authentication during every pointer dereference, and report a temporal violation when the authentication fails. However, this scheme does not protect against spatial violations, e.g., heap overflows, which can overwrite the rest of the heap memory while keeping the ID stored on the heap unchanged. Besides, PTAuth considers the memory overhead of metadata, so it uses the embedded metadata method to store the ID at the beginning of the object, but this method increases the overhead of querying metadata. It is necessary to search backward iteratively to find the valid ID when a pointer inside a heap object is being authenticated. Compared with PTAuth, PACMem's security check is more efficient. According to PTAuth's evaluation, PTAuth tested three test sets from the Juliet test, which are Double-Free, Use-After-Free, and Invalid-Free. According to Table 5, PACMem is effective in detecting not only these three vulnerabilities types, but also six others.

8 Conclusion

Existing memory safety sanitizers either provide partial memory safety guarantees or have excessive performance overheads. Our novel sanitizer PACMem enforces full memory safety by precisely tracking all necessary memory safety metadata of objects and enforcing complete mediation on all pointer dereferences. Further, PACMem utilizes the hardware feature ARM PA to seal metadata directly into pointers and places metadata in a global table indexed by the seal. This mechanism saves the metadata (seal) propagation overhead and enables efficient runtime metadata retrieval and checks. Experiments demonstrate that PACMem has no false positives and negligible false negatives (i.e., missing checks for sub-object overflows) and provides a stronger security guarantee than state-of-the-art sanitizers, including HWASan, ASan, SoftBound+CETS, Memcheck, LowFat, and PTAuth, while introducing lower performance overheads.

Acknowledgement

We would like to thank David Chisnall and all anonymous reviewers for their insightful comments and feedback. This project has received funding, in part, from the National Key Research and Development Program of China (2021YFB2701000), National Natural Science Foundation of China (61972224), Beijing National Research Center for Information Science and Technology (BNRist) under Grant BNR2022RC01006, Alibaba Group through Alibaba Innovative Research Program, and SNSF PCEGP2_186974 and ERC grant number 850868.