PTAuth: Temporal Memory Safety via Robust Points-to Authentication

Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu, *Northeastern University*

https://www.usenix.org/conference/usenixsecurity21/presentation/mirzazade

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the
30th USENIX Security Symposium
is sponsored by USENIX.

PTAuth: Temporal Memory Safety via Robust Points-to Authentication

Reza Mirzazade Farkhani
Northeastern University
mirzazadefarkhani.r@northeastern.edu

Mansour Ahmadi
Northeastern University
Mansosec@gmail.com

Long Lu
Northeastern University
l.lu@northeastern.edu

Abstract

Temporal memory corruptions are commonly exploited software vulnerabilities that can lead to powerful attacks. Despite significant progress made by decades of research on mitigation techniques, existing countermeasures fall short due to either limited coverage or overly high overhead. Furthermore, they require external mechanisms (e.g., spatial memory safety) to protect their metadata. Otherwise, their protection can be bypassed or disabled.

To address these limitations, we present *robust points-to authentication*, a novel runtime scheme for detecting all kinds of temporal memory corruptions. We built a prototype system, called PTAuth, that realizes this scheme on ARM architectures. PTAuth contains a customized compiler for code analysis and instrumentation and a runtime library for performing the points-to authentication as a protected program runs. PTAuth leverages the Pointer Authentication Code (PAC) feature, provided by the ARMv8.3 and later CPUs, which serves as a simple hardware-based encryption primitive. PTAuth uses minimal in-memory metadata and protects its metadata without requiring spatial memory safety. We report our evaluation of PTAuth in terms of security, robustness and performance using 150 vulnerable programs from Juliet test suite and the SPEC CPU2006 benchmarks. PTAuth detects all three categories of heap-based temporal memory corruptions, generates zero false alerts, and slows down program execution by 26% (this number was measured based on software-emulated PAC; it is expected to decrease to 20% when using hardware-based PAC). We also show that PTAuth incurs 2% memory overhead thanks to the efficient use of metadata.

1 Introduction

Memory corruptions remain to be the most commonly exploited software vulnerabilities, despite the significant progress made by decades of research on mitigation techniques. Memory corruptions are caused by programming errors (or bugs) that break the type constraints of data in memory. They serve as the stepping stone for launching almost all types of software attacks, from simple stack smashing to heap spray and to more advanced return-oriented programming (ROP) and code reuse attacks. Generally, memory corruptions exist in two different forms: *spatial* or *temporal*. The former happens when data's spatial boundary is breached. The latter is due to data being used out of its life span.

Temporal memory corruptions may seem less harmful than spatial corruptions. However, they are being increasingly exploited to bypass the state of the art defenses against spatial corruptions or control flow manipulations. Use-after-free (UAF) is the most common type of temporal memory corruption. A recent analysis on the Chromium project shows that 50% of the serious memory safety bugs are UAF issues [28]. To exploit a UAF, an attacker first plants crafted objects in place of expired/freed objects and then waits for the vulnerable program to access the planted objects, and in turn, unknowingly invoke code specified by the attacker. Double-free and invalid-free are two other types of temporal memory corruptions [49] that can provide arbitrary write primitives for attackers.

To counter the powerful and stealthy attacks enabled by temporal memory corruptions, many mitigation or prevention techniques were proposed recently. One approach adopted by these works [37, 44, 58, 62, 65] aims to disable dangling pointers, without which UAF and its variants cannot occur. Though effective, these techniques are either too heavy for real-world deployment [44, 65] or limited in their scope of protection. For instance, DangNull [44] can only protect those pointers that reside on the heap. Techniques solely focusing on preventing dangling pointers, such as Oscar [37], are unable to prevent invalid-free vulnerabilities.

Another line of works on preventing temporal memory corruptions monitors every pointer dereference during runtime and ensures that the to-be-dereferenced pointer indeed points to the expected object (or type) [35, 47, 49]. These techniques are, in principle, more comprehensive than dangling pointer prevention. However, they tend to incur heavier runtime overhead.

Despite the approaches, the aforementioned techniques all

require spatial memory safety to protect their in-memory metadata, whose integrity is critical for the runtime monitoring. This common requirement underlines two limitations of these techniques. First, without external protection, they themselves are not robust against attacks or evasions. Second, requiring spatial safety can significantly increase the already high runtime overhead. Furthermore, many of the existing mitigations against temporal memory corruption, including [44, 62, 65], store a considerable amount of metadata in memory, increasing the memory footprint by as much as 2 times.

Motivated by the limitations of previous works (esp. limited coverage, the requirement of external protection, and high overhead), we present PTAuth, a novel system for dynamically detecting temporal memory corruptions in user-space programs. PTAuth follows the approach of runtime dereference checking. Unlike previous works, PTAuth has built-in protection of its in-memory metadata and thus obviates the need for external mechanisms to provide spatial memory safety. Moreover, PTAuth uses a checking scheme that minimizes metadata size and optimizes metadata placement for better compatibility and handling of data and pointer propagation.

Specifically, during the allocation of every heap object, PTAuth assigns a unique ID to the object and computes a cryptographic authentication code (`AC`) based on the object ID and the base address of the object. PTAuth stores the ID to the beginning of the object. It stores the `AC` to the unused bits of the pointer to the object. As a result, the pointer is "tied to" the object (or the pointee) at the particular location in memory. This points-to relationship can be verified during every pointer dereference by re-computing the `AC`. An `AC` mismatch indicates a temporal memory safety violation.

The in-memory metadata of PTAuth include `AC` for pointers and IDs for objects. Obviously, the robustness of the runtime checks hinges on the integrity of the metadata. PTAuth can detect corrupted or invalid metadata without requiring any form of spatial memory safety, thanks to the design of `AC`. By using a secret key for computing and verifying `AC`, PTAuth prevents attackers from forging or tampering with metadata. We prototyped PTAuth for the latest ARM architecture and employ PAC (pointer authentication code) [16], a hardware-based feature, to implement `AC`. PAC was originally designed for checking the integrity of protected pointers and has been enabled on the latest iOS devices [23]. We repurposed this hardware feature for performing secure encryption (i.e., calculating `AC`) and storing `AC` in unused bits of pointers.

The in-pointer storage of `AC` offers two benefits. First, storing `AC` does not consume additional memory space. Second, an `AC` is propagated automatically when the pointer is copied or moved, without requiring handling or tracking by PTAuth. An object ID is 8-byte long and is stored at the beginning of the object. This distributed placement of object IDs, as opposed to centralized storage, makes the runtime check faster.

In summary, we made the following contributions:

- We designed a novel scheme for dynamically detecting temporal memory corruptions, which overcomes the limitations of previous works and achieves minimal metadata, full coverage, and built-in security against attacks and metadata tempering.
- We built a system for ARM platforms that utilizes PAC to implement the detection scheme in an efficient and secure way.
- We evaluated the prototype using standard benchmarks and compared it with the state-of-the-art temporal corruption detectors, confirming the advantages of our approach.

2 Background

2.1 Exploiting Temporal Memory Bugs

Use-after-free: If a program reuses a pointer after the corresponding buffer had been freed, attackers may plant a crafted object in the same memory location, after the free and before the use, to trick the program into using the crafted object and consequently perform attacker-specified actions. According to recent reports [36, 44], UAF now counts for a majority of software attacks, especially on browsers, mostly because the deployed attack mitigations are unable to detect them. Moreover, most of the recent Android rooting and iOS jailbreaking exploits use UAF as a key part of their attack flows [13].

Double-free: Double-free is a special case of UAF, which occurs when a pointer is freed twice or more. This leads to undefined behaviors [8] and can be exploited to construct arbitrary memory write primitives, with which an attacker can corrupt sensitive information such as code pointers and execute arbitrary code.

Invalid-free: Invalid-free occurs when freeing a pointer that is not pointing to the beginning of an object or a heap object at all (i.e., freeing a pointer that was not returned by an allocator) [8, 49]. Similar to double-free, invalid-free may allow attackers to gain arbitrary memory overwrite abilities. The idea of *House of Spirit* [54] exploitation technique is partly based on exploiting invalid-free errors.

2.2 Pointer Authentication Code on ARMv8

Pointer Authentication Code, or PAC, is a new hardware feature available on ARMv8.3-A and later ARM Cortex-A architectures [10]. PAC is designed for checking the integrity of critical pointers. Compilers or programmers use the corresponding PAC instructions to (1) generate signatures for selected pointers, and (2) verify signatures before signed pointers are used. For instance, in a typical use case of PAC, compilers insert to programs the PAC instructions that, during runtime, sign each return address (*i.e.,* a special code pointer)

before saving it and check the signature before every function return. PAC is designed to detect unexpected or malicious overwrites of pointers. It has been deployed and enabled on the latest iOS devices [23].

PAC generates pointer signatures, or authentication codes, using *QARMA* [30], a family of lightweight block ciphers. *QARMA* takes two 64-bit inputs (one pointer and one context value), encrypts the inputs with a 128-bit key, and outputs a 64-bit signature. A context value is chosen by the programmer or compiler for each pointer. A total of five keys can be set by the OS (*i.e.,* code running at EL1) for encrypting/signing different kinds of pointers. Signatures are truncated and stored in the unused bits of signed pointers (*i.e.,* depending on the virtual address space configuration, 11 to 31 bits in a 64-bit pointer could be unused).

Very recently, ARM announced ARMv8.6-A [25], which introduced some enhancements to PAC. In ARMv8.3, when a pointer authentication process fails, the top bits of the invalid pointer is changed to *0x20*, which makes the pointer invalid to use. In contrast, in ARMv8.6, an exception is thrown when a pointer authentication fails, which prevents an attacker to brute-force the correct signature. Another improvement in ARMv8.6 is that a signature is XORed with the upper bits of the pointer, which help mitigate signature reuse attacks. At the time of writing this paper, no publicly available hardware or simulator supports ARMv8.6. Our design and implementation of PTAuth are based on ARMv8.3. We discuss in §3.5 how our design can be made compatible with ARMv8.6.

Table 1 lists a subset of PAC instructions. Each instruction serves one purpose (signing or authentication), targets one type of pointers (code or data), and uses one of the five keys (*i.e.,* two keys for each pointer type plus a generic key). Differentiating pointer types and having multiple keys help reduce the chance of pointer substitution or reuse attacks. The bottom two instructions in Table 1 are special. `PACGA` is not specific to pointer authentication and can be used as a data encryption instruction on small data objects (16 bytes at most). It uses the generic key and outputs a 32-bit cipher to the upper half of a general-purpose register. `XPAC` removes the signature from a signed pointer without any authentication. Therefore, it does not use any key. PAC is designed for fast and robust checking of pointer integrity. The signing and authentication are performed directly by the CPU without any software-level assistance. The keys are stored in the special CPU registers, which are accessible only to OS or EL1 code and not visible to user-level code.

PAC was originally designed for checking the integrity of pointers and has been mostly used for protecting code pointers. We use PAC as a simple hardware-based primitive for efficiently and securely computing `AC`. The `AC` is computed and verified based on our novel scheme designed for detecting temporal memory corruptions. Compared with PAC, PTAuth achieves a security goal (*i.e.,* enforcing temporal memory safety) that is orthogonal to, and broader than, the original purpose of PAC (*i.e.,* checking pointer value integrity). The recent UAF vulnerability in iOS (CVE-2019-8605 [26]) is a real example that shows PAC is unable to prevent temporal memory corruptions, which are exploited for jailbreaking or compromising iOS devices. In contrast, PTAuth is designed to stop temporal memory corruptions, which remain a type of commonly exploited vulnerabilities today.

Instruction	Key Used	Pointer Type	Purpose
PACIAx	Code.A	Code	Signing
PACIBx	Code.B	Code	Signing
PACDAx	Data.A	Data	Signing
PACDBx	Data.B	Data	Signing
AUTIAx	Code.A	Code	Authentication
AUTIBx	Code.B	Code	Authentication
AUTDAx	Data.A	Data	Authentication
AUTDBx	Data.B	Data	Authentication
PACGA	Generic	Generic	General
XPAC	-	-	Sig. stripping

Table 1: PAC-related instructions.

2.3 Fixed Virtual Platforms (FVP)

The ARMv8.3-A architecture (including PAC) was announced in late 2016 and is expected to enter mass production in 2020 to replace the current mainstream mobile architecture, namely ARMv8.0-A. At the time of writing, no development boards or commercially available SoC (Systems-on-Chip) use ARMv8.3-A. Apple's latest iOS devices, using the A12 Bionic SoC, is based on ARMv8.3-A and supports PAC. However, the SoC and OS are proprietary and cannot be used for testing the prototype of PTAuth.

ARM offers so-called Fixed Virtual Platforms (FVP) for to-be-released architectures [22]. FVP is a full-system simulator that includes processors, memory, and peripherals. It is a functionally accurate model of the simulated hardware. FVP allows for the development and testing of drivers, software, and firmware prior to hardware availability. It is widely used in the industry.

Following this standard practice, we used the ARMv8.3-A FVP when building and evaluating our prototype system. Thanks to FVP's functional accuracy, the evaluation results obtained on FVP are expected to be close to those obtained on actual hardware. We discuss more the implementation and evaluation in §5 and §6, respectively.

3 Design

3.1 System Overview

The goal of PTAuth is to dynamically detect temporal memory corruptions in the heap. The high-level idea is that, upon each pointer dereference (or pointer-based object access), temporal

memory corruption can be detected by checking (1) whether the pointer is pointing to the original or intended object, and (2) whether the metadata or evidence proving the points-to relationship is genuine.

Although the high-level idea is conceptually straightforward, how to realize it in an efficient and secure way is in fact challenging. What metadata are needed for establishing the points-to relationship? How are they computed and where are they stored? How can their integrity be verified? Answers to these design questions determine the efficiency and robustness of PTAuth. For instance, recording too much metadata leads to unnecessarily big memory footprint and redundant checks. Storing metadata separately from objects and pointers may ease metadata protection but significantly increase the overhead for locating and accessing metadata. Storing metadata in-place allows for fast access, but pointer arithmetics may complicate locating metadata. Moreover, in-place metadata is hard to protect and can be easily corrupted.

Our points-to authentication scheme overcomes these challenges and the limitations of previous works. PTAuth randomly generates an ID for each heap object upon its allocation. It also computes a cryptographic authentication code (`AC`) based on the object ID and the base address of the object. The object ID, stored at the beginning of the object, and the `AC`, stored in the unused bits of the object's pointer, together serve as the metadata to establish the verifiable points-to relationship between the object and its pointers. Furthermore, PTAuth can detect forged or corrupted metadata as long as the key for computing `AC` remains confidential and the `AC` computation can only be performed by PTAuth. We discuss the detailed design of the points-to authentication scheme in §3.4.

Our implementation of the points-to authentication scheme takes advantage of the PAC feature on ARM architectures. PTAuth uses PAC as a simple primitive, provided by hardware, for computing and checking `AC` and securing the key. We discuss in §3.5 the use of the PAC instructions and the compiler-based code instrumentation.

Figure 1 presents an overview of the PTAuth system. The PTAuth compiler instruments a protected application by inserting a runtime library and placing necessary hooks before selected `load` and `store` operations. During runtime, the PTAuth library checks the instrumented, pointer-based load-/store operations. The checking is based on a novel scheme that verifies the points-to relationship and the metadata integrity. It uses PAC as the hardware-based authentication primitive. PTAuth also installs a tiny OS patch for managing PAC encryption keys, which are only accessible from the kernel-space (or EL1) as enforced by the architecture. We discuss the design details after explaining the threat model.

3.2 Threat Model

We adopt a threat model common to user-space dynamic memory error checkers. We trust the OS and the underlying hardware (*i.e.,* the TCB). It is technically possible to reduce or remove the trust on OS if a more privileged entity can protect the PAC key management routine (*e.g.,* a hypervisor or EL2), which however is out of the scope for our current design. Our threat model also assumes that the basic defenses against code injection and modification are in place (*e.g.,* DEP and read-only code). This assumption is realistic because such defenses are universally enabled on modern OSes. They are needed for protecting code instrumented by PTAuth (*e.g.,* inline checks cannot be removed or uninstrumented code cannot be injected).

Our threat model also assumes that attackers cannot perform arbitrary memory read when exploiting temporal memory errors. Arbitrary memory read would allow an attacker to read a legitimate `AC` in memory and possibly reuse it, thus bypassing the security check. Assuming the absence of arbitrary memory read in our context is acceptable because finding an `AC` as well as its corresponding object ID in memory can be quite challenging due to ASLR and the indistinguishability between `AC` or ID values and other in-memory data. Moreover, based on previous research [32,40] and real-world attacks [9], attackers often exploit temporal memory errors as a stepping stone to obtain arbitrary memory read abilities, as shown in the high-profile WhatsApp double-free (CVE-2019-11932), Internet Explorer use-after-free (CVE-2013-3893) and iOS use-after-free (CVE-2019-8605 [26,27]) exploits. Therefore, it is realistic to assume attackers cannot perform arbitrary memory read while exploiting temporal memory errors.

Previous works on temporal memory safety [44,49,58,62,65] made all the above assumptions as we do. Additionally, they assumed the absence of spatial memory violations or required an external spatial safety mechanism to protect their metadata. In contrast, PTAuth does not make this assumption or require external spatial safety enforcement. We relaxed the threat model used in the previous work by allowing arbitrary memory overwrite, which an attacker may use to corrupt the metadata. Unlike the previous work, PTAuth has built-in metadata integrity check and is therefore robust against metadata corruption caused by spatial memory errors or attacks.

One might argue that PTAuth can be bypassed by attackers who are able to perform arbitrary memory read and write at the same time. While this argument is technically true, such a powerful attacker does not need to exploit temporal memory vulnerabilities at all, or try to bypass PTAuth, because she already has the ability to directly mount the final-stage attacks, such as code injection or data manipulation.

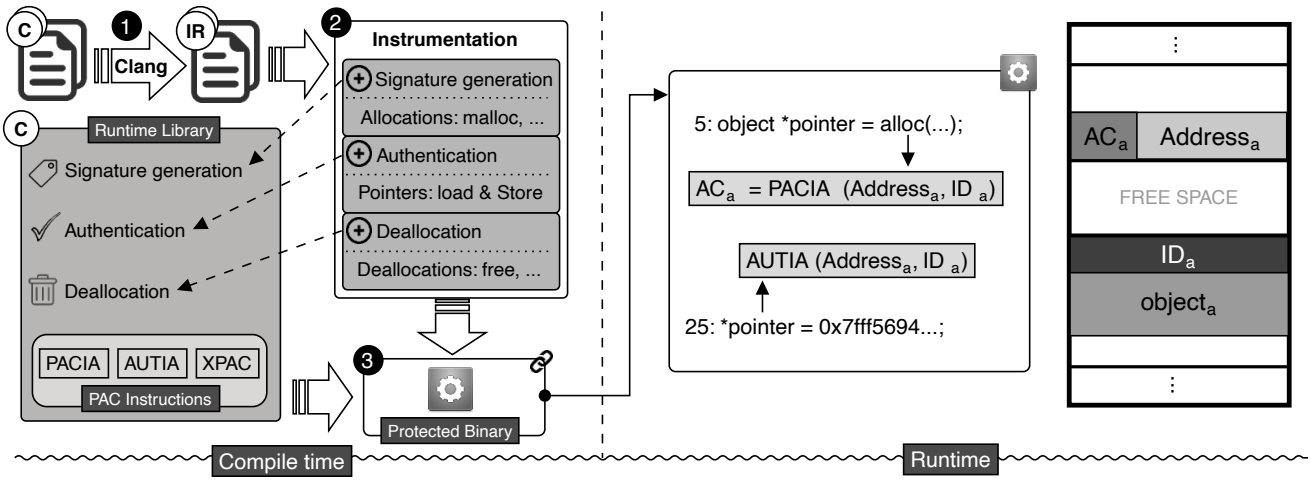

Figure 1: During the compile-time (left), PTAuth instruments a `C` program. It places the hooks in the program that are needed for the PTAuth runtime library to detect heap-based violations of temporal memory safety. The PTAuth library uses a novel authentication scheme (§3.4) that verifies the points-to relationship from pointers to pointees. During runtime (right), the PTAuth library generates signatures (`AC`) for heap objects and their pointers upon memory allocations, checks signatures upon pointer dereferences, and invalidates signatures upon object deallocations (§3.5). The scheme uses PAC as a simple building block for hardware-based metadata signing, storing (for pointers only), and verification.

```
1  int* ptr = (int*)malloc(10);
2  int* qtr = ptr;
3  ...
4  if (error) {
5      free(ptr);
6      ptr = null;
7  }
8  ...
9
10 if (log)
11     logError("Error", qtr);
```

(a) Use-After-Free

```
1  int* ptr = (int*)malloc(10);
2  int* qtr = ptr;
3  ...
4  if (error) {
5      free(ptr);
6      ptr = null;
7  }
8  ...
9
10 cleanCache:
11     free(qtr);
```

(b) Double-free

```
1  char* ptr = (char*)malloc(10);
2  for (; *ptr != '\0'; ptr++){
3      if (*ptr == SEARCH_CHAR)
4      {
5          printf("Match!");
6          break;
7      }
8  }
9  ...
10 cleanCache:
11     free(ptr)
```

(c) Invalid-Free

Figure 2: Examples of double-free, use-after-free and invalid-free temporal memory corruptions, which are undetectable by pointer integrity approaches but detectable by PTAuth.

3.3 Example Vulnerabilities

Before describing our points-to authentication scheme, we present three simple examples of temporal memory corruption below, which help explain why PAC can reliably detect them.

Use-after-free vulnerability: Figure 2 (a) is a typical example of UAF, where a pointer is used after its pointee has been freed. In this case, `qtr`, an alias of `ptr`, is used at Line 11 after `ptr` has been freed at Line 5. Although the programmer nullified the `ptr` at line 6, due to the aliasing, UAF still exists.

Double-free vulnerability: Figure 2 (b) shows a code snippet where a pointer can be freed twice, which may lead to undefined behaviors, including arbitrary memory writes.

Invalid-free vulnerability: Figure 2 (c) demonstrates a case where a pointer is freed while it is not pointing to the beginning of a buffer. This is a special type of temporal memory corruption [8, 49].

3.4 Points-to Authentication Scheme

Our authentication scheme applies to two types of data: objects and data pointers. *Objects* are dynamically allocated data on the heap. *Data pointers* reference the addresses of objects (we only consider pointers to heap objects in this paper). PTAuth verifies the identity of every object and the points-to relationship before it is accessed through a pointer. This verification relies on the `AC` (or authentication code) generated for the object and stored in its pointers.

The *ID* of an object is saved as inline metadata immediately before the object in memory (Figure 3). The ID establishes unique identities for objects and allows for binding pointers to their referenced objects (*i.e.,* making the points-to relationship verifiable), which is essential for detecting temporal memory corruptions. Figure 3 (lower right) shows two objects in the heap with their metadata. The ID is a 64-bit random value generated at the allocation of the object. An `AC` is 16-bit long and stored in the unused bits of a pointer (i.e., 48

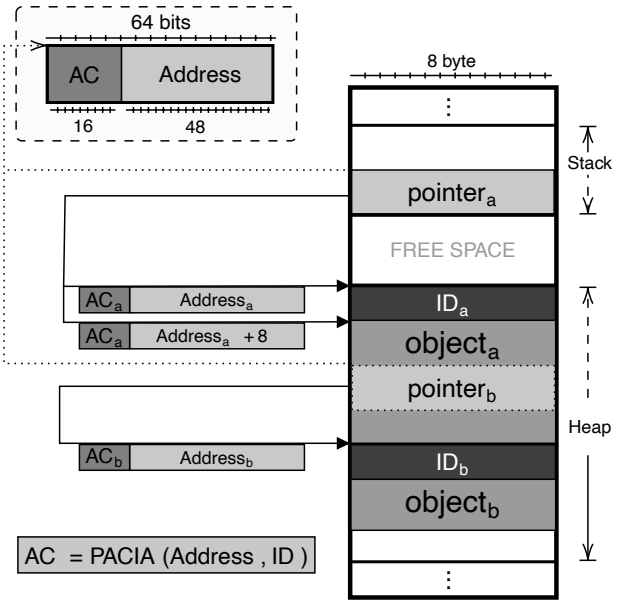

Figure 3: Authentication code (`AC`) and object metadata (ID) defined by PTAuth for pointers and objects. The object metadata is stored in the 8-byte memory proceeding the object. The `AC` is stored in the unused bits of the pointer, which is 16-bit long.

effective bits in a pointer). Unlike the previous works such as DangNull [44], which only protect pointers residing in the heap, PTAuth authenticates data pointers stored everywhere in memory, including heaps, stacks and global regions.

Next, we explain the definition and calculation of `AC`. We then discuss in §3.5 the runtime `AC` generation and the checking mechanism.

Data Pointers: `AC` essentially binds a data pointer to its pointee and makes the binding verifiable. `AC` encodes: (1) the identity of the pointee object, and (2) the base address of the pointee. The ID and the base address together uniquely identify an object in time and space. This definition not only makes the points-to relationship easily verifiable, but also mitigates metadata reuse attacks. Figure 3 (bottom left) shows the computation of `AC` using the `PACIA` instruction. PTAuth performs this computation when an object is allocated. When an object is deallocated or reaches the end of its life cycle, PTAuth simply invalidates its ID (setting it to zero). Upon each pointer dereference, PTAuth recomputes the `AC` and compares it with the `AC` stored in the pointer. A mismatch indicates a temporal memory safety violation. No temporal memory corruption can happen without failing the points-to authentication.

In our scheme, less memory is used for storing the metadata for both pointers and objects than most previous works. Furthermore, there is no assumption that the metadata cannot be tampered with. Last but not least, PTAuth can find the base address of an object reliably with the help of PAC. This is necessary for supporting pointer arithmetic operations, which may shift a pointer to the middle of its pointee, than thus, fail a naive authentication that simply takes the pointer value as the object base address. We discuss the details in §3.5.

The PAC instruction encrypts/signs the inputs (i.e., object ID and base address) using a data pointer keys (`data.A` or `data.B`) and saves the truncated ciphertext to the unused bits in the pointer. Therefore, unlike an object, PTAuth does not need to use extra space for storing `AC` for pointers. An `AC` is generated whenever a pointer takes a new value, which can happen at object allocation or when the pointer is re-assigned to another object (*e.g.,* via the reference operator "`&`").

After a pointer becomes stale when its pointee is freed, any dereference of the dangling pointer will trigger an object ID mismatch, due to either the invalidated ID of the freed object, or a different ID of a new object allocated at the same location. Other temporal memory errors, such as double-free and invalid-free can be detected by PTAuth in the same way.

Code Pointers: Checking the integrity of code pointers is an intended use of PAC and is fairly straightforward. Unlike data pointers, we do not define our own `AC` for code pointers. PTAuth is fully compatible with the intended use of PAC for code pointers for preventing control flow hijacking attacks. They can be used together to thwart a broad range of attacks. We do not consider or claim code pointer authentication as a contribution to this work. For the rest of the paper, we focus our discussion on authenticating the points-to relationships while referring readers to the PAC documentation [12, 16] and PARTS [45] for code pointer authentication.

3.5 Compiler-based Code Instrumentation & Runtime `AC` Checking

To apply the points-to authentication scheme to a given program, PTAuth takes the general approach of inline reference monitoring. Via a custom compilation pass added to LLVM [24], PTAuth instruments the program so that `AC` can be generated and checked at the right moments during program execution. The instrumentation is performed at the LLVM bitcode level, which is close to assembly code while retaining enough type and semantic information for our code analysis and instrumentation. The instrumentation sites are carefully selected to minimize the interception of program execution. Below we discuss in detail the code instrumentation needed for each type of operation on `AC`.

AC Generation: During runtime, PTAuth needs to generate `AC` for data pointers whenever a new points-to relationship is created. To this end, during compilation, PTAuth performs two types of instrumentation. First, it instruments all essential API for heap memory allocation, including `malloc` (the dynamic allocator for heap objects), `calloc` and `realloc`. PTAuth only works on user-space programs and we assume the `ptmalloc` allocator is used. This instrumentation allows PTAuth to intercept all memory allocations, where the object

ID is generated and the AC for the pointer is computed as follows:

```
1  /* Computing AC for Data Pointer */
2  ID = RandomID() // 64-bit
3  AC = PACIA|B <BasePointer><ID>
```

Second, PTAuth instruments object deallocation sites, like free (heap object deallocation). At an object deallocation site, PTAuth simply sets the object ID to zero, which invalidates the object and thus prevents any further use of the object. Figure 3 (upper right) shows an example pointer and its AC. The base address and the ID of the pointee are used as the two inputs to the PACIA|B instruction to generate the AC:

AC Checking: PTAuth performs points-to authentication by checking the AC whenever a pointer-based data access happens (or a pointer reference occurs). During compilation, PTAuth instruments LLVM load and GetElementPtr instructions for pointers. For simplicity, we generally refer to both as load in our discussion. PTAuth verifies the integrity of the pointer and authenticates the AC of the pointer value as follows:

```
1  /* Authenticating AC for Data Pointers */
2  ID = getID(Pointee) // Pointee is an object
3  AUTIA|B <BasePointer><ID>
```

Due to pointer arithmetics, a (legitimate) pointer may sometimes point to the middle, instead of the base, of its pointee. Therefore, during AC checking, PTAuth cannot simply use the value of the pointer as the base address of the to-be-accessed object. A naive solution to this problem is to use additional metadata for recording the object base address for each pointer. However, this not only increases space overhead but creates a more challenging problem of propagating the metadata as pointers are copied or moved.

Backward Search: PTAuth finds the base address of an object during runtime without requiring any additional metadata. For each AC checking, PTAuth, by default, uses the pointer value as the object base address. If the check fails, two possibilities arise. First, the pointer is valid but is pointing to the middle of its pointee (i.e., its value is not the base address, hence the mismatched AC). Second, the pointer is invalid and a temporal memory violation is about to happen. When encountering a failed AC check, PTAuth initially assumes that the first possibility happened. It then starts a backward search from the current pointer location for the based of the object. Since objects are 16-byte aligned in memory, the backward search only looks for the object base addresses divisible by 16. This optimization makes the backward search fast. The search terminates when (1) an AC match occurs (*i.e.,* the correct Object ID and the base address are found), or (2) the search has exceeded the max distance or reached invalid memory, in which case a true temporal memory error is detected.

Our backward search scheme is tested, and works well, on ARMv8.3-A (the latest Cortex-A architecture available today). It is worth noting that the PAC instructions in the future ARMv8.6-A architecture may generate an exception when an authentication fails [25]. Our backward search scheme can work with the exception-enabled PAC instructions by having a tiny kernel patch that masks/disables the corresponding exceptions [20] during the (transient) backward search window. The exception masking code is only callable within the backward search function and thus cannot be abused by attackers. We are unable to evaluate this patch due to the lack of hardware or simulator for ARMv8.6. The rest of the PTAuth design and implementation is compatible with ARMv8.6.

Metadata propagation: Thanks to our in-pointer storage of AC, when a pointer is copied or moved, the metadata of the pointer is automatically propagated without any special handing by PTAuth or any software. As for metadata for objects (*i.e.,* IDs), they are not stored inside objects and thus are not automatically propagated during object duplication or movement. However, this is intended—object metadata should not be propagated when objects are copied or moved. This is because in our points-to authentication scheme, an object ID is assigned to and associated with the allocated buffer, rather than the data stored in that buffer. In contrast, previous works on temporal memory error detection, such as CETS [49], require special handling of metadata propagation at the cost of degraded runtime performance and limited data compatibility.

Handling deallocation: In contrast to pointer dereferencing, where a pointer can point to the middle of an object, for the deallocation procedure, the pointer should always point to the beginning of the object. Otherwise, invalid-free occurs, leading to undefined behaviors and temporal memory errors [8]. Based on this fact, PTAuth only performs one round of AC checking without the backward base address search. If the authentication fails at a deallocation site, it is either a double-free or an invalid-free error. If the authentication succeeds, PTAuth simply sets the object ID to zero (*i.e.,* invalidation) and lets the program execution continue.

Handling reallocation: During reallocation, the base address of an object may or may not change depending on the size of the object and the layout of the memory. PTAuth handles reallocation by instrumenting realloc. If the base of an object has changed, PTAuth nullifies the ID of the old object, generates a new ID, and computes a new AC for the new base pointer. As a result, the existing (stale) pointers to the old object become invalid and cannot be used anymore.

External/uninstrumented Code: During compilation time, PTAuth treats as a blackbox externally linked code or code that cannot be instrumented. This design enables backward and external compatibility. PTAuth instruments the entries to such blackboxes so that immediately before an object or pointer flows into a blackbox (*e.g.,* as an argument to an external function call such as memcpy), PTAuth authenticates the pointer and then strips off its AC, which can be done efficiently

	CETS [49]	DangNull [44]	DangSan [62]	CRCount [58]	PTAuth
Allocation	Generate lock & key	Register pointer	Register pointer	Generate reference counter	Generate ID & AC
Pointer dereference: *p	Comparison of key and lock value	No check	No check	No check	Points-to authentication
Copy ptr arithmetic: p = q+1	Propagate lock address and key	Update register ptr	Update register ptr	Update reference counter	No cost
Deallocation	Invalidate lock	Invalidate pointers	Invalidate pointers	Delayed deallocation	Invalidate ID
Memory overhead	O (# pointers)	O (# pointers)	O (# pointers)	O (# pointers) + Mem leaks	O (# objects)
Metadata handling	Disjoint	Disjoint	Disjoint	Disjoint	Inline
Metadata safety guarantee	×	×	×	×	✔

Table 2: Comparison of our approach with the closely related works, in terms of the use/check, management, and protection of the metadata.

using the `XPAC` instruction. Conversely, when a pointer returns from a blackbox, PTAuth generates the `AC` for it, whose subsequent uses are subject to checks.

3.6 Optimizations

Unnecessary Checks: We optimize the instrumentation strategy by avoiding insertions of unnecessary checks during compilation. The optimization is inspired by the fact that, for any valid pointer, UAF and other temporal memory violations cannot happen through the pointer until it is being freed or later. Therefore, it is not necessary to perform points-to authentication on any use of a pointer that can only take place before the pointer is freed. Obviously, detecting all such pointer uses in a program is an untractable problem [43, 55], which requires perfect alias analysis. However, we can solve this problem within the scope of a function by performing conservative intra-procedural analysis. By tracking a pointer's def-use chain inside a function, we can identify a set of use sites where the pointer and its aliases have not been free or propagated out of the function. PTAuth can safely ignore these use sites during instrumentation (*i.e.,* no runtime check is needed).

```
1  void quantum_gate2 (quantum_reg *reg){
2  int i, j, k, iset;
3  int addsize=0, decsize=0;
4
5  if(reg->num > reg->max )
6        printf("maximum",reg->num);
7
8  else {
9    for(i=0; i<(1 << reg->hashw); i++)
10     reg->hash[i] = 0;
11
12   for(i=0; i<reg->size; i++)
13     quantum_add_hash(reg->node[i].state, i,
           reg);
14     ...
```

Figure 4: Optimization in PTAuth. This example shows that the `reg` pointer is used multiple times in this function. Since the pointer is authenticated before passing to the `quantum_gate2` function, no check on it is needed until Line 13 where the pointer is passed to another function as an argument. Due to the limitation of intra-procedural analysis, we cannot track the pointer into the `quantum_add_hash` function to make sure that it is not being freed. Therefore, After this point, all the temporal checks will be in place.

Figure 4 demonstrates an example of how redundant checks are removed by optimization. In this example, all checks on `reg` up to Line 13 are unnecessary and are omitted by PTAuth. Note that this optimization only works on single-threaded programs. We also extend this optimization to the implementation level. Some frequently used *glibc* functions such as `printf` and `strcpy` never free pointers passed to them as parameters. Therefore, we whitelist such functions and allow the intra-procedural discovery of safe pointer uses to continue beyond such functions.

Global objects: Performing temporal checks on pointers to global objects is also unnecessary because such objects are never deallocated. PTAuth detects those address taken global objects that can be determined statically during the compile-time and remove the checks for them.

3.7 Design Comparison

In Table 2, we compare PTAuth with closely related works in terms of the use/check, management, and protection of the metadata. PTAuth uses inline metadata, which makes the access fast because no heavy lookup is needed. Thanks to the inline metadata, the memory overhead of PTAuth is low and there is no complex handling needed for pointer arithmetics and metadata propagation. PTAuth uses PAC to compute and secure metadata without requiring external spatial safety schemes.

4 Security Analysis

An attacker may attempt to evade PTAuth with the goal of causing temporal memory corruption without being detected. We analyze the possible attacks permitted by our threat model and explain how the design of PTAuth prevents them. Since PTAuth performs load-time authentication and our threat model assumes attackers capable of arbitrarily writing to data memory (*e.g.,* by exploiting certain vulnerabilities), the attacker essentially needs to somehow generate the correct `AC` for the data pointer that she writes before the data is used by the target program or checked by PTAuth. We note that code inject or modification is not allowed under our threat model thanks to DEP and the read-only code region. We identify the following ways that attackers may try to forge the `AC`.

Directly generating AC: One intuitive evasion of PTAuth is to generate the `AC` for the attacker-supplied data, either

offline or dynamically. Offline AC generation does not work because the set of keys used for calculating AC is dynamically generated for each program execution or process and is not static. Alternatively, the attacker may try to directly generate AC on the fly while the target program is running. This is impossible either because the PAC keys are stored in the special CPU register and not accessible from the user space, even if the attacker has the arbitrary memory read capability. Moreover, the attacker cannot inject code and thus cannot directly calculate AC using injected PAC instructions. Also, brute-force is not applicable in this context because one wrong guess can lead to a crash of the process.

Reuse PAC instructions: The attacker's next possible move could be to reuse the existing PAC instructions already loaded in the memory (*e.g.,* those used by PTAuth) for calculating AC on injected data. However, our system can easily get merged with the standard use of PAC for protecting code pointers as well. Therefore, code reuse attacks are prevented thanks to the code pointer integrity check by PAC (*i.e.,* any corrupted return addresses or call/jump targets trigger authentication failures and are detected before the program control flow is hijacked).

ID spray: Another possible attack vector is spraying the ID into the object to misguide the dangling pointers that are pointing to the middle of object. The design of PTAuth considers this attack. Since the AC is bound to the beginning address of an object, even if the correct ID is found in the middle of object, the authentication will fail.

5 System Implementation

We built a prototype for the PTAuth system, including (*i*) a customized compiler for instrumenting and building PTAuth-enabled programs, (*ii*) a runtime library, linked to instrumented programs, for performing dynamic AC generation and authentication, and (*iii*) a set of bootloader and Linux patches necessary for configuring the CPU and enabling the PAC feature [15, 19]. All the system components are implemented in C/C++ with a small set of inline assemblies that directly use the PAC instructions. The PTAuth LLVM pass is approximately 2K lines of C++ code and the runtime library is 1K lines of C code. The current implementation supports C programs. It is based on `ptmalloc` memory allocator from *glibc*. It supports all common memory allocation APIs, such as `malloc`, `calloc`, `realloc` and `free`.

Customized Compiler: Our compiler is based on LLVM 6.0, which already has basic assembler and disassembler support for PAC on ARMv8.3-A. We built the code analysis and instrumentation logic (§3.5) into an LLVM transform pass. It operates on the LLVM bitcode IR. At each instrumentation site, such as pointer `load` and `store`, it inserts a call, based on the type of the instrumented instruction, to the PTAuth runtime library.

Runtime Library: The runtime AC checking logic is built into a dynamically linkable library. It exposes the call gates for the instrumented code to invoke the AC generation and authentication routines. These routines calculate or check AC for different scenarios, as describe in §3.4 and §3.5. The library does not maintain any data internally thanks to the in-place storage of AC and the OS-managed PAC keys. Therefore, no data inside the library needs to be protected or verified. However, we do enable code pointer integrity checking using PAC when compiling the library, which ensures that no control flow hijacking can occur while the library code is running.

OS and bootloader patches: By default, PAC instructions (except for `PACGA` and `XPAC`) are disabled. According to the ARMv8 reference manual [18], to use all PAC instructions and the corresponding key slots, the OS needs to set to 1 the `EnIA, EnIB, EnDA, EnDB` fields in the `SCTLR_EL1` register. Additionally, the `SCR_EL3.APK` and `SCR_EL3.API` registers need to be set to 1 during the system booting stage. These configurations are necessary to fully enable the PAC hardware extension. The OS also needs to generate and manage PAC keys for each process (only OS or code running at EL1 is allowed to manage PAC keys). We implemented these configurations and tasks via two small patches to the bootloader [19] and the Linux kernel. These small patches do not interfere with any bootloader or OS functionalities because (*i*) the configured register fields are reserved exclusively for PAC, and (*ii*) the added PAC key management routine does not interact with the rest of the OS.

We built the patched bootloader and kernel into a system image, which was then installed on the ARMv8.3-A FVP. As discussed in §2.3, FVP is the functional-accurate whole-system simulator for ARM architectures, which emulates processors, memory, and peripherals. We used this prototype and environment for evaluating PTAuth.

6 Evaluation

In this section, we evaluate the prototype of PTAuth in terms of security, runtime overhead and memory overhead. The security evaluation (§6.2) was conducted on the ARM FVP simulator. The performance evaluation (§6.3) was performed on a Raspberry Pi 4 with ARMv8-A Cortex A53 processor (1.5GHz) and 4GB memory, running Gentoo 64-bit Linux (v4.19). We explain the rationale behind this setup in §6.1.

Our experiments aim to show: (*i*) whether PTAuth detects temporal memory corruptions such use-after-free, double-free and invalid-free; (*ii*) how much performance overhead PTAuth incurs during runtime; (*iii*) how much memory overhead PTAuth incurs during runtime. We used Juliet test suite [33] and four real CVEs for security experiments. To evaluate the runtime and space overhead, we used SPEC CPU2006.

```
1  long MASKBITS = 0b000...000111111111111111;
2  void* __pacia(void* ptr,int id){
3      long ptrbits = (unsigned long)ptr &
           MASKBITS;
4      long idbits = id & MASKBITS;
5      long signature = ptrbits ^ idbits;
6      signature = signature << 48;
7      unsigned long ptrWithSign = (unsigned
           long)ptr | signature;
8      return (void*)ptrWithSign;
9  }
```

Figure 5: Software implementation of PACIA instruction as a function.

```
1  #if PACENABLED
2      asm (
3      "mov %x0,%0\n"
4      :
5      : "r" (ptr));
6      asm(
7      "pacia %x0, %x1\n"
8      : "=r" (ptr)
9      : "r" (id));
10 #else
11     ptr =__pacia(ptr,id);
12 #endif
```

Figure 6: When the PACENABLED flag is enabled during the compile-time, actual PAC instructions are generated for the final binary. Otherwise, software implementation of the corresponding instructions is invoked. This implementation helps to test the design on an SoC that does not support the ARMv8.3 instruction set.

Vulnerability	CWE Cat.	# of Prog.	PTAuth	PAC / PARTS [45]
Double-Free	415	50	●	○
Use-After-Free	416	50	●	○
Invalid-Free	761	50	●	○

Table 3: Selection of 150 vulnerable programs from the Juliet Test Suite and detection results.

6.1 Experiment Setup and Methodology

We performed the security evaluation (§6.2) on the FVP simulator that supports PAC. At the time of writing, no publicly available development board supports ARMv8.3 or PAC instructions. Although Apple's A12 Bionic SoC supports PAC instructions, it is a proprietary implementation and we were not able to instrument and run the benchmarks on top of that. We patched the bootloader and OS in the FVP image as described in §5.

We conducted the performance evaluation (§6.3) on a Raspberry Pi 4 (ARMv8-A Cortex A53), rather than FVP. This change of platform is necessary because the benchmarks (SPEC CPU2006) are too heavy to run on the FVP—they often crash or halt the simulator. To allow PTAuth to run on the Raspberry Pi, which does not support PAC, we implemented in software the three PAC instructions used by PTAuth, namely PACIA, AUTIA, and XPAC. The input/output syntax of these functions is identical to that of the original PAC instructions. Figure 5 demonstrates the implementation of PACIA instruction as a C function. The other PAC instructions are implemented based on PACIA. It is worth noting that our software PAC implementation does not contain the exact cryptographic algorithm (*QARMA*) used in PAC instructions. This is because a software implementation of *QARMA* would be much slower than the hardware implementation and thus make it difficult to measure the real performance overhead caused by PTAuth. Instead, we chose a simple encryption and AC computation, keeping the overhead comparable to hardware-based encryption and allowing the performance evaluation to focus on the overhead of PTAuth itself.

Our implementation of PTAuth uses a compile-time flag to indicate whether the compiled binary should use software-emulated PAC or hardware-based PAC instructions. Figure 6 shows an example of the inline assembly.

6.2 Security Evaluation

The security evaluation is a functional test and serves two purposes: (1) testing PTAuth's compatibility with the underlying hardware feature, namely PAC, and (2) testing PTAuth's detection of temporal memory corruptions and its robustness against evasions. Similar to the previous work [45], we chose the ARM FVP simulator for this functional test because no development board with PAC extension exists at the moment and FVP includes ARM's official and fully functional PAC simulation.

We performed the security evaluation using 150 C programs selected from the NIST Juliet test suite [33]. We chose the Juliet Suite for two reasons. First, it is the largest of its kind and contains both vulnerable and non-vulnerable versions of programs. The vulnerable programs, covering the common types of temporal memory corruptions, are ideal for our security evaluation. We used the non-vulnerable/patched counterparts for a compatibility test. Second, unlike the generic CPU benchmarks, the test programs in Juliet were made for security testing without being computationally demanding. They run smoothly on FVP, which is a whole-system (slow) simulator and cannot run computation-intensive programs without halting or crashing. Therefore, using the Juliet programs allows us to focus on evaluating PTAuth in terms of security while avoiding high computation loads that FVP cannot handle. We conducted a separate performance evaluation of PTAuth (§6.3) using much demanding CPU benchmarks.

The 150 Juliet tests include double-free, use-after-free and invalid-free bugs. Table 3 shows the CWE (Common Weakness Enumeration) categories and the number of Juliet tests selected in each vulnerability category. When running with PTAuth enabled, the vulnerable programs all terminated immediately before the bugs were triggered. We also ran the non-vulnerable/patched version of the test programs with PTAuth enabled. All these programs finished properly with-

Application	CVE	Vulnerability Type	Detection
libpng	CVE-2019-7317	UAF	✓
sqllite	CVE-2019-5018	UAF	✓
curl	CVE-2019-5481	DF	✓
libgd	CVE-2019-6978	DF	✓

Table 4: Effectiveness of PTAuth for detecting real-world vulnerabilities.

```
1  if (result != 0)
2  {
3    image->opaque->error_buf = safe_jmpbuf;
4  // calling png_image_free_function() indirectly
5    result = function(arg);
6  }
7  image->opaque->error_buf = saved_error_buf;
```

Figure 7: In CVE-2019-7317, the `png_image_free_function` is called indirectly and the *image* pointer is passed to it as an argument. In this case, the image pointer is freed in the caller and then used in line 7, which is UAF error.

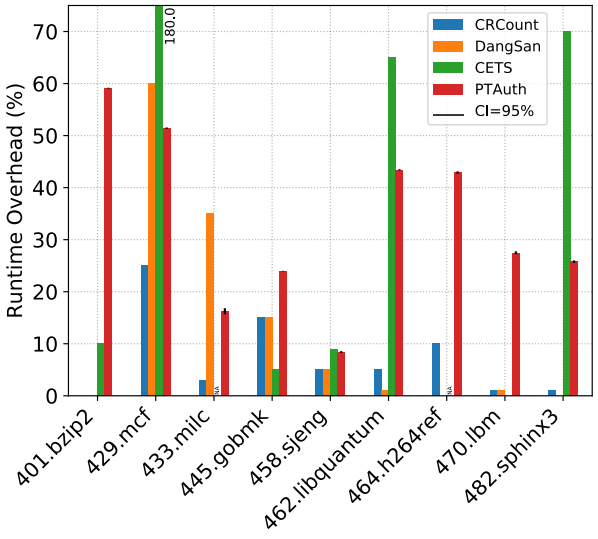

Figure 8: Runtime overhead (95% confidence interval) on SPEC CPU2006 and comparison with CRCount, DangSan and CETS.

out any crash or halt. The result shows that PTAuth achieved a 100% detection accuracy and did not cause any compatibility issues: it did not miss a single temporal memory corruption in any category; it did not alert or crash the programs when no temporal memory corruption was triggered.

The right-half of Table 3 shows a comparison between PTAuth and PAC/PARTS [45]. PAC and PARTS were not designed for detecting temporal memory corruptions and therefore cannot detect any. This comparison underlines our novel use of PAC for addressing a critical security vulnerability class, which is not considered or detectable by the original design of PAC or previous work using PAC.

Case study of real-world vulnerabilities: Besides the Juliet test programs, we also surveyed four recent temporal memory corruption vulnerabilities in real software (Table 4). Since FVP cannot run these entire programs, we performed a manual analysis and verified that PTAuth can prevent the exploitations of all these vulnerabilities. For instance, To exploit CVE-2019-5481, an attacker sends a crafted request, which `realloc()` fails to handle. On the exit path, the pointer is freed. During the cleaning phase, the pointer is freed one more time. Since these two steps are far apart, programmers can easily miss the bug. When PTAuth is enabled, the ID of the object is changed after the first free and thus causes an authentication failure when the second free is about to happen.

Take CVE-2019-7317 as another example, shown in Figure 7. Line 5 indirectly calls `png_image_free_function`, which frees the memory referenced by `arg`, an alias of `image`. Later, Line 7 dereferences `image`, resulting in use-after-free. This bug can be extremely difficult to discover either manually or using analysis tools, due to the layers of function and object aliasing. PTAuth handles aliasing naturally thanks to its points-to authentication scheme. PTAuth can catch this bug right before it is triggered due to the `AC` mismatch.

Robustness evaluation: We created a small set of programs that contain both temporal and spatial memory corruptions to evaluate the robustness of PTAuth. This scenario is analogous to the real-world attacks where a powerful attacker can exploit arbitrary memory write and temporal memory corruptions. We selected 30 programs from Juliet test suite in 3 different categories. Then, we injected memory overwrite vulnerabilities such as buffer-overflow to them, which allow an attacker to overwrite the PTAuth metadata. Such an attacker can bypass previous protections, such as CETS, which simply compare the plain ID of the key and object. However, the attacker cannot bypass PTAuth because she cannot generate valid `AC` without knowing the secret PAC key. As expected, we triggered those vulnerabilities and PTAuth detected all.

6.3 Performance Evaluation

We had to switch from the FVP simulator to a Raspberry Pi 4 (ARMv8-A Cortex A53) for conducting the performance evaluation because the simulator could not run computation-intensive benchmarks. Due to the lack of hardware support for PAC on the Raspberry Pi SoC, we used our own software implementation of PAC in this evaluation. This experiment provides an upper bound of the performance overhead of PTAuth (*i.e.,* the overhead should be lower on devices with hardware PAC support).

We tested PTAuth on the SPEC CPU2006 benchmarks. They are appropriate for the performance evaluation since they are memory- and CPU-intensive. Figure 8 shows the runtime overhead of PTAuth with 95% confidence interval. The interval bars are barely visible in the figure due to the relatively stable results. Figure 13 in Appendix A shows more

clearly, for each benchmark, the concentrated distribution and the narrow standard deviation of the measured overhead. The overhead varies across different benchmarks because the use of data pointers and dynamically allocated objects in some benchmarks is more prevalent than in other benchmarks. For instance, although *mcf* is a small program, it uses many data pointers and requires more `AC` checks than other benchmarks.

We compared PTAuth with the closely related works, including CRCount [58], DangSan [62], and CETS [49]. These prior techniques are either based on pointer invalidation or object (lock) invalidation. In our comparison, we skipped DangNull [44] because DangSan outperforms it. Since the source code of CRCount is not available and DangSan and CETS are not compatible with the ARM architecture, we used the reported numbers in the papers for comparison. The nine C benchmarks in Figure 8 were selected because they are both compatible with our current implementation and were used in the previous works. We note that the compared papers did not use the same set of benchmarks in their evaluation. Some of them did not report the performance numbers for all nine benchmarks. For example, CETS was not evaluated on `433.milc` and `464.h264ref`.

The geometric mean overhead of PTAuth on all benchmarks is 26%. The number around 5% for CRCount, 1% for DangSan, and 10% for CETS. Although PTAuth appears to incur much higher overhead than the others, we note that this comparison is not completely fair because, unlike PTAuth, the other systems require external protection of their metadata (*e.g.,* bound checkers), which incurs additional overhead not captured in this comparison.

For this reason, we conducted another experiment, where we added the reported runtime overhead of SoftBound [48] to the overhead of DangSan, CRCount and CETS. This combined overhead represents what these systems would incur when they are deployed with the required external protection and made as robust as PTAuth. The results are shown in Figure 9. Only three of the nine benchmarks were tested in [48] and thus were included in this comparison. Clearly, PTAuth incurs much less overhead than the other systems on two out of the three benchmarks.

Statistical significance: To prove that our performance result is statistically significant, we perform the following hypothesis testing and show that the performance overhead of PTAuth has a statistically significant upper bound at 42% (with a P-value under 0.05). We construct the *Null Hypothesis* that "the runtime overhead of PTAuth is **not** below 42%". We show below that this Null Hypothesis can be **rejected**. We calculate the Z-score $= \frac{M-\mu}{\frac{\sigma}{\sqrt{n}}}$, where M is the measured average runtime overhead of PTAuth (i.e., 33.2%), σ is the standard deviation of the measured overheads (i.e., 0.159), n is the sample size (i.e., 9), and μ is the overhead bound stated in the Null Hypothesis (i.e., 42%). The calculated Z-score is -1.66038. Its corresponding P-value is 0.04846, which is below the widely

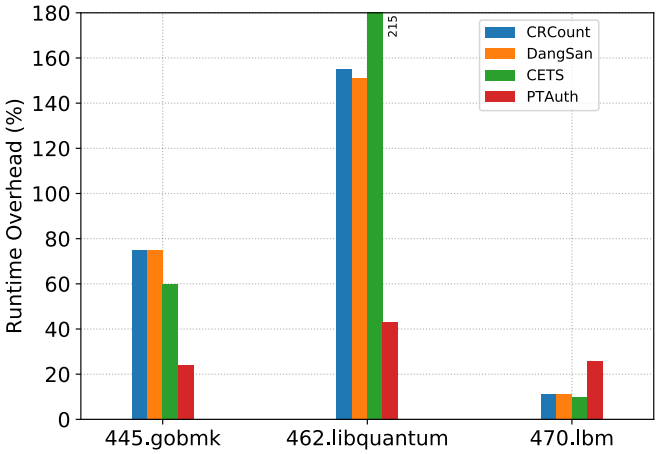

Figure 9: Runtime overhead of temporal memory corruption detectors when they are combined with SoftBound to protect their metadata. PTAuth is a stand-alone system and does not need an external system to protect its metadata.

accepted significance level of 0.05. The result of this hypothesis testing shows that a statistically significant upper bound for PTAuth's overhead can be established at 42%.

Backward search overhead: Backward search incurs the worst-case runtime overhead when many large memory objects exist with many sub-objects referenced directly by pointers. However, in practice, we observed that this worst-case scenario is quite rare and the overhead of backward search is generally low. Figure 10 shows the overhead caused by backward search in each benchmark, as part of the overall overhead. The main reason for the low overhead of backward search is that most large objects are of `struct` type. The fields/sub-objects of the large objects are often accessed via an index from the beginning of the large objects, rather than direct or calculated pointers to the middle of the objects. Therefore, no backward search is needed for those accesses to fields or sub-objects. Also, pointer arithmetics is not frequently used in the benchmarks and regular programs. However, `401.bzip2` and `462.libquantum` contain more pointer arithmetic operations than other benchmarks. Furthermore, our optimization (§3.6) removes some unnecessary checks on pointers.

Overhead under fixed-cycle PAC emulation: In addition to evaluating PTAuth's performance using our software implementation of PAC, we conducted another experiment to estimate the performance of PTAuth running on future hardware with PAC support. This additional experiment was inspired by PARTS [45], where we used equal-cycle `NOP` instructions to replace PAC instructions (assuming each PAC instruction takes 4 CPU cycles as per [45]). Unlike PARTS, our system cannot fully function when PAC instructions are simply replaced with NOPs. This is because the backward search technique requires correct `AC` produced by PAC instructions. Therefore, in this experiment, we disabled the backward search and the corresponding checks on sub-objects. This experiment is only

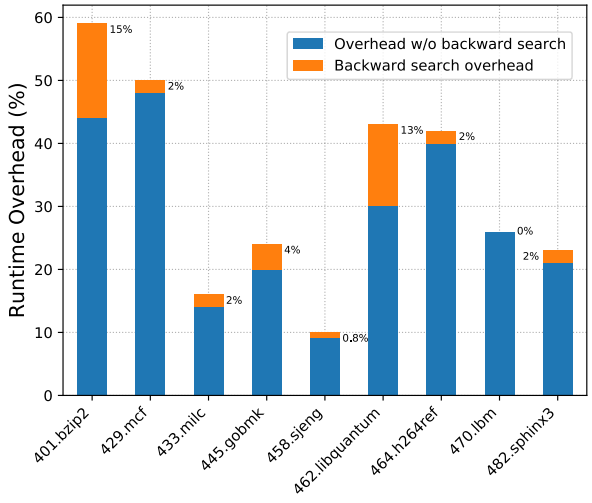

Figure 10: The overhead of backward search in each benchmark.

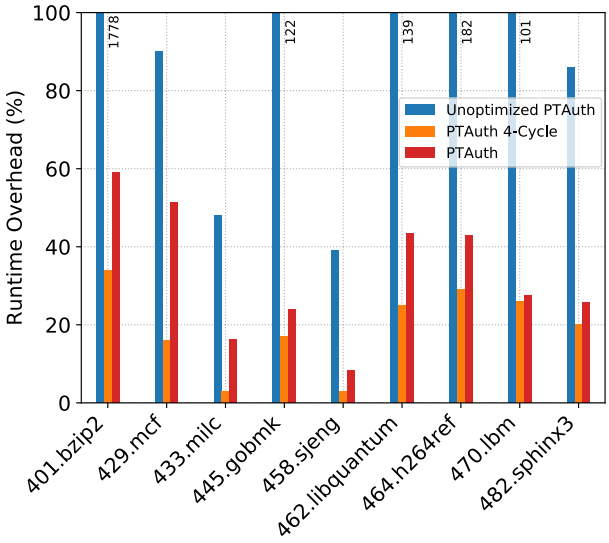

Figure 11: The overhead of PTAuth with 4 CPU cycles and non-optimized implementation

meant to complement the main performance evaluation (the one using the software-based PAC). Figure 11 shows the runtime overhead on the benchmarks assuming each PAC instruction takes exact 4 cycles. Base on this result, we expect the runtime overhead of PTAuth to decrease to 20% when PTAuth runs on devices that support hardware PAC.

Optimization benefit: In order to measure the effectiveness of the optimization described in §3.6, we disabled the optimization during the instrumentation and conducted the experiment again. Figure 11 shows the benefits of the optimization. As expected, the intra-procedural analysis for eliminating unnecessary checks reduces, on average, 72.8% of the overhead.

Memory overhead: In our design, pointer metadata is stored in the unused bits of a pointer and requires no extra memory. The only source of PTAuth's memory overhead is the extra 8-

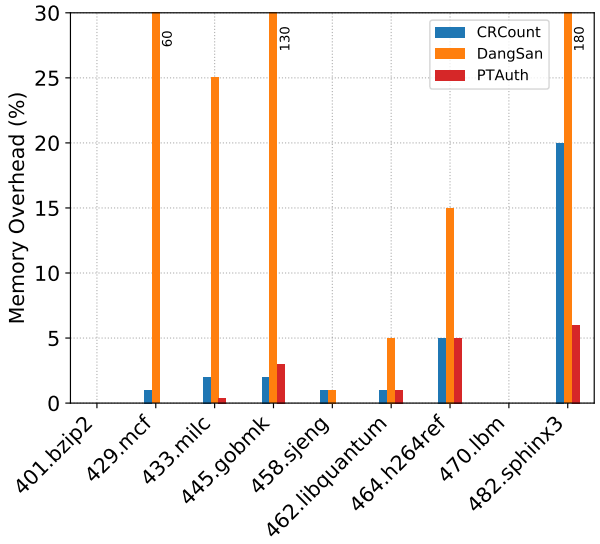

Figure 12: Memory overhead on SPEC CPU2006 and comparison with CRCount and DangSan. CETS have not reported any memory overhead.

byte memory allocated for storing each object ID. We reduce this memory overhead based on the following observation. The `ptmalloc` allocator in the *glibc* of Linux appends extra paddings to objects when object sizes are not 32-byte aligned. For instance, when a 16-byte object is allocated, it is padded to 32 bytes. For objects with such paddings, PTAuth makes use of the padding bytes for storing object IDs, without requiring additional memory space.

To evaluate the memory overhead of PTAuth, we measured the maximum resident set size (Max RSS) of the instrumented SPEC CPU2006 benchmarks. Max RSS (*i.e.,* the peak physical memory allocation for a process) is the metric used by related works. We adopted the same metric to perform a fair comparison. Figure 12 illustrates the memory overhead caused by PTAuth, DangSan, and CRCount on each benchmark. The geometric mean of PTAuth's memory overhead is 2%. This number is 2% for CRCount and 15% for DangSan. CETS did not report memory overhead.

In addition to maximum RSS, we also measured the memory overhead of PTAuth in terms of mean RSS, a metric that captures the memory usage throughout the entire process execution, as opposed to the peak usage at one moment. We calculated mean RSS by taking RSS every five seconds during the lifetime of a process and then computing the mean. The mean RSS overhead of PTAuth is 1%. The related works did not report this number and thus cannot be compared with in this regard.

7 Discussion and Limitation

Multi-threading: Similar to previous works, the current PTAuth prototype does not support multi-threaded programs, mostly due to implementation-level simplifications. To make

PTAuth work on multi-threaded programs, each memory (de)allocation and the resulting metadata updating operation need to be atomic, or the metadata protected by a lock. Without the atomicity or synchronization, PTAuth's metatdata may become stale or invalid when a race condition occurs, leading to missed or falsely detected temporal memory errors. It is worth noting that the unnecessary check removal (one optimization discussed in §3.6) is not threading-safe and needs to be disabled on multi-threaded programs.

Stack use-after-free: Since the deallocation of stack objects is implicitly triggered by function returns, double-free bugs cannot happen to stack objects. However, use-after-free bugs on stack objects, though uncommon, may happen when the address of a stack object is taken and stored in a global variable that is later mistakenly freed. The current design of PTAuth is focused on detecting heap-based temporal errors, which are more prevalent and critical than stack-based use-after-free. In theory, PTAuth can be extended to detect the latter as follows.

We refer to stack objects referenced by global pointers as address-taken objects (*i.e.,* stack objects potentially vulnerable to use-after-free). Since object allocation and deallocation on the stack are different from those on the heap, protecting address-taken objects require special treatment. First, PTAuth needs to identify address-taken objects in stacks via a simple intra-procedural data-flow analysis. Then, PTAuth needs to allocate extra 8 bytes at the beginning of each address-taken object. This extra header is used for storing the metadata, which is initialized upon the creation of the stack frame (*i.e.,* in the prologue of the corresponding function). PTAuth also needs to instrument address-taken operations on stack objects to generate `AC` for the resulting pointers. The authentication scheme for address-taken objects on stacks and their pointers is the same as the scheme for heap objects and pointers. Finally, PTAuth needs to invalidate all metadata of address-taken objects in a function epilogue, similar to what it does upon heap object deallocations.

PAC instructions: In the current implementation, we have used both PAC instructions and the software emulated implementation of the instructions. We used the FVP simulator to run the PAC instruction. However, FVP is not a performance aware simulator. It does not model cycle timing and all the instructions are executed in one processor clock cycle [21]. We also observed that large benchmarks halt the FVP which prevented us from running performance experiments on it. Since A12 Soc is proprietary and there is no public SoC available to test the implementation, the reported runtime overhead is anticipated to be different in real hardware. In other words, the actual PAC instructions are expected to be faster than the software emulated instructions.

We leave these limitations for future work when the real hardware is available.

8 Related work

Safe C: Memory corruption bugs are highly diverse and commonly targeted by software attacks [61]. Prior work introduced memory safety to the C language via a safe type system [39, 42, 50, 60]. These safe languages are immune to temporal vulnerabilities. However, they either impose a significant amount of memory and runtime overhead or they are not applicable to protect legacy C/C++ codes. For instance, Cyclone [42] is a safe dialect of C which is is not applicable to protect legacy codes. It is no longer supported but several ideas of Cyclone have been implemented in Rust [2, 6]. CCured needs some annotations by the programmer. It also uses fat pointers to store metadata which breaks the application binary interface (ABI).

Safe memory allocator: These systems prevent allocated objects from ending up at the same address of freed objects [29,31,52,59]. For instance, DieHard [31] and DieHarder [52] randomize the locations of allocated objects in the heap and consequently provides probabilistic temporal memory safety (*i.e.,* making object reuse or replacement difficult). PartitionAlloc [5] and Internet Explorer isolated heap [7] allocators prevent memory reuse by allocating objects of different types or sizes in separate buckets. Although these schemes have low runtime overhead, it has been shown that they can be bypassed on targeted attacks [11,41]. Moreover, they suffer from a huge memory overhead caused by memory fragmentation.

Memory error detectors: Memory error detectors [34, 51, 56] are widely used among developers. However, due to the high overhead, they are only suitable for debugging or non-production use. AddressSanitizer [56] is a memory error detector that creates shadow memory and red zones around objects. It detects out-of-bounds accesses in heap, stack, and global objects, as well as use-after-free bugs. However, it provides a probabilistic detection system for use-after-free bugs which is susceptible to bypass [63].

Pointer invalidation: Another line of work focused on pointer invalidation. DangNull [44], DangSan [62], FreeSentry [65] and pSweeper [46] explicitly invalidate all the pointers to an object when the lifetime of the object is finished. CRCount [58] uses a reference counting approach for counting the number of pointers to an object. When there is no pointer to an object, then it is freed. In this approach, the pointers are invalidated implicitly during the runtime of the program. This approach suffers from memory leak issue since some pointers are never invalidated. Consequently, the objects will reside in memory for a long time. In general, pointer invalidation systems need to keep a huge amount of metadata in the memory to track the relationship between pointers and objects. Inevitably, those metadata are prone to corruption.

Pointer dereference validation: Some other approaches similar to our design, detect and prevent temporal corruption bugs by pointer dereference validation [35, 49, 64]. CETS [49] pro-

vides temporal safety by assigning a unique identifier to each object and its pointers. The main challenge in this scheme is that extra metadata for the pointers should be stored in the memory. Also, a unique identifier should be assigned to each object and its pointers. Since these metadata are stored disjointly, obtaining these information efficiently during the runtime is challenging. In order to tackle this problem, in our design, we proposed an inline metadata scheme for both pointers and objects. However, inline metadata is prone to corruption by linear overflow. To address this problem, we used PAC to guarantee the integrity of the metadata before using them. To sum up, our approach reduces the high look-up table costs for loading the metadata and provides integrity of the metadata in a unified design.

Hardware-assisted schemes: Similar to PTAuth, there are some approaches that take advantage of hardware to provide temporal safety. Oscar [37], which is the following work of [38], is a page permission-based scheme to prevent temporal memory safety violations in the heap. Basically, Oscar improves the original idea of allocating each object in a separate page (similar to PageHeap and Electric Fence [3,4]) to prevent UAF vulnerabilities.

Another line of work relies on hardware to provide spatial and temporal protections. Hardware-assisted AddressSanitizer (HWASAN) [14,57] is the following work of AddressSanitizer. HWASAN uses address tagging feature [1] to implement a memory safety tool, similar to AddressSanitizer. Memory Tagging Extension (MTE) [17] has been introduced in ARMv8.5 for providing spatial and temporal safety. However, the hardware is not available yet. Intel MPX [53] was introduced by Intel to provide spatial safety. However, due to the high-overhead, it was discontinued by the maintainers.

9 Conclusion

We presented a resilient and efficient points-to authentication scheme called PTAuth, for detecting temporal memory corruptions. By defining the authentication codes (`AC`) for pointers, our scheme allows for convenient and simultaneous checking of metadata integrity and identities when they are being accessed. The unified verification of the two properties (integrity and identity) enables the unified detection of all kinds of temporal memory corruptions in the heap. PTAuth uses PAC on ARMv8.3-A as a basic encryption/signing primitive during `AC` calculation, which is fast and secure thanks to the hardware-level support. PTAuth contains: (*i*) a customized compiler for instrumenting programs with necessary inline checks, (*ii*) a runtime library for `AC` generation and authentication, and (*iii*) a set of OS patches for PAC-related CPU configuration. Our evaluation on 150 vulnerable programs shows that PTAuth detects all 3 categories of temporal memory corruptions with a runtime overhead of 26% (using software-based PAC) and 2% memory overhead.

Acknowledgment

The authors would like to thank the anonymous reviewers for their help with the revision of this paper. This project was supported by the Office of Naval Research (Grant#: N00014-18-1-2043 and N00014-17-1-2891) and the Army Research Office (Grant#: W911NF-18-1-0093). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Arm Cortex-A Series Programmer's Guide for Armv8-A. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/ch12s05s01.html`.

[2] Cyclone is a safe dialect of C. `https://cyclone.thelanguage.org/`.

[3] Electric Fence. `https://elinux.org/index.php?title=Electric_Fence&oldid=369914`.

[4] Microsoft. GFlags and PageHeap. `https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap`.

[5] PartitionAlloc Design. `https://chromium.googlesource.com/chromium/src/+/master/base/allocator/partition_allocator/PartitionAlloc.md`.

[6] Rust. `https://www.rust-lang.org/`.

[7] Understanding IE's New Exploit Mitigations: The Memory Protector and the Isolated Heap. `https://securityintelligence.com/understanding-ies-new-exploit-mitigations-the-memory-protector-and-the-isolated-heap/`.

[8] ISO/IEC 9899 - Programming languages - C. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`, 2005.

[9] ASLR Bypass Apocalypse in Recent Zero-Day Exploits. `https://www.fireeye.com/blog/threat-research/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html`, 2013.

[10] Armv8-A architecture: 2016 additions. `https://community.arm.com/processors/b/blog/posts/armv8-a-architecture-2016-additions`, 2016.

[11] Life After the Isolated Heap. https://googleprojectzero.blogspot.com/2016/03/life-after-isolated-heap.html, 2016.

[12] PAC:Pointer Authentication Code. https://community.arm.com/groups/processors/blog/2016/10/27/armv8-a-architecture-2016-additions, 2016.

[13] Technical Analysis of the Pegasus Exploits on iOS. https://info.lookout.com/rs/051-ESQ-475/images/pegasus-exploits-technical-details.pdf, 2016.

[14] Hardware-assisted AddressSanitizer Design. https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html, 2017.

[15] Linux kernel patch for PAC instructions. https://lore.kernel.org/lkml/1491232765-32501-1-git-send-email-mark.rutland@arm.com/T/#u, 2017.

[16] Pointer Authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, 2017.

[17] Arm A-Profile Architecture Developments 2018: Armv8.5-A. https://community.arm.com/processors/b/blog/posts/arm-a-profile-architecture-2018-developments-armv85a, 2018.

[18] ARM Architecture Reference Manual ARMv8 for ARMv8-A architecture profile. https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf, 2018.

[19] Boot wrapper for Aarch64. https://git.kernel.org/pub/scm/linux/kernel/git/mark/boot-wrapper-aarch64.git/commit/, 2018.

[20] Exception mask registers. https://developer.arm.com/docs/100688/0100/armv8-m-architecture-technical-overview/programmers-model/exception-mask-registers, 2018.

[21] Fast Models Reference Manual Version 10.2. https://developer.arm.com/docs/dui0834/j/versatile-express-model/differences-between-the-ve-hardware-and-the-system-model/restrictions-on-the-processor-models, 2018.

[22] Fixed Virtual Platforms. https://developer.arm.com/products/system-design/fixed-virtual-platforms, 2018.

[23] iOS Security, iOS 12.1. https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf, 2018.

[24] LLVM Project. https://llvm.org/, 2018.

[25] Developments in the Arm A-Profile Architecture: Armv8.6-A. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-architecture-developments-armv8-6-a, 2019.

[26] SockPuppet: A Walkthrough of a Kernel Exploit for iOS 12.4. https://googleprojectzero.blogspot.com/2019/12/sockpuppet-walkthrough-of-kernel.html, 2019.

[27] A survey of recent iOS kernel exploits. https://googleprojectzero.blogspot.com/2020/06/a-survey-of-recent-ios-kernel-exploits.html, 2020.

[28] Memory safety in the Chromium project. https://www.chromium.org/Home/chromium-security/memory-safety, 2020.

[29] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, pages 177–192, 2010.

[30] Roberto Avanzi. The qarma block cipher family. *IACR Transactions on Symmetric Cryptology*, 2017(1):4–44, 2017.

[31] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Acm sigplan notices*, volume 41, pages 158–168. ACM, 2006.

[32] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*, pages 227–242. IEEE, 2014.

[33] Paul E Black. Juliet 1.3 test suite: Changes from 1.2. Technical report, 2018.

[34] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223. IEEE Computer Society, 2011.

[35] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. Cup: Comprehensive user-space protection for c/c++. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 381–392. ACM, 2018.

[36] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143. ACM, 2012.

[37] Thurston HY Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 815–832, 2017.

[38] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 269–280. IEEE, 2006.

[39] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: enforcing alias analysis for weakly typed languages. In *ACM SIGPLAN Notices*, volume 41, pages 144–157. ACM, 2006.

[40] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. How the ELF ruined christmas. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 643–658, 2015.

[41] Abdul-Aziz Hariri, Simon Zuckerbraun, and Brian Gorenc. Abusing silent mitigations. *BlackHat USA*, 2015.

[42] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[43] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[44] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[45] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, 2019.

[46] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1635–1648. ACM, 2018.

[47] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Everything you want to know about pointer-based checking. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[48] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[49] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*, volume 45, pages 31–40. ACM, 2010.

[50] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[51] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[52] Gene Novark and Emery D Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

[53] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. In *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '18, pages 111–112, New York, NY, USA, 2018. ACM.

[54] Phantasmal Phantasmagoria. The Malloc Maleficarum. `https://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt`, 2005.

[55] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[56] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[57] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory tagging and how it improves c/c++ memory safety. *arXiv preprint arXiv:1802.09517*, 2018.

[58] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. Crcount: Pointer invalidation with reference counting to mitigate use-after-free in legacy c/c++. In *NDSS*, 2019.

[59] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2389–2403. ACM, 2017.

[60] Matthew S Simpson and Rajeev K Barua. Memsafe: ensuring the spatial and temporal memory safety of c at runtime. *Software: Practice and Experience*, 43(1):93–128, 2013.

[61] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[62] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 405–419. ACM, 2017.

[63] Eric Wimberley. Bypassing addresssanitizer. `https://dl.packetstormsecurity.net/papers/general/BreakingAddressSanitizer.pdf`, 2013.

[64] Suan Hsi Yong and Susan Horwitz. Protecting c programs from attacks via invalid pointer dereferences. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 307–316. ACM, 2003.

[65] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.

Appendix A Distribution of Runtime Overheads

Figure 13 shows the distribution of runtime overheads obtained by running PTAuth 10 times on all the benchmarks. The green triangles indicate the mean and the red numbers are the standard deviation. Note that the box plots do not use the same scale because the overhead varies significantly across the benchmarks. The standard deviation on all benchmarks are fairly low (i.e., less than 0.6%), indicating that the overhead values distribute closely around the mean. This result confirms that the overhead result is reliable.

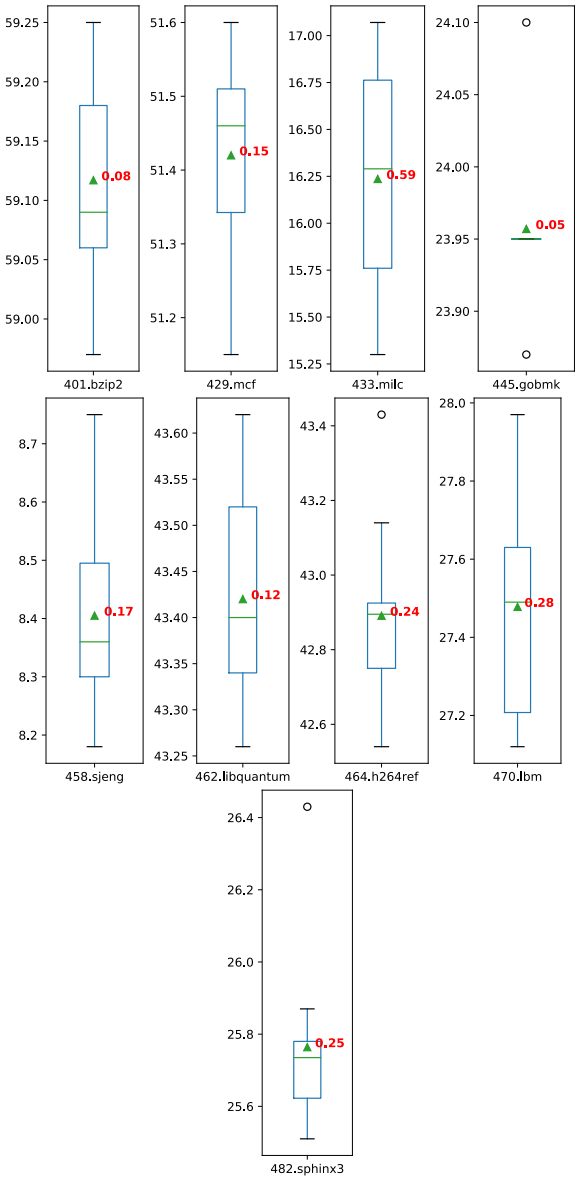

Figure 13: Distribution of PTAuth's runtime overhead on different benchmarks. Per-benchmark scales are used to clearly show the overhead distribution on each individual benchmarks.