PACSafe: Leveraging ARM Pointer Authentication for Memory Safety in C/C++

Konrad Hohentanner
Fraunhofer AISEC, Germany

Philipp Zieris
Fraunhofer AISEC, Germany

Julian Horsch
Fraunhofer AISEC, Germany

Abstract

Memory safety bugs remain in the top ranks of security vulnerabilities, even after decades of research on their detection and prevention. Various mitigations have been proposed for C/C++, ranging from language dialects to instrumentation. Among these, compiler-based instrumentation is particularly promising, not requiring manual code modifications and being able to achieve precise memory safety. Unfortunately, existing compiler-based solutions compromise in many areas, including performance but also usability and memory safety guarantees. New developments in hardware can help improve performance and security of compiler-based memory safety. ARM Pointer Authentication, added in the ARMv8.3 architecture, is intended to enable hardware-assisted Control Flow Integrity (CFI). But since its operations are relatively generic, it also enables other, more comprehensive hardware-supported runtime integrity approaches. As such, we propose PACSafe, a memory safety approach based on ARM Pointer Authentication. PACSafe uses pointer signatures to retrofit full memory safety to C/C++ programs, protecting heap, stack, and globals against temporal and spatial vulnerabilities. We present a full, LLVM-based prototype implementation, running on an M1 MacBook Pro, i.e., on actual ARMv8.3 hardware. Our prototype evaluation shows that the system outperforms similar approaches under real-world conditions. This, together with its compatibility with uninstrumented libraries and cryptographic protection against attacks on metadata, makes PACSafe a viable solution for retrofitting memory safety to C/C++ programs.

1 Introduction

Despite presenting an old and well-known problem, vulnerabilities caused by unsafe programming languages are still omnipresent. In memory-unsafe languages such as C and C++ programming errors can lead to various memory corruptions and, in many cases, to corresponding exploits. MITRE lists out-of-bound writes as number one weakness in its 2021 Common Weakness Enumeration (CWE) top list [11], followed by out-of-bound reads and use-after-free on ranks 3 and 7, respectively. In recent years, safe language alternatives, such as Rust and Swift, emerged but C and C++ are still inevitable, especially in the embedded domain and for low-level components including Operating System (OS) kernels.

Memory corruptions can be grouped into spatial and temporal issues. In a *spatial* memory corruption, a pointer is used to read or write outside the bounds of its pointed-to object, for example, in a *buffer overflow*. In a *temporal* memory corruption, a pointer is used to access an object that has already been freed, for example, in a *use-after-free* attack. Such memory corruptions typically are the first step in more elaborate exploits, where an attacker tries to corrupt a data or code pointer, to leak information or to divert the program's control flow.

In the past decades, numerous approaches have been proposed to mitigate the effects of memory corruption exploits. Some of the resulting protections, such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) [23], have proven to provide reasonable trade-offs and, hence, find wide adoption in current systems. These approaches harden applications by forcing the attacker to reuse existing code and guess code locations, without imposing compatibility issues or performance penalties. Other solutions try to enforce Control Flow Integrity (CFI) using different policies and techniques to detect deviations in indirect control-flow transfers, such as indirect jumps, calls, or function returns [7, 9]. Despite some of these solutions already being supported by major compilers [40, 41], they have not found wide adoption due to limited security guarantees and prevalent compatibility issues [43].

Instead of mitigating the effects of exploited memory corruptions, other approaches aim to prevent the corruption itself by retrofitting C and C++ with memory safety, a thorough enforcement of *temporal* and *spatial* integrity for memory accesses. Early memory safety solutions proposed dialects to the C and C++ languages [20, 30], replacing unsafe language features with safe alternatives, such as fat pointers. However, language dialects impose great restrictions on the program-

mer and on the compatibility with existing code, requiring code to be adapted or rewritten. Later solutions adopted compiler-based or binary-based instrumentation to equip C and C++ applications with memory safety, with the former being the most commonly used form today. Instrumentation-based memory safety approaches refrain from restricting the C and C++ languages and, therefore, are completely transparent to the programmer. To achieve temporal memory safety, solutions typically delay the reuse of freed memory [6, 8, 12, 32, 35, 37], perform garbage collection [20, 30], bind pointers to the lifespan of their objects [5, 29, 36], or invalidate pointers on object deallocation [24, 42, 44]. To achieve spatial memory safety, instrumentation-based solutions either insert protected memory regions between objects [6, 12, 32, 35, 37] or track the bounds of allocated objects and their pointers [5, 8, 15, 16, 20, 28, 30, 36, 45]. However, to provide complete memory safety, protection mechanisms must combine temporal and spatial techniques, which usually hinders their practicability due to significant performance and memory penalties imposed on the protected applications.

In recent years, CPU manufacturers have started to incorporate hardware exploit mitigation extensions into their designs. With the *Pointer Authentication* extension [33] in architecture version ARMv8.3 [4], ARM provides a mechanism in their newer CPU designs that can add a Pointer Authentication Code (PAC) in the upper bits of pointers to enable cryptographic integrity protection. The extension adds new Instruction Set Architecture (ISA) instructions that can be used to generate and verify address signatures stored in the non-address bits of pointers. The key used for signing is managed by higher privileged software, i.e., typically the kernel, and cannot be accessed by the protected application. Existing approaches for the use of PAC primarily consider the realization of CFI policies, which is straightforward for function returns [33] and can be much more complex for indirect jumps and calls [27]. Recently, the more flexible approaches PACStack [26] and PCan [25] have been presented, extending the scope of pointer authentication protected objects to include call stack and stack canaries, respectively. Another approach, PTAuth [17], marks the first use of pointer authentication for memory safety, targeting temporal vulnerabilities only.

We propose *PACSafe*, a concept for utilizing ARM Pointer Authentication to achieve both spatial and temporal memory safety and, hence, provide much stronger security guarantees for protected applications on ARMv8.3 platforms. The basic idea of PACSafe relies on shadowing unsafely accessed memory of an application. The shadow is used to store a unique ID for each memory object, with the ID covering the whole shadow area corresponding to the object. Whenever an object is allocated, the ID is generated and used to generate a PAC specific to pointers to this object. PACSafe ensures that the association of a pointer to an object and its ID never changes. When a pointer is used, its PAC is first checked using the ID of the location that it tries to access. If the pointer has been manipulated to point to another object, the verification fails, since the ID does not match its PAC. When an object is deallocated, its ID is erased from the shadow. This ensures that all remaining pointers to the object stop working, since their PAC cannot be verified without the now erased ID.

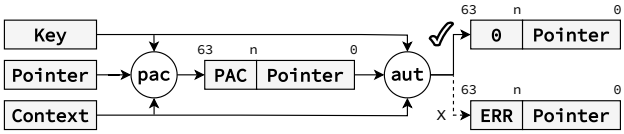

Figure 1: Signing (`pac`) and authenticating (`aut`) a pointer with ARM Pointer Authentication. n is the number of bits used for virtual addressing.

PACSafe achieves complete temporal and spatial memory safety, protecting globals and objects on the heap and the stack. Compared to existing approaches on commodity hardware, PACSafe achieves higher performance and/or better protection against crafted pointers due to its purposeful use of efficient PAC instructions. Additionally, its compatibility with external, uninstrumented libraries makes PACSafe very usable in real-world applications.

In summary, we make the following contributions:

- A mechanism for PAC-assisted temporal and spatial memory safety protection of memory objects.
- The PACSafe concept using this mechanism to protect heap, stack and global objects of C and C++ applications.
- A full LLVM-based prototype implementation of PACSafe.
- A detailed evaluation showing PACSafe's effectiveness and efficiency on a PAC-enabled Apple M1 processor.

The rest of the paper is structured as follows. First, we give background on the ARM Pointer Authentication mechanism in Section 2. In Sections 3 and 4 we then introduce the concept of a memory object life cycle and show how PACSafe adds cryptographic protection during and after an object's lifetime. Section 5 describes the components creating protection across the program, with details to the prototype implementation available in Section 6. We then evaluate PACSafe in Sections 7 and 8, by discussing the security implications and its detection rate, memory overhead and performance. Lastly, Section 9 compares related work approaches with PACSafe.

2 ARM Pointer Authentication

ARM *Pointer Authentication* is a hardware feature in the ARMv8.3 architecture [4, 33]. The feature enables software to cryptographically protect the integrity of pointers and, hence, ensure that they are not maliciously modified. The feature introduces two *key slots* A and B, which are managed by a

privileged component. For example, the keys for user space applications are managed by the kernel, while keys for the kernel are managed by a hypervisor. Typically, the managing component ensures that unique keys are provided for each managed context, e.g., each process, and changed correctly upon context switch.

Both keys can be used in special Pointer Authentication ISA instructions, signing and authenticating pointers as visualized in Figure 1. One group of instructions (`pac[i/d][a/b]`) *signs* pointers (`i`/`d` indicates code or data pointer), using either key A or key B (indicated by `a`/`b`), and stores the signature, called the PAC, in the non-address bits of the pointer. This is possible, since those bits are architecturally unused (with some exceptions). If no architectural tagging is used, all non-address bits except bit 55—which is used to determine if an address is in the *high* or *low* region—are used for storing the PAC. Therefore, the size of the PAC depends on the configured size for virtual addresses (n). With virtual address sizes between 52 and 32 bits, a PAC can have between 11 and 31 bits [33]. On our macOS M1 test system, 47-bit virtual addresses are used, resulting in 16-bit PACs. In addition to the pointer, the signing operation uses a *context* value of 64 bits, which can be used similarly to a salt value. Another group of instructions (`aut[i/d][a/b]`) *verifies* signed pointers, i.e., checks their PAC value, again either using key A or key B and an additional context value. If the key and context are equal to the ones used for creating the PAC, and if the pointer has not been modified in the meantime, the verification succeeds and the pointer is made usable by stripping its PAC bits. Otherwise, the processor replaces the PAC bits with a special bitmask that triggers an exception into the managing component as soon as the pointer is used as address operator in an instruction, e.g., a load/store or branch.

3 Memory Object Life Cycle

In order to describe the PACSafe concept, we first define an abstract memory model. In our memory model, there are two main *entities*:

Object. Any kind of allocated memory. For example, an object can be a variable or a buffer on the stack or in the heap of a program.

Pointer. A reference to an object, technically realized as a variable storing an address. A pointer always belongs to exactly *one* object. A pointer can access all parts of its object but accesses to other objects or accesses after deallocation of the object are illegal.

We define the following abstract *life cycle* for an object:

Allocation. Initially, the program allocates, i.e., reserves, memory for the object. Allocations can be *explicit*, e.g., in form of a specific function call such as `malloc()`, or *implicit*, e.g., in form of stack allocations for local variables at the beginning of functions.

Usage. While a memory region is allocated, it can be used, i.e., written or read, through pointers. Since a pointer always belongs to exactly one object, only accesses through pointers created during allocation or pointers derived from such pointers are legal.

Deallocation. Finally, the object's memory is deallocated and the object may not be used any more by the program. After deallocation, all accesses through pointers belonging to the object are illegal.

A *spatial memory corruption* can occur if a pointer is used to access memory that is not part of the pointer's object, i.e., in a buffer underflow or overflow. A *temporal memory corruption* can occur after deallocation when a pointer to the deallocated object is used for an access.

4 PACSafe Object Life Cycle

PACSafe leverages the ARM Pointer Authentication feature to implement memory safety. Using object metadata, PACSafe creates cryptographically secured pointers that can only be used for accessing the object they belong to while it is still alive. As foundation for its memory safety checks, PACSafe keeps metadata for protected objects in a 1:1 *shadow memory*. In the following, we describe the PACSafe primitives for allocation, pointer usage, and deallocation by introducing the life cycle of a protected object. For now, we do not consider PAC collisions, but discuss them later in Section 7.

Allocation. The first step of the object life cycle is the allocation. The allocation of a PACSafe-protected object is shown in Figure 2. The program first allocates memory for the object as it would normally, but then also generates a unique *identifier* (ID) that is repeatedly written into the corresponding shadow memory. The 32 bit wide ID is generated from a global counter (ctr), incremented after each allocation and initialized to a random value at program start. The ID, in general, enables PACSafe's spatial and temporal protections by marking object dimensions in the shadow memory. As the ID is repeatedly written into the entire shadow memory covered by the newly allocated object, object sizes must be multiples of 32 bit, increasing the allocation size of smaller objects if necessary.

After writing the ID to the shadow memory, instead of the pointer itself, the ID is extended with zero bits and signed using the `pac` operation with a zero context. The resulting PAC is then combined with the original address to create the final PACSafe-pointer used for the rest of the PACSafe object life cycle. As discussed in Section 2, the `pac` operation calculates a cryptographic signature, the PAC, and stores it in

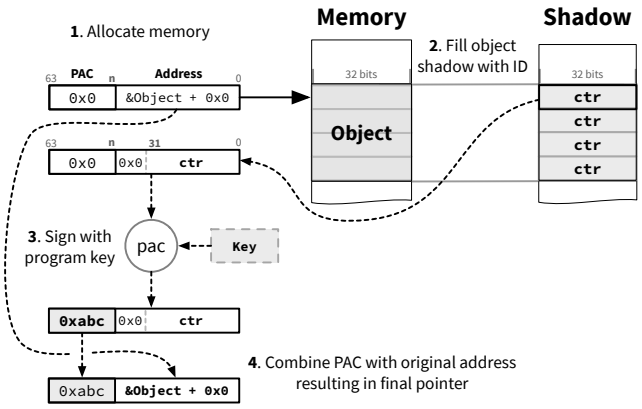

Figure 2: Allocation of an object with PACSafe protection.

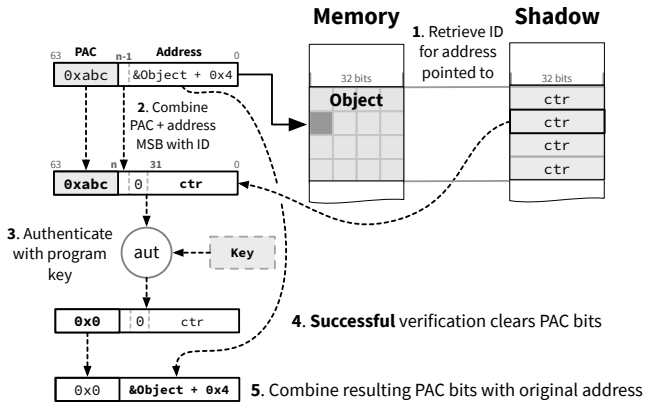

Figure 3: Legal usage of an object with PACSafe protection.

the non-address bits of the pointer. As a cryptographic signature the PAC can take any value including `0x0`. Therefore, to differentiate between pointers to protected or unprotected objects, PACSafe makes use of the ID stored in the object's corresponding shadow memory. Any ID other than `0x0` and `0x1` marks a PACSafe-protected object, the ID `0x0` indicates unallocated or unprotected memory, and the ID `0x1` indicates previously allocated but now free memory. Providing the possibility to mix protected and unprotected objects is crucial for PACSafe's stack protection concept and its support for uninstrumented libraries, both discussed in Section 5.

Usage. After an object has been allocated, it can be used through its PACSafe pointer or any pointer derived from it. For PACSafe-protected objects, any pointer to the object carries a PAC in its non-address bits. Using these PAC bits, PACSafe verifies the integrity of the pointer before allowing access to the protected object. An example of a legal access to an offset of 4 byte into a PACSafe-protected object is shown in Figure 3. First, using the pointer address, the ID corresponding to the pointed-to object is retrieved from the shadow memory. To authenticate the pointer, its address is then substituted with the 32 bits wide ID, only preserving the PAC and the Most Significant Bit (MSB) of the address. As explained in Section 2, the size of virtual addresses n on a particular system can typically be configured between 52 and 32. Since the address MSB must *not* be replaced with the ID, for PACSafe, n has to be between 52 and 33, resulting in a PAC size between 11 and 30 bits. Accordingly, the ID is extended with $(n-1)-32$ zero bits, keeping only the address MSB and, obviously, the PAC intact. Using a 32 bit wide ID ensures that the input for the signing operation is always larger than the resulting PAC, which is important with regards to certain attacks discussed in Section 7. Next, the combined "pointer" is verified using the `aut` operation, reversing the signing operation from the allocation. As our example depicts a legal access, the verification succeeds, clearing the PAC bits in the process. Note that, during the allocation, the address MSB is *fixed* to zero. Therefore, the verification will *always* fail for pointers addressing the upper half of memory, which is important for our shadow memory design, as discussed in Section 5.2. Finally, the resulting PAC bits are combined with the original pointer's address, forming the final pointer used to access the object. To ensure safety, PACSafe restricts its usage to this single access and discards it afterwards, except for some safe optimizations described in Section 5.4. If PACSafe finds the special ID `0x0` reserved for unallocated or unprotected memory, e.g., used by a library, it skips the check. For the special ID `0x1` indicating freed memory of a PACSafe object, the check always fails.

Apart from being used for accessing memory, pointers are also used for deriving other pointers, typically through pointer arithmetic. During such a derivation, the new pointer is created as result of an arithmetic operation on a base pointer. For pointers to PACSafe-protected objects, the arithmetic operation implicitly carries the base pointer's PAC bits to the derived pointer, automatically enabling its integrity protection and locking it to the object it is derived from. All legally derived pointers, i.e., still pointing into their original, alive object, can directly be verified as described above and shown in Figure 3. A more straightforward alternative would be to directly sign addresses, e.g., with a context stored in shadow memory, but such a solution would require pointers to be verified before derivation and re-signed after, hugely increasing the complexity of pointer arithmetic and PACSafe in general. By signing the object's ID instead, PACSafe allows for all pointers to that object to share the same PAC bits, which enables quick uninstrumented pointer arithmetic.

Pointer derivation is prone to creating illegal pointers due to faulty arithmetic operations increasing or decreasing offsets beyond the object's bounds or even overflowing the address into the pointer's PAC bits. For illegal pointers whose PAC bits have been corrupted, the verification trivially fails. For illegal out-of-bounds pointers, an example of the verification that detects the spatial memory violation is shown in Figure 4. Again, the verification process itself is identical to verifying

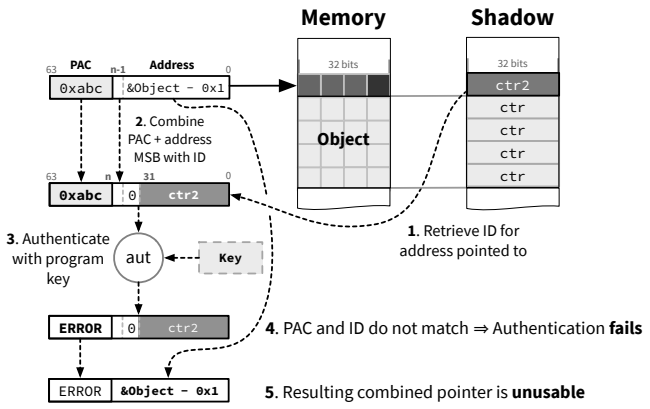

Figure 4: Usage of an object with PACSafe protection violating spatial memory safety.

any other pointer to a PACSafe-protected object. Using the address the pointer points to, the ID is retrieved from the corresponding shadow memory and the pointer's address bits (except the MSB) are substituted with it. However, as the pointer illegally points to a different object, a mismatching ID is retrieved and the `aut` operation fails. In contrast to a successful authentication where the PAC bits of the pointer are cleared, the unsuccessful `aut` operation sets the PAC to an `ERROR` bit combination that ensures that a following access will trigger an exception. After the authentication, the PAC bits are combined with the pointer's original address, forming the final, yet unusable pointer.

Similar to pointer derivation, PACSafe also imposes no restrictions on copying signed pointers. As pointers to PACSafe-protected objects carry their metadata, i.e., the PAC, in their non-address bits, copying these pointers implicitly propagates the metadata as well. Furthermore, PACSafe also imposes no restrictions on unsigned pointers pointing to regular, unprotected objects, e.g., returned by legacy, unprotected libraries. Unprotected objects have their IDs set to `0x0` in the shadow memory, so that PACSafe can omit the verification step for them.

Deallocation. The last step of the object life cycle is the deallocation, in which an object is destroyed by freeing its allocated memory. The deallocation of a PACSafe-protected object is shown in Figure 5. Before the object's pointer can be used for deallocating, PACSafe must verify its spatial and temporal integrity, prohibiting a deallocation through an illegal pointer. For a legal pointer, PACSafe then overwrites the entire shadow memory of the object to be freed with the special ID `0x1`, erasing the object's actual ID and marking the memory area as previously allocated. Finally, the object is freed using the standard deallocation.

Marking previously allocated but now freed memory enables PACSafe to detect temporal memory safety violations, such as *double-free* errors. For an ID other than `0x0`, PACSafe

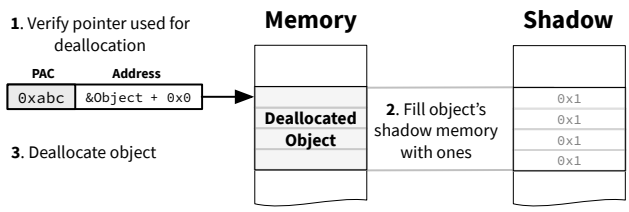

Figure 5: Deallocation of an object with PACSafe protection.

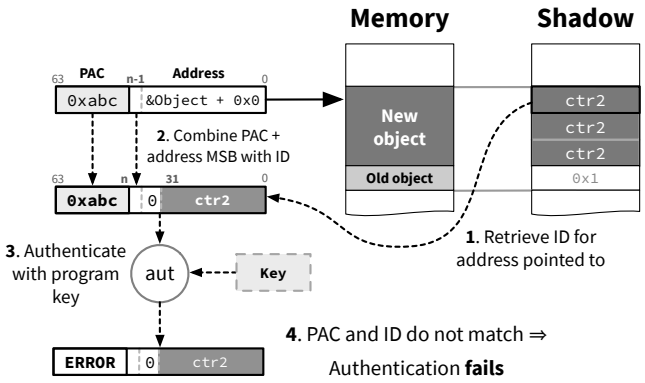

Figure 6: Use-after-free on re-allocated memory as example for a temporal memory violation on a PACSafe-protected object.

performs its pointer verification, detecting dangling pointers to freed memory, as an ID of `0x1` cannot be used to successfully validate any signed pointer. If the freed memory is reallocated to a new object, PACSafe generates a new ID and sets the corresponding shadow memory from `0x1` to the newly generated value. The differing IDs of the old and new object ensure that a temporal memory safety violation on reallocated memory, namely through a *use-after-free* error, is prohibited as well, as visualized in Figure 6.

Summarizing, with its modifications to the object life cycle, PACSafe is able to detect both temporal and spatial memory corruptions. Utilizing the PACSafe object life cycle to achieve memory safety for entire programs is discussed in the next section.

5 PACSafe Program Protection

The PACSafe object life cycle uses per-object metadata to lock pointers to their respective objects: Objects are assigned unique IDs and pointers to these objects are signed using those IDs. In this section, we present how PACSafe embeds this extended object life cycle into programs to achieve full memory safety.

5.1 Enforcing Memory Safety

PACSafe supports both protected and unprotected objects in the same memory space without compromising the security of protected objects. The allocation of protected objects returns pointers signed with their corresponding per-object ID. Redirecting such a pointer to an object with a different ID (including the ID `0x0`, indicating non-PACSafe, unprotected objects) leads to a runtime error. This allows PACSafe to selectively only protect unsafe objects, reducing complexity and increasing performance. Whether an object is unsafe and must therefore be protected by PACSafe, depends on the their location, allocation type, and their usage. In the following, we describe how the different program regions are protected by PACSafe. For this description, we focus on the protection of the main executable, deferring aspects regarding the support of uninstrumented libraries to Section 5.3.

Heap Objects. In PACSafe, all heap objects are protected by default. Allocating objects on the heap from instrumented code always generates unique per-object metadata and results in the returned pointers being signed. The PACSafe object life cycle then guarantees that every use of such pointers is authenticated before allowing access to the corresponding memory objects. As always, pointers without PAC can never access protected objects.

Stack Objects. On the stack, unlike on the heap, the majority of objects, such as primitive type local variables, are safely accessed directly through the stack pointer. Those objects cannot be used for memory corruptions but can themselves be compromised by unsafe objects also stored on the stack. In order to protect the whole stack, it is therefore sufficient to only PACSafe-protect unsafe stack objects, preventing them from compromising each other or the safe stack objects. PACSafe therefore statically identifies unsafe objects and moves their allocation to the protected heap, implicitly transforming them to PACSafe-protected objects. To distinguish between safe and potentially unsafe accesses to the stack, we use the approach proposed by SafeStack [22]. SafeStack determines all register spills, including saved return addresses and frame pointers, and all fixed-size stack variables, whose accesses are statically guaranteed to be in-bounds, to be safe. Variables for which an explicit address is taken at some point are deemed unsafe.

Globals. For globals, PACSafe applies the same partitioning approach as for the stack, only securing, i.e., instrumenting, unsafe globals that might potentially corrupt other objects. In contrast to the stack, instrumented and uninstrumented globals are not separated in memory but stay interleaved in their original locations. Safe and therefore uninstrumented globals include constants and fixed-size objects for which all accesses are statically guaranteed to be in-bounds.

For the objects that have to be protected, their global scope poses an additional challenge compared to stack and heap objects. Since the key and, therefore, the PACs used by PACSafe change on each program run, PACSafe-pointers cannot be generated statically but must always be created at run-time. While pointers for relevant stack and heap allocations are always created at run-time, globals are statically allocated and can be statically addressed, e.g., in non-Position-Independent Code (PIC) executables, not requiring the creation of pointers at run-time.

One solution to support globals with PACSafe would be by using a custom dynamic loader. This loader could generate and store PACSafe-pointers directly into the Global Offset Table (GOT). But since statically addressed globals are not handled via the GOT, the compiler would have to additionally ensure that *all* relevant globals are dynamically linked. Furthermore, since the use of a custom loader makes the solution less practical, we instead chose a self-contained approach solely relying on instrumentation. For that, we introduce the Global PACSafe-Pointer Table (GPPT), which contains a PACSafe-pointer for each protected global object and is accessible as global variable itself. During instrumentation, accesses to the original global variables are then modified to first load the corresponding PACSafe-pointer from the GPPT, and then execute the PACSafe verification to ensure correct spatial and temporal behavior. For global variables protected this way, initialization of the shadow metadata, PAC creation, and storage of the PACSafe-pointer in the GPPT are instrumented at the beginning of the main function of the protected program. Since all accesses to the GPPT are added by PACSafe itself, loads and stores to it are not instrumented with checks.

5.2 Memory Layout and Metadata

As explained in Section 4, PACSafe relies on per-object metadata stored in shadow memory. The shadow must store unique object IDs over the full length of objects and special IDs `0x0` and `0x1` for unallocated and freed memory, respectively. To realize this, PACSafe alters the memory layout of protected programs. In particular, PACSafe divides the entire virtual address space in halves, creating a 1:1 mapping between usable memory in the lower half and corresponding shadow memory in the upper half. For a virtual address size of n bits, the memory layout of a PACSafe-protected program is shown in Figure 7. While the lower half can be used by the program to store code and data, the shadow memory in the upper half is exclusively managed by PACSafe to store metadata for protected objects. Since PACSafe uses a partitioning approach for the stack moving unsafe objects to the heap, only the segments for heap and globals require shadowing. With the 1:1 mapping, addresses of protected objects can easily be translated to their corresponding shadow addresses by setting their most significant bit, i.e., bit $n-1$ in Figure 7. This enables

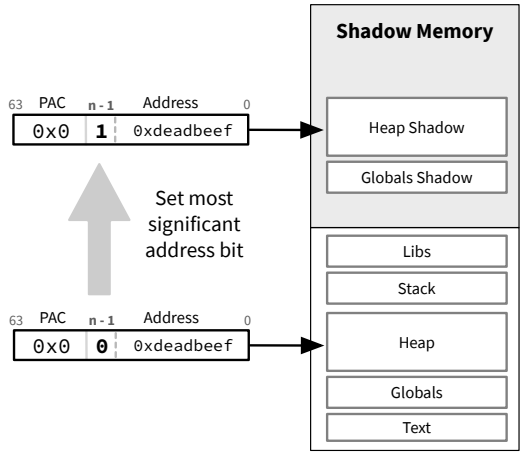

Figure 7: PACSafe shadow memory mapping for virtual address size n. Setting the most significant address bit transforms an address to the corresponding shadow memory address.

PACSafe to perform efficient metadata lookups. Note that the MSB is set, not flipped, so that the mapping is a one way operation and addresses in the shadow map to themselves.

As described in Section 4, PACSafe only generates valid signatures, i.e., working pointers, for the lower half of the address space by forcing the MSB of pointers' addresses to zero during allocations. As the actual address MSB is used during pointer verification, accesses to the upper half, the shadow containing the metadata, are *always* illegal. This is also the case for pointers trying to access their own metadata, which would, without the incorporation of the address MSB, have a PAC matching the shadow ID. Metadata security is further discussed in Section 7.

5.3 Compatibility

With the PACSafe metadata design, we aim to achieve full interoperability with uninstrumented code, i.e., especially dynamically linked libraries. While PACSafe allows libraries to be instrumented and protected as well, which we encourage for optimal protection, in practice, PACSafe-protected programs are likely executed in with unprotected dynamic libraries. Hence, we designed PACSafe not to interfere with objects allocated in unprotected libraries and allow them to coexist with protected objects in the same address space. While PACSafe objects cannot be protected from unchecked accesses from within uninstrumented libraries, PACSafe still ensures that objects are safe from unsafe pointers potentially coming from the libraries.

To still preserve the safety of protected objects, PACSafe intercepts calls to the standard C/C++ allocation routines and ensures that the shadow memory corresponding to the newly allocated object contains the special ID `0x0`, denying the allocation of memory shadowed with an actual ID (which should never happen) and possibly resetting the shadow from special ID `0x1` to `0x0`. The interception is completely transparent to the library and forwards the actual allocation of memory to the originally called routine. Note that only calls from uninstrumented code are intercepted, as instrumented code directly uses the PACSafe versions of these routines. With an ID of `0x0`, PACSafe refrains from verifying pointers before granting memory access, as described in Section 4. Hence, unsigned pointers as returned from the standard allocation routines are still able to access unprotected objects, while access to protected objects is prohibited. If an unprotected object is allocated in a region that was previously occupied by a PACSafe object, resetting its shadow from `0x1` to `0x0` does not compromise temporal safety of the PACSafe object, since potential dangling pointers remain signed and the PAC is only stripped as part of the check. Skipping the check also skips the removal of the signature, which leaves such a pointer unusable, even with a shadow containing `0x0`.

Furthermore, PACSafe is able to detect calls to external code and supports stripping the PAC from pointers arguments. This enables necessary accesses of protected objects from uninstrumented code, achieving PACSafe's full interoperability with unprotected libraries. The differentiation between internal and external functions is possible by postponing PACSafe's instrumentation to Link Time Optimization (LTO), where truly external functions can be correctly identified.

5.4 Optimizations

By default, PACSafe has to instrument each memory access to achieve memory safety. Nonetheless, in several situations checks can be omitted because the underlying memory access is statically safe or redundant, as detailed in the following.

Statically Safe Accesses. As explained in Section 5.1, for stack and globals, PACSafe uses a partitioning approach only protecting objects that are potentially dangerous. The remaining safe objects do not require checks. Since they are shadowed with `0x0` IDs, PACSafe automatically skips verifications at run-time for safe objects, allowing checks to work on memory accesses that can target both protected and unprotected memory. For accesses that are statically ensured to only target safe objects, checks can be omitted at compile-time. To detect some of those cases and optimize them, PACSafe performs an intra-procedural analysis at compile-time.

Redundant Check Removal. If multiple memory accesses use the exact same pointer (the pointer type also determines the access size) and one of the accesses *always* executes before the others, i.e., it *dominates* them, it is sufficient to check only the first access. PACSafe performs an intra-procedural analysis to detect and remove such redundant checks. This

approach was adapted from Softbound's optimization removing redundant spatial checks [28]. Since PACSafe's pointer verification uses a combined spatial and temporal check, our analysis additionally verifies that between two memory accesses with the same pointer, the pointer is not passed to functions possibly freeing the underlying memory object.

Same Lock Optimization. Another intra-procedural optimization of PACSafe is possible for consecutive accesses to the same object. Consecutive memory accesses read or write to the same object, but increase or decrease the pointer each time. In PACSafe, this pointer derivation results in every pointer containing the same PAC and every access through these pointers being legal if the PAC can be authenticated using the pointed-to object's ID. However, as soon as *one* of these pointers has been authenticated successfully, any subsequently derived pointer can trivially be verified by checking whether it points to an object shadowed by the same ID. This *fast* verification is possible, as the relationship between PAC and ID has already been validated and can only be invalidated by a changing ID, i.e., the derived pointer overflowing into the neighboring object.

To apply this optimization, similar to the redundant check removal, PACSafe identifies load and store instructions accessing the same object with different offsets and then instruments the normal pointer verification for the dominating instruction while instrumenting the fast verification for all dominated instructions. Again, the analysis verifies that between the dominating and dominated instructions, the pointer is not passed to any function possibly freeing the underlying memory.

6 Implementation

Our PACSafe prototype is implemented as an extension to the LLVM compiler framework (version 12) and supports macOS on ARMv8.3 hardware, i.e., M1 Macs. Our LLVM extension provides an instrumentation pass and a custom runtime library. While the instrumentation pass inserts the PACSafe protection into the program, the runtime library manages PACSafe at run-time and provides interoperability with unprotected libraries.

6.1 Runtime Library

PACSafe-protected programs are linked against our runtime library based on the LLVM compiler runtime library. Our runtime library initializes and maintains PACSafe's shadow memory and additionally provides wrappers for commonly used functions the C/C++ standard library, such as `memcpy`, to secure calls without instrumentation in the library.

Shadow Memory Allocation. To initialize the shadow memory, the runtime library provides a dedicated constructor, which is executed by the dynamic linker when loading the PACSafe-protected program. This constructor simply allocates the upper half of the virtual memory using `mmap`. If this allocation fails, the runtime library aborts further loading of the protected program.

Shadow Memory Maintenance. PACSafe maintains metadata for heap objects, heap-transformed stack variables, and global variables as described in Sections 4 and 5.1. For allocations on the heap, PACSafe provides safe versions of the standard C/C++ allocation and deallocation routines (i.e., `malloc`, `free`, etc.) that implement the additional handling of metadata. The PACSafe instrumentation pass statically replaces the standard routines with the PACSafe variants, which ensures that uninstrumented libraries still use the standard routines, as required. For global variables, the generation of metadata is directly instrumented into the program. Globals remain allocated for the entire lifetime of the program, so that PACSafe initializes their metadata only once at load-time.

The safe allocation routines, as presented in Section 4, generate the required metadata, store it into the shadow memory, then allocate the actual heap object and sign the corresponding pointer, before returning the signed pointer to the program. The counter used for object ID generation is allocated by the runtime library and incremented each time PACSafe generates a new ID, looping back to zero once reaching its maximum value. To allocate the actual heap object, the safe routines themselves call the standard C/C++ allocation routines. Likewise, the safe deallocation routines overwrite the IDs stored in the shadow memory with `0x1` and delegate the actual deallocation to the standard routines.

Standard Library Wrappers. When linking PACSafe-protected programs against uninstrumented C/C++ standard libraries, library functions taking pointer arguments to manipulate data, such as `memcpy`, `strncpy`, etc., may be used to access protected objects. To guarantee memory safety for protected objects in those cases, PACSafe's runtime library provides wrappers that verify signed pointers before forwarding them to the originally called library function. If the library function also returns a pointer to one of the passed pointer arguments, the initial PAC is reapplied to the pointer before returning. Similar to the safe versions of allocation and deallocation routines, PACSafe's instrumentation pass substitutes calls to the library functions in instrumented code with call to their corresponding wrappers.

6.2 Instrumentation Pass

The PACSafe compiler pass analyzes which load and store instructions require checks (see Section 5.4), instruments them with pointer checks, redirects library calls to our safe

allocation and deallocation routines or to our wrappers for pointer verification, and transforms unsafely accessed stack variables to heap objects. For this transformation our pass replaces the stack variable's *alloca* instruction with a call to our safe allocation routine and additionally inserts a call to our safe deallocation routine after the last use of the variable before the function returns. This guarantees the correctness of the relocated variable's lifetime and enables PACSafe to detect *use-after-scope* errors.

6.3 Language Feature Support

PACSafe is designed to be fully compatible with C and C++. Currently, our PACSafe prototype provides compatibility with the most commonly used functions from the C/C++ standard libraries and supports both user-level and kernel-level multithreading, as discussed in the following. Nonetheless, there are some caveats and limitations that we discuss as well.

Multithreading. Our PACSafe prototype is fully compatible with user-level and kernel-level multithreading. To achieve thread-safety, we have to consider the concurrent allocation and deallocation of PACSafe-protected objects. The verification of signed pointers is always inherently thread-safe, since it does not manipulate metadata and altering the pointer, i.e., clearing or setting its PAC bits, is thread-local. To secure concurrent heap allocations, our runtime library enforces exclusive access to the counter used for ID generation by encapsulating its use with a global mutex. The thread-safety of the actual allocation of memory is guaranteed by the standard C/C++ allocation routines, called by our runtime library before generating the metadata. As the concurrent deallocation of heap objects is a form of double-free error, our runtime library encapsulates its safe deallocation routines with another mutex, preventing concurrent threads from freeing the same object.

External Functions with Variable Argument Lists. Variadic C functions, such as `printf`, take a variable number of arguments. A variadic function uses the list type `va_list` and special macros to access and iterate its variable arguments. Programs sometimes define their own versions of variadic C standard library functions. For example, one of the SPEC CPU 2017 benchmarks implements a custom `printf` function printing to `stdout` and logging to a file at the same time. Since variable arguments cannot directly be forwarded to another variadic function, for those cases the standard library provides non-variadic counterpart functions, such as `vprintf`, that directly take a `va_list` argument. As for all external functions, pointer parameters in the variable arguments (that are dereferenced inside the external library) must be stripped of their PAC to be compatible with the uninstrumented library. If an external uninstrumented variadic function, such as `printf` is called directly from PACSafe-protected code, the PACSafe pass correctly strips the pointer arguments. But, if a program replaces a variadic standard library function and calls the underlying non-variadic, `va_list`-receiving function directly, pointers in the list cannot be automatically stripped. This can be solved by providing wrappers for these standard library functions that unpack the `va_list` according the function semantics and strip pointers. The current PACSafe prototype does not provide such wrappers. Note that this is only an issue when using uninstrumented (standard) libraries, meaning that another, already working solution is to compile the affected library with PACSafe as well.

Non-Local Jumps. With `setjmp` and `longjmp`, the C standard library allows functions to perform non-local jumps, e.g., to return to functions other than their immediate callers. To facilitate these non-local jumps, the C standard library unwinds the stack between the function issuing the jump and the function being jumped to. With PACSafe, unsafe stack variables are relocated onto the heap and their allocation and deallocation are instrumented before their first and after their last use, respectively. To support non-local jumps in PACSafe-protected programs, the deallocation of heap-transformed stack variables must be handled during stack unwinding. Our PACSafe prototype currently does not instrument stack unwinding, leaving non-local jumps as a future improvement.

C++ Exception Handling. Similar to non-local jumps, C++ exception handling requires stack unwinding, which is currently not supported by the PACSafe prototype.

7 Security Discussion

For our discussion, we assume an uncompromised system running programs that contain memory vulnerabilities. An attacker may target these vulnerabilities with the intend to leak program-internal data or corrupt the program's control-flow. However, the attacker cannot interfere with programs through the system, e.g., by manipulating the keys held by the kernel to sign and verify pointers. Within these attacker capabilities, PACSafe is designed to prevent any malicious manipulation of pointers, as discussed in the following.

Illegally Derived and Dangling Pointers. An attacker might compromise a pointer without corrupting the PAC itself (see *Crafted Pointers* for attacks also corrupting the PAC). For example, they might exploit faulty pointer arithmetic to manipulate a pointer's address bits or use an unmodified, but dangling pointer. As described in Sections 4 and 5, PACSafe identifies possibly unsafely accessed objects and locks pointers to their lifetime and bounds using PAC signatures. Safe objects are always accessed safely and cannot be exploited. Dereferences are only allowed for pointers to non-shadow memory (i.e., the lower half of memory) whose PAC matches

the ID in the shadow of the memory pointed to. An exception is ID `0x0`, for which PACSafe skips the check but also skips the removal of the PAC bits, so that signed pointers remain unusable in those cases. Otherwise, illegal accesses via pointers not pointing to their original object only succeed if a *signature collision* occurs, i.e., if the PAC happens to be the same for both memory areas. This is trivially the case if two objects share the same ID, a *metadata collision*, but due to the limited PAC size, can also happen for objects with different IDs. Depending on the virtual address size a system is using, PACSafe's PAC size p is between 11 and 30 bits (tagging disabled). On macOS the PAC is 16 bits wide. The probability for a signature collision when targeting a specific object is $\frac{1}{2^p}$, i.e., only $\frac{1}{2^{16}} = 1.52 \cdot 10^{-5}$ for PACSafe on macOS. We consider this a negligible risk, as these collisions are unpredictable and give an attacker no leeway to time an attack correctly. PACSafe's 32-bit wide IDs ensure that the full PAC space is used for all possible PAC sizes. Metadata collisions might be predictable if the randomly initialized counter for ID generation is exhausted in a single program execution. But since that only happens after $2^{32} - 2$ allocations (and after several signature collisions), we consider this an insignificant risk.

Crafted Pointers. Typical vulnerabilities only allow an attacker to compromise the address bits of a pointer or to use a dangling pointer. Nonetheless, under certain conditions, an attacker might be able to completely *craft* a pointer including its PAC bits. This can, for example, happen via very large pointer arithmetic overflows into the PAC bits or via *intra-object* corruptions, in which the targeted pointer is inside a compound object together with an unsafe member, e.g., an array. With its per-object metadata, as all similar approaches, PACSafe cannot detect intra-object corruptions. Nonetheless, PACSafe's cryptographic pointer protection ensures that an attacker crafting a pointer must guess the right PAC for the object they want to access, with a negligible success probability of $\frac{1}{2^p}$, as described before. As discussed in the next paragraph, this is also the case even if the attacker is able to leak the ID of the targeted object from the shadow memory.

Metadata Security. In contrast to other pointer tagging based approaches, such as HWASan (see Section 9), the cryptographic nature of the PACs ensures that metadata access does not directly compromise the security of corresponding objects. The relationship between metadata and PAC is protected by the key managed by a privileged component and inaccessible to the program. An attacker who is able to write the ID of an object can only access the object in conjunction with a pointer crafting vulnerability (by writing the special ID `0x0` and using a stripped pointer). And leaking an ID of an object does not enable an attacker to generate a valid PACSafe pointer, even with a pointer crafting vulnerability. Nonetheless, as described in Section 5.2, PACSafe prevents direct access to metadata. The memory is split in halves and addresses are mapped to shadow addresses simply by *setting* the MSB of the address. This is a one-way operation, which means that an address in the shadow maps to itself, and a pointer manipulated to point into the shadow is always checked using the ID stored in the pointed to location itself. PACSafe only generates signatures for pointers in the non-shadow memory, i.e., with the address MSB unset. Since the actual address MSB of a pointer is input to the signature verification, a pointer to the shadow, i.e., with the address MSB set, is never successfully verified (assuming no collisions).

Unprotected Libraries. PACSafe allows protected programs to link against unprotected libraries. When linked against unprotected libraries, PACSafe still provides its protection inside the instrumented code, even for pointers coming from the library, as unsigned pointers to protected objects are verified detecting illegal accesses. However, an attacker may illegally access protected objects through vulnerabilities within the uninstrumented code since memory accesses are not checked there.

8 Evaluation

We evaluated our PACSafe prototype on an M1 MacBook Pro 2020 (16 GB memory) with actual hardware support for ARM Pointer Authentication running macOS Big Sur in version 11.3.1. This is in contrast to previous publications using ARM Pointer Authentication [17, 27] which were evaluated using a combination of emulation and simulation.

In our evaluation, we tested whether PACSafe reliably detects temporal and spatial memory corruptions, and how much performance and memory overhead PACSafe induces at runtime. To measure PACSafe's memory safety guarantees we used the Juliet Test Suite[1]. To measure the performance and memory overhead we used the nbench-byte[2] and SPEC CPU 2017[3] benchmark suites.

8.1 Vulnerability Detection Rate

The Juliet Test Suite is a collection of test cases in C and C++ covering a wide variety of CWEs, including temporal and spatial memory corruptions in different flavors, which we used for our evaluation. The suite additionally provides several so called *flow variants* for each test case, wrapping the same vulnerability in different control flow or data flow constructs, e.g., guarding the bug with an `if`-clause. Since most of these variants do not offer differentiation at run-time and are only useful when testing static analysis tools (the original use case for the Juliet Test Suite), we only used the

[1] https://samate.nist.gov/SRD/testsuite.php
[2] https://www.math.utah.edu/~mayer/linux/bmark.html
[3] https://www.spec.org/

Table 1: PACSafe Juliet Test Suite evaluation results.

CWE	Cases	Detected	Secured
121: Stack Overflow	67	90%	100%
122: Heap Overflow	40	80%	*92.5%
124: Buffer Underwrite	16	100%	100%
126: Buffer Overread	13	69.2%	**76.9%
127: Buffer Underread	16	100%	100%
415: Double Free	5	100%	100%
416: Use After Free	6	100%	100%
761: Free Inside Buffer	4	100%	100%

* Remaining tests do *not* trigger an actual memory corruption
** Memory corruptions triggered in uninstrumented `printf`

base variants of the test cases. Furthermore, we excluded some test cases that require user input.

Table 1 summarizes the results of the PACSafe evaluation with the Juliet Test Suite. In our results overview, we differentiate between detected and secured cases. For *detected* cases, PACSafe always triggers an exception when the bug is encountered. For *secured* cases, PACSafe ensures that the bug is not exploitable but does not necessarily detect it, i.e., does not always stop execution. The latter is due to PACSafe's 32-bit padding applied to protected objects and happens if an overflow does not exceed the padding. As such, those situations only affect overflows/overreads and do not compromise the security of the program. PACSafe is able to *secure* all test cases, i.e., provides full memory safety. The remaining, non-secured tests either do not trigger an actual memory corruption (CWE 122) or trigger bugs in the uninstrumented, unwrapped libc function `printf` (CWE 126). Furthermore, PACSafe is able to *detect* all temporal bugs and underreads/underwrites and almost all overflows/overreads. As explained, the remaining, non-detected but secured test cases are overreads/overflows into PACSafe's object padding.

8.2 Performance Overhead

Maintaining metadata and checking memory accesses induces overhead for PACSafe-protected programs. To measure this overhead, we used the nbench-byte and SPEC CPU 2017 benchmark suites, comparing the performance of benchmarks compiled with PACSafe to a baseline of native benchmarks compiled without PACSafe. For our evaluation with SPEC CPU 2017, out of the 17 C/C++ *rate* benchmarks, we had to exclude 6 benchmarks due to PACSafe's own limitations (i.e., benchmarks using non-local jumps and/or external functions with variable argument lists) and 6 additional benchmarks due to incompatibilities with macOS's own use of pointer authentication and LLVM 12 (see Appendix A for more details). For nbench-byte, all 10 benchmarks are compatible with PACSafe and our M1 MacBook Pro test setup. The benchmarks are single-threaded by default. We compiled all benchmarks with an optimization level of `O2` and executed each benchmark 10 times to measure its average performance.

Table 2: Overhead of PACSafe for SPEC CPU 2017 and nbench-byte benchmarks.

Benchmark	Performance	Memory
999.specrand_r	2.0%	7.9%
519.lbm_r	49.9%	99.9%
544.nab_r	106.6%	101.2%
505.mcf_r	146.2%	124.1%
557.xz_r	190.6%	128.8%
Fourier	0.01%	14.8%
Bitfield	1.0%	34.2%
IDEA	16.5%	34.0%
FP Emulation	39.4%	85.9%
Assignment	95.3%	32.9%
Lu Decomposition	101.9%	53.0%
String Sort	103.9%	59.6%
Neural Net	124.6%	41.2%
Huffman	248.9%	37.3%
Numeric Sort	282.6%	41.5%
Average	100.6%	59.7%
Geometric Mean	30.1%	47.1%

The results of our performance evaluation are shown in Table 2, listing the SPEC CPU 2017 benchmarks before the nbench-byte benchmarks. For the SPEC CPU 2017 benchmarks, the evaluation shows an performance overhead of 99% on average, with a geometric mean of 49%. For nbench-byte, the PACSafe-protected benchmarks experience an overhead of 101% on average, with a geometric mean of 23%. The average and geometric mean across both benchmark suites evaluates to 101% and 30%, respectively. Looking at single benchmarks, the overhead varies from negligible to substantial. We attribute this difference to the varying memory usage, with benchmarks, such as the Huffman text and graphics compression algorithm, iterating over large amounts of memory, requiring PACSafe to verify memory accesses frequently and resulting in a significant performance overhead. Other benchmarks, such as the Fourier benchmark, mainly process local, safely accessed variables on the stack, resulting in fewer memory accesses verified by PACSafe and an insignificant performance overhead.

To put the performance of PACSafe into context, in Table 3, we compared it against other memory safety solutions. Since the reproduction of the measurement environment for the other solutions is, in many cases, very difficult, especially since they were mostly not designed for ARM platforms, we rely on the results given by the authors in their respective publications. The comparison shows that PACSafe performs better than all other solutions, with its only compromise being

Table 3: PACSafe performance overhead compared to other memory safety frameworks.

Defense Mechanism	Average	Geo. Mean
PACSafe	101%	30%
CUP	158% [8]	-
ASan	109% [38]	99% [38]
LFP	113%-156% [15]	85% [38]
SoftBound/CETS	116% [29]	-

the loss of *detection* precision (not security) for some very small overflows/overreads (less than 4 bytes). The other approaches either compromise on temporal safety (LFP, CUP, ASan), on spatial safety (ASan), and/or on compatibility (Softbound, CUP), as discussed further in Section 9.

8.3 Memory Overhead

By maintaining object IDs, PACSafe imposes memory overhead on protected programs. Each PACSafe-object is shadowed 1:1, causing 100% overhead. While this relationship is clear, there are several aspects that make an estimation of a program's overall memory much less straightforward, justifying an experimental evaluation: First, PACSafe-objects must be padded to 32-bit boundaries, possibly increasing the overhead beyond 100% for a single object. Second, as explained in Section 5.1, not all objects require explicit protection for the program to be memory safe and PACSafe only protects and shadows heap and *unsafe* stack and global variables, reducing the overall overhead. Third, freeing a PACSafe-object does *not* free its shadow but overwrites it with `0x1` to prevent specific temporal issues with uninstrumented libraries (see Section 4). While we expect the allocator to eventually re-use much of the corresponding memory, which also re-uses its shadow, this behavior can lead to overheads of more than 100% for some programs.

To evaluate PACSafe's memory overhead, we used the same benchmarks as for the performance evaluation, comparing their peak memory consumption using `time -l`[4] against the corresponding non-PACSafe baseline. The results are summarized in Table 2. As expected, the overhead heavily depends on the actual workload, ranging from 7.9% for the `specrand_r` benchmark, not using dynamically allocated memory, to 128.8% for the heap-intensive `xz_r` benchmark. With an average memory overhead of 47.1%, PACSafe positions itself well below Softbound's overhead (without CETS) of 64-87% [28] and ASan's overhead of more than 200% [35]. Note that PACSafe's memory overhead can be further reduced if compatibility with uninstrumented libraries is not required.

[4]Only available on BSD-like systems. Prints the contents of the `rusage` structure.

9 Related Work

Memory safety for C and C++ has been an active field of research for the past decades. The least precise memory safety solutions rely on mapping heap objects to individual memory pages and interleaving them with unmapped, i.e., inaccessible, *guard pages* [12, 14, 32]. These guard pages prevent contiguous overflows between objects, but leave overflows within the object's page and arbitrary accesses skipping guard pages undetected. These solutions are commonly accompanied by *prohibiting* the reuse of mapped memory pages once their objects have been deallocated, providing temporal memory safety at the cost of memory exhaustion.

Similar memory safety solutions with better precision and less memory consumption substitute guard pages with allocating *red-zones* between memory objects [6, 18, 19, 35, 37], such as ASan [35], the best-known and most-adopted memory safety solution today. This relieves stress on the memory, as objects and red-zones can be allocated contiguously on the same memory pages. It further improves the solutions' precision, as protection is not limited to heap objects only and protected objects do not require padding. However, red-zones suffer from the same problem as guard pages, being unable to detect arbitrary accesses skipping them. To provide temporal memory safety, these solutions commonly *delay* the reuse of allocated memory instead of prohibiting reuse altogether. Because metadata for red-zones can be maintained based on the red-zones' location, solutions are highly compatible with uninstrumented code, enabling a seamless integration of red-zones into software requiring uninstrumented third-party libraries. PACSafe maintains per-object metadata and locks pointers to their pointed-to objects, providing superior precision to guard pages, red-zones, and memory reuse delay, as any spatial and temporal corruption between memory objects is prevented. Although PACSafe maintains per-object metadata, compatibility with uninstrumented code is on par with location-based solutions by carefully designing PACSafe to ignore memory objects with metadata set to zero.

Other solutions maintain *per-object* metadata to guarantee spatial memory safety only [2, 13, 15, 16, 21, 34, 45]. They store bounds information of objects in a disjoint data structure and enforce pointer arithmetic to stay within the original object. Out-of-bounds pointers are either invalidated by setting the pointer's value to an unaccessible address or handled explicitly by using additional per-pointer metadata. As these solutions do not attach metadata to pointers themselves (except for out-of-bounds pointers), they generally provide adequate compatibility with uninstrumented code. Bounds information is created during the allocation of objects, which can be intercepted for uninstrumented code as well, giving instrumented code the ability to monitor pointer arithmetic regardless of the origin of objects. However, because pointers lack metadata, these solutions are not able to detect temporal memory corruptions. For spatial memory safety, they provide

almost best possible precision, only leaving intra-object corruptions and overflows into potential padding undetectable. PACSafe also maintains per-object metadata, but locks pointers to their pointed-to objects utilizing unused pointer bits. This eliminates the need to handle out-of-bounds pointers explicitly, and enables PACSafe to provide temporal memory safety and detect corruptions of intra-object pointers. Hence, PACSafe achieves superior precision to other solutions based on per-object metadata. Further, PACSafe has better compatibility with uninstrumented code, as out-of-bounds pointers are handled implicitly and memory objects from uninstrumented code (with their metadata set to zero) are ignored.

Similar to our work, HWASan [36] extends the idea of maintaining metadata per object by utilizing unused pointer bits to store a link to the pointed-to object's metadata. In HWASan, the per-object and per-pointer metadata are identical: For each object, an 8-bit identifier is randomly chosen and stored in the unused bits of pointers to that object. Simultaneously, the identifier is kept in a memory-aligned shadow memory, effectively providing a *lock* to the pointers' *keys* in their unused bits. This allows HWASan to provide temporal memory safety without maintaining individual metadata per pointer. Before dereferencing a pointer and granting memory access, HWASan verifies that the pointer's key is identical to the lock stored in the shadow memory corresponding to the pointed-to object. Further, this enables HWASan to handle out-of-bounds pointers implicitly and ignore pointers from uninstrumented code whose unused bits are not set (i.e., zero). However, with its identifiers of only 8 bits in size, HWASan is prone to metadata collisions leading to undetected memory corruptions. Furthermore, for HWASan, metadata security is critical, whereas PACSafe's cryptographic approach can tolerate, for example, metadata leaks.

The solutions with the best precision possible achieve *complete* spatial and temporal memory safety by maintaining metadata *per pointer* rather than per object. First solutions created fat pointers to store bounds information for spatial safety checks [5, 20, 30, 39] and typically deployed garbage collection for temporal safety [20, 30]. Later, with Softbounds/CETS [28, 29] being the most prominent, solutions opted for keeping bounds information disjoint in memory and added *keys* to the metadata to perform temporal safety checks [1,28,29,31]. This also required additional metadata to store a *lock* per object, matching the key of pointers to that object. By invalidating the lock of deallocated objects, the link to pointers with the same key is broken, making dangling pointers detectable. Further, they are the only solutions capable of detecting intra-object overflow, although to the best of our knowledge no actual implementations have been published to date. However, maintaining metadata per pointer degrades run-time performance significantly and yields poor compatibility with uninstrumented code. When new pointers are created through arithmetic and assignment operations, pointer metadata must be propagated, requiring additional instructions and leaving uninstrumented code incapable of handlich protected pointers. Compared to these solutions, PACSafe maintains per-object metadata, but utilizes unused pointer bits to lock pointers to their pointed-to objects. With this hybrid approach, PACSafe has comparable protection against spatial and temporal memory corruptions, while achieving better run-time performance and providing better compatibility with uninstrumented code.

Finally, some solutions focus on detecting temporal memory corruptions only, either by *tracking pointers* on a per-object basis [10, 24, 42, 44] or, similar to PACSafe, by *locking pointers* to their pointed-to objects [17, 29]. Tracking all pointers pointing to an object is straightforward, as only allocation and pointer derivation operations must be monitored. Whenever an object is deallocated, all registered pointers are immediately invalidated. However, this simple approach cannot be combined or extended with any kind of spatial memory corruption detection, making it inferior to locking-based solutions such as PACSafe. PTAuth [17] utilizes the ARM Pointer Authentication feature to sign heap pointers with a 64-bit identifier prefixed to their pointed-to heap objects. In contrast, PACSafe provides both spatial and temporal memory safety for globals, stack variables, and heap objects alike.

10 Conclusion

We proposed PACSafe, a new memory safety concept for C/C++ programs, leveraging ARM Pointer Authentication. PACSafe is able to provide full memory safety, protecting a program's heap, stack, and globals against both spatial and temporal memory corruptions. Since PACSafe's protection is based on signatures using secret keys, it provides additional security against attacks on metadata compared to similar, software-only approaches. We implemented a fully functional, LLVM-based PACSafe prototype for macOS on M1 Macs. In our evaluation, which, to the best of our knowledge, is the first of its kind on actual hardware, our PACSafe prototype outperforms similar software-based frameworks. This, together with its improved compatibility with uninstrumented libraries, makes PACSafe a viable solution for retrofitting memory safety to C/C++ programs.

Acknowledgments

This work was supported by the Fraunhofer Internal Programs under Grant No. PREPARE 840 231.

References

[1] *An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs*, SIGSOFT '04/FSE-12, New York, NY, USA, October 2004. ACM.

[2] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, SEC '09. USENIX Association, August 2009.

[3] Apple. Preparing your app to work with pointer authentication. https://developer.apple.com/documentation/security/preparing_your_app_to_work_with_pointer_authentication. Accessed: 2022-01-26.

[4] ARM Limited. *ARM Architecture Reference Manual – ARMv8-A, for ARMv8-A architecture profile*, July 2019. ARM DDI 0487E.a (ID070919).

[5] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '94, pages 290–301, New York, NY, USA, June 1994. ACM.

[6] 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*, CGO '11, pages 213–223, Washington, DC, USA, April 2011. IEEE.

[7] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys*, 50(1):16:1–16:33, April 2017.

[8] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Pay er. CUP: Comprehensive user-space protection for C/C++. In *ASIACCS*. ACM, 2018.

[9] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining light on shadow stacks. In *S&P*. IEEE, 2019.

[10] 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*, ISSTA '12, pages 133–143, New York, NY, USA, July 2012. ACM.

[11] MITRE Corporation. 2021 CWE top 25 most dangerous software errors. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html. Accessed: 2022-01-13.

[12] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *Proceedings of the 26th USENIX Security Symposium*, SEC '17, pages 815–832. USENIX Association, August 2017.

[13] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171, New York, NY, USA, May 2006. ACM.

[14] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks*, DSN '06. IEEE, June 2006.

[15] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, CC '16, pages 132–142, New York, NY, USA, 2016. ACM.

[16] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium*, NDSS '17. Internet Society, February 2017.

[17] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. Ptauth: Temporal memory safety via robust points-to authentication. In *30th USENIX Security Symposium (USENIX) Security 21*, 2021.

[18] Niranjan Hasabnis, Ashish Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 135–144, New York, NY, USA, March 2012. ACM.

[19] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*, 1991.

[20] 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*, pages 275–288. USENIX Association, 2002.

[21] Richard W M Jones and Paul H J Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic and Algorithmic Debugging*, AADEBUG '97. Linköping University Electronic Press, 1997.

[22] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 147–163, Berkeley, CA, USA, October 2014. USENIX Association.

[23] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated software diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, S&P '14. IEEE, May 2014.

[24] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, NDSS '15. Internet Society, February 2015.

[25] Hans Liljestrand, Zaheer Gauhar, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. Protecting the stack with paced canaries. In *Proceedings of the 4th Workshop on System Software for Trusted Execution*, SysTEX '19, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Hans Liljestrand, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. Authenticated call stack. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[27] 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, Santa Clara, CA, August 2019. USENIX Association.

[28] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, June 2009. ACM.

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40, New York, NY, USA, June 2010. ACM.

[30] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 128–139, New York, NY, USA, January 2002. ACM.

[31] Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software: Practics and Experience*, 27(1), January 1997.

[32] Bruce Perens. Electric fence malloc debugger. https://linux.die.net/man/3/libefence. Accessed: 2022-01-26.

[33] Qualcomm Technologies, Inc. Pointer Authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf. Accessed: 2022-01-25.

[34] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, NDSS '04. Internet Society, February 2004.

[35] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, ATC '12, pages 309–318, Boston, MA, 2012. USENIX.

[36] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory tagging and how it improves C/C++ memory safety, February 2018.

[37] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, ATC '05. USENIX Association, April 2005.

[38] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for security. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, S&P '19, Los Alamitos, CA, USA, May 2019. IEEE.

[39] Joseph L. Steffen. Adding run-time checking to the portable C compiler. *Software: Practice and Experience*, 22(4), 1992.

[40] The Clang Team. Clang documentation - control flow integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html. Accessed: 2022-01-25.

[41] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*, SEC '14, pages 941–955, Berkeley, CA, USA, August 2014. USENIX Association.

[42] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. DangSan: Scalable Use-after-free Detection. In *Proceedings of the 12th European Conference on Computer Systems*, EuroSys '17, pages 405–419, New York, NY, USA, 2017. ACM.

[43] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software. In *Proceedings of the 28th USENIX Security Symposium*, SEC '19, pages 1805–1821. USENIX Association, August 2019.

[44] Yves Younan. FreeSentry: Protecting against use-after-free vulnerabilities due to dangling pointers. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, NDSS '15. Internet Society, February 2015.

[45] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. PAriCheck: An efficient pointer arithmetic checker for C programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 145–156, New York, NY, USA, April 2010. ACM.

A SPEC CPU 2017 Benchmarks Compatibility

Table 4: PACSafe prototype compatibility with SPEC CPU 2017 benchmarks

Benchmark	Compatible	Reason
500.perlbench_r	-	1, 2
502.gcc_r	-	1, 2
505.mcf_r	✓	-
508.namd_r	-	3
510.parest_r	-	3
511.povray_r	-	3
519.lbm_r	✓	-
520.omnetpp_r	-	3
523.xalancbmk_r	-	3
525.x264_r	-	2
526.blender_r	-	4
531.deepsjeng_r	-	2
538.imagick_r	-	2
541.leela_r	-	2
544.nab_r	✓	-
557.xz_r	✓	-
999.specrand_r	✓	-

1 Non-local jumps
2 External functions with `va_list`
3 Incompatible virtual methods
4 Missing arm64e macOS library

Table 4 lists the SPEC CPU 2017 C/C++ refrate benchmarks and reasons for possible incompatibility with our PACSafe prototype on the MacBook Pro M1 2020 testing setup. Out of the 17 benchmarks, 6 are incompatible with PACSafe (reasons **1** and **2**, see Section 6.3), 6 are incompatible with the testing setup using LLVM 12 even without PACSafe (reasons **3** and **4**), and 5 could be evaluated.

Benchmarks incompatible due to reason **3** contain C++ virtual methods whose declarations differ from their definitions causing them to crash on execution when compiled with LLVM 12 for macOS with pointer authentication enabled. This problem is also mentioned by Apple in [3].

The blender_r benchmark has a dependency to the `__divsc3` function, which is not available in the arm64e system library headers, and could not easily be installed.