TailCheck: A Lightweight Heap Overflow Detection Mechanism with Page Protection and Tagged Pointers

Amogha Udupa Shankaranarayana Gopal, Raveendra Soori, Michael Ferdman, and Dongyoon Lee, *Stony Brook University*

https://www.usenix.org/conference/osdi23/presentation/gopal

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

TailCheck: A Lightweight Heap Overflow Detection Mechanism with Page Protection and Tagged Pointers

Amogha Udupa Shankaranarayana Gopal   Raveendra Soori   Michael Ferdman   Dongyoon Lee

Stony Brook University

Abstract

A heap overflow vulnerability occurs when a program written in an unmanaged language such as C or C++ accesses a memory location beyond an object allocation boundary. Malicious users may exploit this vulnerability to corrupt an adjacent object in memory, creating an entry point for a security attack. Despite decades of research, unfortunately, it still remains challenging to detect heap overflow vulnerabilities in real-world programs at a low cost.

We present TailCheck, a new lightweight heap overflow detection scheme that leverages page protection and pointer tagging. When an object is created, TailCheck allocates an additional page-protected shadow object, called a TailObject, placing the distance from the object to its TailObject as a tag stored in the unused high-order bits of the object pointer. For every access to the original object, TailCheck performs an additional memory access to the TailObject, whose address is computed using the tag. Heap overflows are detected as page faults when an access occurs beyond the TailObject. We evaluated TailCheck with four server applications (apache, nginx, memcached, redis) and the SPEC CPU2017 and SPEC CPU2006 benchmarks, successfully finding heap overflows in SPEC CPU2017 gcc. TailCheck experiences 4% and 3% run-time overhead for the average and tail (99%) latencies for server applications; and only 33% and 29% run-time overhead for SPEC CPU2017 and SPEC CPU2006, respectively, less than the state-of-the-art solution.

1 Introduction

A heap overflow [47,48] is an anomaly that occurs when a program attempts to access a memory location beyond the bounds of its allocated memory. This type of vulnerability is commonly found in programs written in unmanaged languages such as C and C++, as these languages allow programmers to directly manipulate pointers without providing compile-time (*e.g.,* as in Rust) or run-time protection (*e.g.,* as in Java or Go). A malicious user may exploit a heap overflow vulnerability in a C or C++ program to perform a variety of security attacks [57,58], including corrupting code pointers to divert control flow or leaking sensitive information.

Today, many critical software systems—such as server applications and operating systems—are developed in unsafe languages. Programming errors in these systems can therefore lead to heap overflow exploitation. For example, a vulnerability in the `nginx` web server (`CVE-2014-0133`) allowed attackers to send a specially-crafted request that caused a heap overflow, allowing them to execute arbitrary code on the server. The `mysql` database code had a vulnerability (`CVE-2021-2429`) which allowed attackers to send a specially-crafted request that caused a heap overflow, potentially gaining access to the data or taking control of the database. A heap overflow in the PHP programming language related to encryption (`CVE-2022-37454`) could be used to remotely execute arbitrary code on a web server. The prevalence of heap overflow vulnerabilities in deployed software systems highlights the need for effective run-time techniques to protect production systems against exploitation, even when running vulnerable software.

Several systems have made significant strides toward run-time mitigation of heap overflows. AddressSanitizer [54], the state-of-the-practice solution, incurs high run-time overhead: 80% (geometric mean) slowdown for SPEC CPU2006 applications [45]. Modern operating systems offer heap overflow protection by allocating an object at the boundary of a virtual memory page and adding a protected page (with no access permission) after it; this feature is available in Linux as Electric Fence [49] and in Windows as PageHeap [61]. However, allocating just one object per protected page suffers from extremely large memory overhead, along with high run-time cost due to frequent TLB misses. Delta Pointers [28], the state-of-the-art technique, achieves the lowest run-time overhead (35% for SPEC CPU2006), but requires restricting the address space of the protected application. Delta Pointers reserves the N most significant bits (32, by default) for pointer tagging and supports only a $48-N$ bit address space for 64-bit architectures. This limits Delta Pointers' applicability for modern software: it cannot be used for server software with many-gigabyte footprints and even fails for the reference inputs of xz and mcf in the SPEC CPU2017 suite.

In this work, we present TAILCHECK, a new lightweight heap overflow detection scheme that leverages a custom memory allocator, OS-based page protection, and compiler-directed pointer tagging. When an object is created, TAILCHECK allocates an additional shadow object, called a TailObject, at the boundary of a page whose subsequent page is protected by the OS. The TAILCHECK memory allocator returns a tagged pointer in which the otherwise unused most significant bits (*e.g.,* 16 bits for a 64-bit architecture with 48-bit address space) encode the distance from the original object to its TailObject, keeping the address of the original object unmodified in the low-order bits as usual. A TAILCHECK compiler pass instruments each dereference of a tagged pointer, using the embedded tag to compute the shadow address within the corresponding TailObject and inserting an additional memory access to the shadow address alongside each access to the original object. In the event of a heap overflow, the shadow memory accesses reach beyond the bounds of the TailObject, causing a page fault and triggering the OS to terminate the program. This prevents the exploitation of heap overflow vulnerabilities (both over-writes and over-reads) and ensures the integrity and confidentiality of the system.

The TAILCHECK tags allow many objects to share space used by the TailObjects and the OS-protected pages, limiting the memory overhead of the technique and eliminating the performance overheads of frequent system calls to protect pages during memory allocation. To further reduce run-time overhead, TAILCHECK performs three compile-time optimizations to prune the shadow accesses for heap accesses that are statically proven to be safe.

TAILCHECK makes use of well-known page protection and pointer tagging techniques, yet it does not share the limitations of prior solutions. TAILCHECK achieves low run-time overhead by using page protection for heap overflow detection, but unlike Electric Fence and PageHeap, it allows multiple small objects to be co-located on a virtual memory page. TAILCHECK uses pointer tagging, but unlike Delta Pointers, it allows a program to utilize the full address space by only re-purposing the otherwise unused most significant bits.

We implemented TAILCHECK by extending the *mimalloc* allocator [33] and developing LLVM [31] compiler passes for code instrumentation. We evaluated TAILCHECK with four server applications (apache, nginx, memached, redis) and the SPEC CPU2017 and SPEC CPU2006 benchmark suites. Interestingly, TAILCHECK identified an out-of-bounds read in SPEC CPU2017 gcc (v4.5.0), a known bug with an available patch [1], yet the patch is not present in SPEC CPU2017 v1.0.5. For performance, TAILCHECK experiences 4% and 3% run-time overhead for the average and 99% tail latencies for server applications. TAILCHECK exhibits 33% (geometric mean) run-time overhead and 3% memory overhead for SPEC CPU2017. TAILCHECK exhibits 29% (geometric mean) run-time overhead for SPEC CPU2006, lower than Delta Pointers, the state-of-the-art compiler-based solution with the lowest previously-reported run-time overhead (35%).

This paper makes the following contributions:

- To the best of our knowledge, TAILCHECK is the first lightweight heap overflow detection scheme based on page protection that does not place one object per page.
- TAILCHECK introduces a new pointer tagging scheme for heap overflow detection, which encodes distance metadata only in the otherwise unused pointer bits and thus does not restrict the application address space.
- An evaluation of TAILCHECK demonstrates that it incurs low run-time and memory overheads and supports applications with large many-gigabyte memory requirements.

2 Background & Motivation

This section briefly describes the background on heap overflows, discusses the threat model we assume in this work, and highlights the need for a new solution.

2.1 A Heap Overflow Vulnerability

The lack of run-time and compile-time heap overflow protection in C and C++ exposes many critical software systems to security threats. Stack-based buffer overflows have received significant early attention from both academia and industry. Mature mitigations using stack canaries [11] and shadow stacks [60] are readily available: for example, GCC and Clang have built-in support with the -fstack-protector and -fsanitize=safe-stack compiler flags. On the contrary, standard solutions for heap overflows have not yet been settled, with solutions offering trade-offs in run-time and memory overheads, soundness, and completeness (§2.3).

Heap overflows are responsible for many critical real-world security problems. A heap overflow *over-write* is particularly critical as it may allow malicious users to divert the control flow of a victim program or gain privilege escalation. For instance, a heap overflow over-write vulnerability is found in sudo (CVE-2021-3156), a widely-used utility on Unix-like operating systems, which enables a user to run programs with the security privileges of another user. This heap overflow was particularly critical in that an attacker could control not only the size of the buffer that can be overflowed but also the size and contents of the overflow. As a result, when exploited, the vulnerability could allow an unprivileged malicious user to gain root privileges on a vulnerable host.

A heap overflow *over-read* can lead to information leakage. The Heartbleed [19] vulnerability in the popular OpenSSL cryptographic software library (CVE-2014-0160) is a representative example. A missing bounds check in the SSL/TLS heartbeat extension could be exploited to reveal up to 64KB of memory, which may include private keys and other secrets.

2.2 Threat Model

In this work, we address the threat of overflows on heap-allocated objects. We assume an attacker can feed a crafted malicious input to a victim program to exploit a heap overflow vulnerability. We mitigate heap overflows (both over-write and over-read) that occur in application and library code that can be instrumented with our tool, providing integrity and confidentiality when the underlying software contains vulnerable code. We provide no protection for uninstrumented code such as third-party libraries. We do not consider other memory safety violations such as use-after-free or uninitialized read vulnerabilities.

2.3 Motivation: Haven't we solved it yet?

Given the critical implication for security, many solutions have been proposed for mitigating heap overflows. We present several representative works, discussing their limitations and the lessons that we draw upon when designing TAILCHECK. A more comprehensive related work discussion follows in §8.

The idea of leveraging a virtual memory *protected page* to detect a heap overflow dates back to 1987. Electric Fence [49], proposed by Perens and now included in Linux, was the first to place allocated objects immediately before protected pages, which are configured by the OS to trigger a hardware page fault when accessed. Reads or writes beyond the allocated object would land on the protected page, triggering a fault and allowing the OS to mitigate the heap overflow. Successors to Electric Fence, including DUMA [5], DYBOC [56], OSX's libgmalloc [35], and Windows' PageHeap [61] follow a similar design. However, despite its simplicity, the approach of allocating one heap object per virtual memory page has unacceptable memory overhead. Moreover, this approach incurs large run-time overheads from multiple sources: performing system calls to protect a page on every heap allocation is expensive, spreading heap allocations across many pages results in excessive TLB misses, and placing objects at common offsets from the end of memory pages increases data cache contention. For example, Liu *et al.* [36] reports Electric Fence incurs a 7x slowdown for the PARSEC benchmarks [7]. As a result, the idea of using protected pages for run-time heap overflow mitigation has lost its attraction and is rarely found outside of debugging environments. Using protected pages can offer heap overflow protection without explicit metadata lookups and bounds checks, yet require a new solution that supports placing multiple objects per virtual memory page and avoids frequent page protection system calls.

AddressSanitizer (ASan) [54] is an alternative approach with lower run-time overhead. ASan manages a fine-grained inaccessible region called a *redzone* after each allocated object by maintaining a disjoint shadow (metadata) memory space. On each memory access, ASan looks up the metadata space and checks if the target location falls in a redzone. Other prior solutions, notably SoftBound [40], keep *base and bound* metadata in a shadow memory space. On each memory access, SoftBound performs a metadata lookup and explicitly checks that the instrumented access falls within the object bounds. Although the implementation details differ across these systems, they all share two downsides. First, the metadata lookup (comprising additional shift, add, and load instructions) and bounds check (including comparison and branching instructions) have significant run-time overhead. Second, the redzones and metadata store incur significant space overheads. For example, for the SPEC CPU2006 benchmarks, Oleksenko *et al.* [45] report 1.8x run-time and 4x memory overheads for ASan, and 2x run-time and 3x memory overheads for SoftBound. Such performance overheads and memory capacity requirements are unacceptably high for modern large-memory performance critical applications which would most benefit from run-time heap overflow protection.

To reduce the metadata overheads and/or to facilitate metadata lookup, researchers proposed *pointer tagging* techniques that keep metadata in the high-order bits of a pointer itself (*e.g.,* unused 16 bits in 64-bit architecture). For instance, Baggy Bounds [2] tags a pointer with the encoded size of an object. However, Baggy Bounds still requires expensive array table look-ups and bounds checking on pointer arithmetic. Taking one step further, Delta Pointers [28] remove the bounds check (comparison and branching instructions) by transforming the heap overflow detection problem into an integer overflow detection problem, managing pointer tags in a way that would cause out-of-bound pointer arithmetic to set an overflow bit in the tags, thereby making such pointers "uncanonical" and causing the MMU to generate a fault on dereference. Delta Pointers are considered the state-of-the-art software-only solution based on its low run-time overhead, exhibiting only 35% slowdown for SPEC CPU2006 benchmarks. However, Delta Pointer tags must include the distance from the current pointer to the end of an object, requiring significantly more than 16 bits for large objects. To work around this limitation, Delta Pointers shrink the process address space. By default, Delta Pointers use a 32-32 bit split, supporting applications with up to 4GB of address space. Delta Pointers offer a glimpse of a low-overhead solution without metadata lookups or bounds checks, but are still limited in terms of practicality due to its address space restrictions.

3 TAILCHECK Design

TAILCHECK extends the memory allocator and compiler to produce executables that are protected against heap overflow exploitation at run-time. Whenever a programming error leads to an access that overruns the end of a heap-allocated object, TAILCHECK ensures that a hardware page fault is triggered, causing the operating system to mitigate a potential attack by trapping the fault. Although this effect can be achieved by placing each heap object at the end of its own virtual memory

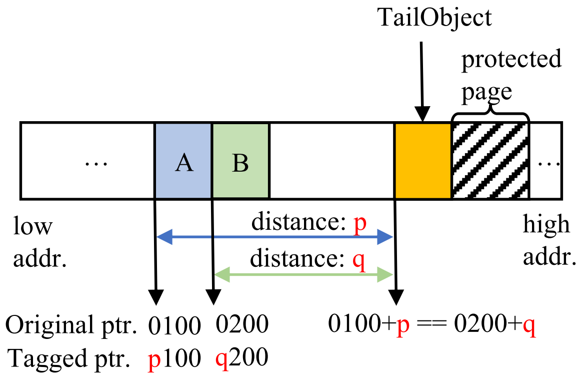

Figure 1: TailCheck allocates a (shadow) TailObject at the boundary of a protected page and tags object pointers with the distance between the original and shadow objects.

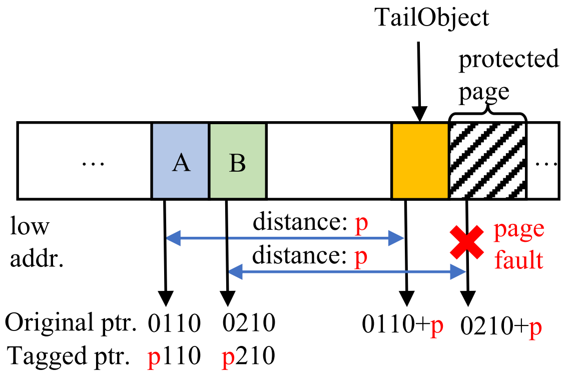

Figure 2: TailCheck adds shadow memory access for the TailObject. A heap overflow beyond object A is detected as a page fault on the shadow access beyond its TailObject.

page (followed by a protected page) [49], such an approach is prohibitively expensive; allocating a virtual memory page for each heap object results in massive heap capacity bloat, while performance is severely impacted at the time of allocation (due to configuring a protected page on each allocation) and also at the time of use of the allocated object (due to a significant increase in TLB and cache pressure).

Rather than coupling each heap object with its own protected page, TailCheck reserves one *TailObject* and a protected page per memory region managed by the heap allocator. Figure 1 depicts this arrangement. When the TailCheck memory allocator requests a region of memory from the OS, it reserves space for the *TailObject* and configures a protected page at the end of the region. After this, the allocator places dynamically allocated heap objects, such as A and B, as usual.

To trigger a page fault on a heap overflow, the TailCheck compiler instruments the application code to perform a shadow memory access to the *TailObject* alongside each *load* and *store* operation to the original heap object. Figure 1 shows that the base address of object A is a known distance p away from the base address of its *TailObject*. The base address of object B is similarly a known distance q from its *TailObject*. To compute the address for the shadow memory access, the compiler simply adds the offset (p or q in this example) to the address of the original access, as shown in Figure 2. Although an access beyond object A (*e.g.,* to the address `p210`) would erroneously read or write data belonging to object B, the corresponding shadow access (*e.g.,* to the address `0210+p`) that TailCheck performs before the original object access will exceed the bounds of the *TailObject*, attempting to access the protected page and triggering a page fault.

To store the distances (such as p and q) from the moment when heap objects are allocated to the time when they are accessed, TailCheck uses tagged pointers. We leverage the otherwise unused high-order bits of pointers to store the distances, using the compiler to emit code that masks these high-order bits before performing an access to the original object and adds the distance encoded in these bits to compute the address of the shadow access. Modern x86 and ARM systems use a 48-bit address space, leaving 16 unused bits in 64-bit pointers, thereby allowing TailCheck to store distances for allocator regions of up to 64KB. A key advantage of this approach is that the distances encoded in the tagged pointers are propagated implicitly, requiring code instrumentation only at the time of pointer dereference or comparison. Notably, TailCheck still protects large heap objects allocated within their own allocator regions by using protected pages and setting the distance for shadow accesses to 0, therefore treating all accesses uniformly, but incurring a small overhead due to the duplication of memory accesses to the large objects.

We note that reserving space for the *TailObjects* is necessary because shadow memory accesses for stores write data into the TailObject space. Although these data are never used, if the space were not reserved, the application could allocate an unrelated object in the same space (at the end of the managed region), causing the shadow stores to clobber that object. An alternative implementation, which uses shadow loads for the corresponding original object stores, is possible. At first glance, this arrangement would eliminate the memory overheads of TailCheck. However, this approach can have significant performance drawbacks because loads are on the critical path of the processor pipeline and have a higher execution cost compared to stores.

3.1 TailCheck Code Instrumentation

We use the compiler to add TailCheck to executables. The compiler performs three tasks: it replaces memory allocation calls with the TailCheck allocator, adds shadow memory accesses at pointer dereference sites, and masks pointer tags when interacting with uninstrumented code (e.g., shared libraries). We detail these tasks below.

Allocator Injection. TAILCHECK uses a custom memory allocator. Unlike regular allocator functions that return a pointer carrying the address of the allocation in virtual memory, the TAILCHECK allocator functions return a tagged pointer. The tag is inserted into the otherwise unused high-order bits of the returned value, and corresponds to the distance between the address of the allocated heap object and the address of its corresponding TailObject.

Replacing the memory allocator provided by the standard library is traditionally done with the dynamic linker. However, TAILCHECK needs its custom allocator only for the heap objects that it protects. To inject its custom allocator during code instrumentation, the TAILCHECK compiler identifies allocator calls (*e.g.,* `malloc`, `new`, `strdup`, etc.) and replaces them with their TAILCHECK allocator equivalents. Any code linked into the executable, but not instrumented with TAILCHECK, continues to use the unmodified system allocator.

For each allocator-managed memory region requested from the OS by the TAILCHECK allocator, we reserve space for the TailObjects at the end of the region and `mprotect` the virtual memory page immediately following the region. TAILCHECK requires for the TailObject reservation at the end of the managed memory region to be as large as the largest object allocated within that region. In practice, it is common for modern allocators to separate memory regions by *size class*, dividing each region into slots of this size and allocating one object per slot. With this strategy, the TAILCHECK allocator can simply reserve the last slot within each managed memory region. Notably, the distance encoded in the pointer tags is computed for the TailObject address equal to *address_of_protected_page minus size_of_allocated_object*, which lands in the middle of the reserved slot whenever the allocated object size is smaller than the size class, or lands at the start of the reserved slot when the allocated object size equals the size class.

When the requested allocation size is larger than the largest allocator size class (*e.g.,* one that requires multiple virtual memory pages), modern allocators switch to a large-object allocation mode where a separate memory region is requested from the OS. When handling large-object requests, the TAILCHECK allocator will `mprotect` a page immediately following each allocated large object and compute the returned pointer as *address_of_protected_page minus size_of_allocated_object*, effectively falling back to the behavior of Electric Fence. For such large-object allocations, the tag of the returned pointer is set to 0.

We note that, when computing TailObject addresses, we round up the *size_of_allocated_object* to honor the original object's memory alignment requirements. This is useful for both correctness and performance, as we want the shadow access instructions to have the same alignment properties as their corresponding original object access instructions. As a result of this rounding, objects whose requested size is not a multiple of their alignment size will have a small (several bytes) region where heap overflows may go undetected with our TAILCHECK implementation. We discuss the generally benign nature of such overflows and the ramifications of adding support for precise overflow detection independent of alignment constraints in §7.2.

Memory Access Instrumentation. The TAILCHECK compiler operates at *module* granularity, treating all pointers local to a module as tagged pointers. On every pointer dereference in the instrumented code, a tagged pointer must be deconstructed into two parts, the object address (by masking the tag) and the shadow access address (by adding the tag to the object address). A compiler pass iterates over the pointer dereferences, inserting compiler IR for tag handling and injecting the TailObject shadow accesses. Each shadow access (load or store) is performed first, immediately followed by the original object access. For both loads and stores, the same store value and load target registers can be used by the two accesses, avoiding tying up additional register resources for the shadow accesses. Compiler optimization passes are performed both before and after the TAILCHECK instrumentation pass, ensuring that all dereferences eliminated by the optimizer are not instrumented and that the address calculation code for handling the tagged pointers is optimized. The shadow accesses are marked as volatile to ensure that they are not moved or eliminated as dead code.

Pointer arithmetic with integers does not require special handling for tagged pointers. However, whenever a tagged pointer is compared (either to another tagged pointer or to `NULL`), the TAILCHECK compiler must mask the tag bits prior to the comparison. Similar to the case of pointer dereference instrumentation, comparison tag manipulation logic is inserted as compiler IR, allowing an optimization pass to minimize the overhead of these operations.

We note that memory access instrumentation does not differentiate between objects allocated with the regular and large-object modes. All pointers within the instrumented code are treated as tagged pointers. For large-object allocations, the tags are set to 0 by the memory allocator, resulting in minor overhead for tag manipulation and harmless shadow accesses that load or store exactly the same address as the original access. Uniform handling of tagged pointers simplifies the implementation, while the overhead of back-to-back instructions that access the same cache line is minimal in modern superscalar out-of-order processors.

Linking with Uninstrumented Code. Although it is theoretically possible to compile all modules of a program with TAILCHECK instrumentation, in practice, we must provide the ability to link against uninstrumented modules, such as shared libraries. In these situations, pointers passed by instrumented code into uninstrumented code (*e.g.,* library function calls that have pointers in their arguments) must have the pointer tags removed. Even if all libraries, including the standard library, are instrumented with TAILCHECK, there are still situations where pointers are passed as arguments to system calls, also requiring the stripping of tags.

The TAILCHECK compiler performs a pass over the instrumented module to identify all function call sites. Calls to functions within the same module receive unmodified tagged pointers as arguments. However, for calls to external functions, compiler IR is inserted to mask the tags of all pointer arguments. Notably, it is safe to pass function pointers of instrumented functions to uninstrumented code (*e.g.,* as callbacks), as any pointer arguments passed by uninstrumented code into these functions will have tags set to 0 and will execute correctly, albeit with harmless duplicated memory accesses as in the case of the large-object allocations.

TAILCHECK keeps pointers without tags in globals because they may be accessed by uninstrumented libraries. To this end, TAILCHECK identifies all pointer stores to global variables in the instrumented code and ensures that the values written into these variables are masked prior to being stored. The TAILCHECK compiler uses the LLVM instruction operand type `GlobalVariable` to identify globals (after calling `stripPointerCasts` on the operand). There is a possibility that TAILCHECK may store a tagged pointer into a global if a program uses a local alias to write to that global, potentially leading uninstrumented code to later dereference a tagged pointer stored in the global. However, we observe that accessing a global variable via a local pointer alias is uncommon in practice: during our evaluation (§6), none of the tested server, SPEC CPU 2017, or SPEC CPU 2006 applications shows any unexpected behavior (*e.g.,* segmentation fault), which would happen if uninstrumented libraries accessed tagged pointers in globals. Thus TAILCHECK does not perform additional pointer alias (provenance) analysis. Furthermore, we treat the *environ* variable as a special case where not only the variable itself, but also the nested pointers within its structure, are written with their tags masked.

Finally, if the standard library is not instrumented with TAILCHECK, there are two classes of commonly used functions (`mem*` and `str*`) that can benefit from special handling, following the practices of previous works [28,29,45]. These functions are often responsible for heap overflows, making it practical to insert bounds checks at their call sites when their function bodies (part of the standard library) are not instrumented. For the *MemIntrinsics* functions (`memcpy`, `memmove`, and `memset`), we inject a TAILCHECK-like bounds check by performing a shadow access to the last byte of the arrays passed as arguments. Notably, we must first check that the *size* argument is non-zero, as the function semantics dictate that no pointer dereferences occur if the size is zero. The common string manipulation functions (*e.g.,* `strstr`, `strchr`, etc.) return pointers. Although instrumented code handles these functions correctly (treating them like other 0-tag pointers), calling these functions effectively removes TAILCHECK protection from the pointers passed to them. To avoid losing heap overflow protection after these function calls, TAILCHECK instruments the call sites of these functions to save the tags of the pointer arguments before the call and re-applies the tags on the returned pointers. Similarly, TAILCHECK masks tags in the return values of those functions that convert a string to a number (*e.g.,* `strtol`) when used in arithmetic operations.

Mixing Memory Allocators Having the same allocator in the application and shared libraries is not a requirement for TAILCHECK. In our setup, we use `LD_LIBRARY_PATH` to ensure that all linked libraries use the TAILCHECK memory allocator, primarily to maintain performance consistency across all experiments. However, it is worth noting that libraries cannot and should not call `free()` on objects they did not allocate themselves [53]. If application code includes such a construct, it would cause failures in many scenarios (e.g., where custom allocators are used), including with the TAILCHECK allocator.

Custom Memory Allocators By default, TAILCHECK protects heap objects that are allocated and deallocated via standard interfaces (*e.g.,* `malloc`, `realloc`, and `free`), replaced by `LD_LIBRARY_PATH`. Thus, there could be a heap protection granularity mismatch if an application uses a custom allocator. For example, for an application-level pool (slab) allocator, TAILCHECK may protect a `malloc`-allocated pool at a coarse granularity, not at the fine-grained custom allocation granularity. Our `nginx` server evaluation (§6) compares two cases with and without application-level pool allocations (by disabling a pool allocator using a debugging flag).

3.2 TAILCHECK Optimizations

To reduce the performance overheads of TAILCHECK, we apply several compiler IR optimizations that reduce the cost of instrumentation. We detail these optimizations below.

Merging Tag Handling. The tags of TAILCHECK tagged pointers remain constant throughout the lifetime of the pointer. When the same pointer is dereferenced multiple times within a function, potentially with different offsets (for accessing different members of the heap object), the operations to compute the TailObject pointers are redundant. To reduce the overhead of this common case, the TAILCHECK compiler instrumentation pass keeps track of the computed TailObject pointers and reuses their already computed values.

Hoisting Tag Handling. Heap pointers are frequently dereferenced inside loops, accessing different locations within the same heap object (*i.e.,* if the object is an array). To reduce the overhead of tag handling, we hoist the computation of the TailObject pointer outside of the loop, leaving only the dereference operations inside the loop body. As a result of this optimization, the TailObject accesses within the loop body exactly mimic the original object accesses, including using the same x86 scale-index-base-displacement for the shadow memory access and pushing all other TAILCHECK instrumentation overheads outside of the loop body.

Statically Safe Dereferences. TAILCHECK is effective at preventing heap overflow exploitation with relatively low overheads at run-time. However, while many pointer dereferences must be verified (*e.g.,* using shadow accesses to the TailObjects), some of the checks are unnecessary because static analysis of the code can guarantee that all accesses remain within the bounds of a heap object. As such, to further reduce the overhead of TAILCHECK, we adopt the *SafeAllocs* [28] static analysis implementation from the Delta Pointers work.

SafeAllocs identifies all heap allocations with statically known sizes and uses the compiler metadata to track object bounds along with the pointer corresponding pointer. Whenever such pointers are dereferenced in the code, the compiler checks if the offset of the dereference can be statistically determined and, if it can be determined and falls within the object bounds, a run-time check is unnecessary.

When SafeAlloc indicates that all accesses to a heap object are known to be safe at compile time, TAILCHECK uses the standard memory allocator for these objects and does not introduce shadow accesses for them. Some heap objects have both dereference sites that are known to be safe and also dereference sites that must be checked at run-time. We statically identify the safe regions at function granularity, avoiding shadow accesses for objects whose accesses are known to be safe. This also requires masking the tag bits of these pointers in the function preambles, as these objects are still allocated using the TAILCHECK custom allocator and the function call sites continue to pass arguments as tagged pointers.

4 TAILCHECK Implementation Details

We develop the TAILCHECK prototype by extending the *mimalloc* allocator [33] and developing LLVM [31] compiler passes for code instrumentation. Tagged pointers are returned by the TAILCHECK allocator for allocations up to 16KB, with all larger requests treated as large-object allocations.

The TAILCHECK instrumentation is performed using three compiler passes. First, a SafeAllocs pass is done to identify optimization opportunities. Then a Call-Site Instrumentation pass replaces memory allocation function calls (`malloc`, `calloc`, `realloc`, `strdup`, `strndup`, and `free`) with the TAILCHECK custom allocator versions of these functions and masks pointer arguments at call sites of external functions. The Dereference Instrumentation pass inserts shadow loads and stores to the TailObjects for all heap objects requiring run-time checks. These passes are performed as part of the link-time optimization, ensuring that all statically linked sub-modules are combined together into one module for TAILCHECK instrumentation before the passes are performed. Standard LLVM compiler optimization passes are performed both before and after the TAILCHECK passes.

We take special care to handle function arguments with the byval attribute. In LLVM, the byval attribute at a call site means that the pointer must be dereferenced and the resulting value copied before being passed as an argument. Because the LLVM byval mechanisms cannot handle tagged pointers, we mask the tags of all pointers with the byval attribute.

All tagged pointer-based solutions present challenges when linking to uninstrumented libraries, as tagged pointers must be masked before being passed to functions in uninstrumented code. Although pointers to data structures have their tags masked at the function call sites by the Call-Site Instrumentation pass, the nested pointers within these data structures are written as tagged pointers by the TAILCHECK instrumented code and cannot be directly dereferenced by the uninstrumented functions. As in prior work [2, 8, 28], we assume that we can soundly enumerate all call sites of external uninstrumented functions that will operate on nested pointers, and inject the necessary instrumentation code to mask nested tagged pointers. Notably, most C++ Standard Template Library (STL) containers do not require masking of nested pointers because they are implemented in header files and thus come within our instrumentation scope. For the select cases we encountered in our benchmark applications that require masking, we manually add the appropriate instrumentation as discussed in §5. In §7.3 we discuss how TAILCHECK may take advantage of the ARM top-byte-ignore Memory Tagging Extension (MTE) [3] and similar features in other ISAs to avoid the need for explicitly masking pointer tags.

5 Evaluation Methodology

We conduct all experiments on a system with an Intel Xeon Gold 5218 CPU. To benchmark TAILCHECK, we use four popular server applications (*apache* v2.4.54, *nginx* v1.22.1, *memcached* v1.6.17, *redis* v7.0.6), as well as the C and C++ applications from the SPEC CPU2017 and SPEC CPU2006 benchmark suites. For SPEC CPU2017, we use the *speed* set and limit applications to one thread.

Server applications often have a custom pool-based allocator. To evaluate potential performance differences between coarse-grained and fine-grained memory allocations, in addition to the *nginx* server results, we also present "*nginx (w/o poolalloc)*," which is compiled with the `-DNGX_DEBUG_PALLOC=1` flag to disable its custom pool allocator and to use `malloc` and `free` directly. *apache* (v2.4.54) and *memcached* (v1.6.17) do not provide similar pool allocation on/off options, while *redis* does not use pool allocation.

To quantify the performance of the web servers *apache*, *nginx*, and *nginx (w/o poolalloc)*, we measure request latency using the `hey` HTTP load generator [16]. We create two workers to repeatedly request a file 256 times per second. We test four different file sizes: 32KB, 128KB, 512KB, and 2MB. We configure apache with two worker threads and nginx with one worker process. For the key value stores, memcached and redis, we measure the request latency with four workers, each requesting 128,000 keys with a 50% get/set ratio. We use a key size of 16 bytes and four different object sizes: 32B, 128B, 512B, and 2KB. For the SPEC CPU2017 and SPEC CPU2006 benchmarks, we measure performance with reference input as wall-clock time of program execution.

For a fair comparison across all systems, we use unmodified *mimalloc* [33] for all evaluated configurations except TAILCHECK. For TAILCHECK, we use unmodified mimalloc for the uninstrumented code and only extend the mimalloc functionality with wrappers for the allocation functions, retaining all of the core functionality of the mimalloc allocator even when called from the instrumented code. The memory overheads we report are measured as peak resident set size.

As part of our evaluation, we include a comparison to Delta Pointers [28] and AddressSanitizer [54]. To ensure fairness, we reproduce the Delta Pointers results in our test environment after enabling only the comparable heap overflow protection features and ensuring that all available optimizations are applied. To make the results directly comparable, we perform this study with the same SPEC CPU2006 benchmark suite that was used in the original Delta Pointers publication. AddressSanitizer is compared for the server applications.

TAILCHECK works for all SPEC CPU 2017 benchmarks using LLVM -O3 with Link Time Optimization (LTO). However, we use -O2 and LTO in our evaluation to make the results directly comparable to the prior work [28]. We introduced specialized handling for the following benchmark applications to address compatibility issues with uninstrumented libraries:

- In the case of 403.gcc, pointers stored in “long long” variables are passed to functions invoked through function pointers. Consequently, an uninstrumented libc function is called with a long long argument containing a tagged pointer. This causes a segmentation fault in the uninstrumented code, with the faulting address being a tagged pointer. Debugging this situation is straightforward, as the stack trace directly points to the problem. To address this, we utilized source instrumentation and manual pointer tag masking in the benchmark sources, similar to techniques applied in previous works [28, 29, 45].
- 520.omnetpp employs a C++ data structure, evbuf, inherited from basic_stringbuf. This object contains a nested tagged pointer, whose information is lost due to C++ inheritance, leading to a tagged pointer being passed to a libstdc++ function. This triggers a segmentation fault, easily identified by the faulting tagged pointer. To overcome this, we explicitly marked the evbuf type to ensure its members are always written as untagged pointers, thereby maintaining the integrity of the passed pointer irrespective of inheritance nuances.

Except for the above two cases, TAILCHECK is compatible with many complex real-world applications, including four servers and all other SPEC CPU 2017 and SPEC CPU 2006 applications. We note that although the two exception cases were easily identifiable and debuggable because they triggered a segmentation fault, it is theoretically possible that a tagged pointer may lead to silent data corruption and exhibit an observable event far later in time. TAILCHECK provides limited support for such cases, and fixing them may require manual code reviews. Indeed, addressing compatibility issues with uninstrumented libraries is a common limitation of pointer tagging-based solutions [2, 8, 28] with a notable exception LowFat [30] (see related work discussion in §8.2).

6 Evaluation Results

Below, we first describe the heap overflow vulnerabilities that were successfully caught by TAILCHECK. We then present the performance and memory overheads of our technique, and explain the impact of the optimizations described in §3.2 which mitigate some of the performance impacts. Finally, we present a comparison with the prior art, demonstrating comparable and lower overheads compared to Delta Pointers, without being subject to its address space limitations.

6.1 Heap Overflow Detection

We developed a set of test cases that exhibit various types of heap overflows to ensure that a segmentation fault is experienced when running such cases when the code is instrumented with TAILCHECK. The cases were drawn from prior empirical studies [36, 65] that analyzed the types and frequencies of heap overflows in 85 CVEs. For example, our test suite includes the following cases:

- Loop accessing heap-allocated arrays, representing 35/85 studied CVE cases (41%).
- memcpy(), memset(), or memmove() into an insufficiently large buffer; 18/85 cases (21%).
- strncpy(), strncmp(), or sprintf() into an insufficiently large buffer; 6/85 cases (7%).
- Incorrect pointer arithmetic; 8/85 cases (9%).
- Accessing a derived class member on a base class object
- Attempting to iterate through char* cast to long*

Beyond the artificial test cases that we created, TAILCHECK also uncovered a heap overflow read in the SPEC CPU2017 gcc application (v4.5.0. function vn_nary_may_trap in tree-ssa-sccvn.c:3365). When instrumented with TAILCHECK, the application triggered a segmentation fault and produced a core file pointing to the error. This heap overflow was present in the code for 16 months before being detected with Valgrind (PR

	average latency	99th% latency	memory
apache	4% (3~6%)	3% (1~5%)	26% (15~32%)
nginx	2% (1~3%)	3% (1~5%)	41% (32~45%)
nginx (w/o poolalloc)	4% (3~6%)	3% (0~6%)	49% (44~52%)
memcached	3% (2~3%)	4% (3~5%)	2% (1~3%)
redis	6% (5~7%)	4% (0~18%)	3% (1~5%)
(Mean)	4%	3%	17%

Table 1: TailCheck runtime overhead (average latency and 99th% latency) and memory overhead on server applications, normalized to an uninstrumented base system. The latencies and memory overhead slightly vary for different file and object sizes tested. The first percentage is a geometric mean and the two numbers in parenthesis represent the range. The overall geometric (Mean) is computed for four servers, excluding nginx (w/o poolalloc), across all input sizes.

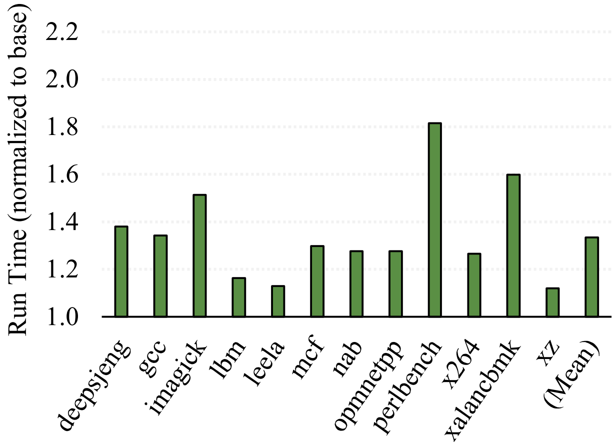

Figure 3: TailCheck run-time overhead on SPEC CPU2017, normalized to an uninstrumented base system.

tree-optimization/44124 [1]). The bug made it into the SPEC CPU2017 suite v1.0.5[1] and, to the best of our knowledge, TailCheck is the first to report it in the literature.

6.2 TailCheck Performance

We present the performance overhead of TailCheck on server applications in Table 1, which shows the average and 99th-percentile latencies. Both latencies vary only slightly for different test input sizes: 32KB, 128KB, 512KB, and 2MB files for web servers and 32B, 128B, 512B, and 2KB objects for key-value stores. There were no noticeable differences for `nginx` with and without application-level pool allocations. `Redis` with 2KB objects shows the highest 99th% latency overhead of 18%, yet with high variance. All the rest, including `redis` with smaller objects, show minor performance degradation ($\leq$7%). An individual 99th-percentile latency result for different file/object sizes can be also found in Figure 5 (for the comparison with AddressSanitizer [54]). The geometric mean across the four servers was 4% and 3% for the average and 99th-percentile latencies, respectively.

The TailCheck performance results for SPEC CPU2017 are shown in Figure 3. The geometric mean of the TailCheck performance overhead for SPEC CPU2017 is 33%, among which `perlbench` shows the highest 1.8x slowdown. We present a performance comparison study with prior art in §6.5. Overall, we find that the combination of these servers and SPEC CPU performance results indicate overheads that are likely low enough to warrant production use of TailCheck for run-time heap overflow detection in security-conscious environments.

6.3 TailCheck Memory Usage

In addition to the performance overheads, TailCheck increases application memory requirements because it reserves space for the TailObjects at the end of each allocator managed region. Table 1 (last column) shows the memory overhead for the server applications. The relative increase in memory usage was small for the key-value store applications, while `nginx` shows the highest overheads. In TailCheck, a protected page for small objects is a virtual page with no access permission and thus does not require a physical page. However, for large objects, TailCheck still requires one TailObject and one protected page. Upon further investigation, we found that at start-up, `nginx` allocates a large number of large objects, incurring a relatively high memory overhead. However, we also observed that once initialized, its peak RSS does not change while serving client requests. *Nginx (w/o poolalloc)* allocates more (non-pool) large objects, showing a slightly higher memory overhead than *nginx* with pool allocations.

Next, we present the memory overheads in Table 2 for SPEC CPU2007 applications. Because TailCheck shares the space of the TailObjects for small objects within a region, the capacity overheads is minimal. The most affected benchmarks (`perlbench`, `gcc`, and `nab`) experience only a 9% increase in the peak RSS. The geometric means were 17% for the servers and 3% for the SPEC CPU2017 applications.

6.4 Analysis of Optimizations.

To better understand the TailCheck performance overheads, we analyze the benefits of the optimizations described in §3.2. For this experiment, we used the SPEC CPU2006 benchmark instead of CPU2017 because when we compare ours with Delta Pointers in §6.5, we want to compare the impact of the same static optimization (SafeAlloc) on ours and Delta Pointers. However, few benchmark applications in SPEC CPU2017 have memory requirements and cannot be supported by Delta

[1] The bug patch has been merged to `gcc` v4.5.1. The ChangeLog of SPEC CPU2017 does not indicate version update or bug fix.

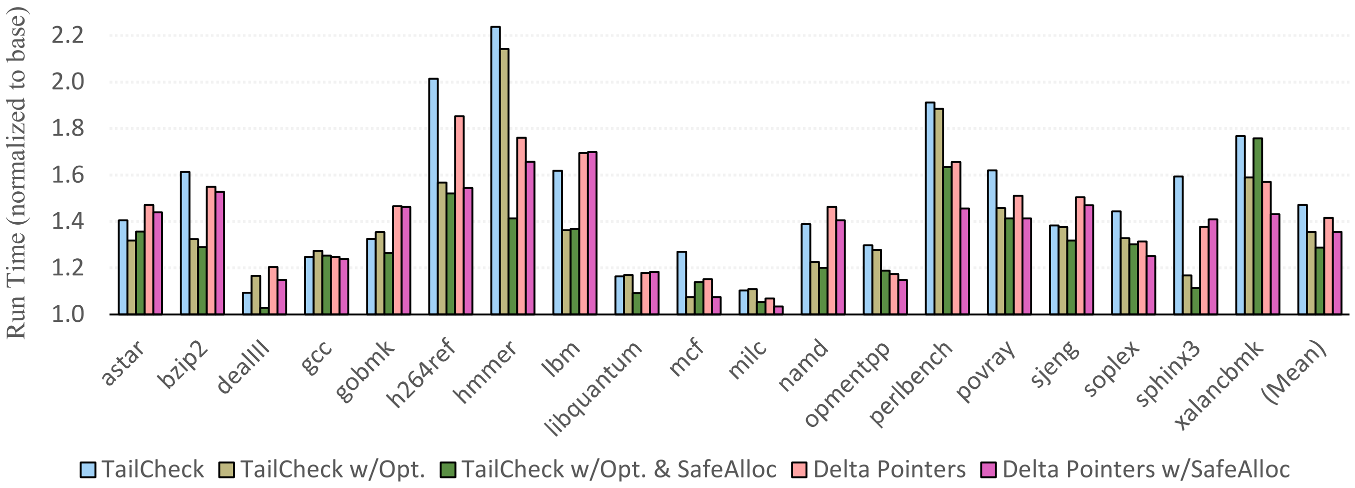

Figure 4: Comparison of run-time overheads on SPEC CPU2006, normalized to an uninstrumented base system: (a) TAILCHECK; (b) TAILCHECK with Opt.; (c) TAILCHECK with Opt. and SafeAlloc; (d) Delta Pointers; and (e) Delta Pointers with SafeAlloc.

application	overhead	application	overhead
deepsjeng	0%	nab	9%
gcc	9%	omentpp	3%
imagick	0%	perlbench	9%
lbm	0%	x264	6%
leela	-1%	xalancbmk	5%
mcf	0%	xz	0%
		geo-mean	3%

Table 2: TAILCHECK memory overhead (peak RSS) on SPEC CPU2017, normalized to an uninstrumented base system.

Pointers:*e.g.*, `xz` and `mcf` with the reference input. Thus, we based our analysis on SPEC CPU2006.

The first three bars in Figure 4 present the impact of optimizations. We first disable the merging of tag handling code when multiple offsets of an object are dereferenced within the same function. Although not drastic, the geometric mean of performance across the SPEC CPU2006 suite improves by 11%, with the biggest gains coming from several applications such as `bzip2`, `h264ref`, and `sphinx3`.

We also examine the benefits of applying SafeAlloc to avoid TAILCHECK instrumentation for heap objects whose accesses are known to be within bounds through static code analysis. Although the gains across all benchmarks are modest, 7% on average, applications such as `hmmer` and `perlbench` exhibit drastic benefits, reducing the run-time overheads by 73% and 24%, respectively. These applications have hot loops iterating over multiple large arrays, allowing SafeAlloc to find a significant number of optimization opportunities.

We note that "Hoisting Tag Handling," as described in 3.2, reduces address calculation overheads, but may also increase register pressure. For loops with a small number of pointer dereferences in the loop body, hoisting the computation of the tail pointer may add register pressure to the program, leading to performance degradation. The TAILCHECK compiler performs a simple count of the number of pointer dereferences, applying hoisting if a loop has two or more dereferences. In some cases (e.g., `xalancbmk`), SafeAlloc eliminates some pointer dereferences, leaving just one dereference in the loop body, bypassing optimization in those loops.

6.5 Comparison with Delta Pointers

This experiment compares TAILCHECK with Delta Pointers, the state-of-the-art compiler-based solution that shares many similarities with TAILCHECK in that both use pointer tagging and do not perform explicit bound checking. We use mimalloc's unmodified allocator for baseline and Delta Pointers performance measurements.

Figure 4 shows the performance comparison. First, when comparing the last two bars, we can find that the SafeAllocs optimization gives roughly the same relative benefit for Delta Pointers (6%) as for TAILCHECK (7%). One thing to note is that the other two tag merging and hoisting optimizations (excluding SafeAllocs) in §3.2 are only applicable to TAILCHECK, but not to Delta Pointers. The reason is that the two optimizations require the tag of a pointer remain unchanged once defined (until an object becomes freed), which is the case for TAILCHECK, but not for Delta Pointers.

Second, when comparing the two fully optimized versions, the 3rd and 5th bars in Figure 4, TAILCHECK exhibits lower runtime overhead than Delta Pointers: 29% vs. 35%. TAILCHECK has lower than or similar runtime overheads than Delta Pointers for most applications. Two exceptions were perlbench and xalancbmk. There are two significant differences in Delta Pointers' and TAILCHECK code instrumentation. Delta Pointers instruments pointer arithmetic to update a pointer tag, while TAILCHECK does not. TAILCHECK adds additional memory operation on a pointer dereference, while Delta Pointers does not. When considering the number of instrumentation as a factor of runtime overhead, TAILCHECK is likely to perform better than Delta Pointers for those applications with more pointer arithmetic and less dereferences.

For reference, we note that Oleksenko *et al.* [45] reported 1.8x, 1.8x, 2x, and >3x runtime overheads for AddressSanitizer [54], Intel's MPX (ICC), SoftBound [40] and SAFECode [15], respectively, for the SPEC CPU2006 applications (in their experimental settings).

Delta Pointers does not incur additional memory overhead, as it does not use a custom allocator with guard pages like TAILCHECK (Table 2). Rather, the major drawback of Delta Pointers is the need to shrink the process address space (e.g., 32-bit tag and 32-bit address space).

6.6 Comparison with AddressSanitizer

Our last experiment compares TAILCHECK with AddressSanitizer [54] for server applications. AddressSanitizer is the state-of-the-practice solution that maintains a disjoint metadata space to distinguish safe regions and (unsafe) redzones. Figure 5 shows the 99th-percentile latency across different file sizes (32KB-2MB) for web servers and object sizes (32B-2KB) for key-value stores. As discussed in §6.2, TAILCHECK shows minor (on average 3%) tail latency degradation. The worst 18% overhead appears only for redis with 2KB objects. On the other hand, AddressSanitizer incurs higher overheads for all cases (on average 16%, up to 51%), reflecting its expensive metadata lookup and checking costs. Likewise, AddressSanitizer shows higher average latencies (not shown) than TAILCHECK: 4% vs. 12% on average; and 7% vs. 56% in the worst case.

7 Discussion

7.1 False Positives and False Negatives

TAILCHECK does not have false positives, assuming there are no use-after-free violations. A shadow memory access computed from a dangling pointer could be wrong if freed and reallocated objects have a different size. Otherwise, the tag in a pointer and the actual distance between a (current) object and its corresponding TailObject always match, and the size of a TailObject is always larger than or equal to that of an original object. Thus, any page fault from a protected page is evidence of a true heap overflow.

We exclude a discussion of potential segmentation faults from passing tagged pointers to uninstrumented code without proper masking. The mechanisms for using tagged pointers in the presence of uninstrumented code are described in §4.

TAILCHECK may have false negatives (miss some heap overflows). First, TAILCHECK is a dynamic tool. It can detect a heap overflow only along the program paths that are explored at run-time, given a test input and environment. Second, TAILCHECK is an instrumentation-based tool and may miss a heap overflow in an object that crosses the instrumented vs. uninstrumented code boundary, such as calls into third-party libraries and assembly code.

Consider two cases, one in which a heap object is created in the instrumented code, but escapes unmasked into uninstrumented code where it is accessed, and vise versa. TAILCHECK cannot detect an overflow in uninstrumented code as there is no shadow TailObject access. Similarly, if an object allocated in the uninstrumented code is passed to instrumented code, TAILCHECK cannot detect an overflow as there is no tag and no corresponding protected page available for the object.

Finally, TAILCHECK relies on a guard page; thus it may fail to detect an overflow beyond the 4KB protected page (similar to AddressSanitizer's 128B redzone [54]). However, it is difficult for a malicious user to exploit this, particularly for small objects, because both the original (manipulated) access and the TailCheck (shadow) access must land on legal memory regions to succeed. The TAILCHECK memory allocator scatters 64KB allocation regions for small objects in the process address space, making the distance between a protected page of a memory region and other valid memory regions non-deterministic. Large objects may be easier to exploit as their original and TailCheck accesses are to the same location. We note that this is different from AddressSanitizer's traditional redzone approach in which a constant length (e.g., 128B) redzone is inserted between adjacent valid memory objects/regions. Prior solutions that use explicit bounds checking (*e.g.,* MPX [45]) or precisely keep track of pointer arithmetic (*e.g.,* Delta Pointers [28]) do not have this limitation.

7.2 Benign False Negatives due to Alignment

x86-64 Linux assumes that heap allocators return 16-byte aligned pointers, allowing the compiler to emit memory instructions based on this assumption. As discussed in §3.1, TAILCHECK enforces the same alignment for TailObjects as for the original objects, allowing the compiler to use the same memory instruction for both original and shadow memory accesses. For objects that are not 16-byte aligned (*e.g.,* 11 byte), the bound of the corresponding 16-byte aligned TailObject (of the same 11 byte size) will not be adjacent to the boundary of a protected page and there will be a gap due to the alignment requirement (5 bytes in this example). This gap may lead to a false negative as the shadow access would not lead to a page fault. Nonetheless, we see this as a "benign" overflow, as the original object would also have the same gap before its adjacent object, due to the same alignment requirement.

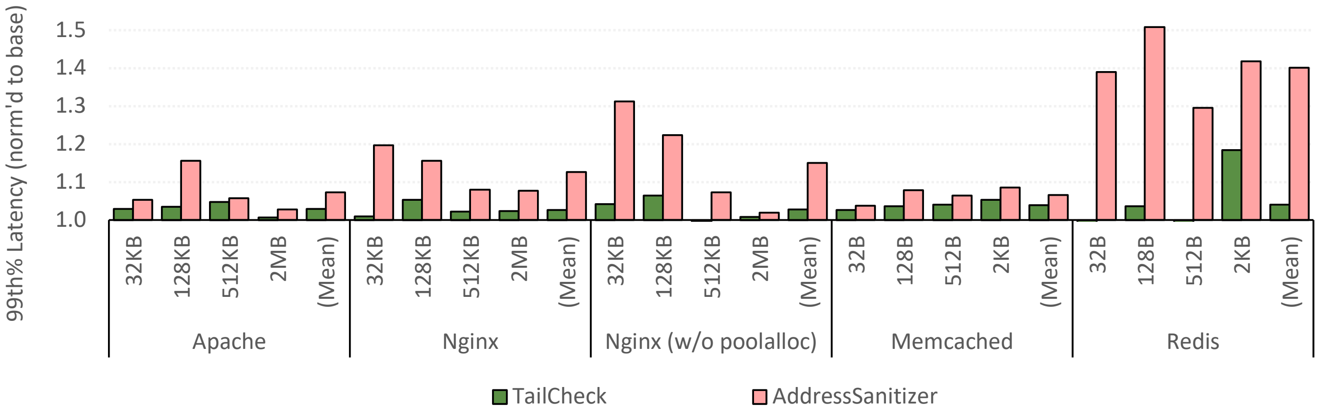

Figure 5: Comparison of 99th-percentile latency on server applications across different file/object sizes, normalized to an uninstrumented base system: (a) TailCheck and (b) AddressSanitizer.

If required, TailCheck can be extended to put the bound of a TailObject immediately before the protected page without a gap. TailCheck would create an unaligned TailObject, and there will not be a false negative due to alignment. However, in this case, the shadow memory access instrumentation pass may need to use different instruction opcodes for the shadow accesses, because the instructions used to access the original object may not support unaligned addresses. Although such a change is possible to eliminate these benign false positives, it is likely to come at a performance cost.

7.3 Potential Hardware Support

TailCheck performance could benefit from the following hardware support. First, TailCheck (on x86-64) must currently mask the tags of pointers before accessing an original object and before passing pointers to uninstrumented code. TailCheck could take advantage of the top-byte-ignore feature of ARM's MTE [3] to avoid masking overhead, similar to HWAsan [55], a hardware-assisted ASan.

Second, TailCheck (on x86-64) relies on load and store instructions to perform shadow memory accesses for overflow detection. It would be sufficient for TailCheck shadow operations to only check for access permission. TailCheck could make use of new pseudo load/store-like instructions which perform virtual to physical address translation and check permissions, without performing an actual memory access or perturbing the data cache, eliminating cache pollution, cache coherence traffic, etc. Such shadow accesses would not modify memory, reducing the TailCheck run-time overhead and reducing the memory overhead because TailObject space would no longer need to be reserved.

Lastly, one can design a hardware TailCheck without compile-time dereference instrumentation. Given a tagged pointer, the memory management unit of a processor can transparently perform a page permission check or a shadow memory access to the TailObject.

7.4 Extensions to Other Memory Safety

Heap Intra-Object Overflow. TailCheck defines a heap object protection granularity at the time of heap allocation. Thus, TailCheck does not protect a more fine-grained subobject from an overflow (*e.g.,* an overflow of an array field of a struct to another field of the same struct) as in per-object bound checking solutions [2, 13, 17, 18, 25, 30, 51, 67]. If desired, TailCheck's compiler instrumentation pass could be extended to "heapify" a subobject, similar to CCured [43]. This is analogous to the additional "bound narrowing" feature in some per-pointer bound checking solutions [40, 45].

Heap Underflow. Though buffer underflow is less critical than overflow in terms of security, if desired, the design of TailCheck could be flipped to "HeadCheck."

Heap Use-After-Free. TailCheck does not support any temporal memory safety, yet it could be combined with existing use-after-free detection schemes that do not use pointer tagging: *e.g.,* Oscar [12] or DangZero [20] that rely on page protection could be a good candidate for integration.

Stack Overflow. TailCheck assumes that stack is protected by other schemes such as stack canaries [11] and shadow stacks [60]. In the current form, TailCheck's instrumentataion pass does not need to distinguish stack and heap objects as any address-taken stack object would hold a tag of 0, leading to harmless redundant memory accesses.

If desired, TailCheck can be extended to support stack overflow protection. The simplest solution is to replace stack allocation with heap allocation, similar to CCured's "heapified" stack [43], at some performance cost. Alternatively, TailCheck could be extended to add a protected page to stack and protect the stack objects similar to heap objects (using a distance tag, a TailObject, and a shadow memory access), with the following instrumentation pass changes. The size of a TailObject (a max of projected objects) can be determined as the sizes of the stack objects are known. The location of TailObject (before a protected page) should be kept in a reserved register or a global variable by instrumenting the entry function: *e.g.,* `main`. For each function, any address-taken stack object (*e.g.,* defined by LLVM's `alloca`) should be instrumented to tag the distance from the stack object (whose address is computed from a stack pointer) to the TailObject (whose address is kept separtely). Then, TAILCHECK can use the same mechanism for stack objects as heap objects. The default size of TAILCHECK's tag is 16 bits, implying that it can support a stack up to 64KB. If a larger stack is needed, the address space should be reduced for a wider tag. Selectively using heapification for a large stack object could be helpful.

8 Related Work

There are hundreds of prior memory safety solutions, with a little bit of exaggeration. This section does not attempt to cover them exhaustively. Instead, we focus on discussing where TAILCHECK sits among these related works.

8.1 Buffer Overflow Detection

The first group maintains "redzone" metadata and checks if a program accesses the red zone on each memory access. Purify [22] is the first to use redzone. LBC [21] introduces a fast path optimization skipping metadata lookup with a random canary. ASan [54] and Valgrind [44] are popular redzone-based tools using static instrumentation and dynamic binary translation, repsectively.

The second group performs explicit "bounds checking." Some maintain per-object bound metadata and perform bounds checking on pointer arithmetic: *e.g.,* J&K [25], CRED [51], D&A [13], Baggy Bounds [2], PAriCheck [67], LowFat [17, 30], and EffectiveSan [18]. Others keep track of per-pointer bound metadata and check bounds on pointer dereferences: *e.g.,* SoftBound [40], SGXBounds [29], MidFat [27], MPX [45], CUP [8], and FRAMER [42]. Static analysis can be used to avoid some bound checks on memory accesses proven to be safe: *e.g.,* PICO [26]. The pointer-based approach has another advantage that makes it easy to support intra-object overflow protection: *e.g.,* an array in a struct.

The third group leverages "page protection": *e.g.,* Electric Fence [49], DUMA [5], DYBOC [56], libgmalloc [35], and PageHeap [61]. They do not maintain redzone/bound metadata nor perform explicit checking as in the above two groups. However, as discussed in §2.3, allocating one object per page incurs huge memory and run-time overheads. Prober [37] shows low overhead but it only protects heap arrays. TAILCHECK proposes a new low-overhead page protection-based solution for all heap objects.

On the other hand, Delta Pointers [28] check the integer overflow of a tagged pointer. It does not make use of a redzone, a bound, or a protected page; and thus does not fall into any of the above groups.

8.2 Pointer Tagging

Many of the above solutions need to maintain some metadata. Some use "fat pointers" that stores the metadata (*e.g.,* base and bound) in separate words alongside the actual pointer value. Examples include Safe-C [4], Cyclone [24], and CCured [43]. CHERI [62, 63] provide hardware support for fat pointers.

Many recent works leverage "pointer tagging" that embeds metadata into some bits of a pointer itself, to avoid a code layout change. For example, Baggy Bounds [2] uses the spare top bits to store the distance between an out of bound pointer and its intended referent. Delta Pointers [28] uses a 32-bit tag to encode the distance from the current pointer to the end of an object. As discussed in §2.3, one common downside of pointer tagging is that it may restrict the address space: *e.g.,* Delta Pointers only support a 4GB of 32-bit address space. This may not be a problem for SGXBounds [29], which is designed for an Intel SGX enclave with already-limited 32-bit address space, and thus it can use a 32-bit tag to store the upper bound of the pointer's referent without any sacrifice. However, other pointer tagging solutions (including Delta Pointers) that require more than 16 unused bits in the current 64 bit architecture cannot be used for general (non-SGX) programs with big memory requirements. TAILCHECK does not share this limitation. Alternatively, CUP [8] takes an extreme design that uses the entire pointer width to store tags. Low-fat pointers [17, 30] store the tag implicitly in the pointer value, and thus can be safely dereferenced without masking.

Several works proposed hardware support for pointer tagging. In-Fat [64] is a hardware extension of EffectiveSan [18], which tags the upper bits of a pointer with an index into a bounds table, performing bounds checking on pointer arithmetic. HeapCheck [52] stores an index into a per-pointer bounds table, and checks bounds on pointer dereferences. PACMem [34] leverages ARMv8 AArch64 Pointer Authentication (PA) [50], computing cryptographic hashes based on the value of pointers (and other contexts) for pointer integrity. PACMem seals object metadata into the high-order bits of pointers via PA and uses the seal as the index to retrieve it. The tagged PA codes are propagated by hardware along with the pointers. No-Fat [23] supports low-fat pointers [17, 30].

8.3 Use-After-Free Detection

Existing use-after-free solutions can be categorized into three groups based on their detection techniques. Some solutions such as CETS [41], ViK [10] and xTag [6] tag the allocated memory and the pointer with a unique identifier (referred to as lock and key), and check if the tags of pointer and memory match on dereference. Any mismatch indicates that a pointer used for deference is a dangling pointer. ARM's Memory Tagging Extension (MTE) [3] and SPARCS's Silicon Secured Memory (SSM) [46] provide hardware support to assign random 4-bit tags to object-pointer pairs to probabilistically find use-after-free bugs on tag mismatch. HWAsan [55] an extension of ASan with ARM's MTE makes use of its top-bit-ignore feature and avoids masking on memory dereference.

Other solutions such as Undangle [9], FreeSentry [66], DangNull [32], DangSan [59], BOGO [68] maintain metadata to find and invalidate dangling pointers on free. Then a use-after-free is detected as an invalid pointer use: *e.g.,* null pointer dereference.

Yet others such as D&A [14], Oscar [12], and DangZero [20] leverage page protection: a page becomes inaccessible after a free. Oscar [12] reduces physical memory and run-time overhead by mapping multiple virtual pages into a single physical page. DangZero [20] further lowers run-time overhead by directly accessing the page tables with support from virtualization extensions and a privileged backend (*e.g.,* Kernel Mode Linux). TAILCHECK does not provide use-after-free detection, but its page-based approach makes it possible to integrate the above page-based use-after-free solutions to achieve both spatial and temporal memory safety. We leave this to future work.

8.4 Uninitialized Memory Read

Uninitialized memory reads can lead to information leakage, similar to buffer overflow reads. Purify [22] and Valgrind [44] detect an uninitialized memory read by maintaining and checking (initialized vs. uninitialized) state metadata at a byte or bit granularity, respectively. UniSan [38] uses data-flow analysis to zero-out variables that might be disclosed to an attacker. SafeInit [39] modifies the compiler and heap allocator to ensure that all stack/heap regions be initialized.

9 Conclusions

Heap overflow vulnerabilities leave many software systems exposed to security attacks and exploitation. This work presented TAILCHECK, a novel heap overflow mitigation scheme that leverages a custom memory allocator, OS-based page protection, and compiler-directed pointer tagging. TAILCHECK achieves low run-time overhead by detecting heap overflows using page protection, without maintaining bound metadata and without performing explicit bounds checks. TAILCHECK uses pointer tagging and shadow memory accesses to detect overflows, allowing multiple original objects to share a single TailObject, which reduces both performance and memory overheads compared to the previously explored techniques. The results of our experimental evaluation demonstrate the effectiveness and efficiency of TAILCHECK in detecting heap overflows in C and C++ programs.

Acknowledgements

We would like to thank the anonymous reviewers for their feedback on the draft of this work, and James Mickens for acting as our shepherd. This work was supported in part by the National Science Foundation under Grant No. CCF-2153747, CNS-2135157, CNS-2214980, CCF-2153297, and CNS-1763680.

References

[1] re PR tree-optimization/44124. https://gcc.gnu.org/git/?p=gcc.git&a=commit;h=4d085b9ee391f005d209512de3ea283fde49d42e.

[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 *USENIX Security Symposium*, volume 10, 2009.

[3] ARM. Armv8.5-a memory tagging extension. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[4] Todd M Austin, Scott E Breach, and Gurindar S Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[5] Hayati Aygün. Detect Unintended Memory Access (D.U.M.A.). https://duma.sourceforge.io/, 2022. [Online; accessed 29-Nov-2022].

[6] Lukas Bernhard, Michael Rodler, Thorsten Holz, and Lucas Davit. xtag: Mitigating use-after-free vulnerabilities via software-based pointer tagging on intel x86-64. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 502–519, 2022.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery.

[8] 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*, ASIACCS '18, page 381–392, New York, NY, USA, 2018. Association for Computing Machinery.

[9] 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 2012, page 133–143, New York, NY, USA, 2012. Association for Computing Machinery.

[10] Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Vik: Practical mitigation of temporal memory safety violations through object id inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 271–284, New York, NY, USA, 2022. Association for Computing Machinery.

[11] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[12] 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.

[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*, pages 162–171, 2006.

[14] 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.

[15] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, page 144–157, New York, NY, USA, 2006. Association for Computing Machinery.

[16] Jaana Dogan. hey: Http load generator. https://github.com/rakyll/hey.

[17] 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 2016, page 132–142, New York, NY, USA, 2016. Association for Computing Machinery.

[18] Gregory J. Duck and Roland H. C. Yap. Effectivesan: Type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 181–195, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, page 475–488, New York, NY, USA, 2014. Association for Computing Machinery.

[20] Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. Dangzero: Efficient use-after-free detection via direct page table access. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1307–1322, New York, NY, USA, 2022. Association for Computing Machinery.

[21] Niranjan Hasabnis, Ashish Misra, and R Sekar. Lightweight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 135–144, 2012.

[22] Reed Hastings. Purify: Fast detection of memory leaks and access errors. In *Proc. 1992 Winter USENIX Conference*, pages 125–136, 1992.

[23] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. No-fat: Architectural support for low overhead memory safety checks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 916–929, 2021.

[24] 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.

[25] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *AADEBUG*, volume 97, pages 13–26, 1997.

[26] Tina Jung, Fabian Ritter, and Sebastian Hack. Pico: A presburger in-bounds check optimization for compiler-based memory safety instrumentations. 18(4), jul 2021.

[27] Taddeus Kroes, Koen Koning, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Fast and generic metadata management with mid-fat pointers. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[28] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[29] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 205–221, 2017.

[30] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 721–732, 2013.

[31] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[32] 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.

[33] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. In *Asian Symposium on Programming Languages and Systems*, pages 244–265. Springer, 2019.

[34] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1901–1915, New York, NY, USA, 2022. Association for Computing Machinery.

[35] libgmalloc(3) [osx man page]. (guard malloc), an aggressive debugging malloc library. https://www.unix.com/man-page/osx/3/libgmalloc/.

[36] Hongyu Liu, Ruiqin Tian, Bin Ren, and Tongping Liu. Prober: Practically defending overflows with page protection. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 1116–1128, New York, NY, USA, 2021. Association for Computing Machinery.

[37] Hongyu Liu, Ruiqin Tian, Bin Ren, and Tongping Liu. Prober: Practically defending overflows with page protection. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 1116–1128, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 920–932, New York, NY, USA, 2016. Association for Computing Machinery.

[39] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. Safelnit: Comprehensive and practical mitigation of uninitialized read vulnerabilities. In *NDSS*, volume 17, pages 1–15, 2017.

[40] Santosh Nagarakatte, Jianzhou Zhao, Milo MK 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*, pages 245–258, 2009.

[41] 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, page 31–40, New York, NY, USA, 2010. Association for Computing Machinery.

[42] Myoung Jin Nam, Periklis Akritidis, and David J Greaves. Framer: A tagged-pointer capability system with memory safety applications. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 612–626, New York, NY, USA, 2019. Association for Computing Machinery.

[43] 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*, pages 128–139, 2002.

[44] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[45] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):1–30, 2018.

[46] Oracle. Sparc m7 silicon secured memory (ssm). https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html.

[47] Mitre Org. CVE Buffer Overflow. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer+overflow, 2022. [Online; accessed 29-Nov-2022].

[48] Mitre Org. CWE-787. https://cwe.mitre.org/data/definitions/787.html, 2022. [Online; accessed 29-Nov-2022].

[49] Bruce Perens. Electric Fence. https://linux.die.net/man/3/efence/, 1987. [Online; accessed 19-Nov-2022].

[50] Qualcomm. Pointer authentication on armv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointerauthentication-on-armv8-3.pdf, 2017.

[51] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *NDSS*, volume 4, pages 159–169, 2004.

[52] Gururaj Saileshwar, Rick Boivie, Tong Chen, Benjamin Segal, and Alper Buyuktosunoglu. Heapcheck: Low-cost hardware support for memory safety. *ACM Trans. Archit. Code Optim.*, 19(1), jan 2022.

[53] R.C. Seacord. *Secure Coding in C and C++*. SEI series in software engineering. Addison-Wesley, 2013.

[54] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

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

[56] Stelios Sidiroglou, Giannis Giovanidis, and Angelos D. Keromytis. A dynamic mechanism for recovering from buffer overflow attacks. In *Proceedings of the 8th International Conference on Information Security*, ISC'05, page 1–15, Berlin, Heidelberg, 2005. Springer-Verlag.

[57] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295, 2019.

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

[59] 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*, EuroSys '17, page 405–419, New York, NY, USA, 2017. Association for Computing Machinery.

[60] Vendicator. Stack Shield: A "stack smashing" technique protection tool for linux. https://www.angelfire.com/sk/stackshield/, 2000.

[61] Microsoft Windows. PageHeap. https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap, 2022. [Online; accessed 21-Nov-2022].

[62] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.

[63] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 545–557, New York, NY, USA, 2019. Association for Computing Machinery.

[64] Shengjie Xu, Wei Huang, and David Lie. In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 224–240, New York, NY, USA, 2021. Association for Computing Machinery.

[65] Tao Ye, Lingming Zhang, Linzhang Wang, and Xuandong Li. An empirical study on detecting and fixing buffer overflow bugs. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 91–101, 2016.

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

[67] 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*, pages 145–156, 2010.

[68] Tong Zhang, Dongyoon Lee, and Changhee Jung. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 631–644, New York, NY, USA, 2019. Association for Computing Machinery.