HeapCheck: Low-cost Hardware Support for Memory Safety

GURURAJ SAILESHWAR, Georgia Tech, USA
RICK BOIVIE, TONG CHEN, BENJAMIN SEGAL, and ALPER BUYUKTOSUNOGLU, IBM Research, USA

Programs written in C/C++ are vulnerable to memory-safety errors like buffer-overflows and use-after-free. While several mechanisms to detect such errors have been previously proposed, they suffer from a variety of drawbacks, including poor performance, imprecise or probabilistic detection of errors, or requiring invasive changes to the ISA, binary-layout, or source-code that results in compatibility issues. As a result, memory-safety errors continue to be hard to detect and a principal cause of security problems.

In this work, we present a minimally invasive and low-cost hardware-based memory-safety checking framework for detecting out-of-bounds accesses and use-after-free errors. The key idea of our mechanism is to re-purpose some of the "unused bits" in a pointer in 64-bit architectures to store an index into a bounds information table that can be used to catch out-bounds errors and use-after-free errors without any change to the binary layout. Using this memory-safety checking framework, we enable HeapCheck, a design for detecting Out-of-bounds and Use-after-free accesses for heap-objects, that are responsible for the majority of memory-safety errors in the wild. Our evaluations using C/C++ SPEC CPU 2017 workloads on Gem5 show that our solution incurs 1.5% slowdown on average, using an 8 KB on-chip SRAM cache for caching bounds-information. Our mechanism allows detection of out-of-bounds errors in user-code as well as in unmodified shared-library functions. Our mechanism has detected out-of-bounds accesses in 87 lines of code in the SPEC CPU 2017 benchmarks, primarily in Glibc v2.27 functions, that, to our knowledge, have not been previously detected even with popular tools like Address Sanitizer.

CCS Concepts: • **Computer systems organization** → *Processors and memory architectures*; • **Software and its engineering** → *Software testing and debugging*; • **Security and privacy** → *Vulnerability management;*

Additional Key Words and Phrases: Memory safety, hardware bounds checking, software security

Extension of Conference Paper: R. Boivie, G. Saileshwar, T. Chen, B. Segal, and A. Buyuktosunoglu, "Hardware Support for Low-cost Memory Safety," in *Proceedings of the 2021 51st Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S'21).* In this extension, we include the following new contributions: (1) new details about the software runtime, changes to the program memory layout and impact of caching on bounds-checks; (2) new security results demonstrating the exploits that HeapCheck can detect; (3) new out-of-bounds references detected by HeapCheck in common Glibc functions; (4) new evaluations of HeapCheck with bounds caches of different configurations; (5) comparison with recent memory-safety works highlighting how HeapCheck achieves better performance and compatibility.

Authors' addresses: G. Saileshwar, Georgia Tech, 266 Ferst Dr. NW, KACB 1210 Atlanta, Georgia 30332; email: gururaj.s@gatech.edu; R. Boivie, T. Chen, B. Segal, and A. Buyuktosunoglu, IBM T. J. Watson Research Center 1101 Kitchawan Road, PO Box 218 Yorktown Heights, New York 10598-0218; emails: {rhboivie, chentong, bpsegal, alperb}@us.ibm.com.

1544-3566/2022/01-ART10 $15.00
https://doi.org/10.1145/3495152

ACM Reference format:
Gururaj Saileshwar, Rick Boivie, Tong Chen, Benjamin Segal, and Alper Buyuktosunoglu. 2022. HeapCheck: Low-cost Hardware Support for Memory Safety. *ACM Trans. Archit. Code Optim.* 19, 1, Article 10 (January 2022), 24 pages.
https://doi.org/10.1145/3495152

1 INTRODUCTION

Applications written in memory-unsafe languages like C/C++ are vulnerable to memory-safety errors like buffer overflows, use-after-free, and others. Such errors have been exploited in numerous attacks in the past [44], including high-profile attacks, such as the Morris worm [30] and Heartbleed [1], and are ranked by MITRE [2] to be some of the most dangerous software bugs. As shown in Figure 1, a recent study [22] by Microsoft revealed that such errors continue to be the root cause of approximately 70% of the CVEs identified in their production software; in particular, the errors specific to heap objects, including heap corruption, out-of-bounds accesses, and use-after-free, caused almost 50% of the CVEs in 2019.

In recent years, numerous software tools such as ASAN [37], Valgrind [27], Dr Memory [6], and so on, have been proposed to detect memory-safety errors without invasive changes to the source code or compatibility issues. However, such software-only solutions incur prohibitive performance overheads (e.g., ASAN slows down execution by 2–3×), which has limited their use case to debugging and testing purposes at development time. As a result, any bugs that escape into real-world software continue to cause security vulnerabilities. However, if memory-safety solutions can detect bugs with low overhead, then they can be enabled even in production environments where performance is critical, to detect and prevent such bugs at runtime. For example, Google recently announced Android support [38] for ARM's Memory Tagging [4] feature, which provides probabilistic detection of bugs at negligible performance overheads, with the goal of enabling in-production detection of bugs [39]. However, MTE and several recent hardware-based solutions [4, 29, 35, 42] providing probabilistic detection of bugs trade off detection capability for performance and only detect some bugs while missing others.

More principled bounds-checking-based solutions in hardware [9, 18, 23, 28, 47, 48] have also been proposed that enforce object bounds and detect memory-safety errors with high coverage. However, even the state-of-the-art HW-based bounds-checking solutions such as Chex86 [40] and AOS [18] continue to incur slowdown of 8–15% that may limit their adoption in production environments. Additionally, solutions like AOS rely on new ISA instructions for capturing bounds information when objects are created, thus requiring source code rewrite or recompilation, resulting in incomplete security for legacy code.

Ideally, we seek a solution capable of precise detection of memory-safety bugs with near-0% slowdown to enable adoption at software run time and without invasive ISA changes (no application changes) to ensure security for even legacy code. This can enable always-on memory safety in production software where performance is critical. Such a performant memory-safety bug detector in hardware can also accelerate software testing solutions like sanitizers and fuzzers [16], which holds the promise of effectively increasing the coverage of such software testing solutions. To that end, this article presents a HW-based bounds-checking framework optimized for negligible slowdown, negligible ISA changes, and no changes to application source code or binary layouts.

The crux of our solution is re-purposing architecturally visible "unused bits" in a pointer (modern 64-bit architectures have pointers typically storing virtual addresses with 48-bits of information or less) to store an index to a bounds-metadata table, which tracks the range of addresses legitimately accessible via the pointer. As shown in Figure 2, our framework (a) allocates an

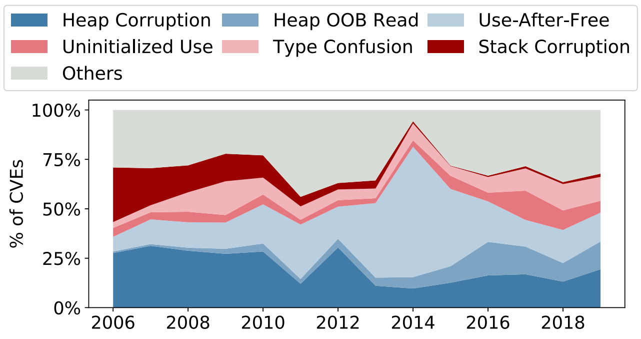


Fig. 1. Data on the root cause of CVEs from a recent study [22] by Microsoft shows memory-safety errors (all non-grey colors above) cause >70% of the CVEs, with errors like Heap Corruption, Heap OOB Reads, and Use-after-free (in blue color above) found in almost 50% of the CVEs.

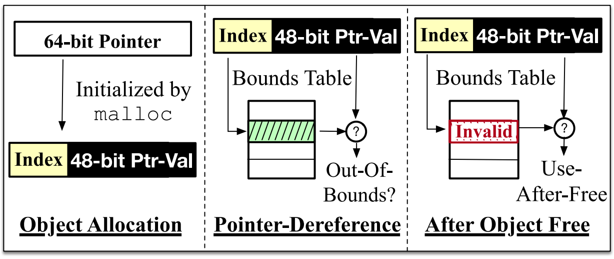


Fig. 2. HeapCheck detects spatial and temporal safety errors for heap objects by reusing the top bits of object pointers to store an index to a table containing bounds metadata. On pointer accesses, hardware performs bounds-checking by accessing the bounds metadata using the index bits in pointer.

entry in a per-process bounds-table to store the bounds-information when an object is allocated; (b) re-purposes the unused top bits of the object pointer to store the index to the bounds-table entry; (c) automatically propagates the index when subsequent addresses are derived via assignment statements or pointer arithmetic; and (d) executes bounds-checking in hardware on load and store instructions by using the index bits in an address to access the bounds metadata. We use this framework to design *HeapCheck*, a spatial and temporal safety solution for heap objects capable of detecting errors like heap out-of-bounds reads and writes, and use-after-free. We focus on heap errors primarily as they make up almost 50% of patched CVEs by Microsoft [22] and almost 60% of the memory-safety bugs detected by Google's OSS-Fuzz [36]; however, our framework can also be extended to detect errors for stack and global objects using the same principles with additional compiler support.

To ensure the slowdown for our solution is negligible, we make several key design choices for our bounds metadata. First, our bounds metadata is associated per object (and not per memory-word unlike prior shadow-memory-based solutions like Hardbound [9] or Watchdog[23]), which allows all pointers associated with a given buffer to have the same index, and thus the bounds table accesses for different words of a single object enjoy excellent temporal locality. Second, our flat linear bounds-table design enables fast access with a single lookup unlike prior approaches requiring expensive multi-level table-accesses [18, 28, 40]. Additionally, the linear bounds table encourages excellent spatial locality as nearby objects also have their bounds-table entries close to each other (within the same cache line), unlike comparable works like AOS [18] using hashed bounds-table designs that lose such locality and incur slowdown.[1] These design choices make it highly likely for the bounds information of a pointer to be available in an on-chip Bounds-information (BI) cache: our simulations show a greater than 98% hit-rate for bounds-table accesses in a fast 8 KB on-chip BICache. Thus, bounds-checking for a majority of the loads or stores can be done without added delays of accessing the memory for obtaining bounds. Finally, our bounds-table index is propagated "automatically" via program semantics on pointer assignments, or pointer arithmetic during array indexing, or when a pointer is passed to a function, without any extra overhead (as all pointer arithmetic is 64-bits, the top bits flow automatically), unlike prior solutions that require extra instructions [3, 10, 24, 50], or extra micro ops and table lookups to propagate pointer metadata like in the recent Chex86 [40].

With all these optimizations combined, our scheme achieves an average slowdown of 1.5% across SPEC-2017 benchmarks, which is more than 5× lower than the prior best bounds-checking solution [18]. While the rich body of previous works in this space have admittedly also used similar ideas like reusing pointer bits for metadata [10, 18] and storing bounds in shadow regions [9, 18, 23, 40], to our knowledge, no prior work has achieved the trifecta of near-zero slowdown, exact bounds enforcement, and seamless backward compatibility, unlike our work.

Our solution also ensures seamless compatibility with legacy code by avoiding changes to the application source code or binary layout, unlike prior works on hardware-based memory safety [18, 23, 47]. Our software instrumentation responsible for bounds table management is packaged as a shared library that just needs to be linked to existing source code. The software instrumentation intercepts calls to memory allocator functions and allocates bounds-table entries on `mallocs`, and invalidates entries on `frees`. On pointer dereference, the index of a bounds-table entry in the top bits of an object pointer is transparently used to access the bounds-table in hardware and execute the bounds-check, to flag out-of-bounds accesses and detect use-after-free errors after an object is freed (as the index in a dangling pointer points to an invalid bounds-table entry).

Overall, this work makes the following contributions:

(1) We propose a low cost, minimally invasive bounds-checking framework by reusing the top bits of an address for bounds-checking metadata.

(2) We implement HeapCheck, our bounds-checking framework for heap objects using LLVM instrumentation to intercept `malloc`/`free` calls in programs and a shared library for tracking bounds, ensuring no application code change is needed for backwards compatibility.

(3) We model our bounds-checking hardware in Gem5 and show our solution has an overall slowdown of 1.5% average, based on evaluations of C/C++ SPEC-CPU2017 benchmarks, using an 8 KB on-chip cache for bounds-metadata.

(4) We demonstrate that our solution precisely detects errors like *heap out-of-bounds reads* and *writes*, *use-after-free*, *invalid-free*, and *double-free*, and we show it detects 25 of 25 heap exploits from the How2Heap exploit suite [41].

(5) We also show our framework can detect memory-safety errors when pointers are passed to unmodified shared libraries, that tools like ASAN [37] cannot. Our solution identified out-of-bounds references in 87 lines of code in SPEC-CPU2017 programs, mainly in Glibc-v2.27 functions where aggressively optimized SIMD instructions access out-of-bounds memory; to our knowledge, these have not been detected in prior work.

[1]AOS [18] uses pointer bits to store keys to a hash table with bounds metadata and uses ARM's Pointer-Authentication ISA extensions (which is not compatible with legacy code) to generate random hash-table keys; the random hash-table accesses are often without spatial locality in the caches, resulting in higher average slowdown of 8.4% and >100% in worst case (versus 1.5% on average and 6% in worst case for our work). We discuss this in Section 6.1.

2 BACKGROUND AND MOTIVATION

We first discuss the problem of memory safety, and then we describe prior solutions and their limitations to motivate our work.

2.1 Problem: Spatial and Temporal Safety

Applications written in C/C++, where pointer manipulation is permissible without safety checks, are prone to memory errors where pointers dereference invalid memory regions. A *spatial error* (out-of-bounds access) can arise from pointer arithmetic using unvalidated inputs that causes a buffer pointer to access memory beyond the buffer bounds. Similarly, a *temporal error* (e.g., use-after-free) can result from a read or write using a dangling pointer (a pointer to a freed object whose memory has been subsequently reused). Memory leakage and corruption due to such errors has been exploited by attacks that break data confidentiality [1], attempt privilege escalation [34], break system integrity [12], and so on. While existing software testing mechanisms use sanitizers [43] and fuzzing [45] to detect memory-safety errors, such mechanisms are primarily software-based and impose large slowdown, thus limiting their applicability. Thus, improving the speed of memory-safety error detection (with hardware support) can considerably improve software reliability and security.

2.1.1 *Threat Model.*

Our threat model focuses on attacks on victim applications that contain memory-safety bugs, which may be exploited by an adversary passing unchecked data inputs to the victim code (e.g., Heartbleed [1]). The adversary itself is unprivileged and does not have arbitrary code execution capability or arbitrary memory read/write capability in the address space of the victim process. We assume the victim code itself is non-malicious and the adversary can only try to exploit bugs in the victim code by passing malicious inputs to it. In this work, we focus on memory-safety bugs (both spatial and temporal) in heap objects, but the solutions we develop are generally applicable to stack or global regions as well. Note that the focus of our solution is mainly on architecturally visible out-of-bounds accesses, and we leave speculative out-of-bounds accesses (like Spectre-v1) out of scope like several prior memory-safety solutions. This is because transient-execution attacks constitute a broader set of vulnerabilities than just Spectre-v1, which bounds-checking might possibly address, and holistic solutions for safe speculation [20, 51] also mitigate Spectre-v1. Hence, we keep Spectre-v1 out of focus for this work by default; but we discuss the implication of our solution on Spectre v1 in Section 5.4.

2.2 Hardware Solutions for Detecting Memory-safety Errors

2.2.1 *Probabilistic Solutions.*

Such solutions either use trip wires or tagged-memory to probabilistically detect memory accesses that cross object-bounds. Trip-wire-based solutions (such as REST [42] and Caliform [35]) insert red-zones or trip-wires around objects to detect common spatial bugs that go beyond object-bounds by a small amount. However, hardware-based memory-tagging solutions like ARM's MTE [4] and SPARCS's SSM [29] proposed assigning random 4-bit tags or "colors" to object-pointer pairs, in an attempt to probabilistically detect bugs based on "color" mismatch. While such solutions are easy to adopt due to minimal slowdown or compatibility issues, they are unable to provide complete coverage for error-detection, by design.

2.2.2 *Bounds-checking-based Solutions.*

Such solutions provide precise enforcement of safe program behavior by tracking the object base and bounds and enforcing bounds-checks on all object accesses. These approaches can be grouped based on the location of their bounds metadata, as shown in Figure 3.

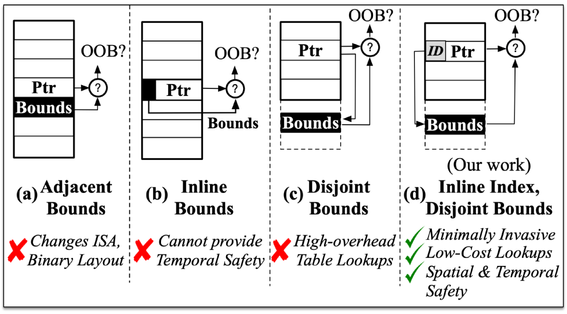
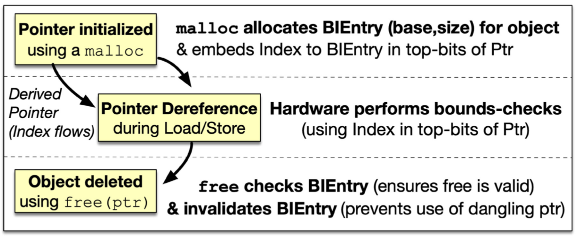


Fig. 3. Categories of bounds-checking-based solutions used for detecting memory-safety errors.

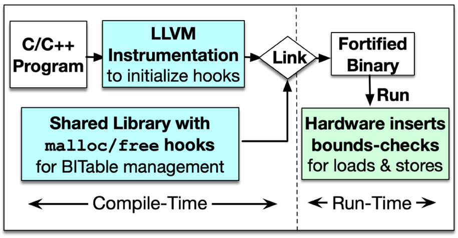
Adjacent Bounds: *Fat-pointer* solutions [17, 26], shown in Figure 3(a), store the base and bounds metadata in separate words alongside the actual pointer value, allowing execution of a bounds-check to detect spatial errors when the actual pointer is dereferenced. CHERI [47] implements such a design with hardware-based bounds-checking, replacing pointers with 256-bit "capabilities" that include the address, bounds, permission bits and other metadata needed for fine-grain compartmentalization and bounds-checking; subsequent works proposed a compressed encoding scheme [46] to reduce capability size to 128-bits and a capability revocation mechanism [48] for temporal safety. Unfortunately, such solutions require invasive changes to the ISA, source code and the binary layout impacting compatibility with legacy code.

Inline Bounds: *Low-fat Pointer* solutions [10, 11, 19], shown in Figure 3(b), encode the object bounds inline within the pointer without impacting binary layout. Kwon et al. [19] use a compact floating-point format to store (the least significant bits of) the object base and bounds addresses in the top 18 bits of a 64-bit pointer. Other works [10, 11] allocate objects in size-specific partitions of memory at a size aligned base-address, to implicitly encode the base and bounds in the pointer value. These works track pointer arithmetic (either in hardware [19] or via explicit instructions [10, 11]) to ensure the pointer never crosses the inline bounds. Unfortunately, such solutions do not provide temporal safety, as the checks using the bounds within a dangling pointer continue to pass, even after the memory it references has been freed.

Disjoint Bounds: Other solutions store bounds metadata in a disjoint table in *shadow memory* to avoid changing the binary layout, as shown in Figure 3(c). The bounds-table is typically indexed using the pointer-value, as a linear table lookup [9, 23] or using a multi-level trie lookup [28, 40]. Spatial errors are detected by executing a bounds-check with a table-lookup in hardware on pointer dereferences. The resulting overheads are lower than comparable software-based solutions [3, 24, 25, 50] as bounds are propagated and checked using micro-code or dedicated hardware. However, such solutions continue to incur moderate to high slowdown due to expensive table lookups using the pointer value, to access the bounds metadata. Hardbound [9] stores the bounds metadata in shadow memory and requires a shadow memory access for a bounds-check on each regular memory access. However, this design only provides spatial safety. Watchdog [23] extends this to provide spatial and temporal safety, by storing unique identifiers for pointers that are only valid for object lifetime, along with bounds metadata, and incurs a higher slowdown of 24%. Intel's MPX [28] requires a two-level table lookup to obtain the bounds metadata, incurring a much higher slowdown of 50% on average. Recently, Chex86 [40] proposed spatial and temporal safety by associating pointers with capabilities (containing bounds metadata) stored in a separate table, that is accessed based on an identifier (capability-ID). Unfortunately, this adds another layer of indirection for a bounds-check, requiring in the worst-case, a five-level table lookup to identify


Pointer initialized using a malloc
malloc allocates BIEntry (base,size) for object & embeds Index to BIEntry in top-bits of Ptr
Derived Pointer (Index flows)
Pointer Dereference during Load/Store
Hardware performs bounds-checks (using Index in top-bits of Ptr)
Object deleted using free(ptr)
free checks BIEntry (ensures free is valid) & invalidates BIEntry (prevents use of dangling ptr)

(a) Life-cycle of a Pointer in HeapCheck


C/C++ Program
LLVM Instrumentation to initialize hooks
Link
Fortified Binary
Run
Shared Library with malloc/free hooks for BITable management
Hardware inserts bounds-checks for loads & stores
Compile-Time
Run-Time

(b) Software and Hardware support required

Fig. 4. Overview of HeapCheck. (a) Life-cycle of a heap-object pointer, and associated operations with bounds-information entry (BIEntry) in the bounds-table (BITable) to enforce memory safety. (b) Changes in SW and HW made to enable HeapCheck.

the capability-ID for a pointer, and then a lookup to retrieve the capability itself, resulting in a worst-case slowdown of 40%.

2.3 Goal and Insight

Our goal is to enable a hardware-based bounds-checker that can precisely detect spatial errors, and also efficiently invalidate bounds for dangling pointers to detect temporal errors. For practical adoption, we would like our mechanism to have negligible performance overheads, not require adding any new instructions to the ISA, and avoid any changes to the application source code or the binary layout. To that end, we adopt a configuration shown in Figure 3(d), that associates a pointer with a unique inline identifier, which is used to index into a disjoint bounds-table. While such a configuration was recently used [18] to provide memory safety, we observe there is significant room to reduce overheads and improve compatibility. We describe our solution next.

3 DESIGN

We first provide an overview of our solution HeapCheck, and then describe the software and hardware components that enable our solution.

3.1 Overview of Bounds-checking with HeapCheck

High-level Idea: The main principle of HeapCheck is to store the bounds metadata of an object throughout its lifetime in a per process **bounds-information table (BITable)**, within the program's virtual address space, and perform hardware-based bounds-checks on all object accesses at runtime. Figure 4(a) shows the life-cycle of a pointer during program runtime. When an object is created, a **bounds-information entry (BIEntry)** is created in the BITable to store its base-address and the size, and the index of the corresponding entry of the BITable is embedded within the top bits of the pointer.[2] When the pointer is dereferenced, the hardware uses the index within the top bits to access the corresponding BIEntry and perform a bounds-check to detect out-of-bounds accesses. When an object is freed, its BIEntry is invalidated, allowing detection of temporal errors if dangling pointers to such freed objects are used subsequently.

Implementation: The organization of our implementation is shown in Figure 4(b). The software manages the BITable: we use hooks for malloc and free functions to intercept calls to these functions, and perform associated BITable operations such as allocation and invalidation of BIEntries. We define these hooks in a shared library that can be added by the linker during program

[2]Our design does not require the index bits to be contiguous or be the top bits of the pointer. However, for simplicity of implementation, we reserved the top 24-bits for index bits.

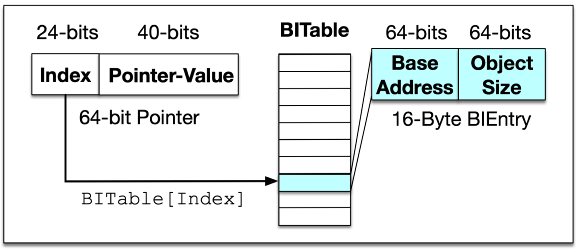


Fig. 5. Design of the Bounds-information Table (BITable).

compilation, without requiring any changes to the source code and without any compatibility issues due to changes to the binary layout. The hardware, on which this binary runs, transparently executes the bounds-checks for every load or store to detect memory-safety violations: we modify the load and store execution in hardware to additionally also access the BITable and obtain the bounds metadata for the bounds-check; we add a **bounds-information cache (BICache)** to limit the slowdown from accesses to the BITable in memory.

3.2 Software Support to Enable the Bounds-table

The BITable is maintained within a program's virtual address space and stores the bounds metadata for each heap object in the program. It is accessed on each load or store to a heap object to execute the bounds-check. We describe the design and implementation of the BITable below.

BITable Design. The BITable is organized as a linear table of 16-byte BIEntries, with each BIEntry containing a 64-bit base-address and a 64-bit object-size, as shown in Figure 5 (a more optimized design could accommodate a BIEntry in 12-bytes by storing 48-bit base and size fields). Each BIEntry is associated with a heap object; accessing the BIEntry corresponding to an object requires a single table-lookup ($BITable[index]$), using the index that is embedded into the pointer during object allocation, unlike prior works [28, 40] that access bounds metadata with multi-level table lookup using the pointer value itself.

BITable Implementation. The space for the BITable is reserved using an `mmap` at program initialization, with the `MAP_ANONYMOUS` flag, that results in physical pages being allocated lazily on access. Hence, the memory consumed by the BITable grows proportional to the number of `malloced` objects in the program. The virtual address of the base of the BITable and its size are stored in custom MSRs, i.e., model-specific registers ($BTBASE$, $BTSIZE$) in hardware that can be written to or read from using `wrmsr` and `rdmsr` instructions respectively; the OS saves and restores these registers on context switches (the BITable of multiple processes can co-exist in DRAM and does not need to be swapped out on context-switches). This allows the hardware to calculate the virtual address of a BIEntry as $BTBASE \mid (index << 4)$ while executing the bounds-check; to ensure such a concatenation of base and index is possible, the `mmap` ensures the base of the BITable is aligned to a power-of-2 larger than the table-size, so that the $BTBASE$ lower bits are 0s.

The BITable needs to be at least as large as the maximum number of live objects in the program (objects that are malloced but not yet freed). We observe that the SPEC CPU2017 workloads with the *ref* data-set have a maximum of 2.4M live objects in any program (despite hundreds of millions of objects being malloced/freed throughout the benchmark lifetime). With an abundance of caution, we set the BITable size to be 16M entries to support up to 16M live objects (malloced but not yet freed) at any point in time. We discuss how an even higher number of live objects can be supported in Section 3.4.

Listing 1. malloc_hook.

```
 .. //get a free BIEntry
 //index: the new entry
 p = real_malloc(size);
 writeBIEntry(index,p,size);
 return(p | (index << 40);
}
```

Listing 2. free_hook.

```
 //p: ptr passed to free
 index = p >> 40;
 p = p & 0xFFFFFFFFFF;
 readBIEntry(index, p);
 .. //check p matches BIEntry
 writeBIEntry(index, 0, 0);
 .. //add index to free list
 real_free(p);
}
```

BIEntry Allocation and Invalidation. We use LLVM-based instrumentation to initialize our malloc/free hooks, before main is executed. These hooks intercept subsequent calls to malloc and free from the program, where we create or delete the BIEntry and then call the internal memory allocation functions (real_malloc and real_free). We implement writeBIEntry and readBIEntry functions, responsible for writing a new entry in the BITable and reading an entry, respectively, using array accesses, as inline functions in our shared library.

Listings 1, 2 show the HeapCheck function hooks called on malloc/free. When a malloc is intercepted, we call the real_malloc, and store the returned base-address and the requested object-size in a BIEntry using writeBIEntry. For the first 16M mallocs, we use a new BIEntry in the table, or one of the entries invalidated on a free in FIFO order. The index of this BIEntry is then embedded in the top 24 bits of the pointer and returned to the program. When a free is intercepted, the index in the top bits of the pointer is used to get the BIEntry using readBIEntry and verify that the pointer value matches the object base. Then, the BIEntry is invalidated (base and size set to 0), and the real_free is called to free the object.

Program Memory Layout. Our scheme uses the top 24-bits of a pointer to store the index of a BIEntry. On 64-bit Linux systems with a four-level page-table, programs typically use a 48-bit user VA-space. Since our index bits overlap with the top 8-bits of a 48-bit user address, we need to constrain our program memory layout as shown in Figure 6, to avoid collisions of program addresses with heap object pointers containing the 24-bit index. In this layout, the heap grows upwards from 0x0 to 0xFFFFFFFFFF and the top 24-bits of a heap pointer (which are 0x000000 by default) can be used by the BITable index. To identify stack/global addresses, we ensure they have "0x00007F" in the top 24 bits (we keep the BITable index 0x7F unused), and hence the stack grows downwards from 0x7FFFFFFFFFFF to 0x7F0000000000. Thus, the stack/global region is limited to a 40-bit space (1 TB) and combined with a 1 TB heap, this limits the total VA space to 2 TB.[3] To support this program layout, we skip the usage of index value 0x7F for heap objects to avoid collision with stack addresses and also the index values from 0xFFFF80 – 0xFFFFFF to avoid

[3]The amount of VA space is configurable. For example, a program can have 8 TB VA space while using a flat bounds-table with 4M entries, which is sufficient for SPEC-CPU2017 applications (we use a bounds-table with 16M entries by default to keep some buffer). The VA space per program can be even further increased with a two-level bounds-table design: a smaller flat table and an overflow table, as discussed in Section 3.4.

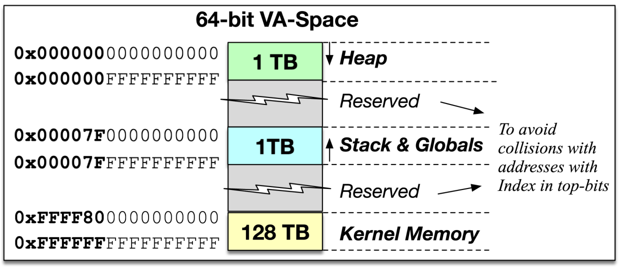


Fig. 6. Layout of a program's 64-bit Virtual-Address Space.

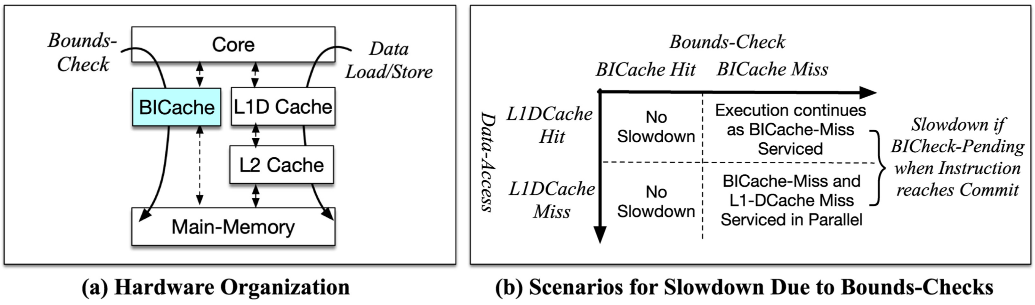

Bounds-Check
Core
Data Load/Store
BICache
L1D Cache
L2 Cache
Main-Memory
(a) Hardware Organization
Bounds-Check
BICache Hit
BICache Miss
Data-Access
L1DCache Hit
L1DCache Miss
No Slowdown
No Slowdown
Execution continues as BICache-Miss Serviced
BICache-Miss and L1-DCache Miss Serviced in Parallel
Slowdown if BICheck-Pending when Instruction reaches Commit
(b) Scenarios for Slowdown Due to Bounds-Checks

Fig. 7. (a) Hardware for HeapCheck includes a dedicated BICache for caching BITable entries, that is accessed for Bounds-check in parallel to L1-Dcache on Loads/Stores. (b) Assuming TLB-Hit for Data Access and Bounds-check, slowdown is incurred only if Bounds-check has a BICache Miss and the check is pending by the time the load/store instruction reaches commit stage.

collision with kernel addresses. The index `0x0` is reserved for the NULL pointer (pointer value "0") to support free called on NULL pointer (that is valid program behavior [5]).

3.3 Hardware-based Bounds-checks

HeapCheck detects out-of-bounds accesses and use-after-free references via bounds-checks performed transparently by the hardware on loads and stores. We describe the design and implementation of bounds-checks below.

Design of Bounds-checks. In HeapCheck, all loads and stores to heap objects (identified by the presence of an index in the top 24-bits) have a bounds-check included as a part of the load/store execution. The bounds-check involves using the index to lookup the corresponding BITable entry, obtaining the base and size of the object, and asserting that the access is within $[base, base + size]$; else, an out-of-bounds exception is triggered. If the BIEntry base and size are 0, then a use-after-free exception is raised. If the BIEntry has been reused, then a dangling pointer access is still detected due to mismatch in bounds with a high probability, but flagged as an out-of-bounds exception.

Implementation of the Bounds-checks. For the bounds-check, the address of the BIEntry for the bounds-check is calculated using the BITableBase register and index bits from the load/store address as $BTBASE \mid (Index << 4)$, and the address translation is identical to a regular load. The execution of the bounds-check begins when the virtual address of a load/store is ready, and continues in parallel to the load/store execution without impacting its execution. The bounds-check is only on the critical path of load/store commit stage, which commits a load/store only if the check passes; else it stalls until the bounds-check completes. The bounds-check itself requires simple logic: OR and Left-shift to calculate BITable-Address, in addition to a 48-bit Adder and a Comparator for the bounds-check (the adder can be avoided if the bounds metadata stores the end address of the object instead of the object size); this requires less than a few thousand logic gates of extra hardware.

Design of BICache. To minimize performance impact, we cache BITable entries in a dedicated BIcache, as shown in Figure 7, so that bounds-checks that hit in the BIcache have no impact on load/store latency. Our design uses a dedicated 8 KB eight-way BICache for the BITable entries. All accesses to the BITable generated in hardware for bounds-checks are routed through the BICache; similarly all loads and stores to the BITable generated from software for BITable management on malloc and free are routed through the BICache by the hardware (knowing the range of the BITable programmed in the MSRs). The design of the BICache is Virtually Indexed Physically Tagged (VIPT) and identical to the L1-Dcache with the same latency (but much smaller in size), to ensure that load/store execution perceives no performance impact if the bounds-check gets a hit in the BICache. On a miss, the entries can be accessed from the L2, Last-level cache, or the main memory as applicable; for ease of implementation in our simulated design, we directly service BICache misses from the main-memory. We extend the load/store queue entries to store the status of pending bounds-checks (storing a 48-bit BIEntry address, a 1-bit *checkIssued* flag, and a 1-bit *checkComplete* flag), in case a BICache miss delays the bounds-check.

3.4 Discussion

Scaling to Larger Number of BIEntries: To enable fast access to bounds metadata, we choose a flat-table-based design for the BITable supporting 4–16M entries using 22 to 24 top bits of a pointer as index bits; this restricts a program to 2–8 TB of VA space. However, it is also possible to support programs needing a larger number of BIEntries or systems where fewer pointer bits may be used as index, using a two level BITable design. Such a design would have (a) a flat BITable indexed by the top bits of the pointer, storing the bounds for as many object bounds as possible, and (b) a slower overflow table indexed by the pointer value like prior designs like Chex86 [40] or Intel's MPX [28], storing the bounds for the remaining objects. Workloads with fewer heap objects could exclusively use the flat BITable and have no performance degradation (e.g., almost half of the SPEC-CPU2017 workloads have less than 64K active objects at any time). As the number of active objects in a program increases beyond the capacity of the flat table, the performance would degrade gracefully, with the worst-case slowdown being no worse than prior designs. For all of the SPEC CPU2017 workloads, a single 4M entry BITable (using 22 top bits of the pointer as index) was sufficient for the bounds of all objects (<2.5M entries were used at the maximum) and an overflow table was not required. For workloads requiring a larger number of BIEntries, future work can explore hybrid bounds-table designs using fewer pointer bits as index.

Compatibility with Multi-Threading: Our `malloc` and `free` hook functions can be implemented in a thread-safe manner by using locks to ensure atomic updates to the BITable. Additionally, the coherence between the BICaches (having a VIPT design) across different cores is maintained using the existing cache-coherence fabric in hardware. The updates to the BITable from one core are reflected in accesses from other cores, without any extra software intervention. As long as the program itself is written in a thread-safe manner (e.g., no data race between a `free` and an access to the same object from different threads), and the internal memory allocator itself is thread-safe, our bounds-checking framework retains compatibility with multi-threaded programs.

4 ANALYSIS OF BUG DETECTION WITH HEAPCHECK

In this section, we first discuss the types of memory-safety bugs HeapCheck is able to detect, and then we discuss new out-of-bounds references we detected using HeapCheck.

4.1 Types of Memory-safety Bugs Detected

We tested HeapCheck with 25 programs from the How2Heap [41] exploit suite containing heap spatial and temporal safety bugs like out-of-bounds accesses, use-after-free, invalid-free, and double-free. HeapCheck detected the bugs in all 25 of these programs. Of these, we detected an Out-of-bounds access in eight exploits, Use-after-free in ten exploits, and Invalid/Double-Free in seven exploits.

The mechanism of detecting each of these classes of bugs is as follows:

- **Out-of-bounds Accesses:** On loads and stores, the bounds-check, performed in parallel to the load/store, checks the BIEntry and ensures the access is within object bounds. This allows HeapCheck to detect any out-of-bounds accesses before they occur.
- **Use-after-free:** If the bounds-check on a load/store finds the BIEntry to have base=0 and size=0, then the object was either recently freed or the BIEntry was uninitialized (object was never malloced). We flag both scenarios as errors. If the BIEntry is re-allocated between a Free and a Use-after-free, then the Use-after-free is detected as an out-of-bounds access with a very high probability, as the chance of the index being reused for an object overlapping in virtual memory with the freed object is quite low.
- **Double-free and Invalid-free:** On a free, it is verified that the pointer-to-be-freed matches the object-base in the BIEntry. If a mismatch is detected, it is indicative of an invalid or a double-free bug. If the index in the top-bits of the pointer is not a valid value, or if the BIEntry object-base does not match, then an invalid-free bug is flagged; else, if the BIEntry base and size are 0, then a double-free bug is flagged.

4.2 Out-of-bounds References in Glibc and SPEC-CPU2017

We modeled HeapCheck bounds-checks in Gem5 and tested it with 13 C/C++ SPEC-CPU2017 binaries compiled using clang-11 and Glibc-v2.27. HeapCheck detected out-of-bounds reads in several SPEC benchmarks when pointer-accesses were checked against the allocation-bounds of an object (the 16-Byte aligned size allocated by malloc is stored in the BIEntry); note that our framework also supports byte-granularity bounds-enforcement by storing object-size requested by the program in the BIEntry instead of allocated-size.

Figure 8 shows the functions where we observe out-of-bounds accesses across 13 SPEC C/C++ binaries run for 11 billion instructions on Gem5 simulator with our HeapCheck framework. Overall, we observed out-of-bounds reads in 87 lines of code—80 of these were in highly optimized Glibc-v2.27 functions used for string handling (including strlen, strchr, strcmp, etc.) and 7 lines of code in four user-functions of *blender* program. The maximum number of bytes by which these accesses go out-of-bounds is 62 bytes for the Glibc functions and only 4 bytes for the functions in *blender*.

All of these out-of-bounds references were observed to be due to SIMD instructions that load data from the memory to a SIMD register. Figure 9 shows the instructions for which we observe out-of-bounds accesses. The instructions in the Glibc string handling functions include SIMD move (`MOVDQA`, `MOVDQU`, `MOVHPD`, `MOVLPD`) or compare (`PCMPEQB`) or minimum (`PMINUP`) instructions, while those in blender included SIMD arithmetic instructions (`MUSS`, `ADSS`, `SUBSS`). On inspection, the out-of-bounds references in *blender* occurred when the compiler used unaligned 16-byte SIMD loads to access memory at the boundary of an object resulting in partially out-of-bounds accesses, when compiled with the O3 flag; these disappeared on using the O0 flag as SIMD arithmetic was not used. The out-of-bounds references in Glibc functions appear to be more serious as they were

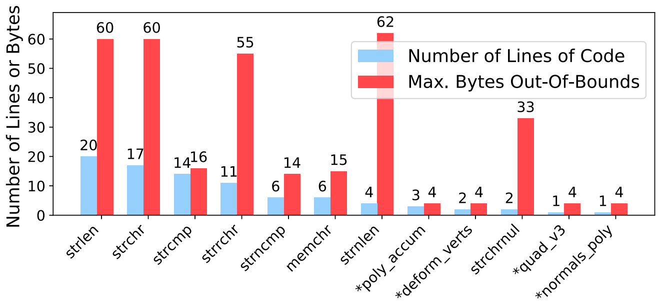


Fig. 8. Functions in SPEC-CPU2017 programs where Out-of-bounds accesses were detected with HeapCheck. Such accesses were observed either in heavily optimized Glibc functions or in functions of the blender program.

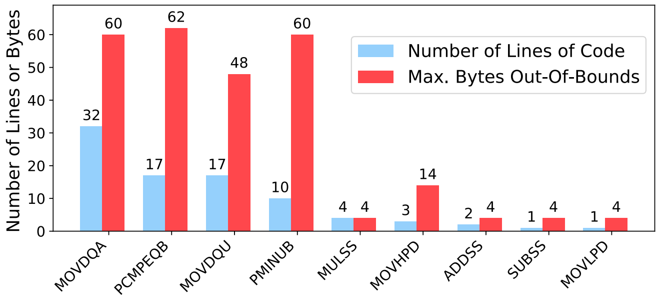


Fig. 9. Instructions in SPEC-CPU2017 applications that caused Out-of-bounds accesses. All such out-of-bounds references are due to SIMD instructions in optimized Glibc functions or SIMD instructions inserted by the compiler when O3 flag is used.

in the shared-library (`libc.a`) distributed with Ubuntu 18.04, and as they access memory that is out-of-bounds by up to 62 bytes.

We manually inspected *strlen*, the function with the maximum number of lines of code (20) that have out-of-bounds accesses: a majority of these (11 of 20) are due to the instruction PCMPEQB, which performs byte-wise comparison of 16 B memory and register operands. *strlen* uses these instructions to perform fast checks for the "`\0`" (null) character in an input string, to compute the string length. Listing 3 shows the assembly code for *strlen* generated from the object dump of `libc.a` in Glibc v2.27. The code aggressively issues three 16-byte comparisons (up to four in other code sequences) before using a `test` and a `jump` instruction to stop the comparison on a null, which can end up accessing memory up to 47-bytes outside of a string object (up to 63-bytes out-of-bounds with four 16-byte comparisons). This behavior has been acknowledged by one of the authors of the strlen routine in a *Stack Overflow* post [7], but so far it is assumed to not result in any vulnerability as the illegally referenced data on out-of-bounds reads is not explicitly used by subsequent instructions. However, such behavior of installing illegally obtained data in registers is quite risky as such illegal data are accessible speculatively and can be leaked out via transient-execution attacks like Spectre v2. Future works could demonstrate proof-of-concept attacks leveraging these security weaknesses.

We also checked if the default version of **Address Sanitizer (ASAN)** [37] is able to detect these out-of-bound references by running these binaries with ASAN. However, ASAN is unable to detect such out-of-bounds accesses in shared library code like Glibc by default, as it requires shared libraries to be recompiled to detect any out-of-bounds accesses in them unlike our work, which detects these without any additional requirements; also ASAN by default cannot detect unaligned loads accessing out-of-bounds memory [32] that we detect in *blender* with HeapCheck.

Listing 3. Disassembly of `__strlen_sse2` from `libc.a`.

```
0x3e: pcmpeqb  0x10(%rax),%xmm1 //16-byte compare
0x43: pcmpeqb  0x20(%rax),%xmm2 //16-byte compare
0x48: pcmpeqb  0x30(%rax),%xmm3 //16-byte compare
0x4d: pmovmskb %xmm1,%edx //16-bit result in edx
0x51: pmovmskb %xmm2,%r8d //16-bit result in rd8
0x56: pmovmskb %xmm3,%ecx //16-bit result in ecx
...
// code-block assembles 48-bit result in rdx
..
0x79: test     %rdx,%rdx //checks if '\0' found
0x7c: je       100 <__strlen_sse2+0x100> //jump
```

4.3 Protection of Bounds Metadata from Adversarial Tampering

Our assumption is that the victim program with the memory-safety bugs is itself not malicious and the adversary only interacts with the victim program via malicious data inputs to exploit the memory-safety bugs (a classic example of such a setting being the Heartbleed vulnerability). So the adversary does not have the ability to perform arbitrary read or write accesses nor the ability to execute arbitrary code in the victim's address space. So the bounds metadata in the victim code is safe from direct reads or tampering by an adversary.

As a defense in depth strategy, below we describe extensions to further fortify the HeapCheck metadata in the victim from corruption by any unknown means. The two types of metadata used by HeapCheck, are the BIEntry in the BITable containing the bounds metadata, and the index bits in the pointer used to access the bounds.

Protecting the BITable: To further protect the BITable, the BITable itself can be sandboxed within the victim address space to prevent spurious reads or writes from code outside the trustworthy HeapCheck library code. This can be achieved by having the HeapCheck library code use the `mprotect` syscall to set/reset the read and write permission bits (in the page table) of the BIEntry page before/after the BIEntry update on mallocs and frees. This ensures that the read and write permissions for the BITable are always set to false outside the trustworthy library code and any spurious access from software to the region causes an exception. Similar sandboxing can also be achieved using Intel's **Memory Protection Keys (MPK)** [31] with negligible overhead (as it changes permissions without relying on modifying the page table). Accesses to the BITable for bounds-checking, which are inserted within the hardware automatically on data loads/stores, can continue unhindered by safely ignoring these permission bits.

Protecting the Pointer Index-bits: While an adversary cannot directly read or modify the index bits of the pointer, there is a possibility that the pointer arithmetic may be susceptible to an overflow or underflow, which could corrupt the index-bits. However, this would typically be flagged as a bounds-mismatch on a pointer dereference as the index bits would point to a different entry with unpredictable bounds. This can also be prevented by using guard-bits before and after the index bits, which would prevent such overflow or underflow from affecting the index bits.

5 PERFORMANCE EVALUATIONS

We first discuss our evaluation methodology, and then we discuss the overheads of our software and hardware modifications.

5.1 Methodology

We package the software changes for HeapCheck (including the malloc/free hooks) as a shared library and use instrumentation added with LLVM10 to add an initialization function before the program *main*. The hardware changes for HeapCheck are modeled in Gem5 v20.0 [21]. For our performance evaluations, we use 13 C/C++ benchmarks available in SPEC-CPU2017 [8] with the *ref* dataset. We evaluate the overheads of our software instrumentation by running the instrumented binaries to completion on a native machine (Intel Xeon CPU E-2174G at 3.80 GHz), and comparing them against uninstrumented binaries. For hardware overheads, we use the instrumented binary and run it with and without the bounds-checks on Gem5 in System-Call Emulation mode. We fast-forward the first 10 billion instructions to skip the initialization phase and warmup the caches, and track statistics for 1 billion instructions. The hardware configuration we use for Gem5 is shown in Table 1.

5.2 Performance Overheads

First, we discuss the overheads of the software instrumentation needed for BITable management, then the overheads of hardware-based bounds-checks and then dissect the overheads for the worst-performing workloads and evaluate sensitivity of performance to cache size.

Software Instrumentation Overheads: We first evaluate the slowdown due to the malloc/free instrumentation. Figure 10 shows the execution time of applications linked with our shared library intercepting malloc/free calls to update the BITable, normalized to the execution time of uninstrumented binaries. To run these instrumented binaries natively, we allocate BIEntries on mallocs, but do not embed the index bits in the pointer; on frees, we delete a random BIEntry. On average, the SW instrumentation for BITable management (without bounds-checks) adds only 0.5% slowdown across all programs. Workloads with the high malloc frequency (*gcc*, *perlbench*) have slowdowns of up to 1.8%–2.4% due to increased cache accesses for BITable updates. Other workloads with fewer mallocs see negligible performance impact.

Hardware Bounds-check Overheads: Using the instrumented binaries, we evaluate the slowdown due to the hardware-based bounds-checks in Gem5. Figure 11 shows the execution time for 1-billion instructions of our instrumented binaries running with Bounds-checks, normalized to execution time of the same binary without Bounds-checks. On average, the bounds-checks add only 1% slowdown. The main driver of these overheads is the memory accesses incurred by bounds-checks due to misses in the BICache. Workloads such as *xalancbmk*, *gcc* and *parest*, with high frequency of mallocs, tend to have smaller buffers and hence fewer buffer accesses sharing the same index. This results in larger working sets of bounds metadata, causing relatively higher BICache miss rates (2% to 16%) and higher slowdown (1% to 6%). Other workloads, with higher than 99% BICache hit rate, have negligible slowdown. Overall, our design achieves exceptionally high locality of BITable accesses (average BICache hit rate of 97%); with such locality, any performance impact on TLB is negligible, especially if the BITable uses large pages.

Dissecting Worst-case Slowdown from Bounds-checks: To understand the slowdown in the worst-performing workloads like *xalancbmk*, *gcc*, *parest*, we study the scenarios in which they incur BICache misses. More than 98% of the BICache misses are incurred by bounds-checks on load operations (only 2% of the misses occur on stores). Figure 12 shows the break-down of BICache-Miss for load operations based on where the load was serviced from (by absolute numbers and by percentage). *xalancbmk* has the most BICache misses (as it has the highest miss rate), and consequently the highest slowdown. However, *gcc* has a higher miss rate than *parest*, but incurs

Table 1. Gem5 Hardware Configuration

Processor	
Core	Single-core, Out-of-order Execution, 3.5 GHz ROB—192 entries, LQ—32 entries, SQ—32 entries
Cache-Hierarchy	
L1-DCache, L1-ICache	32 KB/core, 8-way, 64 B line-size, 2-cycle latency
BICache	8 KB/core, 8-way, 64 B line-size, 2-cycle latency
L2-Cache	2 MB/core, 16-way, 64 B line-size, 20-cycle latency
DRAM Memory-System	
Bus frequency	1200 MHz (DDR 2.4 GHz)
DRAM Timings	tCL=14 ns, tRCD=14 ns, tRP=14 ns
DRAM Organization	1-channel, 1 KB Row-Buffer

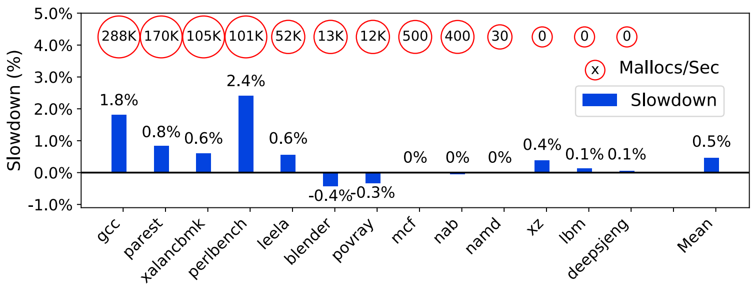


Fig. 10. Slowdown due to Software Instrumentation for BITable management, with native execution (averaged over five runs). On average, SW instrumentation adds 0.5% slowdown.

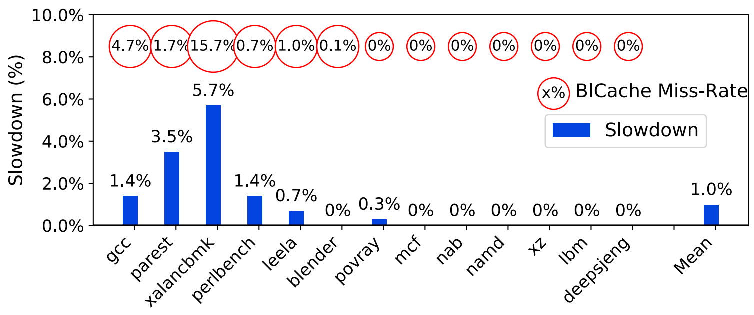


Fig. 11. Performance Impact of Hardware Bounds-checks, modeled in Gem5. On average, the bounds-checks add only 1% slowdown, owing to high BICache hit rates (98%).

lesser slowdown—because it has a much lesser fraction of BICache misses when the load is an L1-Hit (that impacts performance more than a BICache-Miss on a load that was an L1-Miss). This lack of locality in BICache accesses on L1-Cache Hits is the main driver for the slowdown for *xalancbmk* and *parest*. However, this can be addressed by making the BIEntry allocation algorithm

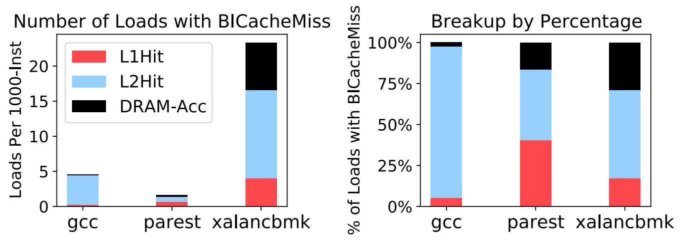


Fig. 12. Breakdown of Scenarios where Bounds-checks on Loads have a BICache Miss.

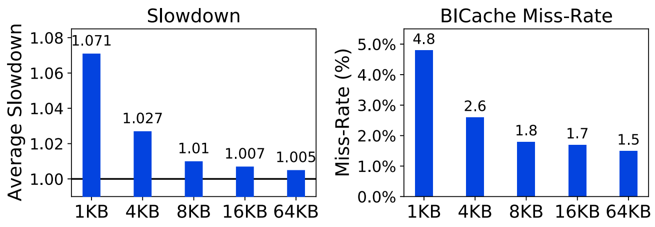


Fig. 13. Slowdown due to Bounds-checks vs. BICache Size.

locality-sensitive, especially for sub-cacheline objects that are common in these benchmarks, to reduce the overheads for these workloads.

Sensitivity to BICache Size: While our evaluations use a default BICache size of 8 KB, we also analyzed the overheads with other BICache sizes. Figure 13 shows the slowdown due to Bounds-checks and BICache-Miss rates as the BICache size is varied from 1 KB to 64 KB. As the BICache size increases, slowdown decreases from 7% (1 KB) to 3% (4 KB) to 1% (8 KB). With further increase in BICache size, the decrease in slowdown is marginal. This is because beyond 8 KB, the average BICache-Miss rates do not decrease much, as the miss rates for most workloads is less than 1%.

5.3 Memory Overheads

Figure 14 shows the overheads in memory consumption of HeapCheck. In our default design, the BITable needs additional memory of up to 256 MB (16M entries with 16-Byte per entry) and consumes 39% extra memory on average across SPEC-2017 benchmarks; this is comparable to the overhead (38%) of prior work Chex86 [40] and much lower than that of several prior bounds-checking solutions in Table 2, which consume 50%–90% extra memory on average. Moreover, a memory optimized implementation of our BITable (BITable-MemOpt) is also possible with 12-Byte entries (storing 48-bit base and bounds) and 8M entries, which only adds 17% extra memory on average. These overheads are negligible in comparison to prior shadow-memory-based solutions like ASAN that increase memory consumption to more than 300%. This is because, unlike ASAN where the memory overhead is proportional to the program memory footprint, HeapCheck's overheads are proportional to the number of objects allocated in a program; additionally, even though a `mmap` reserves the whole BITable in HeapCheck at the beginning, the lazy physical page allocation ensures physical memory is consumed only for the BIEntries that are initialized and accessed.

With regards to memory-bandwidth consumption, a simplistic design where the BICache misses are serviced directly by the main-memory can increase the memory bandwidth by 29%. A practical design however would cache BICache entries in the L2 or the LLC, and misses from the BICache would get serviced from these caches; this can significantly reduce the extra memory accesses to ensure that it does not cause any significant performance impact.

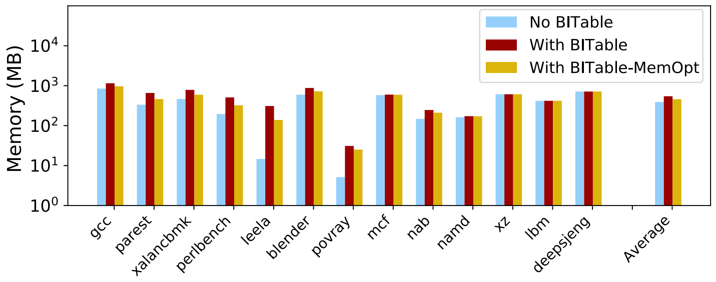


Fig. 14. Memory Consumption with and without HeapCheck.

5.4 Discussion: Mitigating Spectre v1 with Bounds-checking

Like several prior memory-safety works, our threat model primarily focuses on memory-safety vulnerabilities, and transient execution attacks are out of scope. Hence our design executes the BITable access for bounds-checks and the associated data access in parallel during speculative execution and serializes them only at commit time. But note that this is not the main driver of performance of our design. Our low performance overheads originate from the fact that the BITable accesses have a BICache hit at an average hit rate of 98%, which is serviced in a single-cycle.

Our design also holds the promise of preventing Spectre-v1 if the BITable access and data access are serialized during out-of-order execution. The performance impact of such serialization promises to be minimal compared to comparable solutions as our bounds-metadata is more likely to be found in the BICache (our 8 KB BICache has 98% avg hit rate). However, note that bounds-checking solutions cannot prevent the majority of transient execution attack variants (e.g., Spectre-v2,v3,v4, SpectreRSB, MDS, LVI, etc.), and holistic defenses are likely to be required to completely mitigate such vulnerabilities. Hence, we choose to keep transient execution attacks out of our threat model. Future works may explore how our work can support defenses for transient execution attacks.

6 RELATED WORK

In this section, we describe prior solutions for memory safety, and compare our work against them.

6.1 Bounds-checking-based Solutions

AOS [18] proposes using the unused pointer bits to store a **Pointer-Authentication-Code (PAC)**, which functions as a key to a hashed bounds table entry with the associated bounds. But the hashed bounds table design in AOS may cause both performance and security challenges, while its requirement of adding new PAC generation instructions into a program causes compatibility issues. We discuss these concerns in detail next:

- **Performance:** The hashed bounds-table accesses in AOS result in random access patterns lacking in spatial locality, as objects that lie in the same memory cache line or neighboring cache lines have their bounds-entries scattered across the memory without locality. Moreover, the 16-bit PAC, used as the bounds-index, only has 64K different values. If a program has 2–4M active objects at a time (we observe 3 SPEC-CPU17 workloads in this range), then each PAC is shared by 32–64 bounds. As a cacheline stores eight bounds entries (8-byte each), the lookup of a single object-bounds could require accessing 4 to 8 cache lines from memory. This could result in extremely poor cache hit rates and a high slowdown in the worst-case. AOS (Section IX-A in Reference [18]) acknowledges that it suffers a worst-case slowdown

Table 2. Comparison with Prior Solutions Based on Type of Memory Safety, Backward Compatibility, Slowdown, and Memory Overhead

Type	Solution	Spatial Safety	Temporal Safety	Backward Compatibility	Slowdown	Memory Overhead
Bounds-checking	**HeapCheck**	**Heap**	**Heap**	✓	**1.5% (SPEC-2017)**	**17% (SPEC-2017)**
Bounds-checking	AOS [18]	Heap	Heap	X	8.4% (SPEC-2006)	Not available
Bounds-checking	Chex86 [40]	✓	✓	✓	15% (SPEC-2017)	38% (SPEC-2017)
Bounds-checking	Hardbound [9]	✓	X	X	5–9% (Olden)	55% (Olden)
Bounds-checking	Watchdog [23]	✓	✓	X	24% (SPEC-2000)	56% (SPEC-2000)
Bounds-checking	MPX [28]	✓	X	X	50% (SPEC-2006)	90% (SPEC-2006)
Bounds-checking	BOGO [52]	✓	✓	X	60% (SPEC-2006)	36% (SPEC-2006)
Bounds-checking	CHERI [47]	✓	X	X	18% (Olden)	90% (Olden)
Bounds-checking	CHERIvoke [48]	X	✓	X	5% (SPEC-2006)	12.5% (SPEC-2006)
Bounds-checking	Cornucopia [13]	X	✓	X	2% (SPEC-20006)	33% (SPEC-20006)
Trip-wire	REST [42]	Linear	Until Realloc	X	25% (SPEC-2006)	Not-available
Trip-wire	Caliform [35]	Linear	Until Realloc	X	14% (SPEC-2006)	Not-available
Memory-Bug Detector	ASAN [37]	Linear	Until Realloc	✓	75% (SPEC-2006)	220% (SPEC-2006)

Backward Compatibility is defined as no new instructions added to the binary and no change in binary layout. Slowdown and Memory Overhead use values reported in prior works.

of >2X for *gcc* due to "cache pollution caused by bounds metadata." In contrast, HeapCheck uses a linear bounds-table design that enjoys excellent bounds-cache locality (98% hit rate in 8 KB cache) and only has a worst-case slowdown of 6% (and 1.8% for *gcc*).

- **Security:** AOS can suffer from PAC collisions for applications with millions of active objects, as the 16-bit PAC only has 64K unique values. Such PAC collisions have a possibility of resulting in bounds-mismatches with security implications. AOS (Section VI in Reference [18]) acknowledges that "PAC collisions can result in a false-positive," which results in a non-zero probability of false declaration of an attack. In contrast, our linear bounds table design does not incur such false positives as it ensures that all objects are provided unique index values.
- **Compatibility:** AOS requires adding new instructions into the program source code (i.e., using ARM's ISA extensions) for generating the PAC for a pointer. Hence, its protection is not backwards compatible with applications that do not have their source code available or where recompilation is not possible. In contrast, HeapCheck is particularly designed to not require any new instructions or changes to the source code or binary layouts, to ensure pointers in legacy code and shared libraries are also protected.

Chex86[40] provides spatial and temporal safety by storing object-bounds in a capability table, indexed with a capability-ID. The key challenge with such a design is the additional overhead of mapping capability-IDs to pointers. Transfer of capability-IDs between pointers occurs rule-based for pointers transferred via register operations. For pointers derived from memory locations (e.g., pointers spilled to stack, pointer-chasing patterns), Chex86 requires speculation hardware to predict the capability-ID and a lookup to a five-level Pointer-Alias Table using the pointer-value to confirm the predicted value is correct. Mis-speculations of the capability-ID can delay data accesses considerably as the five-level alias-table lookups and the capability-table lookup fall on the critical path of the data access, causing high slowdown. In comparison, HeapCheck has automatic metadata tracking for *all* pointers (even spilled pointers or pointer chasing patterns) as the index to the bounds metadata is always inline with the pointer-value in the top-bits and propagates automatically during all pointer operations (ALU operations or pointer read/writes). This limits the slowdown of HeapCheck.

Hardbound [9] proposed hardware-based bounds-checks by storing bounds-metadata in shadow-memory and provided spatial safety. **Watchdog** [23] extended this design to also provide temporal safety at added cost (24% average slowdown) by associating pointers with a unique identifier that can be revoked (also stored in the shadow-memory). The main problem with such solutions is that each memory word of the program requires a shadow-memory entry storing the bounds. So the memory overhead becomes proportional to the program memory footprint in such solutions. Our solution is quite different as it stores the bounds-metadata per object and hence our memory-overhead is proportional to the number of objects in the program (much smaller than memory footprint). Consequently, Hardbound and Watchdog have average memory overhead of >50%, while ours is much smaller, 17% on average. Additionally, shadow-memory bounds-metadata accesses in Hardbound and Watchdog have limited locality, as different shadow-memory locations are accessed for loads/stores to different words of a single object, resulting in high slowdown. In our work, loads/stores to different words of a single object access the same BITable entry, enabling bounds-checks to have more than 98% hit-rate in the BICache and have negligible slowdown.

MPX [28] and **BOGO** [52] offer spatial safety and temporal safety, but incur high overheads (50–60% on average) as their bounds-checks require extra explicit instructions and involve expensive table-lookup as the bounds-table is organized as a two-level trie. In contrast, our bounds-checks require at-most a single table-lookup that has high temporal locality and is inserted transparently in hardware during load/store execution.

Fat-pointer-based solutions like **CHERI** [46–48] provide memory safety at the cost of invasive changes to the ISA and the binary layout, that impacts compatibility with legacy code. **Low-fat Pointers** [19] avoid compatibility issues by implicitly encoding the object size in the pointer value, enabled by placing the object in a suitable region of memory. While this provides spatial safety, this comes at the cost of being incapable of providing temporal safety. This is because accesses to dangling (freed) pointers are always valid, as there is no way to revoke the pointer value, and the bounds encoded in it, on frees. **In-fat Pointers** [49] is an extension of low-fat pointers, that provides spatial safety at sub-object granularity, but similarly does not provide temporal safety. In comparison, our work only stores the index to the bounds-metadata in the top bits of the pointer and stores the actual bounds metadata in a disjoint table. So it is able to maintain compatibility with legacy shared libraries and provide both temporal and spatial memory safety for all heap object pointers, even those passed to legacy shared libraries, while ensuring negligible slowdown.

No-fat [14] is a contemporary work, that uses a binning memory allocator to implicitly encode the object size and the base in the pointer value itself for spatial safety. But because this metadata is implicit, it needs to perform bounds-checks on all pointer arithmetic and when the pointer is written to memory or passed to a function, that can result in performance overheads (8% on average). As, HeapCheck stores the bounds-table index explicitly in the top bits of the pointer, it can provide memory safety with only bounds-checks on pointer dereference, resulting in lesser slowdown (less than 2% on average). Additionally, HeapCheck has better compatibility with legacy code, as No-fat requires insertion of new ISA instructions for bounds-checking in application code and subsequent recompilation.

6.2 Probabilistic Solutions

Trip-wire-based solutions such as **REST** [42] and **Caliform** [35] provide low-cost detection of memory errors (2%–18% slowdown), by inserting magic values (trip-wires) at an object or sub-object granularity and checking for them in hardware, to detect out-of-bounds accesses that activate such trip-wires. However, such solutions cannot detect larger out-of-bounds accesses, that access memory beyond the trip-wire. Our solution provides precise detection of all out-of-bounds accesses, at better performance.

Memory Tagging-based solutions like **SPARC's SSM** [29] or **ARM's MTE** [4, 38], assign a tag or "color" for an object-pointer pair, maintain these tags separately for both, and check if the tags of a pointer and the accessed memory match on a pointer dereference. While such solutions have negligible slowdown (<5% [39]), they are only able to detect errors probabilistically as they use 4-bit tags (stored in the top-bits of the pointer), that are reused for different objects, leading to false negatives. Our work also repurposes the pointer-bits, but in contrast, uses them to store the index to the actual bounds-information; hence our work provides precise enforcement of object-bounds (higher coverage) at comparable slowdowns.

6.3 Other Solutions

Frameworks such as **Address Sanitizer** [37], **Valgrind** [27], and others, are powerful software tools detecting memory-safety bugs (and other memory errors), as they provide rich debugging information both for heap and stack, and are compatible with existing source code (only requiring recompilation). However, they can incur prohibitive overheads (ASAN-75%, Valgrind 4-22x) that makes them better suited for development/test than for runtime protection of operational systems.

Other solutions providing code and data pointer integrity, like **ARM's Pointer Authentication** Code [33] and **Zero** [15], prevent pointer tampering and exploits due to memory-safety vulnerabilities like control-flow or data-flow hijacking, return oriented programming, and others. However, such solutions do not prevent the root-cause, i.e., buffer-overflows or use-after-free like bugs, and continue to be vulnerable to corruption of non-pointer data structures due to such bugs.

7 CONCLUSION

Memory-safety bugs in C/C++ programs have been a leading cause of vulnerabilities for over three decades; yet an effective and practical-to-adopt memory-safety solution has thus far been elusive. In this work, we presented a hardware-based solution that prevents errors like buffer overflows and use-after-free in heap objects virtually for free. Our solution, HeapCheck, has minimal performance overhead (1.5% slowdown), maintains compatibility with legacy code and has detected 87 lines of code with out-of-bounds reads in Glibc functions and SPEC-CPU2017 workloads, and provides an effective and low-cost solution to detect and prevent memory-safety bugs.

REFERENCES

[1] NIST. 2014. Buffer Overflow (BOF) Examples—Heartbleed. Retrieved from https://samate.nist.gov/BF/Examples/BOF.html.

[2] CWE. 2019. 2019 CWE Top 25 Most Dangerous Software Errors. Retrieved from https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html.

[3] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, Fabian Monrose (Ed.). USENIX Association, 51–66. Retrieved from http://www.usenix.org/events/sec09/tech/full_papers/akritidis.pdf.

[4] ARM. 2019. Armv8.5-A Memory Tagging Extension. Retrieved from https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[5] Jeffrey P. Bigham. 2006. Calling free() on a NULL pointer. Retrieved from http://www.manticmoo.com/articles/jeff/programming/c/free-with-null-pointer.php.

[6] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *Proceedings of the 9th International Symposium on Code Generation and Optimization (CGO'11)*. IEEE, 213–223. https://doi.org/10.1109/CGO.2011.5764689

[7] Stephen Canon. 2020. Vectorized strlen getting away with reading unallocated memory. Retrieved from https://stackoverflow.com/a/25574201/1011788.

[8] Standard Performance Evaluation Corporation. SPEC CPU 2017. Retrieved from https://www.spec.org/cpu2017/.

[9] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: Architectural support for spatial safety of the C programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, Susan J. Eggers and James R. Larus (Eds.). ACM, 103–114. https://doi.org/10.1145/1346281.1346295

[10] Gregory J. Duck and Roland H. C. Yap. 2016. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction (CC'16)*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 132–142. https://doi.org/10.1145/2892208.2892212

[11] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. 2017. Stack bounds protection with low fat pointers. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17)*. The Internet Society. Retrieved from https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/stack-object-protection-low-fat-pointers/.

[12] Eclypsium. 2020. There's a Hole in the Boot—Boothole (CVE-2020-10713). Retrieved from https://eclypsium.com/2020/07/29/theres-a-hole-in-the-boot/.

[13] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, et al. 2020. Cornucopia: Temporal safety for CHERI heaps. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'20)*. 1507–1524.

[14] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. 2021. No-FAT: Architectural support for low overhead memory-safety checks. In *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*. 916–929. https://doi.org/10.1109/ISCA52012.2021.00076

[15] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, and Simha Sethumadhavan. 2021. ZeRO: Zero-overhead resilient operation under pointer integrity attacks. In *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*. 999–1012. https://doi.org/10.1109/ISCA52012.2021.00082

[16] Yuseok Jeon, Wookhyun Han, Nathan Burow, and Mathias Payer. 2020. FuZZan: Efficient sanitizer metadata design for fuzzing. In *Proceedings of the USENIX Annual Technical Conference (USENIX-ATC'20)*.

[17] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A safe dialect of C. In *Proceedings of the General Track USENIX Annual Technical Conference*, Carla Schlatter Ellis (Ed.). USENIX, 275–288. Retrieved from http://www.usenix.org/publications/library/proceedings/usenix02/jim.html.

[18] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. 2020. Hardware-based always-on heap memory safety. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*. IEEE, 1153–1166.

[19] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight Jr., and André DeHon. 2013. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM, 721–732. https://doi.org/10.1145/2508859.2516713

[20] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. 2021. DOLMA: Securing speculation with the principle of transient non-observability. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security'21)*.

[21] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. Retrieved from https://arxiv.org/abs/2007.03152.

[22] Matt Miller. 2020. SSTIC-2020. Pursuing Durably Safe Systems Software. Retrieved from https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2020_06_SSTIC.

[23] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA'12)*. IEEE, 189–200. https://doi.org/10.1109/ISCA.2012.6237017

[24] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, Michael Hind and Amer Diwan (Eds.). ACM, 245–258. https://doi.org/10.1145/1542476.1542504

[25] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: Compiler enforced temporal safety for C. In *Proceedings of the 9th International Symposium on Memory Management (ISMM'10)*, Jan Vitek and Doug Lea (Eds.). ACM, 31–40. https://doi.org/10.1145/1806651.1806657

[26] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, John Launchbury and John C. Mitchell (Eds.). ACM, 128–139. https://doi.org/10.1145/503272.503286

[27] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 89–100. https://doi.org/10.1145/1250734.1250746

[28] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX explained: A cross-layer analysis of the intel MPX system stack. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 2 (2018), 28:1–28:30. https://doi.org/10.1145/3224423

[29] Oracle. 2015. Hardware-Assisted Checking Using Silicon Secured Memory (SSM). Retrieved from https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html.

[30] Hilarie K. Orman. 2003. The morris worm: A fifteen-year perspective. *IEEE Secur. Priv.* 1, 5 (2003), 35–43. https://doi.org/10.1109/MSECP.2003.1236233

[31] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software abstraction for Intel memory protection keys (intel MPK). In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'19)*. 241–254.

[32] Mitch Phillips. 2012. Address Sanitizer Algorithm. Retrieved from https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm.

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

[34] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.* 15, 1 (2012), 2:1–2:34. https://doi.org/10.1145/2133375.2133377

[35] Hiroshi Sasaki, Miguel A. Arroyo, M. Tarek Ibn Ziad, Koustubha Bhat, Kanad Sinha, and Simha Sethumadhavan. 2019. Practical byte-granular memory blacklisting using califorms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. ACM, 558–571. https://doi.org/10.1145/3352460.3358299

[36] Kostya Serebryany. 2017. OSS-fuzz—Google's continuous fuzzing service for open source software. USENIX Association, Vancouver, BC.

[37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. Address sanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. Retrieved from https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[38] Kostya Serebryany and Sudhi Herle. 2019. Adopting the Arm Memory Tagging Extension in Android. Retrieved from https://security.googleblog.com/2019/08/adopting-arm-memory-tagging-extension.html.

[39] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. Retrieved from https://arXiv:cs.CR/1802.09517.

[40] Rasool Sharifi and Ashish Venkat. 2020. CHEx86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture, (ISCA'20)*. IEEE, 762–775. https://doi.org/10.1109/ISCA45697.2020.00068

[41] Shellphish. 2020. How2Heap Github Repository. Retrieved from https://github.com/shellphish/how2heap.

[42] Kanad Sinha and Simha Sethumadhavan. 2018. Practical memory safety with REST. In *Proceedings of the 45th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'18)*, Murali Annavaram, Timothy Mark Pinkston, and Babak Falsafi (Eds.). IEEE, 600–611. https://doi.org/10.1109/ISCA.2018.00056

[43] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for security. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'19)*. 1275–1295.

[44] Laszlo Szekeres, Mathias Payer, Tao Wei, and R. Sekar. 2014. Eternal war in memory. *IEEE Secur. Priv.* 12, 3 (2014), 45–53. https://doi.org/10.1109/MSP.2014.44

[45] Pengfei Wang and Xu Zhou. 2020. SoK: The progress, challenges, and perspectives of directed greybox fuzzing. Retrieved from https://arXiv:2005.11907.

[46] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony C. J. Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel Wesley Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. 2019. CHERI Concentrate: Practical compressed capabilities. *IEEE Trans. Comput.* 68, 10 (2019), 1455–1469. https://doi.org/10.1109/TC.2019.2914037

[47] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC

in an age of risk. In *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA'14)*. IEEE, 457–468. https://doi.org/10.1109/ISCA.2014.6853201

[48] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel Wesley Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising pointer revocation using CHERI capabilities for temporal memory safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. ACM, 545–557. https://doi.org/10.1145/3352460.3358288

[49] Shengjie Xu, Wei Huang, and David Lie. 2021. 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)*. ACM, New York, NY, 224–240. https://doi.org/10.1145/3445814.3446761

[50] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. 2010. PAriCheck: An efficient pointer arithmetic checker for c programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS'10)*, Dengguo Feng, David A. Basin, and Peng Liu (Eds.). ACM, 145–156. https://doi.org/10.1145/1755688.1755707

[51] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative taint tracking (STT) a comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 954–968.

[52] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2019. BOGO: Buy spatial memory safety, get temporal memory safety (almost) free. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 631–644. https://doi.org/10.1145/3297858.3304017

Received May 2021; revised September 2021; accepted October 2021