Hardware Support for Low-Cost Memory Safety

Rick Boivie [†], IBM Research, rhboivie@us.ibm.com
Gururaj Saileshwar [†] [*], IBM Research, gururaj.s@gatech.edu
Tong Chen, IBM Research, chentong@us.ibm.com
Benjamin Segal, IBM Research, bpsegal@us.ibm.com
Alper Buyuktosunoglu, IBM Research, alperb@us.ibm.com

***Abstract*—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 binary-layout or source-code. Consequently, memory-safety errors continue to exist in production-software and are a principal cause of security problems.**

In our project at IBM, we worked on a minimally-invasive and low-cost hardware-based bounds-checking framework for preventing out-of-bounds accesses and use-after-free errors. The key idea is to re-purpose ”unused bits” in a pointer to store an index into a bounds-information table that can be used to catch out-of-bounds errors and use-after-free errors without any change to the binary layout. Using this bounds-checking framework, we implement a design for preventing Out-of-Bounds accesses and Use-After-Free for heap-objects, that are responsible for the majority of memory-safety errors in the wild.

I. 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 [22], including high-profile attacks such as the Morris worm [17] and Heartbleed [2], and are ranked by MITRE [1] to be some of the most dangerous software bugs. A recent study by Microsoft revealed that such errors continue to be the root cause of approximately 70% of the CVEs identified in their production-software, as shown in Figure 1. In particular, errors affecting heap-objects like heap-corruption, heap out-of-bounds accesses, and use-after-free, caused 50% of the CVEs in 2019 in Microsoft's software (a study by Google showed heap-errors make up 60% of memory-safety bugs detected in their software [19]). Most recently, a heap-error bug existed in plain sight for nearly 10 years, that allowed local users to gain root access through a privilege escalation attack [3]. Hence, the main focus of this work is on heap-errors (although our ideas are equally applicable to memory safety for globals and stack objects.).

II. Pitfalls of Prior Memory Safety Solutions

Prior solutions for providing memory safety at runtime can be classified into two categories: those that provide *probabilistic-detection* and those that provide *bounds-checking based prevention*. **Probabilistic-Detection based approaches** allow detection of (some) memory errors – by inserting trip-wires around objects, or randomizing memory layouts or

[†] Equal contribution.
[*] Gururaj Saileshwar is currently affiliated with Georgia Tech

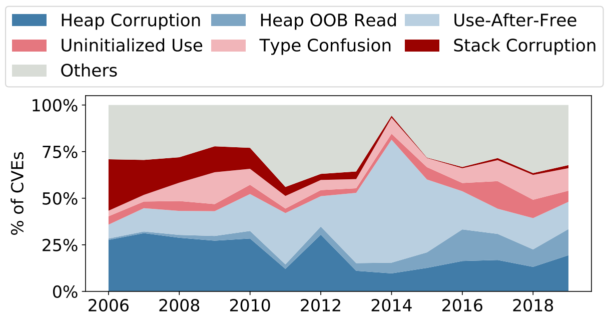

Fig. 1. A recent study from Microsoft [11] on the root cause of CVEs shows that memory-safety errors cause >70% of the CVEs, with Heap-Corruption, Heap OOB Read, and Use-After-Free errors causing almost 50% of the CVEs.

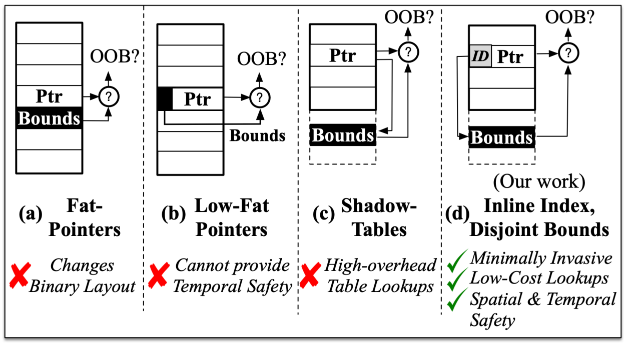

Fig. 2. Pitfalls of prior bounds-checking based solutions.

associating pointers and objects with ”colors” that can be matched. While these solutions are typically easy to adopt (without prohibitive slowdown or compatibility issues), they lack complete coverage and allow errors to remain undetected. **Bounds-Checking based solutions** enforce safe program behavior by verifying that pointer dereferences are within valid object-bounds, and allow precise enforcement of spatial and temporal memory-safety. Unfortunately, such solutions face two drawbacks that make them difficult to adopt:

1. Incompatibility with Library-Code: *Fat-pointer* based solutions [8], [15] including CHERI [23], as shown in Figure 2(a), store bounds-information for a pointer in a separate word alongside the actual pointer value, so that a bounds-check can be performed when the pointer is dereferenced. However, this requires changes to the binary layout that is incompatible with legacy library-code. *Low-Fat Pointers* [9] avoid binary changes by storing bounds implicitly within the pointer (as in Figure 2(b)), but cannot provide temporal safety, such as use-after-free, as they are unable to detect dangling pointers.

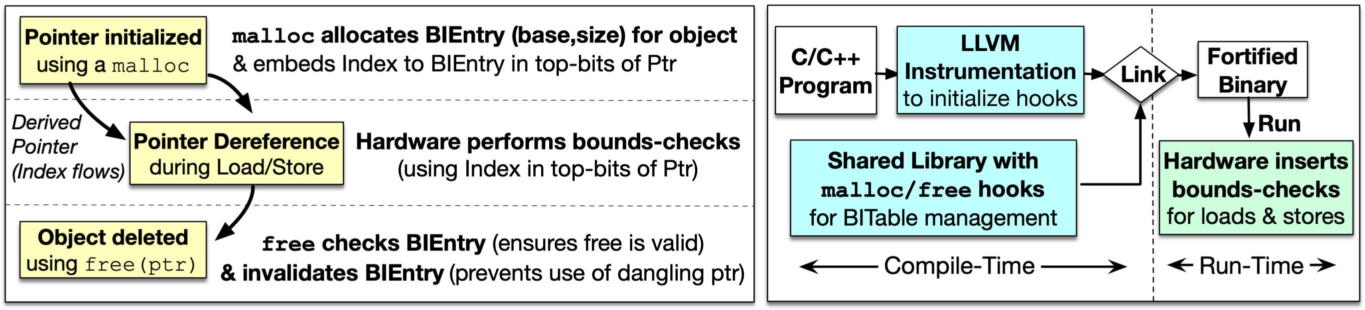
Pointer initialized using a malloc
malloc allocates BIEntry (base,size) for object & embeds Index to BIEntry in top-bits of Ptr
Derived Pointer (Index flows)
Pointer Dereference during Load/Store
Hardware performs bounds-checks (using Index in top-bits of Ptr)
Object deleted using free(ptr)
free checks 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. 3. Overview of our proposal. (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 our proposal.

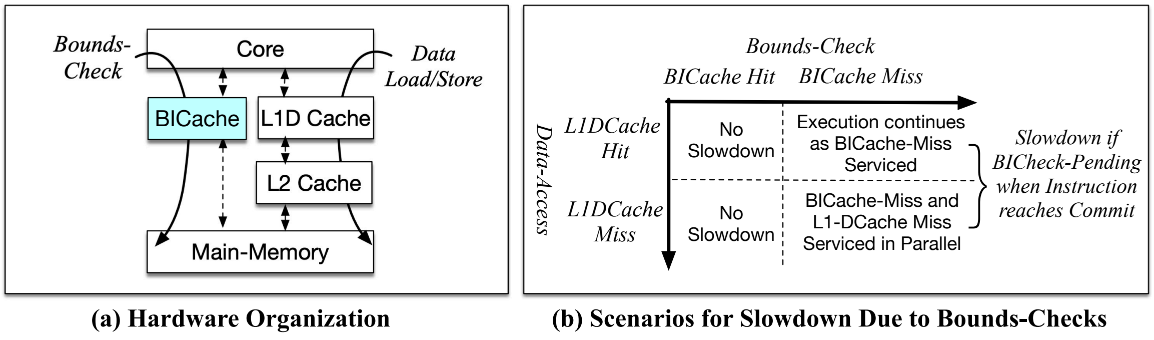
Bounds-Check
Core
Data Load/Store
BICache
L1D Cache
L2 Cache
Main-Memory
(a) Hardware Organization
Bounds-Check
BICache Hit
BICache Miss
Data-Access
L1DCache Hit
L1DCache Miss
No Slowdown
No Slowdown
Execution continues as BICache-Miss Serviced
BICache-Miss and L1-DCache Miss Serviced in Parallel
Slowdown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

Fig. 4. (a) Hardware organization (b) Scenarios for Slowdown Due to Bounds-Checks

2. High Performance Overhead: *Shadow-table* based solutions maintain bounds-metadata in a table in shadow memory that is indexed using the pointer-value to minimize changes to binary layout. However, the extra memory accesses for bounds-checks due to expensive (and in some cases multi-level) table-lookups using the pointer-value, leads to high slowdowns – MPX [16] and BOGO [24] incur 50%–60% average slowdown, Watchdog [14] incurs 24% slowdown, Softbound+CETS [12], [13] incurs 116% slowdown; recent work Chex86 incurs 14% slowdown on average, but a higher worst-case slowdown of 40%.

III. Our Approach

The goal in our project is to provide strong memory safety for heap-objects with hardware-based bounds-checking and prevent errors like out-of-bounds access and use-after-free. At the same time, to enable adoption, we want to achieve this with negligible slowdown and with minimal changes to the binary layout. To that end, we develop a minimally invasive bounds-metadata organization, as shown in Figure 2(d), where a pointer is associated with a unique inline identifier, used to index into a bounds table in a low-cost manner.

The main idea of our proposal 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 enforce hardware-based bounds-checks on all object accesses at runtime. Figure 3(a) shows the life-cycle of a pointer during program runtime. When an object is created, an entry is created (BIEntry) in the BITable to store its base-address and size, and the index of the corresponding entry in the BITable is embedded within the top bits of the object's address. When the address 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 freed objects are used subsequently.

We divide the responsibilities between the software and hardware as shown in Figure 3(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 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 and store to detect memory safety violations: we modify the load and store execution in hardware to access an entry in the BITable and use that information to check bounds on loads and stores; we add a bounds-information cache (BICache), shown in Figure 4(a) to minimize the impact of accesses to the BITable in memory. Hardware in our proposal includes a dedicated BICache for caching BITable entries, that is accessed for Bounds-Check in parallel to the L1-Dcache on Loads/Stores. As shown in Figure 4(b), assuming TLB-Hit for Data-Access and Bounds-Check, slowdown is incurred only if the Bounds-Check has a BICache-Miss and the check is still pending by the time the load/store instruction reaches commit-stage.

Overall, the benefits of our design over prior works include:

- Unchanged binary layout, that retains compatibility with library-code, unlike prior fat-pointer based approaches [8], [15], [23].
- No overhead for propagation of the index which happens "automatically" on pointer assignments and pointer arithmetic, and even type cast, unlike prior solutions which require extra instructions [4], [6], [13], [16] or micro-ops [14], [20] to propagate bounds information.
- Providing temporal safety at no extra-cost, as the location of the bounds-information (determined by the index) is independent of the pointer-value. So our solution maintains invalid-bounds status for dangling pointers even after the freed memory is reused, unlike prior shadow-table based bounds-checking solutions [5], [9], [14] where the bounds-metadata location is linked to pointer-value.

IV. Performance Evaluations

We package the software changes for our proposal (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 our proposal are modeled in Gem5 v20.0 [10].

We tested our proposal with the exploits from the How2Heap [21] exploit suite that leverage heap spatial and temporal safety bugs like out-of-bounds accesses, use-after-free, invalid-free, and double-free. Our proposal was able to detect the bugs in all 25 of these programs and raise an exception to terminate the program before the objective of the exploit is achieved.

For our performance evaluations, we also use 13 C/C++ benchmarks available in SPEC-CPU2017 [7] with the *ref* dataset. We show that our framework prevents memory safety errors when pointers are passed to un-instrumented library-code. Our solution identified memory-safety bugs in 87 lines of code in commonly-used Glibc-v2.27 and SPEC-CPU2017 functions, with aggressively optimized SIMD instructions accessing out-of-bounds memory, that, to our knowledge, were previously undetected even with state-of-the-art tools like Address Sanitizer [18].

We first evaluated the slowdown due to the malloc/free instrumentation. Figure 5 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. On average, the SW instrumentation for BITable management (without bounds-checks) adds only 0.5% slowdown across all programs.

We then evaluated the slowdown due to the hardware bounds-checks with an 8Kbyte BICache. Note that in our design we can get a high hit rate in a fairly small BICache

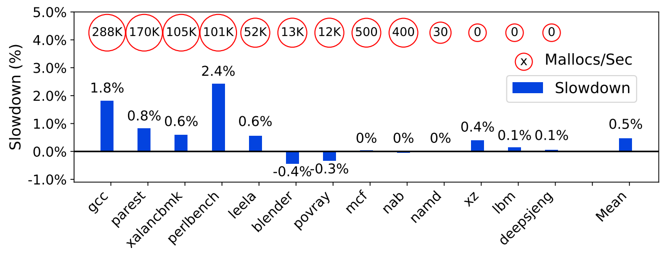

Fig. 5. Performance Impact of Software Instrumentation for BITable management, modeled using native execution. On average, the SW instrumentation adds 0.7% slowdown.

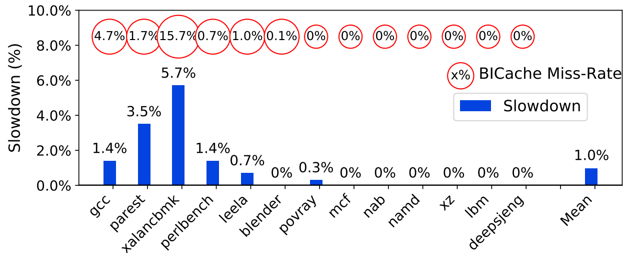

Fig. 6. Performance Impact of Hardware Bounds-Checks. On average, the bounds-checks add only 1% slowdown, owing to high BICache hit-rates (larger than 98%).

since all the addresses within a given buffer and all the pointers to the buffer have the same value in their index bits and use the same entry in the BICache. As the BICache is only a few KBs in size, it can even be saved and restored on a context-switch. (Even if the BIcache is not preserved on a context-switch, since all the addresses for a single buffer share a single entry in the BICache, the BICache will tend to be quickly re-filled with the necessary bounds information when a process is re-dispatched.) Figure 6 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 higher BICache miss-rates (2% to 16%) and higher slowdown (1% to 6%). Other workloads, with more than 99% BICache hit-rate, have negligible slowdown.

V. Lessons Learned and Challenges Ahead

While memory safety bugs in C/C++ programs have been a leading cause of vulnerabilities for over three decades, an effective memory safety solution, that is also practical to adopt, has thus far been elusive. Our project resulted in a low-cost, hardware-based solution for bounds-checking that provides precise detection of errors like buffer-overflows and use-after-free in heap-objects. Our practical solution has minimal performance overhead (less than 2% slowdown), and maintains compatibility with legacy library-code. Throughout the project we have learned valuable lessons:

1) With a hardware/software co-designed approach, it is feasible to implement a low-cost hardware solution to memory safety problems that have been with us for more than 30 years with minimal performance impact.
2) Precise enforcement of spatial and temporal memory-safety is a must; otherwise errors can remain undetected and the investment will be wasted.
3) Changes to source-code or binary-layout should be avoided. Compatibility is key for adoption.
4) Propagating pointer-metadata should be efficient. Our solution does not incur any overhead for propagation of the index which happens "automatically" when one pointer is assigned to another, passed in a function call, or used to compute another address in array indexing or pointer arithmetic. This is unlike prior solutions – and programming languages that provide memory safety – which require extra instructions to propagate metadata.
5) The performance impact of the actual bounds-checking should be minimal. Extra memory accesses for bounds information should be minimized. In our solution, since all the addresses associated with a given buffer have the same index, the bounds information for an address is often available in the on-chip BIcache.

Although our approach seems to provide an effective and low-overhead solution, several challenges remain:

1) If the number of "live" objects exceeds the size of the BITable, e.g. if the number of live objects exceeds the number that can be represented by the index bits in a pointer, an overflow table similar to the tables used in prior work could be used. If 24 index bits are available, an overflow table could be used when the number of live objects exceeds 16M.
2) The Bounds-checking framework should be compatible with multi-threaded workloads. To support such workloads, the malloc and free functions should be implemented in a thread-safe manner by using locks to ensure atomic updates to the BITable.
3) Our approach can protect a program from a wide variety of memory-safety bugs and thus provides some protection for the integrity of the BITable. Additional protection for the BITable can be provided with some additional overhead.
4) Bound cache management is an area that could benefit from further study. For example, a separate prefetch engine for the BICache could be desirable to increase BICache hit rate and take BICache load off the critical path of the actual load or store.
5) It is possible, as a result of array indexing or pointer arithmetic, that the arithmetic will overflow and corrupt the index bits in a pointer which could cause an out-of-bounds reference or a use-after-free reference to be undetected. While the likelihood of this is low, our approach could be improved to eliminate this possibility.
6) Our technique can provide precise detection of out-of-bound references even when objects are nested. We intend to explore this use case in the future.

VI. Conclusions

We presented a low-cost hardware-based solution for bounds-checking that detects errors like buffer-overflows and use-after-free in heap-objects. Our proposal has minimal performance overhead (less than 2% slowdown), maintains compatibility with legacy library-code and has detected 87 lines of code with memory safety bugs in Glibc functions and SPEC-CPU2017 workloads, that, to our knowledge, were previously undetected. This can provide an effective, low-cost, deployable solution that can protect software from an important problem that has existed for more than 30 years.

References

[1] 2019 CWE Top 25 Most Dangerous Software Errors. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html.

[2] Buffer Overflow (BOF) Examples - Heartbleed. https://samate.nist.gov/BF/Examples/BOF.html.

[3] Heap-based Buffer Overflow Leads to Privilege Escalation. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156.

[4] Periklis Akritidis and et al. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX 2009*.

[5] Joe Devietti and et al. Hardbound: architectural support for spatial safety of the c programming language. *ASPLOS 2008*.

[6] Gregory J Duck and et al. Heap bounds protection with low fat pointers. In *CC 2016*.

[7] https://www.spec.org/cpu2017/. Spec benchmarks.

[8] Trevor Jim and et al. Cyclone: A safe dialect of c. In *USENIX 2002*.

[9] Albert Kwon and et al. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *SIGSAC 2013*.

[10] Jason Lowe-Power and et al. The gem5 simulator: Version 20.0+. *arXiv:2007.03152*, 2020.

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

[12] Santosh Nagarakatte and et al. Cets: compiler enforced temporal safety for c. In *ISMM 2010*.

[13] Santosh Nagarakatte and et al. Softbound: Highly compatible and complete spatial memory safety for c. In *PLDI 2009*.

[14] Santosh Nagarakatte and et al. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *ISCA 2012*.

[15] George C Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *POPL 2002*.

[16] Oleksii Oleksenko and et al. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *POMACS 2018*.

[17] Hilarie Orman. The Morris worm: A fifteen-year perspective. *IEEE Security & Privacy*, 1(5):35–43, 2003.

[18] Konstantin Serebryany and et al. Address sanitizer: A fast address sanity checker. In *USENIX ATC*, 2012.

[19] Kostya Serebryany. Oss-fuzz - google's continuous fuzzing service for open source software. 2017.

[20] Rasool Sharifi and et al. Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities. In *ISCA 2020*.

[21] Shellphish. How2heap github repository. https://github.com/shellphish/how2heap, (accessed August 1, 2020).

[22] Laszlo Szekeres and et al. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*.

[23] Jonathan Woodruff and et al. The cheri capability model: Revisiting risc in an age of risk. In *ISCA 2014*.

[24] Tong Zhang and et al. Bogo: buy spatial memory safety, get temporal memory safety (almost) free. In *ASPLOS 2019*.