N00B: Bounds Checking for the Masses

Adriaan Jacobs
DistriNet, KU Leuven
Belgium

Carlo Ramponi
University of Trento
Italy

Jonas Roels
DistriNet, KU Leuven
Belgium

Bruno Crispo
University of Trento
Italy

Silviu Vlasceanu
Huawei Research
Germany

Mahmoud Ammar
Huawei Research
Germany

Stijn Volckaert
DistriNet, KU Leuven
Belgium

Abstract

Spatial memory safety vulnerabilities continue to plague systems-level code. Decades of bounds checking research has not produced a widely adopted solution: performance overhead plateaus at roughly 50%, and a narrow focus on optimization neglects compatibility with legacy code.

These overlooked functional aspects are nevertheless key to rapid adoption. Hardening the massive existing C/C++ ecosystem demands broad real-world source compatibility, and seamless binary interoperability can unlock *selective hardening*, prioritizing vulnerability-ripe code without penalizing the entire application.

In this paper, we show how existing work fails to meet these constraints. We then introduce N00B, a new bounds checker that is, for the first time, both legacy C/C++-compatible *and* selectively applicable. N00B achieves this through a novel *hybrid* bounds checking approach: coarse-grained pointer arithmetic constraints keep pointers near their referents, enabling the storage of precise *relative* bounds in the limited space of ignorable upper bits. This hybrid strategy ensures all N00B-hardened pointers remain natively dereferenceable—crucial for interoperability—while tolerating benign out-of-bounds pointer motion in legacy C/C++.

We evaluate N00B on standard compatibility, security, and performance benchmarks. The results show that despite significant functional improvements, N00B's geomean run-time overhead remains competitive at around 50%.

CCS Concepts

• **Security and privacy → Systems security**.

Keywords

Bounds checking, memory safety, systems security

CCS'26, The Hague, NL

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Adriaan Jacobs, Carlo Ramponi, Jonas Roels, Bruno Crispo, Silviu Vlasceanu, Mahmoud Ammar, and Stijn Volckaert. 2026. N00B: Bounds Checking for the Masses. In *Proceedings of Conference on Computer and Communications Security (CCS'26).* ACM, New York, NY, USA, 20 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Almost four decades ago, Robert T. Morris first popularized the exploitative potential of buffer overflows with his infamous *worm* [121]. Since then, memory safety issues have remained a central security risk [64, 87, 104, 112, 130], especially for critical infrastructure such as hospitals [38] and government systems [20], which are often targeted by state-sponsored hackers [55]. Recent evidence suggests that over 80% of zero-day vulnerabilities actively exploited in the wild are memory errors [8, 20, 49, 87].

Among these memory errors, *spatial* memory safety issues [88, 90] have proven particularly difficult to mitigate over the years [92]. They occur when a program dereferences a pointer outside the bounds of its intended referent [60]. Many spatial safety defenses have been proposed [3, 5, 12, 30, 31, 41, 42, 52, 58, 60, 63, 65, 69, 70, 75, 79, 94, 97, 99, 114, 146, 148], yet few are adopted in practice. There are a number of reasons for this. A first deterrent is run-time overhead [124], which, despite decades of optimization work [3, 18, 42, 47, 51, 61, 69, 79–81, 94, 97, 122, 134, 142, 143, 149], has stagnated at roughly 50% on compute-intensive benchmarks [34, 44, 79, 147]. Theoretically, this overhead could be curbed via selective hardening: prioritizing vulnerability-ripe or "cold" modules to achieve significant security gains for a fraction of the overhead. Not all vulnerable code is equally performance-sensitive, and evidence suggests that vulnerabilities tend to accumulate in less-executed, under-tested code [133]. However, selective hardening still remains blocked by the second major deterrent: source incompatibility. Bounds checkers today *either* support good interoperability with external/unhardened code (and are thus selectively applicable) *or* are source-compatible with real-world C/C++. Without source compatibility, incompatible code snippets must be dynamically detected from program crashes [35] and manually patched out [34, 86], which slows adoption and undermines robustness.

In this paper, we argue that this stalemate stems from inherent limitations of the reigning bounds-checking paradigms. "Pointer-based" bounds checkers encode bounds information in each pointer [5,

12, 58, 63, 65, 66, 68, 69, 75, 93, 94, 99, 100, 105, 106, 115, 125, 136], which invariably breaks the native pointer layout and thus precludes binary interoperability with external code (Section 2.1). In contrast, "object-based" bounds checkers track bounds information per object [3, 28, 34, 35, 60, 79, 114, 146, 147], which largely preserves the native pointer layout, but must keep pointers in-bounds at all times (via pointer arithmetic constraints) to recover the correct metadata during checks. This requirement breaks source compatibility because it conflicts with common pointer-manipulation idioms that temporarily create out-of-bounds pointers without dereferencing them (Section 2.2). Hence, no current bounds checker is both source-compatible *and* selectively applicable today.

To address this gap, we introduce a new bounds checker design, called `N00B`, that uses a novel *hybrid* approach to bounds checking (Section 4): we arrange objects into coarse-grained memory arenas to which pointer arithmetic is constrained, and within which we can encode small enough *relative* bounds information to fit in the ignored upper bits of pointer masking extensions on all major processor architectures.

We implement an efficient prototype that uses judicious object alignment to optimize bounds checks and evaluate it on AArch64 Top Byte Ignore (TBI) and Intel Linear Address Masking (LAM) platforms. We show that `N00B`'s full instrumentation overhead is on par with state-of-the-art bounds checkers (Section 6), while avoiding the functional issues that typically plague high-performance mechanisms.

Finally, we demonstrate the power of selective hardening by applying `N00B` only to FFmpeg's `libavformat` component [37]. This configuration retains over 98% of the baseline application throughput while successfully thwarting `libavformat` exploits (Section 7), confirming that selective hardening can effectively neutralize the performance overhead of spatial safety.

2 Functional Challenges in Bounds Checking

In this section, we analyze why existing bounds checkers fail to meet *functional* requirements, like compatibility with common but less-than-well-defined C/C++ code constructs, or interoperability with uninstrumented external code. Table 1 summarizes our findings.

2.1 Selective Applicability

The need for selective applicability and interaction with external, uninstrumented code imposes several challenging constraints on the design of bounds checkers.

ABI compatibility. "Fat Pointers" [5, 58, 65, 99] intuitively store bounds alongside pointers and copy them around together. However, they famously break ABI compatibility: uninstrumented external code, unaware of the structural changes, miscalculates offsets in updated data structures, resulting in incorrect behavior.

Bounds propagation. SoftBound [94] restores ABI compatibility by storing bounds disjointly in a map indexed by pointer location. However, like fat pointers, SoftBound still requires all pointer copies to be explicitly instrumented to propagate the disjoint bounds, which is challenging in type-unsafe code [62], and simply does not happen in external code [106]. Hence, a selectively applicable bounds checker must encode all necessary bounds information inside the native pointer type, so that it can be *implicitly* propagated by native pointer copies, even in external code.

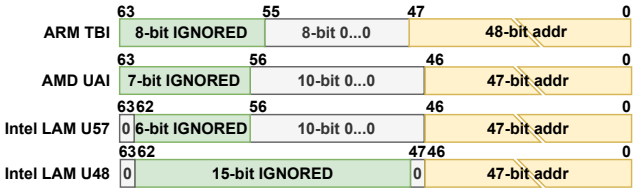


Figure 1: Linux 4-level paging user-space pointer layout under different pointer masking extensions.

Limited upper bits. However, in-pointer space is scarce. On 4-level paging systems, 47/48 bits are reserved for addressing, leaving only 16/17 bits for metadata (Figure 1). A popular approach is to encode a bounds table index in these bits [12, 66, 68, 115, 132, 136], which can then be used to query the full bounds information through a table lookup. Aside from performance concerns, this limits the number of concurrently allocated objects (which must all receive a unique index) to $2^{16} - 1 = 65535$, vastly insufficient for many real-world applications [115]. For instance, the reference workload of SPEC CPU2017 already allocates up to 2.4M concurrently live objects. Some defenses do shrink the available virtual address space to free up more upper bits [6, 70, 75], but this, in turn, conflicts with the trend toward larger address spaces in modern processors, i.e., 5-level paging [144].

Pointer masking. Another problem with populating the upper bits is that it creates non-canonical pointers that trap on dereference. This is especially problematic in uninstrumented external code, where explicit masking operations cannot be inserted. Many bounds checkers try to solve this by simply masking off the top bits when passing pointers as arguments to external functions [3, 12, 41, 63, 68, 69, 75, 132]. This is not an adequate solution.

First, argument masking is insufficient. It fails to catch pointers escaping via memory [41]—in itself a challenging problem due to the undecidability of aliasing [111]—or by returning from a callback function. Second, it creates different coexisting versions of the same pointer (with and without top bits) which breaks pointer comparisons and subtractions. These correctness issues are hard to fix in instrumented code [63, 69], let alone external code. Third, masking permanently strips protection. Since defenses typically disable checks on bare pointers to avoid crashes [115, 132], every interaction with external libraries progressively erodes safety.

Hardware pointer masking extensions such as Intel LAM [22] and ARM TBI [77] alleviate all these issues by removing the need for pointer masking, but contain insufficient space (6–15 bits) to encode full bounds information for any non-trivial application [66, 115].

Object-based bounds checking. Jones and Kelly [60] introduced "object-based" bounds checking as an alternative that avoids the bounds encoding issues of pointer-based bounds checkers. Their approach validates pointer *arithmetic* rather than *dereference*. By strictly enforcing that pointers never drift out-of-bounds, the design guarantees that all in-bounds pointers can reliably locate their bounds metadata (stored relative to the object), eliminating the

Table 1: N00B vs other relevant bounds checkers with similar goals and operational constraints.

Approach	Type	Hardware Assistance	Pointer Instrumentation				Target Memory			Compatibility					Run-Time Overhead
			Deref.	Arith.	Escape	Copy	Stack	Heap	Global	Ext. code interop	Benign OoB ptrs	Maximal Address Space (2^x)	Max. #Live Objects	Max. Alloc. Size (GB)	
Fat Pointers [5, 58, 65, 99]	P	-	✓	-	-	-	✓	✓	✓	○	●	▲	▲	▲	●
SoftBound [94]	P	-	✓	-	✓	✓	✓	✓	✓	◐	●	▲	▲	▲	●
MESH [132]	P	-	✓	✗	✓	-	✗	✓	✗	◐	●	48	2^{16}	▲	●
PACMem [75]	P	ARM PA	✓	✗	✓	-	✓	✓	✓	◐	●	39	2^{24}	4 GB	●
CUP [12]/Midfat [68]	P	-	✓	✓	✓	-	✓	✓	✓	◐	●	▲	2^{31}	4 GB	●
CGuard [63]	P	-	✓	✓	✓	-	✓	✓	✓	○	◐‡	48	▲	▲	●
Delta Pointers [69]	P	-	✓	✓	✓	-	✓	✓	✓	○	◐‡	32	▲	2 GB	●
J&K [60]	O	-	-	✓	-	-	✓	✓	✓	●	○†	▲	▲	▲	●
CRED [114]	O	-	✓	✓	-	-	✓	✓	✓	●	◐	▲	▲	▲	●
PariCheck [146]	O	-	-	✓	-	-	✓	✓	✓	●	◐	38	▲	▲	●
Baggy Bounds$^{64\text{-bit}}$ [3]	O	-	-	✓	-	-	✓	✓	✓	●	◐	38	▲	▲	●
Low-Fat [31, 33, 34]	O	-	-	✓	-	-	✓	✓	✓	●	○	▲	▲	32 GB	●
CAMP [79]	O	-	-	✓	-	-	✗	✓	✗	●	○†	▲	▲	▲	●
ShadowBound [147]	O	-	-	✓	-	-	✗	✓	✗	●	○	▲	▲	8 GB	●
FRAMER/Miu [96, 97]	P+O	-	✓	✓	✓	-	✓	✓	✓	◐	◐‡	▲	▲	▲	●
N00B	P+O	TBI/UAI/LAM	✓	✓	-	-	✓	✓	✗	●	◐‡	▲	▲	16 GB	●

Legend: P Pointer-based, **O** Object-based, **R** Red-zone-based (aka Tripwire), ✓ Has this feature, ✗ Lacks this feature, ✓ Has this feature (but is fundamentally flawed), ✗ Lacks this feature (but no fundamental issue in the design), - Feature is not required, ● Full Compatibility, ◐ Partial Compatibility, ○ No Compatibility, ● <30% (average) overhead, ● <50% (average) overhead, ● >50% (average) overhead, ▲ The maximum possible by the underlying OS/architecture, † Only off by One, ‡ Within limited range. If the defense includes more than just spatial checks, we model their spatial component alone.

need to embed bounds *inside* the pointer; the pointer itself *becomes* the metadata handle.

Subsequent object-based bounds checker innovations aligned objects to their size [3], and encoded size information in the pointer using only a few bits [34], such that object base pointers could be recovered by aligning in-bounds pointers down, fully eliminating the need for disjoint metadata lookups. Notably, Low-Fat Pointers [34] allocates objects at specific addresses such that the most significant bits of the effective address contain the size of the object (see the "N" field in Figure 2), which can be used to align in-bounds pointers down to retrieve the object's base pointer without using any upper bits or disjoint metadata. This breakthrough encoding remains one of the fastest and most popular bounds checker designs to date [62].

2.2 Pragmatic C/C++ Compatibility

Real-world C and C++ programs routinely contain undefined or implementation-defined behavior [17], in particular by assuming that pointers and integers behave interchangeably. Many bounds checkers break this assumption.

Benign OoB pointers. Specifically, object-based bounds checkers cannot handle pointer arithmetic that generates an out-of-bounds (OoB) address, even if that pointer is never dereferenced. While the C standard allows pointers to drift "one past the last element of the array object" [56], real-world programs generate pointers far beyond these limits [17, 35, 62, 84, 95, 114]. Typical use cases include reverse array iteration (creating a pointer to *before* the start of the object), and subtracting a fixed offset `K` from a base pointer so that `ptr[K]` accesses the 0th element [35]. Chisnall et al. found benign OoB pointers in over half of the code bases they surveyed [17], at a rate of roughly 1 instance per 1,000 LoC. Duck et al. encountered them in 9 SPEC CPU2006 benchmarks [35], which matches our own experience (Section 6.1). The prevalence of this idiom need not surprise; 73% of systems developers surveyed by Memarian et al. [84] believed that temporarily going OoB and returning in-bounds was well-defined, standards-compliant behavior.

Mitigations. "Rectifying" this misconception across millions of lines of code is a slow, manual, and error-prone process, without any guarantee of getting complete coverage. Since the offending behavior appears as harmless pointer arithmetic or indexing, it is also hard to statically lint for. Previous work used dynamic profiling tools to identify these benign OoB cases and exclude them from object-based instrumentation [35], but this approach naturally has coverage limitations.

Jones and Kelly originally proposed to pad all objects with a single byte to keep "off-by-one" OoB pointers within bounds [60]. However, this is insufficient when pointers exceed the bounds by more than one byte or underflow the object instead [28, 60, 114, 146]. Baggy Bounds Checking [3] replaces OoB pointers with an invalid value instead, so that they can exist harmlessly unless dereferenced [60]. However, this approach might cause functional issues during pointer comparisons [3]. Alternatively, CRED [114] and PariCheck [146] preserve the pointer value but track its invalidity in disjoint metadata, at the cost of expensive per-dereference lookups.

All of these "OoB poisoning" approaches face additional challenges when the benign OoB pointer is later brought back into bounds (e.g., in the `ptr[K]` case). In such cases, the bounds checker must transparently "un-poison" the OoB pointer, which requires (i) tracking the pointer's intended referent while it remains OoB and (ii) precisely detecting all pointer arithmetic to instrument the operation that brings the pointer back into bounds. Problem (i) reintroduces the familiar bounds encoding and propagation challenges of pointer-based schemes (see Section 2.1), and problem (ii) is challenging due to the type-unsafe nature of C/C++ and the lack of information at compile time.

Pointer-based bounds checkers. In contrast, pointer-based bounds checking mechanisms discussed in Section 2.1 impose no constraints on the value of the effective address, allowing them to support these quirks without issue. At the same time, this agnosticism toward the effective address is precisely what prevents

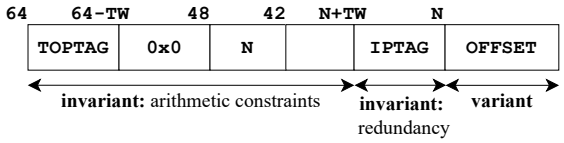


Figure 2: N00B pointer layout. Object sizes are powers of two, and objects are aligned to their size. Thus, *in-bounds* pointer arithmetic only modifies the least-significant N bits, shown as OFFSET. Nearby *out-of-bounds* arithmetic modifies the iptag, which N00B can detect during dereference by comparing it with the toptag. Larger arithmetic modifying bits above the iptag is disallowed through pointer arithmetic checks.

them from efficiently encoding bounds information, leading to the interoperability issues described in Section 2.1.

3 Threat Model

We consider adversaries that aim to exploit spatial memory safety vulnerabilities in the user-space program protected by N00B by providing malicious input. These vulnerabilities could allow them to read or write arbitrary memory locations. N00B's goal is to prevent such exploits by enforcing bounds checks on memory accesses. However, N00B only protects against vulnerabilities within the hardened source code, excluding potential vulnerabilities in external libraries. We specifically focus on preventing overflows that cross allocation boundaries, as these constitute most buffer overflow exploits observed in practice. While intra-allocation overflows between struct members or array elements are still exploitable [40], they fall outside the scope of N00B (see Section 8).

4 The N00B Design

In a sense, the distinction between pointer-based and object-based approaches can be viewed through their handling of benign OoB pointers. Pointer-based approaches allow them without restriction, leading to referent-tracking and bounds-encoding issues in large address spaces. In contrast, object-based approaches explicitly disallow them, causing source code compatibility issues.

N00B explores a middle ground. Our key insight is that enforcing coarse-grained, wider-than-object bounds via arithmetic constraints can help alleviate the inherent limitations of both approaches. It permits a limited range of benign OoB arithmetic around objects within the wider bounds, preventing crashes. Yet, it also still restricts the effective address range of the pointer, allowing us to encode compact, *relative* bounds information within the limited number of unused bits of commodity pointer masking extensions, keeping pointers natively dereferencable. In effect, N00B's "hybrid" approach implements a small pointer-based bounds checker *inside* a wider object-based bounds checker to achieve a design that is both selectively applicable *and* source-compatible. The rest of this section explains the specific design decisions used to make N00B fast and to tailor it towards existing pointer masking extensions.

4.1 Hybrid Bounds Checking

N00B *efficiently* enforces *hybrid* bounds without relying on disjoint metadata lookups. Our approach builds on the observation that small pointer arithmetic primarily modifies the least significant bits of pointer values, while the more significant bits typically require larger offsets to change. When objects are aligned to their size, and their size is a power of two, this distinction manifests explicitly in the pointer value. As illustrated in Figure 2, for an aligned object of size 2^N, all in-bounds pointer arithmetic modifies only the least significant N bits, leaving the upper $64 - N$ bits intact. Thus, as noted by previous work [14, 74], the challenge of spatial memory safety enforcement can be reduced to maintaining the invariance of these more significant bits between allocation and dereference. To delineate variant from invariant bits, N can be encoded within the effective address bits of the pointer by allocating objects into specific, large address ranges according to their size [31, 34, 44, 46]. In N00B, as shown in Figure 2, N is encoded in a fixed location within the effective address of the pointer (`ptr[48:42]`).

Directly enforcing the invariance of the more significant $64 - N$ bits through pointer arithmetic checks would yield an efficient 2^N-aligned object-based bounds checker [3], such as Low-Fat Pointers [34], which fails to handle benign OoB pointers. Instead, our second observation is that we can ensure invariance for a subset of these bits through *redundancy*, by duplicating them into the upper unused bits of the pointer. During object creation, we extract the least significant `TAG_WIDTH` (`TW`) number of bits of the invariant part of the pointer (`ptr[N+TW:N]`), and store them in the top bits of the pointer (`ptr[64:64-TW]`). We refer to these extracted bits as the "in-pointer tag" (`iptag`), while their copy in the pointer's top bits is called `toptag`. The `iptag` can be modified by pointer arithmetic without losing track of the intended referent, as its original value is backed up in the `toptag`. The more significant bits—specifically, `N+TW` and above, including the bits that encode the size information (`N`) and the `toptag` itself—are strictly checked for invariance during pointer arithmetic. Notably, under N00B, the least significant `N+TW` bits may vary across pointer arithmetic, making them unsuitable to hold trustworthy metadata. Therefore, these bits must not overlap with the pointer's size information, i.e., the N field (`ptr[48:42]`). This constrains the maximum value of `N+TW` to 42, yielding a maximum allocation value of $2^{42-8=34}$=16GB on platforms like AArch64 where `TW=8`, which is sufficient for many applications.

4.2 N00B's Memory Layout

N00B, like several other bounds checkers [3, 14, 34], opts to align objects and bounds to powers of two, which allows its checks to be efficiently implemented through simple bitwise operations. The performance gains this affords are typically worth the increased memory usage and cache footprint. We implement a custom memory allocator, `N00Balloc`, that organizes memory into arenas, each containing 2^{TW} objects of the same size 2^N. By also aligning arenas to their power of two size (2^{TW+N}), we can efficiently determine whether two pointers point to the same arena by simply comparing their upper bits, as shown below:

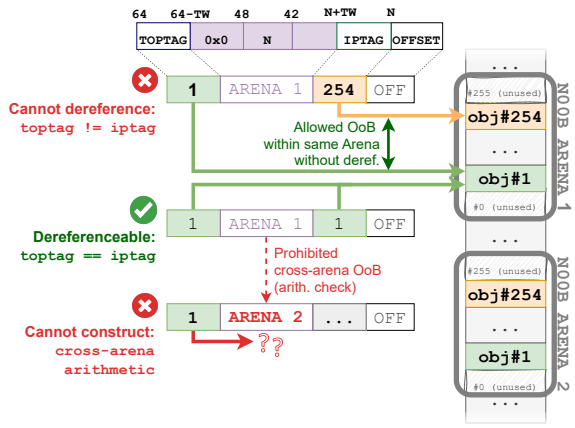


Figure 3: N00B arenas are constructed such that pointer tags never repeat. Coarse-grained bounds checks restrict pointer arithmetic to within arena boundaries.

```
base_arena = base >> (N + TW);
ptr_arena = ptr >> (N + TW);
if (base_arena != ptr_arena) trap();
```

This efficient arena check also means that not all pointers benefit from the same amount of benign OoB leeway. As seen in Figure 3, pointers to objects at the very start or end of an arena may face arithmetic boundaries at their own allocation boundaries, imposing tight arithmetic constraints akin to pure object-based bounds checkers. To avoid such issues, N00B simply does not allocate objects at either arena boundary. This grants all objects at least one object size worth of benign OoB leeway, which naturally grows with larger object sizes. For performance reasons, N00B extends this leeway to the size of a cache line (64 bytes) for the smallest allocations.

Stack Objects. Similar to previous work [31, 44, 46, 86], N00B creates hardened pointers for stack objects by allocating them on dedicated, disjoint, per-size N00B-managed stacks. As such, the layout of hardened pointers to stack objects is identical to that of heap pointers, avoiding slow disambiguation checks. Using LLVM's `StackSafetyAnalysis`, we identify stack objects that might be subject to OoB indexing and allocate them on disjoint stacks of appropriate sizes instead, much like SafeStack [26]. All stack objects that we do not explicitly move remain on the native program stack, which is located in non-N00B-managed memory.

4.3 Selective Hardening

Under selective application, N00B-hardened pointers will escape into external code, which must be able to use them without any instrumentation. N00B solves this by design, because it only embeds metadata in *ignorable* fields of pointers (see Figure 2): the `toptag` in the ignored upper pointer bits [22, 27, 77], and the `power` and `iptag` bits inside the effective address, avoiding the need for masking. Hence, all N00B-hardened pointers are valid native pointers.

Conversely, external code may provide non-N00B-hardened pointers to N00B-hardened code, which the N00B bounds checks must transparently ignore. To handle this, N00B imposes a program-wide memory layout constraint that all non-N00B-managed memory must be allocated *above* a certain, fixed address. The address is chosen such that its embedded size information (the `N` field) appears huge to N00B bounds checks, e.g., 48, which tricks them into transparently enforcing wide, compatible bounds on these memory regions. We apply this via a linker script, or via a small custom chainloader called `N00Bloader` that `mmap`-s the entire N00B-managed virtual memory area at startup, before handing off control to the actual program loader, which will only load regions above the reserved N00B-managed virtual memory area. This ensures that external global data and the native program stack are allocated at a N00B-ignorable address. After startup, `N00Balloc` interposes all dynamic allocations and places them in the N00B-managed region.

Key to N00B's selective effectiveness is that it attaches and maintain bounds information for as many objects in the program as possible, e.g., by interposing all heap allocations, including those by external code. This is useful because it ensures that even selective N00B bounds checks get to actually perform useful bounds checking work on as many objects as possible. As Section 6.2 shows, `N00Balloc` is fast enough to make this a viable default.

Selectivity Guidance. N00B supports selective hardening as a practical adoption requirement to avoid total recompilation of all process code, including mixed-language components, assembly routines, and closed-source dependencies. However, even within recompilable code, selective hardening can also be a way to lower overhead, by targeting only specific code modules [13, 123].

For one, developers can prioritize code that is easiest to exploit or most likely to contain vulnerability entry points, e.g., code that directly parses untrusted input. This selection can be informed by semantic knowledge about the program [13, 123], but also more quantifiably guided via historical CVE metrics per code module, or live bug reports of known yet unpatched vulnerabilities [10]. In Section 7, we evaluate such a selective deployment of N00B on a parsing component of FFmpeg [37].

Conversely, developers could purposefully exclude certain known-performance-sensitive components to meet an overhead budget. The performance characteristics of software components are easier to measure than their security properties, and many prominent historical bugs have appeared in "cold" code [133], e.g. HeartBleed [98].

Finally, memory-unsafe components of applications written in safe languages can form a natural selective hardening target [113], closing the memory safety gap and helping prevent cross-language attacks [85]. The resulting program is spatially memory-safe, but incurs only selective overhead.

4.4 Efficient Hybrid Bounds Checks

Performing bounds checks on pointer arithmetic *and* dereference risks introducing significant performance overhead. We carefully designed an efficient hybrid check that *simultaneously* performs both checks at dereference sites, and integrates their operation. To use it, we must first be able to perform the arithmetic check at dereference sites.

Tracking Base Pointers. N00B employs an approach similar to CGuard's "static-base" tracking [63] to provide the original, unmodified, intra-procedural *base* pointer value of pointers during

their dereference. We first identify the sources of pointer values within a function, such as memory loads, function calls, or function arguments. We then introduce an additional variable to track these pointers' original value as they propagate through `select` instructions, arithmetic, or other operations, as illustrated below.

```
1 bb1:
2       base1 = load mem1;
3       ptr1 = base1 + offset;
4       jmp %deref;
5 bb2:
6       ptr2 = arg2 + offset;
7       jmp %deref;
8 deref:        ; preds: %bb1, %bb2
9       ptr = phi [ptr1, ptr2];
10      noob.base = phi [base1, arg2];
11      check_arith(noob.base, ptr);
```

The key advantage of this base-pointer tracking mechanism is that it enables all arithmetic between pointer introduction and dereference to be checked in one step, together with the dereference check at the dereference sites, rather than checking each arithmetic operation individually.

To guarantee that all such intra-procedural base pointers contain valid metadata, N00B must additionally check intra-procedural pointer *escapes*, such as function argument passing, storing pointers to memory, or returning them from functions. Traditional object-based bounds checkers often face significant compatibility challenges when doing so [34, 150], as pointers may be benignly OoB at this point, and they require them to be strictly in-bounds. N00B overcomes this limitation because it only has to enforce that escaping pointers remain within the same arena as their base value, not within the same object. As long as pointers stay within arena bounds, the correct intended referent can always be retrieved during bounds checks.

Efficient Checks. N00B's hybrid bounds checks work by using the `toptag` and arena information of the trusted base pointer to reconstruct the intended referent of the offset pointer. Then, a single comparison performs *simultaneous* arithmetic and dereference checks (Listing 4.1): it verifies that the offset pointer's `iptag` matches the base pointer's `toptag` (pointer-based check), *and* ensures that the more significant bits (above the `iptag`) of the base and offset pointers match, including the embedded size information and the `toptag` (object-based check). This efficient check comprehensively enforces the invariance of all non-offset bits in pointers, from allocation to dereference, without prematurely crashing on intermediary OoB pointers that are never dereferenced.

Note that most of the work in this check (lines 1 through 4) merely reconstructs the intended referent value, and doing so happens solely using the base pointer's metadata. As such, this work can often be performed upfront, e.g., outside a loop, and even reused among multiple bounds checks that share the same base pointer. We take advantage of this feature in Section 5.2.

Simply moving to this hybrid check eliminated a third of N00B's overhead across the SPEC CPU2006 C/C++ benchmark suite, compared to doing separate arithmetic and dereference checks.

```
1 N = (base >> 42) & 0x7f;
2 referent = (base >> N) & ~TAG_MAX; /*arena base*/
3 toptag = base >> (64 - TW);        /*toptag*/
4 referent |= toptag;                /*orig obj*/
5 if (referent != (ptr >> N)) trap();
```

Listing 4.1: Hybrid N00B bounds check.

5 Implementation

We implemented N00B in a 4.7 KLoC out-of-tree LTO plugin for the LLVM 15 compiler, and `N00Balloc` in 580 lines of C++ library code. Our implementation targets 64-bit Linux environments with 4- or 5-level paging, which is common for modern desktops and servers [144].

5.1 Platform-Specific Considerations

All major processor architectures support pointer masking extensions that are compatible with N00B, summarized in Figure 1. On ARM, N00B uses the Top-Byte Ignore (TBI) extension, which has been supported since ARMv8 (2011) [77], making it available on many commodity AArch64 processors. On AMD64, we use AMD's Upper-Address Ignore (UAI) [27] or Intel's Linear Address Masking (LAM) [22] extensions. UAI ignores the upper 7 bits of a pointer, whereas TBI ignores 8. LAM provides configurable ignoring of either 6 or 15 bits, but unlike UAI and TBI, it applies to different bit ranges: either `ptr[62:57]` or `ptr[62:48]` are ignored.

Delayed kernel support. We implemented and functionally evaluated N00B for all three platforms. However, Linux kernel support for the AMD64 pointer masking extensions has been delayed, in part because they have been shown to facilitate speculative leakage attacks [50], and require a hardware fix. In addition, kernel engineers have also objected more fundamentally to aspects of AMD UAI's design [21], such as masking the most significant pointer bit (see Figure 1) which is traditionally used inside the kernel to distinguish user and kernel pointers in security checks. We still verified the correctness of our implementation for small programs by manually enabling the UAI bit in the appropriate MSRs, but more complex programs require kernel support, e.g., to handle tagged pointers as syscall arguments. Hence, we use a compatibility mode on AMD platforms, where bounds checks decode all in-pointer `N` values to 48, which transparently enforces large bounds, allowing us to keep the `toptag` empty. We verified that this compatibility mode does not modify the generated bounds-checking instruction sequence. Thus, we are confident that our AMD UAI performance results (Section A) accurately reflect what will be observed in the future with kernel support.

For Intel LAM, we selected the U48 variant to implement N00B on top of, although we still only use 8-bit wide `toptag`s. U48 is also not available yet in the mainline kernel, but out-of-tree patches by Intel engineers do exist[1], which we use. Integrating with LAMU57 in the future should be straightforward given N00B's configurable `toptag` size.

[1] https://git.kernel.org/pub/scm/linux/kernel/git/kas/linux.git/log/?h=lam-old

Limited AMD64 Address Space. On AMD64 Linux platforms, user space is only 128TB large [25], which is half of that on AArch64 [24]. We retain the same pointer layout as AArch64, but with one less bit in the N field (6->5). While this cuts the number of encodable size classes in half, we ensure that sufficiently large objects can still be hardened. The 5 N bits encode 32 size classes, one of which must be reserved for non-N00B-managed memory. With the remaining 31 values, we can achieve a maximum size class of 34 if we decode all in-pointer N values to N+4. This yields a minimum allocation size of $2^4 = 16B$ and a maximum of $2^{34} = 16GB$. Larger objects need not crash the process, but can simply be forwarded to the default system allocator in non-N00B-managed memory.

Since the remaining non-N00B-managed N value of 35 is insufficient to ignore all bounds checks transparently, like in AArch64, we introduce a small 32-byte table that maps in-pointer N fields to actual size values, similar to previous work [34]. This allows us to map the non-N00B-managed N value to 48, transparently ignoring bounds checks on non-N00B-managed memory once again.

5.2 Compiler Optimizations

We integrate several compiler optimizations during LTO when hardening applications with N00B. We inject our instrumentation at the FullLinkTimeOptimizationLast extension point, after all optimizations on the underlying application have completed, to prevent interference [62].

Pointer Safety Analysis. We developed an interprocedural backward data flow analysis to prove that each memory access's pointer operand remains within the bounds of every allocation it may derive from. The analysis tracks constant pointer offsets and uses LLVM's ScalarEvolution (SCEV) [1] to estimate extremum values for non-constant offsets. We conduct the analysis twice to check for overflow and underflow, considering maximum and minimum offset values, respectively, and cache results to ensure efficient analysis times. We extend our analysis interprocedurally on function arguments if we can conservatively determine all call sites. We also look through direct calls to functions defined in the same LTO module. We stop at memory loads of pointer values, since *Reaching Definitions Analysis* is expensive and imprecise [2]. Across the C/C++ SPEC CPU 2006 benchmarks, our analysis proves an average of 47% of memory accesses safe, including those dominated by checks on the same pointer, as is common in prior work [3, 34, 62, 94]. We provide per-benchmark statistics in Section B, Table 11.

Loop Optimizations. The remaining memory accesses must be bounds checked. Since checks within loops execute more frequently, they contribute significantly to N00B's overhead. To mitigate this, we implement common optimizations to summarize loop memory access patterns and hoist checks out of the loop body [44, 134, 149]. We avoid hoisting checks on memory accesses that do not post-dominate the loop's preheader, as they may not execute and could hold OoB or invalid pointer values, which should not trigger a bounds check failure if not dereferenced.

When the base pointer of a dereferenced pointer is known before the loop, we precompute all necessary information for bounds checks upfront, emitting only a conditional branch inside the loop. This optimization works even for loop-dependent pointer values, as the original referent is reconstructed solely from the tracked base pointer. For such cases, the instrumentation inside the loop simplifies to:

```
1 if (precomputed_referent != (ptr >> N)) trap();
```

This is a remarkably efficient bounds check that fully capitalizes on the structure of N00B's hybrid checks (Section 4.4).

Unfortunately, many other loop-bound memory accesses remain unhoistable and unanalyzable, even if they post-dominate the preheader. For instance, pointer operands that are loaded from memory on each iteration, such as during a linked-list traversal, offer no hoisting opportunity. In our experiments, LLVM's ScalarEvolution cannot provide usable value-range information for 65% of such non-provably-safe loop-bound memory accesses (see Table 11). The prevalence of these *unpredictable* memory accesses underscores the importance of using efficient bounds-checking instruction sequences (Listing 4.1).

6 Experimental Evaluation

We evaluate N00B's performance and source compatibility on the C and C++ benchmarks of the SPEC CPU2006 and SPECspeed 2017 suites, as they are standard compute-intensive benchmarks in bounds-checking research and contain diverse systems-level codebases that commonly require manual patches to eliminate compatibility issues [63, 69, 86]. We evaluate on the following platforms:

P1 A HiSilicon Kunpeng-920 5250 server with 96 TSV110 ARMv8.2 cores at 2.6GHz and 512GB of RAM, running Ubuntu 22.04.5 on Linux kernel 5.15.0. This platform fully supports N00B through Top-Byte Ignore (TBI).

P2 An Intel Core Ultra 9 285K Arrow Lake desktop with 8 Lion Cove "P" cores at 3.7GHz and 64GB of RAM, running Ubuntu 24.04.3 LTS on a custom Linux 6.1.0-rc3 kernel that includes Intel patches to enable LAM U48 support[2]. This platform fully supports N00B through Intel Linear Address Masking (LAM).

P3 An AMD Ryzen 7 7800X3D desktop with 8 Zen4 AMD64 cores at 4.2GHz and 64GB of RAM, running Ubuntu 24.04.1 on Linux kernel 6.8.0. This platform will support N00B in the future through AMD Upper Address Ignore (UAI), but currently still lacks software support in the kernel.

P4 An AMD Ryzen 7 PRO 4750G desktop with 8 Zen2 AMD64 cores at 3.6GHz and 32GB of RAM, running Ubuntu 20.04.3 on Linux kernel 5.11.0. This platform lacks UAI hardware support for N00B, but provides a contrasting measurement to the 7800X3D from the same microarchitectural family. The 7800X3D features a huge 96MB L3 cache ("3D V-Cache"), which may partially hide N00Balloc's overhead.

P5 An Intel Core i7-12700 Alder Lake desktop with 8 Golden Cove AMD64 "P" cores at 2.1GHz and 32GB of RAM, running Ubuntu 20.04.5 on Linux kernel 6.12.0. This platform also does not support N00B because it predates LAM, but provides an estimate of N00B's overhead on a different Intel microarchitecture.

Together, these platforms cover both major 64-bit ISAs (AMD64 and AArch64) from three different vendors (HiSilicon, AMD, and Intel), five distinct microarchitectures for four product categories

[2] https://git.kernel.org/pub/scm/linux/kernel/git/kas/linux.git/commit/?h=lam-old&id=e64abde4d7ab69eee2b4d11651d989c451930292

(server, gaming, workstation, and desktop), and three implementations of pointer masking (TBI, LAM, and UAI). This variety demonstrates the versatility of N00B and helps us prevent measurement bias [127].

6.1 Source Compatibility

We compare N00B's compatibility with Low-Fat Pointers, a representative object-based bounds checker that struggles to handle benign OoB pointers. To ensure a fair comparison [62], we reimplement Low-Fat Pointers ourselves by modifying N00B to disable `toptag` checking and to narrow object-based arithmetic checks to object boundaries rather than arena boundaries. This essentially yields a power-of-2-aligned version of Low-Fat Pointers [34], which already provides some padding around non-pow2-sized objects that should help with preventing crashes from small OoB pointer motion.

SPEC CPU2006. In the initial run, N00B crashed on two out of the 19 SPEC CPU2006 programs (`403.gcc` and `450.soplex`), while Low-Fat Pointers crashed on five (two shared with N00B, and additionally `400.perlbench`, `483.xalancbmk`, and `447.dealII`). Upon closer inspection, all bugs in `403.gcc` were well-known snippets of undefined behavior that GCC developers themselves had already patched out over 20 years ago (see Table 2), but that did not make it into the SPEC CPU2006 release. All of them are related to far-out-of-bounds pointers, and were repeatedly noted in previous work [34, 63, 86]. In particular, TDI [86], which supports 4GB (!) of OoB pointer arithmetic leeway, applied two identical patches to mitigate this problem. Indeed, we did not implement any patches ourselves to make `403.gcc` compatible with N00B: all detected issues had already been independently discovered and purposefully fixed by the developers, indicating that they already wanted to eliminate this undefined behavior from their code. After applying these small patches, N00B runs `403.gcc` without further issue, but Low-Fat Pointers still crashes on the remaining smaller OoB cases in `403.gcc` that N00B's OoB leniency transparently handles.

The second crashing program was `450.soplex`, which, as also observed by prior work [34], uses *inter-object relative pointers*:

```
1 offset = realloc(array, grow) - array;
2 for (i = 0; i < size; i++)
3     array[i].ptr += offset;
```

The offset computation on line 1 constitutes undefined behavior in C [56] (see Q9 in [84]), since it brings pointers outside of their intended referent, and even dereferences them while OoB. This pattern using `realloc` is recurring but uncommon: CHERI developers found only a handful instances of it during their experience porting > 2 MLoC worth of FreeBSD desktop applications [137], all of which took only minor changes to fix. From N00B's perspective, the fixup operation on line 3 is erroneous since it brings the `ptr` values out of their arena. Inter-object offsets are generally not supported by bounds checking toolchains, including CHERI [140], since they essentially constitute a developer-intended memory error. They also confuse compiler alias analyses and may lead to miscompilations [84], since they semantically implement pointer assignment but do so syntactically via pointer arithmetic.

Nonetheless, some bounds checkers inadvertently do support relative pointers. Red-zone sanitizers, such as AddressSanitizer [118], can only distinguish between accesses to valid objects and accesses to padding between objects, and thus place no constraints on pointer arithmetic or relative pointers. In addition, some pointer-based bounds checkers that fully embed bounds information in the native pointer also forego arithmetic checks [66, 75, 115, 132, 136], which allows the fixup operation on line 3 to update the upper bits "correctly". In theory, such updates should not be allowed at all, because they also allow an attacker to add a well-calculated offset to any pointer to obtain a valid pointer to a different object, which is the very definition of a memory error. In practice, these bounds checkers do allow them, either because they assume attackers will struggle to inject such large offsets in benign applications [69, 132], or because the contents of the top bits are randomized and hard to guess [53, 66, 74, 75, 126]. Previous work has warned against this negligence [57], and even successfully exploited it [48, 54]. Hence, we consider supporting relative pointers a design choice that reduces the security of the bounds checker. N00B, like CHERI [140], does not support this pattern.

Table 2: Applied N00B compatibility patches.

Program	Revision	Year	Message
403.gcc	r62672	2003	"Don't use offset pointers."
403.gcc	r89543	2004	"Avoid undefined pointer arithmetic on qty_table"
403.gcc	r79945	2004	"Don't play queer offsetting games with reg_known_value"

N00B successfully ran all 17 other benchmarks in the SPEC CPU2006 suite without issue, while Low-Fat crashed on three more of them due to benign OoB pointer motion.

SPECspeed 2017. N00B ran all 12 C/C++ programs in the SPECspeed 2017 suite without issues. In contrast, Low-Fat Pointers still crashed on three programs: `600.perlbench_s`, `623.xalancbmk_s`, and `641.leela_s`, all due to benign OoB pointer motion.

Summary. In total, N00B only applied an existing source compatibility patch for 1/30 programs across both SPEC suites, and ran the other 29 programs completely stock, while Low-Fat Pointers crashed on 7 additional programs. This demonstrates the efficacy of N00B's benign OoB leeway.

To estimate the porting effort for Low-Fat, we tried to estimate roughly how many unique issues remain in the programs where it causes crashes. We modified N00B to record every unique bounds check involving a pointer whose `toptag` no longer matches its `iptag`, since that indicates that the pointer is temporarily OoB. Table 3 shows the results. `600.perlbench_s` stands out, with 807 unique locations where it deals with benign OoB pointers. Upon further investigation, all these OoB pointers appear to be underflowing their intended referents at various negative offsets.

This measurement is crude; it counts binary program locations rather than source code locations, and it only includes the coverage of the SPEC reference workloads, which are unlikely to trigger all offending code. Nevertheless, it reinforces the idea that adopting an object-based bounds checker to harden real-world C/C++ code is an uphill battle. In contrast, N00B simply handles all these cases through the pointer leeway provided by its arenas.

6.2 Performance

On all evaluation platforms, we isolate a single (performance) core, fix its speed to its base clock, and pin the benchmark program

Table 3: Unique benign out-of-bounds cases in SPEC programs that Low-Fat Pointers crashes on.

Benchmark	Unique occurrences
400.perlbench	8
403.gcc	45
483.xalancbmk	49
447.dealII	11
600.perlbench_s	807
623.xalancbmk_s	1
641.leela_s	7

to it to avoid scheduling interference. We disable hyperthreading, processor turbos, and Dynamic Voltage Frequency Scaling (DVFS). These are all standard measures to improve the reproducibility of performance benchmarks [7]. We compile all benchmarks with `-flto -O2 -mcpu/-march=native`. Due to space constraints, we only show per-benchmark details in Figure 4 for the platforms on which N00B is fully functional *today*, i.e., the Kunpeng-920 (**P1**) and the Core Ultra 9 285K (**P2**). The other platforms are in Section A.

6.2.1 Impact of N00Balloc. We measure the impact of N00Balloc across hardware platforms with widely varying L3 cache sizes, from 96MB, 48MB, and 36MB for the 7800X3D, Kunpeng, and Core Ultra, respectively, to 25MB on the i7-12700 and just 8MB on the Ryzen 4750G.

Overall, the impact of N00Balloc is insignificant (±0% geomean). However, some applications show overhead spikes under N00Balloc. We investigate 447.dealII's case in detail, because it shows a remarkably high overhead of 65% on the cache-constrained Ryzen 4750G. We find that it spends a lot of time iterating over small individually allocated tree nodes that are just a little larger than 32 bytes, and thus get aligned up to an entire cache line by N00Balloc. This increases the cache footprint of 447.dealII significantly, causing 2.7x more cache misses. On other platforms, 447.dealII is much less impacted, down to just 5% for the Core Ultra.

Artificial baseline. On most other benchmarks, N00Balloc either performs similarly to, or outperforms, the default system allocator. It is not uncommon for academic allocator prototypes to do so [33, 86], as they are not tuned for the same general-purpose workloads. However, this complicates measuring the overhead of N00B's full instrumentation [91]. If we use the native measurement as a baseline, speedups in N00Balloc may partially hide N00B's overhead. However, using the N00Balloc measurement as a baseline eliminates the overhead of N00Balloc, including its effect on memory layout. Therefore, we created an artificial baseline that consists of the fastest measurements from both N00Balloc and native on each benchmark. This artificial baseline essentially assumes a worst-case scenario that ignores any speedups N00Balloc would provide, but includes all slowdowns. We still provide dedicated N00Balloc overhead measurements for comparison. Unless explicitly stated otherwise, all N00B overhead figures are calculated using this artificial baseline.

GCC regression. Finally, we observed a regression in both the 2006 and 2017 versions of GCC, where its garbage collector implementation maintains page-table-like structures for each 4GB region of virtual memory. Heap pointers are assigned to memory regions based on their upper 32 address bits, which usually yields a short list of regions, since most heap pointers do not differ much in their most significant bits [9]. However, in the case of N00B, the upper bits of pointers differ in the toptag and N bits, which broke this assumption of GCC's code, yielding a very large list of regions and extreme overhead during simple lookups, slowing down the program by over 3.7x overall. We created a small 3-line patch to strip pointers of their N00B toptag before looking up their "page table", which eliminated this overhead entirely. Reported overhead results for GCC include this patch. Note that the patch does not affect N00B's security guarantees: without a valid toptag, the stripped pointers can never be dereferenced.

Any use of the upper pointer bits in GCC might trigger this regression. In fact, previous bounds checkers observed the same problem [63, 69], and applied similar patches.

6.2.2 Memory and Code Size Overhead. Table 4 shows the results for peak Resident Set Size (RSS) and code size overheads on the Kunpeng-920. Peak RSS overhead is slightly lower on the AMD64 platforms, but code size overhead is higher (see Section A Tables 8 to 10). Averaged across all benchmarks and platforms, N00B's peak RSS overhead is 50.6%. Code size overhead is 79% on AArch64, and 113% on AMD64.

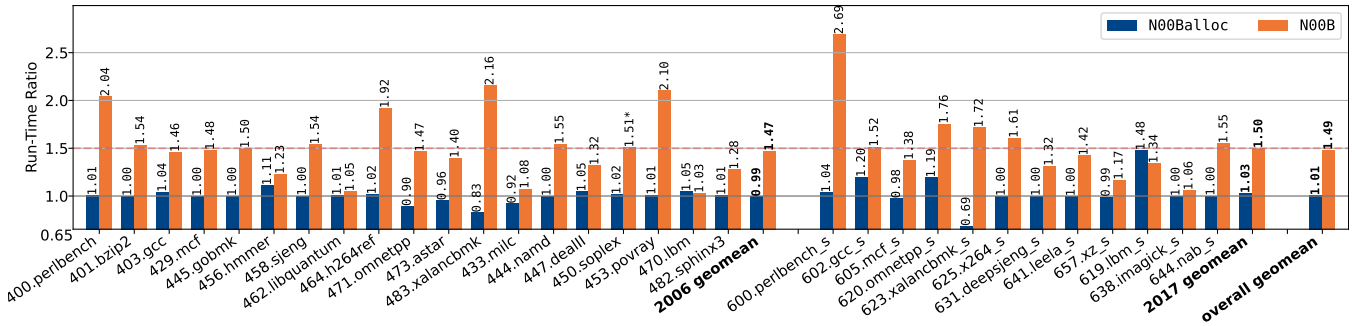
While power-of-2 over-allocation should theoretically only cause a maximum 2x memory overhead, we observe a few outliers, with up to ≥ 10× peak RSS overhead on 462.libquantum, 464.h264ref, and 433.milc. These applications do not exhibit high run-time overhead under N00Balloc, suggesting that their Working Set Size (WSS) is not significantly increased. Instead, we attribute this increased RSS peak to N00Balloc's reluctance to unmap arenas after use. Our benchmarking setup does not trigger the kernel to reclaim physical memory from these stale arenas, causing peaks in memory usage across different size classes to linger and accumulate, even if the underlying memory is not in use.

Hence, Peak RSS is a misleading metric, which does more to measure the kernel's reclamation policy than it does the memory efficiency of a user-space program [59]. We include it because it is a standard metric, but did not modify N00Balloc's implementation to optimize it.

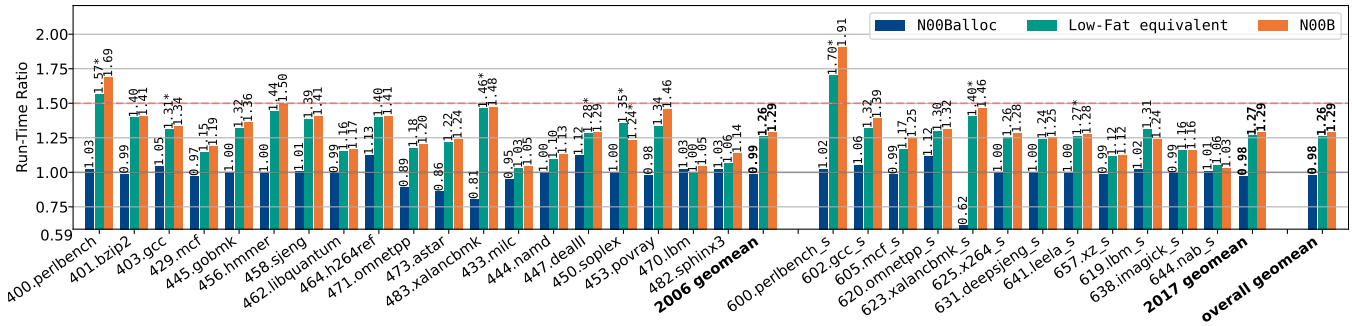
6.2.3 N00B's Run-Time Overhead. With full checking enabled, N00B achieves 29% geomean run-time overhead across SPEC CPU 2006 and 2017 on the Kunpeng, and 49% on the Core Ultra.

Platform differences. The overhead on Kunpeng is exceedingly low for full inter-object spatial memory safety enforcement, and much lower than on all AMD64 platforms. Disregarding the impact of N00Balloc, which is determined mainly by cache size, we attribute the differences between platforms to varying baseline saturations of the ALU execution units. With less baseline contention, N00B's bounds checks can be scheduled on free execution units without delaying baseline operations, which increases the Instructions Per Cycle (IPC), crucial for keeping the overhead of bounds checkers at bay [105]. Hence, we speculate that the Core Ultra's microarchitecture fits the SPEC workloads better, which creates more contention between N00B operations and baseline operations on the ALU units.

Comparison with state of the art. Yet, even on the 7800X3D, which shows the highest overhead among all evaluated platforms



(a) Intel Core Ultra 9 285K (AMD64-Lion Cove).



(b) HiSilicon Kunpeng-920 (AArch64-TSV110).

Figure 4: Run-time overhead on SPEC CPU 2006 and SPECspeed 2017. "⋆" = compatibility mode.

(54%), N00B's slowdown remains on par with the state of the art. In fact, we are aware of only five other software-based bounds checkers that have achieved comparable overhead (Table 5), but all come with significant practical drawbacks.

Delta Pointers [69] (which only checks upper bounds) and CGuard [63] both encode bounds as pointer deltas, requiring updates on all arithmetic operations. This necessitates precise instrumentation of every pointer-arithmetic instruction—a challenge even with full source-code access [86]. Furthermore, both store their pointer deltas in non-ignored upper pointer bits, creating interoperability challenges with external code. Unsurprisingly, they require extensive patches to run SPEC benchmarks.

Conversely, Low-Fat Pointers [34], CAMP [79], and ShadowBound [147] are object-based bounds checkers that lack support for benign OoB pointers, as demonstrated by our Low-Fat evaluation in Section 6.1. ShadowBound and CAMP are additionally exclusive to the heap by design, and include aggressive optimizations to eliminate checks on non-heap memory accesses, making direct comparisons difficult.

This review highlights the unique and advantageous space that N00B's hybrid design occupies. It achieves overhead comparable to the most efficient prior works, while avoiding their practical limitations regarding compatibility and interoperability. This is largely made possible by the efficient implementation of N00B's hybrid pointer- and object-based checks (Section 4.4), which prevents a naive doubling of overhead. The most compelling evidence for this is our Low-Fat equivalent: although its arithmetic checks perform only half the work of N00B's hybrid checks, the reduction in overhead is relatively minor (8pp on average). In return, N00B can natively run 7 additional SPEC benchmarks.

6.3 Security

We used the Juliet 1.3 suite of C/C++ vulnerabilities by NIST [101] to confirm that N00B stops spatial memory errors, similar to related work [12, 42, 46, 52, 74, 75]. We ran these tests on **P1** (ARM TBI). Table 6 shows the results for the relevant test cases, i.e., those that perform out-of-bounds accesses. We consider a few instances of the test cases out of scope. N00B is not designed to handle intra-allocation overflows [40], which we discuss further in Section 8. Additionally, some test cases only trigger bugs in external code, which is also out of scope.

N00B correctly mitigated all in-scope vulnerabilities without triggering false positives. We manually analyzed the behavior of each test case to check whether and how the vulnerability was mitigated. Some test cases were mitigated without a visible N00B bounds check failure, since they did not escape the padded power-of-2 object bounds provided by N00Balloc. Since this padding contains no useful information and is never benignly read by the program, accesses to this padding are not exploitable by attackers.

We also found that one test case did not trigger any bug, regardless of whether N00B was present or not. The test case attempted to create a 32-bit integer overflow during heap allocation, but failed to do so on 64-bit architectures. Upon reporting this bug, NIST acknowledged the issue. N00B successfully detects the error in our modified test case that does overflow the size variable.

7 Selectively Hardening FFmpeg

To evaluate the selective hardening benefits of N00B in a real-world scenario, we selected FFmpeg [37], a highly popular multimedia processing library, written in C and assembly, that is both performance-sensitive and security-critical. Typical FFmpeg deployments handle

Table 4: Peak Resident Set Size (RSS) and `.text` section size overhead on the Kunpeng-920.

	Peak RSS		.text Size	
Benchmark	Baseline (MB)	N00B	Baseline (KB)	N00B
400.perlbench	663.8	1.19×	1223.23	1.92×
401.bzip2	611.0	1.01×	63.26	2.01×
403.gcc	95.7	2.14×	3409.11	1.82×
429.mcf	1677.4	1.00×	10.17	1.99×
445.gobmk	29.8	1.10×	2076.16	1.66×
456.hmmer	4.8	1.79×	155.83	1.78×
458.sjeng	176.1	1.01×	127.96	1.64×
462.libquantum	97.5	9.36×	20.45	1.72×
464.h264ref	64.6	8.65×	555.52	2.26×
471.omnetpp	171.3	1.20×	720.57	1.49×
473.astar	134.7	1.07×	32.37	1.50×
483.xalancbmk	420.2	1.52×	3479.56	1.82×
433.milc	680.5	11.50×	98.18	1.90×
444.namd	48.4	1.03×	175.31	1.97×
447.dealII	794.6	1.53×	469.24	1.87×
450.soplex	423.6	1.64×	309.29	2.04×
453.povray	7.3	1.51×	1064.83	1.67×
470.lbm	410.2	1.01×	10.20	2.17×
482.sphinx3	43.8	2.03×	155.29	1.72×
SPEC CPU2006		**1.80×**		**1.83×**
600.perlbench_s	89.8	1.26×	2297.45	1.79×
602.gcc_s	3100.7	1.16×	9575.29	1.84×
605.mcf_s	3960.9	3.18×	18.47	1.88×
620.omnetpp_s	241.4	1.15×	1915.46	1.45×
623.xalancbmk_s	478.9	1.45×	4010.11	2.02×
625.x264_s	155.7	1.03×	550.46	1.90×
631.deepsjeng_s	6881.3	1.00×	71.34	1.56×
641.leela_s	25.1	1.23×	84.08	1.58×
657.xz_s	4201.5	1.15×	125.29	1.77×
619.lbm_s	3224.0	1.00×	10.30	2.11×
638.imagick_s	7006.7	1.20×	1203.27	1.62×
644.nab_s	557.2	1.31×	86.22	1.61×
SPECspeed 2017		**1.27×**		**1.75×**

Table 5: Run-time overhead comparison of N00B versus the fastest current software-based bounds checkers.

Bounds Checker	SPEC CPU2006	SPECspeed 2017
CAMP [79]	76.1% [44]	48.6% [44]
ShadowBound [147]	20.1% [44]	5.72%
Low-Fat [31]	26–43%	27–42%
Delta Pointers [69]	35% [46]	32.4% [43]
CGuard [63]	–	42.1%
N00B	**29–54%**	**29–54%**

Table 6: Juliet Test Suite detection and mitigation results.

Description (CWE)	Total	Out of Scope	Mitigated due to N00Balloc	Detected	Missed
Stack buffer overflow (CWE121)	58	41	9	8	0
Heap buffer overflow (CWE122)	63	42	11	10	0
Write-What-Where (CWE123)	1	0	0	1	0
Buffer underwrite (CWE124)	19	16	0	3	0
Buffer overread (CWE126)	11	8	1	2	0
Buffer underread (CWE127)	19	16	0	3	0
Integer overflow (CWE680)	6	0	0	6	0

complex media input from untrusted sources, e.g., when probing or processing user-uploaded files, with malicious actors looking to exploit bugs in the parsers or codecs. At the same time, being a media processing tool, almost all FFmpeg operations are highly performance-sensitive.

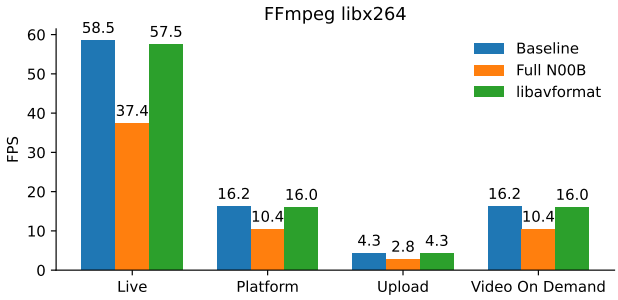


Figure 5: FFmpeg x264 throughput without hardening, with full N00B hardening, and with partial N00B hardening of `libavformat`.

In this section, we evaluate a selective hardening of FFmpeg with N00B, comparing the performance against both a baseline and a fully N00B-hardened version. We selected `libavformat` as our target for selective hardening, as it directly interfaces with untrusted input (media container metadata), but executes far less frequently than, e.g., the per-pixel operations of the video decoder.

Setup. FFmpeg's fastest codecs are notoriously implemented in assembly, which cannot be instrumented by N00B, so we use the `-disable-asm` configure flag to build a C-only baseline version. This allows us to accurately measure the overhead of the hardening instrumentation. In addition, to avoid interference from multiple threads, we perform a single-threaded benchmark of FFmpeg 5.1.8. We use the Phoronix Test Suite (PTS) [107], which uses vbench [82] under the hood, to supply the benchmark harness and workloads, and select the x264 [131] codec as our benchmark target.

We create a full LTO build of FFmpeg and x264 together, so all three benchmarks start from the exact same build setup. The baseline version compiles using the default `-O2 -march=native` pipeline in Clang 15.0.7, the full N00B instrumented version additionally hardens the complete FFmpeg+x264 LTO module, and the selective `libavformat` version applies stack object hardening to the complete LTO module, but only adds bounds checks to functions that originally came from files inside the `libavformat/` folder. The current N00B prototype enables this through the `NOOB_SELECT=<module>` environment variable, which, when set, selectively instruments all LLVM Functions whose debug metadata indicates that their original filename contained the `<module>` string. For `libavformat`, this resulted in only around 11% of the bounds check coverage that would be applied under full instrumentation.

Security. To validate the security efficacy of our selective approach, we verified that it prevents CVE-2022-2566 [102]. This vulnerability is a heap out-of-bounds write caused by an integer overflow during heap allocation. The vulnerability is considered highly severe: supplying a vulnerable version of FFmpeg with a single maliciously crafted MP4 already suffices to enable code execution. This is exactly the type of vulnerability that suits selective N00B: a high-severity, non-linear buffer overflow (which could bypass simpler red-zone defenses [46, 150]) located in a security-critical but less performance-sensitive component.

Using the public PoC[3], we confirm that our selectively hardened FFmpeg indeed detects the memory error.

Performance. We then run the complete PTS x264 benchmark on each configuration. We use **P2** (Intel LAM) for these benchmarks. Figure 5 shows the results. Full N00B hardening reduces Frames Per Second (FPS) to 64.2% (geomean), which corresponds to a runtime overhead of 55.8%, in line with the SPEC results. However, hardening `libavformat` alone recovers nearly all the baseline performance, at **98.6% FPS**. Hence, selective N00B application incurs just a minor slowdown while comprehensively hardening FFmpeg's input parsing logic, and protecting it from high-severity vulnerabilities like CVE-2022-2566.

8 Discussion

Being a research prototype, N00B lacks features that require additional engineering effort to implement. Notably, it does not enforce strict bounds for **global** arrays, as embedding the `toptag` in global addresses involves handling various linkage and symbol types [36]. While our performance results do include the cost of bounds checks on globals, including their alignment, we map them in non-N00B-managed memory and enforce always-passing bounds. Recent work has addressed global variables similarly [46].

In addition, N00B does not currently detect **sub-object overflows** [40], such as bounds violations between struct members or array elements. This limitation is shared with many low-overhead, software-based bounds checkers, as sub-object protection introduces significant design challenges.

Two main approaches exist. One transforms struct member arrays into distinct allocations using source code rewriting tools like `buf2ptr` [151], and protects them through inter-object bounds enforcement instead [150]. While technically feasible for N00B, this approach breaks ABI compatibility by altering data-structure layouts, requiring universal recompilation similar to fat pointers. The second approach, *bounds narrowing*, derives and enforces fine-grained bounds for sub-object pointers. The main challenges here are (i) correctly inferring sub-object bounds based on unreliable static type information [109], (ii) attaching those bounds to escaping interior pointers, and (iii) accommodating programming practices that intentionally violate sub-object bounds, such as Linux's infamous `container_of` macro [67, 83, 94].

While these challenges are largely orthogonal to the practice of allocation-granular bounds checking, challenge (ii) can benefit from a bounds checker's ability to represent arbitrary sub-object bounds flexibly. Doing so typically precludes the use of clever size-alignment techniques that nearly all efficient bounds checkers without disjoint bounds metadata rely on [3, 34, 71, 79, 150], including N00B, since sub-object alignment would break ABI. Hence, defenses that do support bounds narrowing tend to be either hardware-accelerated [15, 140, 141] or incur significant slowdown due to additional metadata queries [16, 32, 94, 97]. For these reasons, along with the static analysis and compatibility issues, bounds narrowing is generally an opt-in feature [139].

For sub-object overflow detection in N00B, we therefore look towards orthogonal mechanisms that can complement N00B's rigid allocation-granular bounds. An attractive option in this regard is (hardware) memory tagging. Even a single tag bit could deterministically stop linear buffer overflows among sub-object members [51, 76], and more could provide even stronger protection [32, 51]. These memory tags would have to share the upper pointer bits with N00B's `toptag`, but N00B's configurable `TAG_WIDTH` does make that possible. Exploring this integration on hardware with native tagging support (e.g., ARM MTE) constitutes a promising direction for future work.

Additionally, spatial safety is not the only form of memory safety. **Temporal memory errors**, also known as "Use-After-Free" (UAF) issues [89], occur when a pointer is dereferenced after its *intended referent* has already been deallocated and its memory reused for a different object [39]. UAF vulnerabilities form the basis for many expressive attacks [119], particularly those looking to craft malicious type-confused objects [86, 116, 128]. They generally remain limited to heap-allocated data [18], where UAF-aware heap allocators can temper them with type-safe memory reuse [86, 128, 135], dangling pointer tracking [11, 29, 45, 110, 120, 145] or invalidation [79, 129], probabilistic memory quarantining [103, 118], and one-time allocation in the most extreme case [23, 138].

Type-safe memory reuse and memory quarantining, in particular, have gradually found their way into major software projects [4, 19, 73, 78, 108]. While these approaches are not bulletproof, they help inhibit an important attack vector at low overhead. N00B's arenas already require same-size objects, which would be trivially satisfied by same-type objects as well. Hence, N00B could share the memory layout of type-homogeneous memory pools, and through its spatial safety checks help avoid cross-pool overflows [86].

Finally, some platforms may **lack kernel or hardware support** for pointer masking. Here, we want to explore the possibility of virtual page aliasing to embed the toptag into the effective address bits, similar to xTag [6]. Still, address space exhaustion may be a challenge: the effective address bits must now accomodate `power`, `iptag` *and* `toptag` bits, which may require tag width reductions, or could only be feasible on 5-level-paging platforms.

9 Related Work

We cover most directly related work in Table 1 and Section 2.

Similar Ideas. A select number of papers have implemented similar techniques to N00B. **FRAMER** [97] identifies imaginary aligned regions of 2^N size around memory objects, similar to C^3 [74], which it uses as a reference frame within which a metadata object can be located by embedding its offset to the start of the frame into the upper bits of the pointer. The metadata object contains precise bounds information for the intended referent, which it enforces at dereferences. Similar to N00B's arenas, FRAMER must prevent pointers from escaping their "frame" to avoid invalidating the upper pointer bits, and it does so through arithmetic constraints. The original paper proposed enforcing allocation-granular object-based bounds checks instead of frame boundaries [97], which is overly strict and resulted in classic object-based compatibility issues. Recent follow-up work [96] updated the implementation. Regardless, FRAMER still uses an additional metadata object to store precise bounds information at the cost of an extra indirection through memory for every bounds check, leading to high overhead (> 200% on average).

[3] https://github.com/advisories/GHSA-m864-5788-g574

ZOMETag [117] reserves aligned 4GB "zones" within which up to 16 different objects can be allocated simultaneously. Each object in a zone receives a unique 4-bit tag using ARM MTE. During pointer arithmetic, ZOMETag implements highly efficient arithmetic constraints that prevent pointers from escaping the 4GB zone, similar to TDI [86]. While the spirit of the design aligns well with N00B, ZOMETag strikes a more esoteric balance between compatibility and efficiency. The 4GB zones are overly large and vastly under-utilized: given the prevalence of small objects [97], supporting only 16 per zone creates immensely sparse memory layouts and significantly increases TLB pressure. ZOMETag also incurs additional overhead from explicit tagging during allocation, which is especially impactful for stack objects [46]. We believe ZOMETag could benefit from N00B's *implicit* in-pointer tags, which would eliminate explicit tagging operations and increase the number of simultaneously allocated objects per zone 16-fold.

StickyTags [46] tags objects using predictable 4-bit ARM MTE tags based on the object's location. This creates fixed "red zones" of 16×`obj_size` bytes around the object, which can stop linear buffer overflows and non-linear overflows with offsets smaller than the red zone size. Conceptually and from a security perspective, it resembles N00B without arithmetic checks, although N00B would still enforce much larger red zones on AArch64 with its 8-bit tags. Given the almost negligible overhead on ARM MTE-enabled platforms (4%), StickyTags provides an attractive security-overhead trade-off on hardware that supports it. The authors also evaluated an x86 version with classical in-memory red zones of adjustable size, which incurred higher overhead. This x86 version, in particular, may benefit from using N00B's implicit in-pointer tags, given StickyTags' already-deterministic nature. In addition, on MTE-enabled hardware, one could combine implicit 4-bit N00B-like "implicit" sticky tags for spatial memory safety with probabilistic explicit MTE tags for temporal memory safety.

Sanitizers. N00B is an exploit mitigation designed for production use against sophisticated adversaries. In contrast, bug-detection tools like AddressSanitizer [118] focus on detecting as many accidental memory errors as possible during development or fuzz-testing. Often implemented using red zones ("tripwires" [93]) [42, 44, 51, 118, 149], they simply distinguish accesses to allocated objects from accesses to inserted padding in between objects, without tracking pointer bounds or checking arithmetic, which avoids the functional issues we described in Section 2.

While N00B provides better spatial security at a lower overhead, its power-of-2 over-alignment may miss small overflows, making it less suitable for a debugging scenario. However, previous work shows that such defenses can still be adapted for testing purposes by filling excessive padding bytes with canaries [72] or fine-grained bounds information [44], or by adapting checks to variable object sizes [34]. We aim to explore these options in a future extension.

10 Conclusion

In this paper, we identified fundamental limitations of dominant bounds checking paradigms and introduced *hybrid* bounds checking as a best-of-both-worlds alternative. We overcame long-standing *functional* challenges when hardening legacy C/C++ programs and incomplete code bases, and suppressed run-time overhead through efficient bounds checks and selective hardening.

The result is truly a bounds checker for the masses. N00B runs on commodity hardware, using extensions available on all major architectures, and pragmatically supports common C/C++ programming constructs and deployment realities that stymied previous work. Our evaluation on standard benchmark suites shows that N00B's overhead remains competitive with state-of-the-art alternatives, while demonstrating significant *functional* advantages.

Open Science

N00B's full source code and results are available at https://github.com/adriaanjacobs/NOOB.

Acknowledgments

We thank the anonymous reviewers for their feedback. We also thank everyone who gave feedback on earlier drafts, including Anton Schelfhout, Dimitris Stavrakakis, Cristiano Giuffrida, Jonas Vinck, and Alicia Andries. We thank Ruben Sturm, Ruben Mechelinck, and Thomas Frans for their linker and loader expertise that helped us develop the N00Bloader, and Tianchi Yu for helping brainstorm many a pointer layout. Finally, we thank Alyssa Milburn for being so open and clear about SPEC compatibility issues in the TDI paper [86], and Floris Gorter for a much-needed critical look at benchmarking practices in recent bounds checking work [44].

This research is partially funded by the Internal Funds KU Leuven, and by the Cybersecurity Research Program Flanders. Adriaan Jacobs and Carlo Ramponi performed part of this work during an internship at the Huawei Research Center in Munich.

References

[1] J. Absar. 2018. Scalar Evolution - Demystified. *EuroLLVM Developers' Meeting* (2018).
[2] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, & tools.* Pearson Education India.
[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 *USENIX Security Symposium*, Vol. 10. 96.
[4] Apple Security Research. 2022. Towards the next generation of XNU memory safety: kalloc_type. https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/
[5] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. Association for Computing Machinery, New York, NY, USA, 290–301. doi:10.1145/178243.178446
[6] Lukas Bernhard, Michael Rodler, Thorsten Holz, and Lucas Davit. 2022. xTag: Mitigating use-after-free vulnerabilities via software-based pointer tagging on Intel X86-64. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 502–519.
[7] Dirk Beyer, Stefan Löwe, and Philipp Wendler. 2019. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer* 21, 1 (2019), 1–29.
[8] Joe Bialek, Ken Johnson, Matt Miller, and Tony Chen. 2020. Security Analysis of Memory Tagging. https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20memory%20tagging.pdf.
[9] Lorenzo Binosi, Gregorio Barzasi, Michele Carminati, Stefano Zanero, and Mario Polino. 2024. The Illusion of Randomness: An Empirical Analysis of Address Space Layout Randomization Implementations. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 1360–1374.
[10] Johannes Blaser, Floris Gorter, Klaus v. Gleissenthall, and Herbert Bos. 2026. QuickSafe: Targeted Hardening Against Memory Corruption . In *2026 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1580–1598. doi:10.1109/SP63933.2026.00085
[11] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience* 18, 9 (1988), 807–820.

[12] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. 2018. CUP: Comprehensive User-Space Protection for C/C++. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS '18)*. Association for Computing Machinery, New York, NY, USA, 381–392. doi:10.1145/3196494.3196540

[13] Scott A. Carr and Mathias Payer. 2017. DataShield: Configurable Data Confidentiality and Integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates) *(ASIA CCS '17)*. Association for Computing Machinery, New York, NY, USA, 193–204. doi:10.1145/3052973.3052983

[14] Nicholas P Carter, Stephen W Keckler, and William J Dally. 1994. Hardware support for fast capability-based addressing. *ACM SIGOPS Operating Systems Review* 28, 5 (1994), 319–327.

[15] Yu-Chang Chen and Shih-Wei Li. 2024. Hemate: Enhancing heap security through isolating primitive types with arm memory tagging extension. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*. 1–11.

[16] Zhe Chen, Rui Yan, Yingzi Ma, Yulei Sui, and Jingling Xue. 2024. A smart status based monitoring algorithm for the dynamic analysis of memory safety. *ACM Transactions on Software Engineering and Methodology* 33, 4 (2024), 1–47.

[17] David Chisnall, Colin Rothwell, Robert NM Watson, Jonathan Woodruff, Munraj Vadera, Simon W Moore, Michael Roe, Brooks Davis, and Peter G Neumann. 2015. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 117–130.

[18] Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. 2022. ViK: practical mitigation of temporal memory safety violations through object ID inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 271–284.

[19] Chromium Project. 2025. PartitionAlloc Design. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/base/allocator/partition_allocator/PartitionAlloc.md

[20] CISA, FBI, ASD's ACSC, and CCCS. 2024. Exploring Memory Safety in Critical Open Source Projects. https://www.cisa.gov/sites/default/files/2024-06/joint-guidance-exploring-memory-safety-in-critical-open-source-projects-508c.pdf.

[21] Jonathan Corbet. 2022. Pointer tagging for x86 systems. *LWN.net* (2022). https://lwn.net/Articles/888914/

[22] Intel Corporation. 2025. *Intel 64 and IA-32 Architectures Software Developer's Manual*. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[23] Thurston HY Dang, Petros Maniatis, and David Wagner. 2017. Oscar: A practical {Page-Permissions-Based} scheme for thwarting dangling pointers. In *26th USENIX security symposium (USENIX security 17)*. 815–832.

[24] Linux developers. 2025. Memory Layout on AArch64 Linux. https://docs.kernel.org/arch/arm64/memory.html.

[25] Linux developers. 2025. Memory Management - Complete virtual memory map with 4-level page tables. https://docs.kernel.org/arch/x86/x86_64/mm.html#complete-virtual-memory-map-with-4-level-page-tables.

[26] LLVM Developers. 2025. SafeStack - Clang documentation. https://clang.llvm.org/docs/SafeStack.html.

[27] Advanced Micro Devices. 2025. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf

[28] Dinakar Dhurjati and Vikram Adve. 2006. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th international conference on Software engineering*. 162–171.

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

[30] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 144–157. doi:10.1145/1133981.1133999

[31] Gregory Duck, Roland Yap, and Lorenzo Cavallaro. 2017. Stack object protection with low fat pointers. In *NDSS Symposium 2017*.

[32] Gregory J Duck and Roland HC Yap. 2018. EffectiveSan: type and memory error detection using dynamically typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 181–195.

[33] Gregory J Duck and Roland HC Yap. 2018. An extended low fat allocator API and applications. *arXiv preprint arXiv:1804.04812* (2018).

[34] 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 2016)*. Association for Computing Machinery, New York, NY, USA, 132–142. doi:10.1145/2892208.2892212

[35] Gregory J. Duck, Yuntong Zhang, and Roland H. C. Yap. 2022. Hardening binaries against more memory errors. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 117–131. doi:10.1145/3492321.3519580

[36] Evgeny777. 2019. [HWASAN] Instrument globals. https://reviews.llvm.org/D56672.

[37] FFmpeg Developers. 2025. FFmpeg. https://ffmpeg.org

[38] Matthew Field. 2018. WannaCry cyber attack cost the NHS £92m as 19,000 appointments cancelled. https://www.telegraph.co.uk/technology/2018/10/11/wannacry-cyber-attack-cost-nhs-92m-19000-appointments-cancelled/.

[39] 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 *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 608–625.

[40] Ronald Gil, Hamed Okhravi, and Howard Shrobe. 2018. There's a Hole in the Bottom of the C: On the Effectiveness of Allocation Protection. In *2018 IEEE Cybersecurity Development (SecDev)*. IEEE, 102–109.

[41] Amogha Udupa Shankaranarayana Gopal, Raveendra Soori, Michael Ferdman, and Dongyoon Lee. 2023. TAILCHECK: A Lightweight Heap Overflow Detection Mechanism with Page Protection and Tagged Pointers. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*.

[42] Floris Gorter, Enrico Barberis, Raphael Isemann, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. 2023. FloatZone: Accelerating Memory Error Detection using the Floating Point Unit. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 805–822. https://www.usenix.org/conference/usenixsecurity23/presentation/gorter

[43] Floris Gorter and Cristiano Giuffrida. 2025. Canon Pointers: Conditional Delta Tagging for Compatible Buffer Overflow Protection. *PLAS, October* (2025).

[44] Floris Gorter and Cristiano Giuffrida. 2025. RangeSanitizer: Detecting Memory Errors with Efficient Range Checks. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association, Seattle, WA, 4501–4519. https://www.usenix.org/conference/usenixsecurity25/presentation/gorter

[45] Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2022. Dangzero: Efficient use-after-free detection via direct page table access. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1307–1322.

[46] Floris Gorter, Taddeus Kroes, Herbert Bos, and Cristiano Giuffrida. 2024. Sticky Tags: Efficient and Deterministic Spatial Memory Error Mitigation using Persistent Memory Tags. In *2024 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 217–217.

[47] Istvan Haller, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. 2016. METAlloc: Efficient and Comprehensive Metadata Management for Software Security Hardening. In *Proceedings of the 9th European Workshop on System Security (EuroSec '16)*. Association for Computing Machinery, New York, NY, USA, Article 5, 6 pages. doi:10.1145/2905760.2905766

[48] Mohamed Tarek Bnziad Mohamed Hassan. 2022. *Hardware-Software Co-design for Practical Memory Safety*. Ph. D. Dissertation. Columbia University.

[49] Ben Hawkes. 2019. News and updates from the Project Zero team at Google. https://googleprojectzero.blogspot.com/p/0day.html.

[50] Mathé Hertogh, Sander Wiebing, and Cristiano Giuffrida. 2024. Leaky address masking: Exploiting unmasked spectre gadgets with noncanonical address translation. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3773–3788.

[51] Konrad Hohentanner, Florian Kasten, and Lukas Auer. 2023. HWASanIO: Detecting C/C++ Intra-object Overflows with Memory Shading. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (Orlando, FL, USA) *(SOAP 2023)*. Association for Computing Machinery, New York, NY, USA, 27–33. doi:10.1145/3589250.3596139

[52] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. 2023. CryptSan: Leveraging ARM Pointer Authentication for Memory Safety in C/C++. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23)*. Association for Computing Machinery, New York, NY, USA, 1530–1539. doi:10.1145/3555776.3577635

[53] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. 2023. CryptSan: Leveraging ARM pointer authentication for memory safety in C/C++. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. 1530–1539.

[54] Mohamed Tarek Ibn Ziad, Evgeny Manzhosov, and Simha Sethumadhavan. 2022. C-4: Compromising cryptographic capability computing. Work in progress.

[55] SOCRadar® Cyber Intelligence Inc. 2023. Exploring the Top Vulnerabilities Exploited by State-Sponsored Threat Actors. https://socradar.io/exploring-the-top-vulnerabilities-exploited-by-state-sponsored-threat-actors/.

[56] ISO. 2011. *C11 Standard*. /bib/iso/C11/n1570.pdf ISO/IEC 9899:2011.

[57] Adriaan Jacobs and Stijn Volckaert. 2024. Not Quite Write: On the Effectiveness of {Store-Only} Bounds Checking. In *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*. 171–187.

[58] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02)*. USENIX Association, USA, 275–288.

[59] Mark S Johnstone and Paul R Wilson. 1998. The memory fragmentation problem: Solved? *ACM Sigplan Notices* 34, 3 (1998), 26–36.

[60] Richard WM Jones and Paul HJ Kelly. 1997. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs.. In *AADEBUG*, Vol. 97. 13–26.

[61] Tina Jung, Fabian Ritter, and Sebastian Hack. 2021. PICO: A Presburger In-Bounds Check Optimization for Compiler-Based Memory Safety Instrumentations. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 4 (2021), 1–27.

[62] Tina Jung, Fabian Ritter, and Sebastian Hack. 2025. Memory Safety Instrumentations in Practice: Usability, Performance, and Security Guarantees. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 390–404.

[63] Piyus Kedia, Rahul Purandare, Udit Agarwal, and Rishabh. 2023. CGuard: Scalable and Precise Object Bounds Protection for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1307–1318.

[64] Paul Kehrer. 2021. Memory Unsafety in Apple's Operating Systems. https://langui.sh/2021/12/13/apple-memory-safety/.

[65] Samuel C Kendall. 1983. Bcc: Runtime checking for C programs. In *Proceedings of the USENIX Summer Conference*. 5–16.

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

[67] Greg Kroah-Hartman. 2005. The container_of() Macro. http://www.kroah.com/log/linux/container_of.html.

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

[69] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta Pointers: Buffer Overflow Checks without the Checks. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 22, 14 pages. doi:10.1145/3190508.3190553

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

[71] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre 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 2013 ACM SIGSAC conference on Computer & communications security*. 721–732.

[72] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. 2022. Securing gpu via region-based bounds checking. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 27–41.

[73] Daan Leijen, Benjamin Zorn, and Leonardo De Moura. 2019. Mimalloc: Free list sharding in action. In *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*. Springer, 244–265.

[74] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M. Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. 2021. Cryptographic Capability Computing. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 253–267. doi:10.1145/3466752.3480076

[75] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. 2022. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. ACM, 1901–1915. doi:10.1145/3548606.3560598

[76] Hans Liljestrand, Carlos Chinea, Rémi Denis-Courmont, Jan-Erik Ekberg, and N Asokan. 2022. Color My World: Deterministic Tagging for Memory Safety. *arXiv preprint arXiv:2204.03781* (2022).

[77] ARM Limited. 2025. *ARM Architecture Reference Manual, Armv8, for Armv8-A architecture profile*. https://developer.arm.com/documentation/ddi0487/latest/

[78] Zhenpeng Lin. 2021. How AUTOSLAB Changes the Memory Unsafety Game. https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game

[79] Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing. 2024. CAMP: Compiler and Allocator-based Heap Memory Protection. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 4015–4032. https://www.usenix.org/conference/usenixsecurity24/presentation/lin-zhenpeng

[80] Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. 2024. GIANTSAN: Efficient Memory Sanitization with Segment Folding. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 433–449.

[81] Zhengyang Liu and John Criswell. 2017. Flexible and efficient memory object metadata. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM 2017)*. Association for Computing Machinery, New York, NY, USA, 36–46. doi:10.1145/3092255.3092268

[82] Andrea Lottarini, Alex Ramirez, Joel Coburn, Martha A Kim, Parthasarathy Ranganathan, Daniel Stodolsky, and Mark Wachsler. 2018. vbench: Benchmarking video transcoding in the cloud. *ACM SIGPLAN Notices* 53, 2 (2018), 797–809.

[83] Kangjie Lu. 2023. Practical program modularization with type-based dependence analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1256–1270.

[84] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/2908080.2908081

[85] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. 2022. Cross-Language Attacks.. In *NDSS*. 1–18.

[86] A. Milburn, E. van der Kouwe, and C. Giuffrida. 2022. Mitigating Information Leakage Vulnerabilities with Type-based Data Isolation. In *2022 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Los Alamitos, CA, USA, 259–275. doi:10.1109/SP46214.2022.00016

[87] Matt Miller. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape, 2019. *BlueHatIL 2019* (2019). https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf

[88] MITRE. 2024. CWE-125: Out-of-bounds Read. https://cwe.mitre.org/data/definitions/125.html.

[89] MITRE. 2024. CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html.

[90] MITRE. 2024. CWE-787: Out-of-bounds Write. https://cwe.mitre.org/data/definitions/787.html.

[91] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. 2009. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices* 44, 3 (2009), 265–276.

[92] Yeoul Na. 2023. -fbounds-safety. Enforcing bounds safety for production C code. *EuroLLVM Developers' Meeting* (11 May 2023). https://llvm.org/devmtg/2023-05/slides/TechnicalTalks-May11/01-Na-fbounds-safety.pdf

[93] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2015. Everything you want to know about pointer-based checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[94] 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 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 245–258. doi:10.1145/1542476.1542504

[95] Myoung Jin Nam. 2020. *Inline and Sideline Approaches for Low-Cost Memory Safety in C*. Ph. D. Dissertation. University of Cambridge, Cambridge, United Kingdom. doi:10.17863/CAM.64624

[96] Myoung Jin Nam. 2024. FRAMER/Miu: Tagged Pointer-based Capability and Fundamental Cost of Memory Safety & Coherence (Position Paper). *arXiv preprint arXiv:2408.15219* (2024).

[97] Myoung Jin Nam, Periklis Akritidis, and David J Greaves. 2019. FRAMER: A Tagged-Pointer Capability System with Memory Safety Applications. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 612–626. doi:10.1145/3359789.3359799

[98] National Vulnerability Database. 2014. CVE-2014-0160 Detail. https://nvd.nist.gov/vuln/detail/CVE-2014-0160.

[99] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.* 27, 3 (may 2005), 477–526. doi:10.1145/1065887.1065892

[100] Xiaofan Nie, Liwei Chen, and Gang Shi. 2022. PointerChecker: Tag-Based and Hardware-Assisted Memory Safety against Memory Corruption. In *2022 IEEE 31st Asian Test Symposium (ATS)*. IEEE, 96–101.

[101] NIST. 2017. Juliet C/C++ 1.3. https://samate.nist.gov/SARD/test-suites/112

[102] NIST National Vulnerability Database. 2022. CVE-2022-2566: Heap Out-of-Bounds Write in FFmpeg. https://nvd.nist.gov/vuln/detail/CVE-2022-2566.

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

[104] White House Office of the National Cyber Director (ONCD). 2024. *BACK TO THE BUILDING BLOCKS: A PATH TOWARD SECURE AND MEASURABLE SOFTWARE*. Technical Report. ONCD. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

[105] 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. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2018).

[106] Benjamin Orthen, Oliver Braunsdorf, Philipp Zieris, and Julian Horsch. 2024. SoftBound+ CETS revisited: More than a decade later. In *Proceedings of the 17th European Workshop on Systems Security*. 22–28.

[107] Phoronix Media. [n. d.]. Phoronix Test Suite. https://www.phoronix-test-suite.com/

[108] Filip Pizlo. 2022. All About Libpas, Phil's Super Fast Malloc. https://github.com/WebKit/WebKit/blob/main/Source/bmalloc/libpas/Documentation.md Accessed: 2025-03-12.

[109] qwattash. 2019. Subobject bounds on union member should not inherit bounds of union type. https://github.com/CTSRD-CHERI/llvm-project/issues/333.

[110] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. 2009. Precise garbage collection for C. In *Proceedings of the 2009 international symposium on Memory management*. 39–48.

[111] Ganesan Ramalingam. 1994. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 5 (1994), 1467–1471.

[112] Alex Rebert and Christoph Kern. 2024. *Secure by Design: Google's Perspective on Memory Safety*. Technical Report. Google Security Engineering.

[113] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping Safe Rust Safe with Galeed. In *Proceedings of the 37th Annual Computer Security Applications Conference* (Virtual Event, USA) *(ACSAC '21)*. Association for Computing Machinery, New York, NY, USA, 824–836. doi:10.1145/3485832.3485903

[114] Olatunji Ruwase and Monica S Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *NDSS*, Vol. 2004. 159–169.

[115] 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 (2022), 10:1–10:24. doi:10.1145/3495152

[116] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 745–762.

[117] Jiwon Seo, Junseung You, Donghyun Kwon, Yeongpil Cho, and Yunheung Paek. 2023. ZOMETAG: Zone-Based Memory Tagging for Fast, Deterministic Detection of Spatial Memory Violations on ARM. *IEEE Transactions on Information Forensics and Security* 18 (2023), 4915–4928. doi:10.1109/TIFS.2023.3299454

[118] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[119] Fermin J Serna. 2012. The info leak era on software exploitation. *Black Hat USA* 7 (2012).

[120] Zekun Shen and Brendan Dolan-Gavitt. 2020. HeapExpo: Pinpointing Promoted Pointers to Prevent Use-After-Free Vulnerabilities. In *Proceedings of the 36th Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 454–465. doi:10.1145/3427228.3427645

[121] Eugene H. Spafford. 1989. The internet worm program: an analysis. *SIGCOMM Comput. Commun. Rev.* 19, 1 (jan 1989), 17–57. doi:10.1145/66093.66095

[122] Yulei Sui, Ding Ye, Yu Su, and Jingling Xue. 2016. Eliminating Redundant Bounds Checks in Dynamic Buffer Overflow Detection Using Weakest Preconditions. *IEEE Transactions on Reliability* 65, 4 (2016), 1682–1699. doi:10.1109/TR.2016.2570538

[123] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. 2020. OAT: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1433–1449.

[124] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (SP)*. 48–62. doi:10.1109/SP.2013.13

[125] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W Keckler, and Mark Stephenson. 2023. CuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 124–147.

[126] Martin Unterguggenberger, David Schrammel, Lukas Lamster, Pascal Nasahl, and Stefan Mangard. 2023. Cryptographically Enforced Memory Safety. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). 889–903. doi:10.1145/3576915.3623138

[127] Erik van der Kouwe, Gernot Heiser, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2019. SoK: Benchmarking Flaws in Systems Security. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 310–325. doi:10.1109/EuroSP.2019.00031

[128] Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. 2018. Type-After-Type: Practical and Complete Type-Safe Memory Reuse. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 17–27. doi:10.1145/3274694.3274705

[129] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 405–419. doi:10.1145/3064176.3064211

[130] Jeff Vander Stoep and Chong Zhang. 2019. Queue the Hardening Enhancements. https://android-developers.googleblog.com/2019/05/queue-hardening-enhancements.html.

[131] VideoLAN. [n. d.]. x264. https://www.videolan.org/developers/x264.html

[132] Emanuel Q. Vintila, Philipp Zieris, and Julian Horsch. 2021. MESH: A Memory-Efficient Safe Heap for C/C++. In *Proceedings of the 16th International Conference on Availability, Reliability and Security* (Vienna, Austria) *(ARES '21)*. Association for Computing Machinery, New York, NY, USA, Article 16, 10 pages. doi:10.1145/3465481.3465760

[133] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. 2015. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 866–879.

[134] Haojie Wang, Jidong Zhai, Xiongchao Tang, Bowen Yu, Xiaosong Ma, and Wenguang Chen. 2018. Spindle: Informed memory access monitoring. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 561–574.

[135] Ruizhe Wang, Meng Xu, and N Asokan. 2024. SeMalloc: Semantics-Informed Memory Allocator. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 1375–1389.

[136] Xiaolei Wang, Bin Zhang, Chaojing Tang, and Long Zhang. 2024. Highly Comprehensive and Efficient Memory Safety Enforcement with Pointer Tagging. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 74–81.

[137] Robert NM Watson, Ben Laurie, and Alex Richardson. 2021. Assessing the viability of an open-source CHERI desktop software ecosystem. *Capabilities Limited, Sep* 17 (2021).

[138] Brian Wickman, Hong Hu, Insu Yun, Daehee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. 2021. Preventing Use-After-Free Attacks with Fast Forward Allocation. In *30th USENIX Security Symposium (USENIX Security 21)*. 2453–2470.

[139] Jonathan Woodruff and Paul Metzger. 2023. SOSP 2023 CHERI Exercises. Tutorial at ACM SIGOPS Symposium on Operating Systems Principles (SOSP). https://www.cl.cam.ac.uk/~pffm2/sosp2023_cheri_tutorial/

[140] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 457–468.

[141] 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*. 224–240.

[142] Shengjie Xu, Eric Liu, Wei Huang, and David Lie. 2023. MIFP: Selective Fat-Pointer Bounds Compression for Accurate Bounds Checking. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 609–622.

[143] Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. 2017. SIMBER: Eliminating redundant memory bound checks via statistical inference. In *ICT Systems Security and Privacy Protection: 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, May 29-31, 2017, Proceedings 32*. Springer, 413–426.

[144] Huaisheng Ye. 2021. Introduction to 5-level paging in 3rd gen intel xeon scalable processors with linux.

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

[146] 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* (Beijing, China) *(ASIACCS '10)*. Association for Computing Machinery, New York, NY, USA, 145–156. doi:10.1145/1755688.1755707

[147] Zheng Yu, Ganxiang Yang, and Xinyu Xing. 2024. ShadowBound: Efficient Heap Memory Protection Through Advanced Metadata Management and Customized Compiler Optimization. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 7177–7193. https://www.usenix.org/conference/usenixsecurity24/presentation/yu-zheng

[148] Yiyu Zhang, Tianyi Liu, Zewen Sun, Zhe Chen, Xuandong Li, and Zhiqiang Zuo. 2023. Catamaran: Low-Overhead Memory Safety Enforcement via Parallel Acceleration. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 816–828. doi:10.1145/3597926.3598098

[149] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. 2022. Debloating Address Sanitizer. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4345–4363. https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-yuchen

[150] 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 *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 916–929.

[151] Mohamed Tarek Ibn Ziad, Miguel A Arroyo, and Simha Sethumadhavan. 2020. SPAM: stateless permutation of application memory. *arXiv preprint arXiv:2007.13808* (2020).

A Detailed Performance Results on Additional Hardware

Table 7: Memory overhead on the Intel Core Ultra 9 285K.

	Peak RSS		.text Size	
Benchmark	Baseline (MB)	N00B	Baseline (KB)	N00B
600.perlbench_s	90.2	1.27×	2407.61	2.29×
602.gcc_s	3101.3	1.16×	9933.81	2.47×
605.mcf_s	3961.5	3.18×	19.44	2.35×
620.omnetpp_s	242.0	1.15×	1898.04	2.01×
623.xalancbmk_s	479.3	1.45×	4065.33	2.91×
625.x264_s	156.4	1.03×	634.05	2.33×
631.deepsjeng_s	6881.8	1.00×	77.07	1.66×
641.leela_s	25.6	1.24×	90.90	1.97×
657.xz_s	4201.7	1.15×	135.32	1.95×
619.lbm_s	3224.5	1.00×	14.50	2.33×
638.imagick_s	7007.5	1.20×	1510.52	1.89×
644.nab_s	558.5	1.31×	93.74	1.96×
SPECspeed 2017		**1.27×**		**2.15×**

Table 8: Memory overhead on the Ryzen 7 7800X3D.

	Peak RSS		.text Size	
Benchmark	Baseline (MB)	N00B	Baseline (KB)	N00B
400.perlbench	665.0	1.20×	1597.00	2.75×
401.bzip2	611.5	1.01×	220.96	2.59×
403.gcc	97.3	2.03×	4430.49	2.37×
429.mcf	1678.0	1.00×	21.35	2.42×
445.gobmk	31.0	1.14×	2643.34	1.90×
456.hmmer	5.2	1.83×	276.26	2.74×
458.sjeng	176.5	1.01×	162.97	1.85×
462.libquantum	98.0	9.31×	52.76	1.36×
464.h264ref	66.2	8.20×	1568.14	3.54×
471.omnetpp	172.0	1.22×	745.02	2.03×
473.astar	135.2	1.08×	43.64	1.80×
483.xalancbmk	420.4	1.53×	3781.40	2.56×
433.milc	681.2	5.99×	223.42	2.74×
444.namd	49.1	1.04×	783.35	2.43×
447.dealII	795.9	1.55×	1268.18	2.85×
450.soplex	424.7	1.64×	797.97	3.08×
453.povray	8.6	1.52×	1764.53	2.18×
470.lbm	410.9	1.00×	26.64	1.47×
482.sphinx3	44.6	1.61×	359.88	2.74×
SPEC CPU2006		**1.72×**		**2.32×**
600.perlbench_s	90.7	1.27×	2678.35	2.33×
602.gcc_s	3102.6	1.15×	11320.87	2.36×
605.mcf_s	3961.5	3.18×	53.97	2.60×
620.omnetpp_s	242.6	1.17×	2015.01	1.96×
623.xalancbmk_s	480.7	1.46×	5429.78	3.27×
625.x264_s	157.0	1.03×	1125.27	2.40×
631.deepsjeng_s	6881.9	1.00×	127.17	1.63×
641.leela_s	25.9	1.22×	185.33	1.88×
657.xz_s	4201.9	1.15×	215.99	2.20×
619.lbm_s	3224.7	1.00×	30.28	1.37×
638.imagick_s	7008.5	1.20×	3146.04	2.04×
644.nab_s	558.1	1.31×	379.82	1.81×
SPECspeed 2017		**1.27×**		**2.10×**

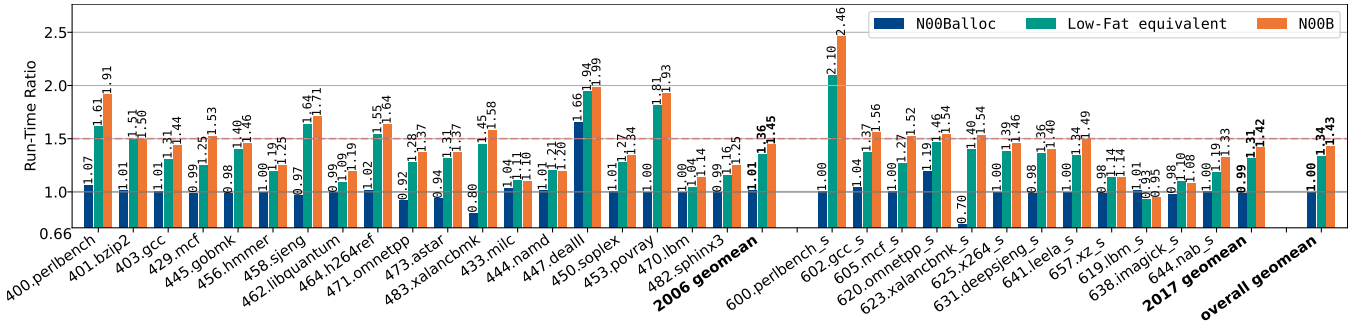


Figure 6: SPEC CPU2006 and SPECspeed 2017 run-time overhead results on the AMD Ryzen 7 PRO 4750G (AMD64-Zen2).

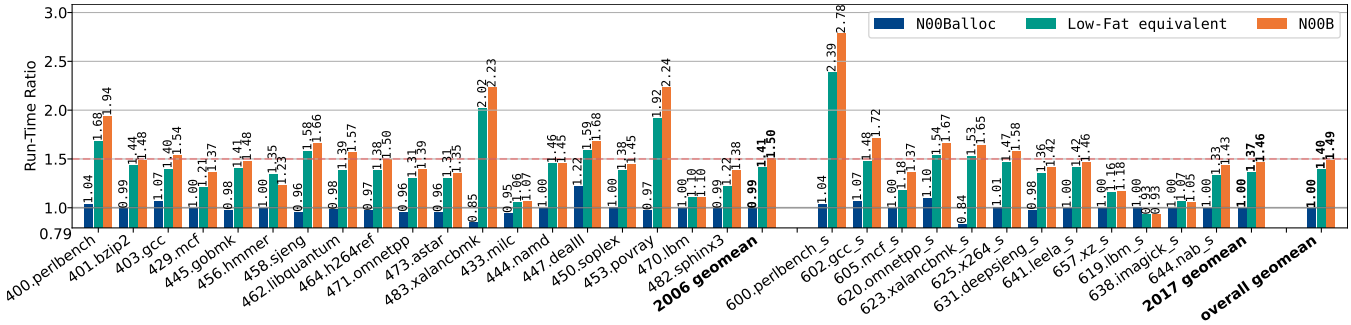


Figure 7: SPEC CPU2006 and SPECspeed 2017 run-time overhead results on the Intel Core i7-12700 (AMD64-Golden Cove).

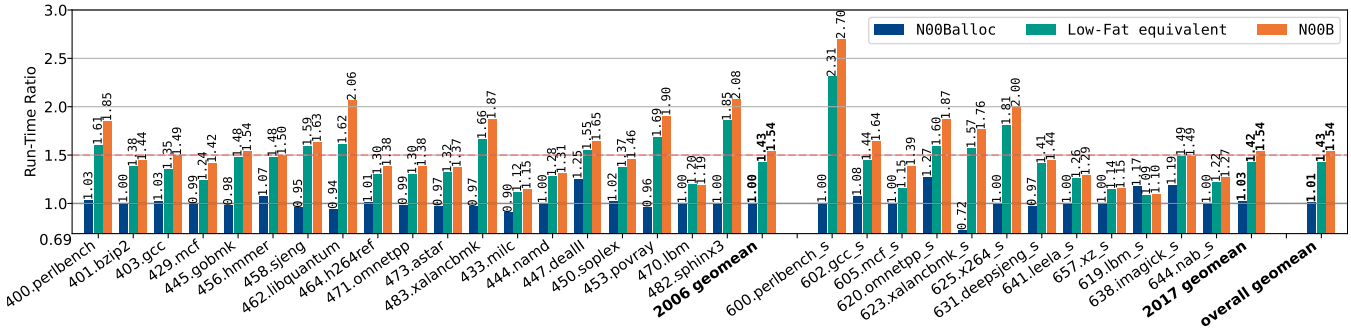


Figure 8: AMD Ryzen 7 7800X3D (AMD64-Zen4).

Table 9: Memory overhead on the Ryzen 7 PRO 4750G.

	Peak RSS		.text Size	
Benchmark	Baseline (MB)	N00B	Baseline (KB)	N00B
400.perlbench	664.4	1.20×	1212.62	2.47×
401.bzip2	611.1	1.01×	67.40	2.33×
403.gcc	96.3	2.03×	3619.68	2.29×
429.mcf	1677.7	1.00×	10.63	2.33×
445.gobmk	30.3	1.13×	2044.42	1.84×
456.hmmer	5.2	1.75×	166.32	2.11×
458.sjeng	176.4	1.01×	120.02	1.88×
462.libquantum	97.8	9.33×	22.62	1.90×
464.h264ref	65.0	8.30×	588.38	2.64×
471.omnetpp	171.7	1.22×	738.21	2.02×
473.astar	134.5	1.09×	33.88	1.78×
483.xalancbmk	419.8	1.53×	3473.33	2.38×
433.milc	680.9	5.99×	102.48	2.08×
444.namd	48.1	1.05×	221.06	2.01×
447.dealII	794.9	1.55×	511.93	2.22×
450.soplex	423.9	1.64×	319.74	2.65×
453.povray	7.3	1.66×	1218.13	1.87×
470.lbm	410.6	1.00×	16.35	2.54×
482.sphinx3	44.4	1.60×	162.97	2.12×
SPEC CPU2006		**1.73×**		**2.17×**
600.perlbench_s	90.0	1.27×	2398.35	2.17×
602.gcc_s	3101.5	1.15×	9880.80	2.25×
605.mcf_s	3961.3	3.18×	19.42	2.02×
620.omnetpp_s	241.5	1.17×	1886.27	1.90×
623.xalancbmk_s	479.2	1.46×	3970.94	2.56×
625.x264_s	156.2	1.03×	629.38	2.05×
631.deepsjeng_s	6880.9	1.00×	75.87	1.64×
641.leela_s	25.3	1.23×	93.26	1.82×
657.xz_s	4201.6	1.15×	133.63	1.88×
619.lbm_s	3224.3	1.00×	14.43	2.12×
638.imagick_s	7007.5	1.20×	1497.34	1.82×
644.nab_s	558.3	1.31×	93.08	1.79×
SPECspeed 2017		**1.27×**		**1.99×**

Table 10: Memory overhead on the i7-12700.

	Peak RSS		.text Size	
Benchmark	Baseline (MB)	N00B	Baseline (KB)	N00B
400.perlbench	663.8	1.20×	1258.11	2.54×
401.bzip2	611.3	1.01×	91.48	2.26×
403.gcc	96.3	2.03×	3692.95	2.32×
429.mcf	1677.7	1.00×	14.59	2.02×
445.gobmk	30.3	1.13×	2079.65	1.86×
456.hmmer	5.1	1.82×	178.58	2.26×
458.sjeng	176.4	1.01×	123.16	1.88×
462.libquantum	97.7	9.34×	25.18	1.89×
464.h264ref	65.2	8.29×	669.65	2.96×
471.omnetpp	172.0	1.22×	740.06	2.04×
473.astar	135.0	1.08×	34.70	1.82×
483.xalancbmk	420.0	1.53×	3525.65	2.42×
433.milc	681.0	5.99×	128.60	2.43×
444.namd	48.9	1.03×	274.37	2.17×
447.dealII	795.2	1.55×	580.32	2.35×
450.soplex	424.2	1.64×	361.85	2.77×
453.povray	8.1	1.52×	1247.45	1.93×
470.lbm	410.6	1.01×	17.22	1.77×
482.sphinx3	44.4	1.60×	178.53	2.23×
SPEC CPU2006		**1.72×**		**2.18×**
600.perlbench_s	90.1	1.27×	2440.74	2.20×
602.gcc_s	3101.7	1.15×	10029.29	2.28×
605.mcf_s	3961.2	3.18×	22.86	2.17×
620.omnetpp_s	242.0	1.17×	1899.63	1.92×
623.xalancbmk_s	479.5	1.46×	4154.21	2.72×
625.x264_s	156.3	1.03×	683.88	2.11×
631.deepsjeng_s	6881.7	1.00×	93.04	1.52×
641.leela_s	25.5	1.23×	97.11	1.84×
657.xz_s	4201.6	1.15×	145.48	1.90×
619.lbm_s	3224.3	1.00×	18.24	1.65×
638.imagick_s	7007.4	1.20×	1622.85	1.87×
644.nab_s	557.7	1.31×	104.98	2.02×
SPECspeed 2017		**1.27×**		**2.00×**

B Detailed Statistics of N00B's Static Analysis

Benchmark	Total	In-Bounds	Dominated	Loop-Bound	Unanalyzable
453.povray	62989	31446	7188	9987	6562
400.perlbench	74231	27659	8465	10904	8001
471.omnetpp	31822	14960	3013	5435	4746
482.sphinx3	6276	2039	698	2685	1742
473.astar	1573	1072	68	339	295
462.libquantum	464	354	15	77	51
470.lbm	357	52	37	263	3
483.xalancbmk	158303	23796	30468	31380	20211
433.milc	5362	1637	945	2404	683
429.mcf	559	174	54	320	199
403.gcc	190467	54763	22868	41981	36666
445.gobmk	27600	9644	1795	7173	4307
401.bzip2	4575	425	1728	2052	1378
444.namd	8603	1339	1206	5105	3795
456.hmmer	7442	2586	1121	3033	2034
458.sjeng	5048	2819	128	1148	817
450.soplex	19365	1728	3782	7816	4965
464.h264ref	42288	16223	2942	16783	12942
447.dealII	26869	8324	3237	11182	7976

Table 11: Static Memory Access Statistics on SPEC CPU2006, categorized by safety status. The columns represent (respectively): the total number of accesses, those provably in-bounds, those dominated by a bounds check, those not provably in-bounds and within loops, and those within loops that were not further summarizable.