Not Quite Write: On the Effectiveness of Store-Only Bounds Checking

Adriaan Jacobs and Stijn Volckaert, ***DistriNet, KU Leuven***

https://www.usenix.org/conference/woot24/presentation/jacobs

This paper is included in the Proceedings of the 18th USENIX WOOT Conference on Offensive Technologies.

August 12–13, 2024 • Philadelphia, PA, USA

ISBN 978-1-939133-43-4

Open access to the Proceedings of the 18th USENIX WOOT Conference on Offensive Technologies is sponsored by USENIX.

Not Quite Write: On the Effectiveness of Store-Only Bounds Checking

Adriaan Jacobs
DistriNet, KU Leuven

Stijn Volckaert
DistriNet, KU Leuven

Abstract

Compiler-based memory safety enforcement for unsafe C/C++ code has historically suffered from prohibitively high overhead. Despite regular advances in compiler optimization and increasing hardware resources and hardware support, most applications require too many checks to guarantee complete memory safety at an acceptable performance level. Consequently, researchers often propose relaxed policies where not all memory accesses undergo equally rigorous checking. One common suggestion is to omit pointer validity checks for memory loads. This omission significantly reduces the number of necessary checks and, thus, overhead. Moreover, it should *only* sacrifice the detection of pure information disclosure vulnerabilities through invalid reads, which are left unchecked.

This work challenges the perceived security benefits of store-only bounds checking. We show that invalid reads often suffice to take control of memory writes and bypass store-only validity checks. We empirically demonstrate the problem on SoftBound and qualitatively analyze the impact on a broad scope of other work. We also perform a large-scale evaluation on 1,000 popular C/C++ repositories and show that real-world code readily satisfies the necessary preconditions for store-only bypasses. Finally, we briefly discuss possible defenses and adaptations that let complete bounds checkers regain a part of the store-only overhead reduction potential without dramatically losing security.

1 Introduction

Memory-unsafe programming languages continue to dominate the composition of our modern software stack, from bootloaders, Operating System (OS) kernels, and system libraries to user-facing applications like web servers and browsers. Programs written in these languages often contain memory errors such as out-of-bounds (OOB) accesses [70, 72] or use-after-free bugs (UAF) [71], which can be exploited by attackers to leak or corrupt sensitive data, or to force the victim program to execute attacker-chosen code [102].

Owing to these security risks, government bodies [80] and industry leaders [89] are increasingly pushing for more memory safety in critical infrastructure and systems-level software, encouraging the use of safe languages instead, such as Rust [22]. Software vendors have already adopted these recommendations for new software projects [61, 99]. However, for a vast amount of already existing C and C++ code, translating it into safer languages is not feasible any time soon [86], leaving a mountain of unsafe code currently deployed in production environments for which no clear solution exists.

Researchers and practitioners from academia and industry alike have come up with many attempts to minimize the security impact of this unsafety through compiler transformations that automatically *harden* the code against memory error exploitation, e.g., by inserting checks on memory accesses or indirect control flow transfers. One such approach, which has been thoroughly investigated for decades [97, 102], is to retrofit memory safety into these languages by (semi-)automatically instrumenting memory accesses with run-time checks that validate pointer bounds (*spatial memory safety*) and object lifetimes (*temporal memory safety*) [6, 9, 12, 14, 18, 25–27, 30–32, 35, 36, 42, 49, 51, 54, 57–59, 63, 66, 74, 77–79, 90, 93, 94, 103, 116, 120, 121]. We broadly refer to these memory safety enforcement mechanisms as "bounds checkers" for short.

The design and implementation of bounds checkers has been a long-standing and highly active area of research, fueled by the promise of strong memory safety but plagued by prohibitive run-time overhead and compatibility issues. Despite steady advances over time, from optimizing the storage structure of bounds and lifetime metadata [18, 39, 41, 67, 68, 75, 78, 111], to avoiding the branch predictor pollution of typical compare-and-branch instrumentation [6, 18, 36, 58], or maximally reducing the number of redundant checks through static code analysis and optimization [12, 18, 44, 45, 52, 66, 100, 108, 112, 117, 122], the overhead of comprehensive memory safety enforcement remains well outside the stringent performance budget of typ-

ical production deployments [102]. For this reason, some prior work proposes to deliberately sacrifice *some* security coverage to reduce the performance impact by selectively eliding validity checks on memory accesses whose protection is explicitly scoped out [15, 63, 75, 77, 78], or whose performance impact is considered disproportional to their security benefit [38, 50, 107].

One commonly suggested strategy is to place validity checks on memory writes alone [36, 63, 75, 77, 78, 81, 94], which significantly reduces performance overhead, as most programs tend to read memory far more often than they write to it [65, 81, 84, 115]. Naturally, this comes at the cost of leaving pure information disclosure vulnerabilities out of scope. Major security crises like Heartbleed demonstrate that such confidentiality breaches are not necessarily of lesser impact [87]. Still, they only represent a minority of possible attacks, while their mitigation frequently requires more than double the amount of validity checks [78, 81]. Hence, *store-only* bounds checking is often heralded as a straightforward option to curb overhead while keeping the vast majority of memory vulnerabilities at bay by ensuring that attackers can never abuse invalid memory writes to corrupt program memory.

> Store-only checking, which allows read operations on out-of-bound locations and with dangling pointers, is sufficient to prevent all memory corruption-based security vulnerabilities.
>
> Nagarakatte et al. [77]

In this work, we argue that **the intended integrity assurance of store-only bounds checking does not hold in practice**, as a direct consequence of the lack of protection on memory reads. In short, the core issue is that store-only bounds checkers do not suffice to secure the data and pointer flow of the program, while their protection guarantees assume they do. As just one striking consequence of this, we show that attackers can corrupt arbitrary memory locations using *protected* writes by loading valid pointers through invalid memory reads. Our key finding is that the substituted, invalidly loaded pointer will always pass the store-only validity check, *regardless of the bounds checker design or implementation*. In summary, we make the following contributions:

- We outline four types of attacks that can corrupt memory under store-only bounds checking, including one that abuses protected writes.
- We empirically validate our attack on a SoftBound-hardened program [75], and qualitatively analyze the susceptibility of a representative selection of other work.
- We estimate the real-world feasibility of our attacks by analyzing a large corpus of open-source code.
- We reflect on the security assurances of store-only bounds checking and discuss possible improvements.

Listing 1: SoftBound's pointer propagation instrumentation, adapted from [75].

```
ptr = *some_loc; // pointer load
bounds = lookup(some_loc)->bounds;

*other_loc = ptr; // pointer store
lookup(other_loc)->bounds = bounds;
```

2 Background

Bounds checkers check if the pointers a program dereferences still point to their *intended referent* [51]. This intended referent is usually the object whose address the program initially derived the pointer value from. Two main bounds-checking approaches can guarantee complete memory safety.

Pointer-based approaches explicitly track the intended referent for each pointer as run-time information, either in a disjoint metadata structure [25, 75, 76, 79, 81], or encoded as part of the pointer itself (so-called fat pointers[1]) [9, 14, 49, 79, 109]. In the former case, the referent metadata is indexed using the address of pointers in memory, and the compiler explicitly instruments pointer copies in the program so they update the metadata. Listing 1 shows the way this explicit propagation happens in SoftBound [75].

Object-based approaches instead restrict pointer arithmetic such that the program can always recover the address of the original referent during memory accesses [26, 30, 51, 90]. Typically, this means not permitting a pointer to escape the original bounds of its allocation [30, 78]. The program associates safety metadata with every object's base address and inspects the currently-pointed-to object's metadata whenever it performs pointer arithmetic. Pointer propagation through memory requires no special handling, as the program can retrieve the allocation bounds based on the pointer value alone.

It is worth noting that many bounds checkers, especially recent ones [32, 58, 78, 93], do not perfectly fit either of these categories but instead appear more of a hybrid. For instance, Delta Pointers track the original referent per pointer through a relative distance metric in the pointer's unused top bits (Pointer-Based) but can only do so for a limited range of pointer arithmetic, after which the original referent is lost (Object-Based) [58].

Secondly, as mentioned in Section 1, developers do not always operate bounds checkers in their most secure, full-coverage mode due to overhead concerns. Instead, bounds checkers sometimes omit some checks, allowing developers to accept a limited security risk to improve run-time performance. For instance, Wagner et al. argue that the most frequently executed memory accesses are the best candidates for

[1] For brevity, we also include "diet" pointers that do not extend the native pointer width in this category.

bounds check elision [107], since they contribute to the overhead the most. Yet, the code that contains these accesses is likely the least bug-prone and best-tested code in the program, given its frequent execution.

A more popular way to deploy bounds checkers selectively is to restrict the checks to memory writes alone [63, 75, 77, 78, 81, 94], or to memory accesses that are permitted to access a certain amount of sensitive data [3, 15, 60, 101], or only certain regions of memory, e.g., the heap [30, 35, 45, 66]. The overhead reduction factor naturally depends on the amount of memory accesses that are left unchecked.

Store-only bounds checkers frequently report reduced overheads by a factor 2 or more [63, 75], while preventing all out-of-bounds or dangling pointer writes. Prior work has presented this as an attractive performance-security tradeoff [77, 81], primarily due to the ease of converting any bounds checker design to a store-only working mode. Hence, although not all published memory safety enforcement work includes dedicated discussions and benchmarks of store-only operating modes, the prevailing notion seems that any bounds checker can readily be operated in a store-only mode when performance requirements dictate so, with limited security impact.

3 Risks of Store-Only Bounds Checking

The central thesis of this paper is that by leaving memory reads uninstrumented and freely exploitable, store-only bounds checkers give up much more security guarantees than "merely" the detection of pure information disclosure vulnerabilities such as Heartbleed [33, 87]. In this section, we describe several *additional* vulnerabilities and attack vectors spawned by the lack of protection on memory reads. In particular, we show that attackers can still arbitrarily corrupt memory *despite* passing all store-only validity checks.

3.1 Threat Model and Assumptions

Throughout this paper, we assume that (i) the program contains exploitable memory reads (e.g., out-of-bounds accesses or reads through dangling pointers), and (ii) the program uses a bounds checker of any type (i.e., pointer- or object-based, or a combination of both) to protect its memory writes. As we aim to break the intended integrity assurance of the store-only working mode, we do not rely on sub-object overflows [34] or vulnerabilities in external code or unprotected memory regions since prior work usually considers such vulnerabilities out of scope [30, 35, 45, 66]. We also assume that the attacker knows the details of the deployed store-only hardening and will adapt the attack to its design and implementation characteristics. Finally, as repeatedly demonstrated by previous work [73, 95, 98], we assume that any Address Space Layout Randomization (ASLR) [83] can readily be bypassed through information disclosure as a result of invalid memory reads.

3.2 Invalid Pointer Loads

A first, highly impactful security issue with store-only bounds checking appears when the program loads pointers from memory through exploitable memory reads such as the one shown in Listing 2. Attackers that can control the read on line 3 can choose which pointer to load from memory and, thus, which pointer gets dereferenced in the later memory write. Crucial here is that, as long as the loaded pointer points to a valid live object, the memory write will *always pass the store-only validity check* on line 5. The fundamental problem is that omitting the validity check for the memory read allows attackers to load a pointer value illegitimately, yet ensures that the pointer has valid bounds information when the program performs the store validity check. This is true even if the loaded pointer propagates through an arbitrary number of assignment statements before it reaches the final store instruction because the bounds checker will propagate the pointer metadata along the way if necessary.

Listing 2: A vulnerable code pattern under store-only hardening, with SoftBound instrumentation in red.

```
1 // exploitable pointer load
2 ptr = array[i];
3 bounds = lookup(&array[i])->bounds;
4 // ...
5 assert_in_bounds(ptr, bounds);
6 *ptr = ...;
```

Taking SoftBound as an example, Listing 2 shows that the bounds of the pointer at the `&array[i]` memory location are loaded and then checked against the value of the loaded pointer itself. Given control over `i`, attackers can choose which pointer is loaded, and due to the dynamic bounds propagation, SoftBound will look up the correct bounds associated with the accessed memory location. We stress that this is not a design or implementation issue with SoftBound; these are the intended bounds propagation rules for any bounds checker, regardless of object- or pointer-orientation. In Section 6, we describe the same issues against other types of bounds checkers.

To exploit this issue in practice, attackers must procure a valid pointer in the program to use as a substitute for (one of the) intended pointer values. Operating a bounds checker in store-only mode dramatically facilitates the search for these valid pointers since attackers can freely disclose large swaths of application memory through invalid reads, explicitly permitted by the threat model of these bounds checkers [77, 81]. Even without such capabilities, and depending on the type of victim application, offline analysis on a local binary may be sufficient to find useful pointers near the exploitable memory read location. Such an attack would not even require defeating ASLR in the first place, as a form of "Position-Independent

Address Reuse" [37].

Alternatively, attackers can *craft* valid pointers and inject them in attacker-controlled memory regions as part of the payload. This crafting option gives attackers even greater flexibility to meet the constraints of the invalid memory load. As far as we are aware, only pointer-based bounds checkers that maintain disjoint metadata keyed on pointer addresses, e.g., SoftBound [75], may be able to reject such crafted pointers during the store-only validity check, because the disjoint metadata will only contain entries for addresses of existing, valid pointers in the program. Any crafted pointers will not have corresponding entries in the metadata, and, as such, fail the metadata lookup itself. We note that modern pointer-based bounds checkers rarely use disjoint metadata, as it hurts cache locality [2, 81], can be a concurrency bottleneck in multi-threaded programs, and leads to compatibility issues when the bounds checker cannot reliably instrument all pointer copies that should update the metadata, e.g., in external code [42]. Hence, most modern bounds checkers [63, 78, 81] fail to detect the use of attacker-crafted pointers.

3.3 Arbitrary Code Execution Without Memory Corruption

No amount of validity checks on memory writes can help prevent exploits that solely use invalid memory reads. Existing store-only bounds checkers explicitly consider this in the case of information disclosure vulnerabilities [63, 75, 77, 78, 81, 94], but overlook the broader implications of memory-unsafe information flow. Consider the below snippet:

```
func = array[i];
func(args);
```

As previous work also noted [60, 102], code patterns such as the above allow attackers to substitute `func` for other code pointers, including crafted ones, by merely abusing a single memory read. Such invalid function pointer reads suggest that developers should *at least* complement store-only bounds checks with defenses like Control Flow Integrity (CFI) [1]. In the original Code Pointer Integrity (CPI) paper [60], the protection against invalid code pointer reads is the precise difference between CPI and its less secure Code Pointer Separation (CPS) variant.

> Store-only checking provides much better safety than control-flow integrity with similar performance overheads.
>
> Nagarakatte et al. [77]

Interestingly, SoftBound also associates metadata with function pointers [75], much like data pointers, and checks on indirect calls whether the called address has a corresponding metadata entry. As acknowledged by the authors, any valid function pointer can still be substituted, enabling expressive Whole-Function Reuse (WFR) [88, 92, 104]. More modern bounds checkers typically do not include any checks on indirect branches at all since the memory safety offered by the bounds checking itself should suffice to stop the initial memory error leading to a code pointer overwrite.

3.4 Invalidly Loading Non-Pointer Data

Further generalizing the implications of memory-unsafe information flow, attackers can also abuse invalid memory reads to load plain, non-pointer data from an attacker-controlled source. These invalid reads include pure information disclosure vulnerabilities like Heartbleed. However, they can also be used to create a *write-what* primitive where there previously existed none, as shown below:

```
1 int adminLvl = dangling_ptr->lvl;
2 if (adminLvl > 2)
3     system("/bin/bash");
4 globalAdminLvl = adminLvl;
```

The use-after-free vulnerability on line 1 allows attackers to take control of the value of the `adminLvl` variable following the invalid load, typically by placing payload data at the `dangling_ptr` location. Because that memory load is left unchecked under store-only bounds checking, this snippet allows attackers to control a privilege flag without corrupting it, solely through an invalid read. In this case, the attack results in a privilege escalation. Note how this attack allows attackers to overwrite memory, e.g., the `globalAdminLvl` on line 4. Such an overwrite is entirely memory-safe.

3.5 Breaching Pointer Confidentiality

Some bounds checkers embed metadata in pointers (e.g., by writing a key tag into their top bits) but, for the sake of compatibility, still allow the program to perform arbitrary pointer arithmetic [42, 62, 63, 91, 105]. Unconstrained, this pointer arithmetic could overwrite the metadata. Any such design implicitly introduces a confidentiality requirement on pointer values. Consider the below snippet:

```
int* adminLvl = ...;
ptr = &array[i];
*ptr = ...;
```

If attackers can leak the `adminLvl` pointer value and the base address of the `array`, they can fill the difference between both in as `i`. The resulting `ptr` will then be equal to `array+(adminLvl-array) = adminLvl`, which will be a valid pointer to dereference, including all the necessary in-pointer metadata.

To defend against this type of attack, affected bounds checkers enforce the confidentiality of pointer values by checking memory reads to prevent information disclosure. In contrast, a

store-only deployment explicitly breaches this confidentiality by eliding checks on memory reads, massively exacerbating the applicability of this attack.

> No matter how tempting it may sound to protect only writes, one must remember that buffer-overread vulnerabilities will slip away from such low-overhead checking.
>
> Oleksenko et al. [81]

With the advent of low-latency cryptographic block ciphers in commodity hardware [10, 11, 56], we notice a growing trend towards such in-pointer metadata designs without pointer arithmetic restrictions [42, 62, 63, 105]. We want to stress that, even with full and store protection, these schemes still struggle to guarantee pointer confidentiality when the program is prone to sub-object overflows [34], or when it inadvertently leaks pointer values without violating memory safety. Concurrent work [40, 46] already exploits this precise weakness of the C^3 defense [62]. On top of this, store-only checking grants attackers reliable access to confidential pointer values via information disclosure, thus presenting a clear security incompatibility with this emerging trend in low-overhead bounds checker design.

4 Ubiquity in Real-World Code

To assess whether existing code contains the necessary patterns to enable our store-only bypass techniques, we conducted an evaluation of the 1,000 most-starred C and C++ GitHub repositories. We tried to automatically identify the generic vulnerable patterns described in Section 3 using custom CodeQL[2] queries. We excluded two patterns from this search. We did not search for invalid loads of non-pointer data (Section 3.4), since its exploitable use, e.g., bypassing a privilege check, is highly application-specific and hard to infer automatically for a broad range of software. In addition, we also disregarded pointer arithmetic sites that are prone to the attack we described in Section 3.5 since we have no way of realistically estimating attacker control over the pointer offset.

Instead, we looked for loads of pointers that are later dereferenced in a memory write (Section 3.2), or called indirectly (Section 3.3). We excluded patterns where the load operation was obviously safe (e.g., direct loads from a scalar local variable). Instead, we focused on patterns (specifically on reads from arrays), of which we assume a substantial portion are exploitable. We then evaluated how many of them suit the requirements of store-only bypasses. This selection targets a large class of spatial C and C++ vulnerabilities but may miss potential Use-After-Free (UAF) issues, which can also appear without any indexing operations. However, these temporal safety issues are much harder to distinguish from obviously-safe pointer loads statically.

We match every pattern that contains direct data flow from a loaded pointer value to the pointer operand of a memory write (*unsafe data pointer loads*) or an indirect call (*unsafe funcptr loads*). Figure 1 shows that the former pattern occurs broadly across the entire suite of evaluated repositories. In addition, many repositories have frequent occurrences, e.g., 1,000 or more for over half of the evaluated programs. In contrast, the function pointer load pattern occurs less frequently, in large part because indirect calls occur less frequently than memory accesses. Hence, the store-only bypass based on invalidly loaded data pointers significantly increases the attacker's options when facing a store-only bounds-checked program.

5 Assurances of Store-Only Bounds Checking

Given the store-only bounds-checking risks we describe in Section 3, one may ask whether the utility of store-only bounds checking is defeated entirely. In this section, we analyze the expressiveness of the arbitrary write primitive granted through our store-only bypass, and discuss cases where store-only bounds checking is still useful.

After gaining some control over the target object of the memory write, attackers can corrupt address data to bootstrap a more powerful primitive [43], or corrupt key data structures directly, e.g., security-sensitive configuration data [16], syscall arguments [43], or syscall-guard variables [113]. Alternatively, attackers may seek arbitrary code execution by corrupting a code pointer in the program [13, 82, 85]. Most of these are already accessible through valid pointers in the program, so attackers can disclose the target corruption address more easily and obtain valid pointers to bypass the store-only validity checks. However, some objects never appear as valid overwrite targets in the bounds-checking metadata because no instrumented write should ever be able to target them. We describe a few examples here.

Return Addresses Overwriting return addresses can be difficult under store-only hardening since they are not part of any live object. In addition, some bounds checkers "heapify" [79] stack allocations to better control their memory layout [29], or to simplify instrumentation. This effectively leaves the return addresses on a safe stack [60], of which the location may be harder to disclose, and, in turn, complicates the task of crafting valid pointers. However, we find no such restrictions for the corruption of function pointers, i.e., forward-edge control flow hijacking, which is equally expressive [13].

Bounds Metadata An attractive option for adversaries looking to bootstrap an initial store-only bypass into a more expressive primitive may be to target the bounds or lifetime metadata itself. Once again, however, no pointers will naturally occur in the program for which any bounds checker

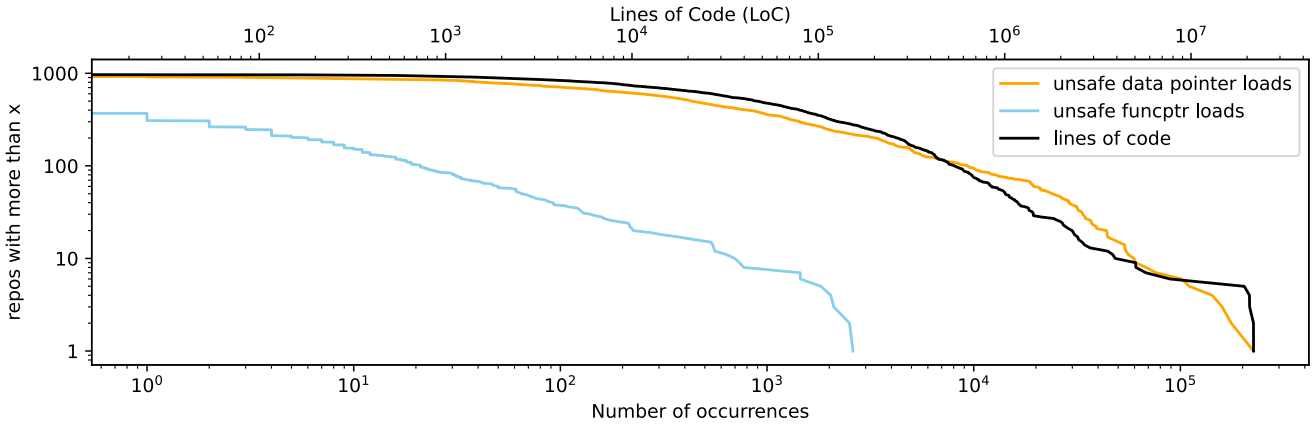

Figure 1: Occurrences of the enabling code patterns in the 1,000 most popular C/C++ GitHub repositories. The top axis counts the black graph, representing the distribution of the lines of code of each repository. The bottom axis counts the yellow and blue graphs, which show the distribution of unsafe data pointer loads, and unsafe function ptr loads respectively.

metadata is a valid target. Some defenses, e.g., CryptSan [42] and Mid-Fat Pointers [57], even include dedicated Software Fault Isolation (SFI) to explicitly outlaw invalid accesses to the metadata, as a defense-in-depth measure.

Safe Objects As a way to reduce run-time overhead [29, 106] or to provide isolation [44, 45, 60, 64], defenses statically identify objects that can never be the source of memory errors, because they are provably always accessed within their bounds. Bounds checkers do so too, e.g., to avoid heapify-ing many stack objects for performance reasons [29]. Similar to return addresses, this leaves them separated on a safer stack, which no bounds-checked memory writes can target.

For each of these cases, we notice a large difference between centralized/disjoint and decentralized/inline metadata. For a typical fat pointer approach, e.g., Austin et al.'s [9], attackers can craft fat pointers with any bounds attached to them, including those permitting access to the stack, bounds table metadata, or any other illegitimate targets, e.g., the Global Offset Table (GOT) [118]. After disclosing the address of the target object, such approaches pave the way for expressive exploitation.

On the other hand, many bounds checkers contain at least some metadata that is not kept inline with the pointer, and thus hinders straightforward crafting. Among "diet pointer" schemes, i.e., those that do not extend the native pointer width, many include a small *metadata key* as part of the pointer [18, 63, 78, 79, 91, 110], which can be used to retrieve complete metadata information during validity checks. These make it harder to craft arbitrary pointers to illegitimate targets, since it may require crafting a metadata entry, too. If all metadata is stored in a centralized, disjoint location [63, 91], this is near impossible using store-only hardened writes alone. Alternatively, metadata can be stored inline with the objects too, e.g., as allocation headers or footers [18, 78, 79]. Adversaries must then be able to craft the metadata in the expected location, typically near the target object, to accompany the crafted pointer. For some illegitimate targets, e.g., return addresses, this can still be feasible when there are enough attacker-controlled regions available nearby.

Note that our discussion in this section primarily concerns *illegitimate* corruption targets, such as return addresses, to which the bounds checker will never create any valid pointers, as they are not supposed to be overwritten by application-level memory writes. All other corruption targets, e.g., function pointers, access control data, configuration data, etc., can generally be targeted through hardened memory writes using both crafted and reused pointers. In addition, store-only security risks that do not depend on invalid memory writes are not affected by any limitations of the store-only bypass primitive. For instance, loading attacker-chosen function pointers enables expressive control flow hijacking, with which these "illegitimate" targets can still be corrupted.

6 Analysis of Existing Store-Only Bounds Checkers

We reviewed several prominent bounds checkers that include a store-only mode and analyzed their susceptibility to the security risks we identified in Section 3. We summarize our findings in this section. Table 1 shows the condensed results, with the properties of each evaluated defense, and the bypass expressiveness it grants.

Property	SoftBound [75]	FRAMER [78]	PACMem [63]	Intel MPX [81]
Hardware	None	None	Commodity	Commodity
Type	Pointer-based	Object-based	Pointer-based	Pointer-based
Per-Pointer Metadata	Disjoint	In-pointer	In-pointer	Disjoint
Per-Object Metadata	None	Inline	Disjoint	None
Pointer Reuse	✓	✓	✓	✓
Pointer Crafting	✗	✓	✓	✓
Illegitimate Targets	✗	✗	✗	✓

Table 1: Comparison of selected bounds checkers that offer a store-only working mode. We highlight their respective design properties and the expressiveness of the store-only bypass technique under each.

Listing 3: Vulnerable program.

```
1  int* adminLvl = ...; // *adminLvl = 0
2  struct user {
3      int age;
4  }* users[NUM_USERS] = ...;
5
6  id = input_user();
7  age = input_user();
8  // exploitable memory read
9  struct user* user = users[id];
10 // checked memory write
11 user->age = age;
12
13 if (*adminLevel > 2) {
14     printf("Shell for admin: \n");
15     system("/bin/bash");
16 }
```

SoftBound [75] SoftBound is one of the most well-known spatial memory safety defenses in academic literature, with much derivative work reusing or extending its techniques [15, 100, 101]. The basic design is pointer-based with disjoint, centralized metadata. A large table, indexed by the storage locations of program pointers, contains information about the bounds of their intended referents. When pointers move around in memory, SoftBound updates the metadata to move around with them, and when they are loaded from memory, their bounds metadata is, too. This propagation mechanism allows SoftBound to check every pointer against the bounds of its intended referent on memory accesses without constraining or checking pointer arithmetic.

SoftBound, and the later review article by the same authors [77], includes an evaluation of a store-only working mode that reduces run-time overhead by a factor of 2 or more. Using SoftBounds' open-source prototype [23], we empirically validated our store-only bounds check bypass on a manually written vulnerable program, shown in Listing 3. On line 9, attackers can use the `id` variable to control the loaded pointer from the `users` array. After defeating ASLR and disclosing the addresses of the objects involved, attackers can load the `adminLvl` pointer on line 9 by out-of-bounds indexing the `users` array, such that the bounds-checked write on line 11 overwrites the admin level, leading to privilege escalation in this case. We modeled this example after the IE God Mode bug [7], where a single variable controlled the privilege level of *VBScript* code executing in a sandbox.

We confirmed that we were able to successfully exploit the native program, without any hardening applied, by passing it the correct offset value for `id`, e.g., `&adminLvl - &users`, and supplying an `age` larger than 2. When we repeated this experiment on a fully hardened program version, SoftBound successfully detected the exploitation at the initial out-of-bounds memory read on line 9. We then turned off checks on memory reads and were able to exploit the program again, using the same technique as with the native program. During the memory read, SoftBound looked up the bounds associated with the actually-accessed memory location, i.e., `&adminLvl`, and enforced those at line 11. Naturally, these bounds were valid for the memory write to `*adminLvl`.

FRAMER [78] FRAMER is a spatial-only bounds checker which implements a mostly object-based design. Small in-pointer metadata keys track the location of per-object bounds information, which is typically located close to the object. FRAMER restricts pointer arithmetic to preserve the metadata key and, thus, to remember the intended referent at all times. FRAMER also supports a store-only working mode, which incurs less than a third of the performance overhead of its full instrumentation version.

As pointers store metadata keys in the unused top bits, they contain all the necessary information to pass the validity check. Naturally, pointer reuse is possible here to obtain valid substitute pointers, like in the previous SoftBound exploitation example. In addition, attackers can trivially craft pointers with arbitrary metadata keys in the upper bits. The possibility of pointer crafting makes FRAMER even more suitable for store-only attack bypasses than SoftBound.

PACMem [63] PACMem uses ARM's Pointer Authentication (PA) feature [8] to bind pointers to their disjoint per-object metadata entries cryptographically. During allocation, PACMem generates a Pointer Authentication Code (PAC) based on the object's full validity metadata (base pointer, allocation size, and a randomly generated temporal identifier called a "birthmark") and places it into the top bits of the pointer. PACMem also stores per-object validity metadata in a linear table, indexed by the PAC of pointers during memory accesses. If the PAC does not match the looked-up validity metadata, PACMem knows the pointer is no longer tracking its intended referent, either due to out-of-bounds indexing or due to an intervening deallocation.

The authors also evaluate PACMem in a store-only working mode, which more than halves run-time overhead. From a store-only bypass perspective, PACMem behaves very similarly to FRAMER. The PAC is essentially a metadata key, protected from corruption through cryptographic integrity checks, for which FRAMER uses pointer arithmetic checks instead. In both cases, the metadata keys are revealed when the attacker can leak memory contents, and attackers can craft pointers using any metadata key to grant access to any bounds stored in the metadata table. Hence, this design permits both crafting and reuse to obtain valid pointers. To reiterate, what we describe as *pointer crafting* still requires the disclosure of authenticated pointers to the target object first to craft an identical copy in a different place. However, this is entirely in the scope of the store-only threat model, as mentioned in Section 3.1.

Finally, PACMem is the only one out of our evaluated schemes that suffers from the breach of its implicit pointer confidentiality. As discussed in Section 3.5, pointers contain their own metadata tags, and PACMem permits arbitrary pointer arithmetic. Hence, leaking two PACMem pointers and computing their offset gives attackers an index value with which they can construct pointer A from pointer B and vice versa.

Intel MPX [81] The now-deprecated Intel Memory Protection Extensions (MPX) were a hardware feature of select Intel CPU microarchitectures that included dedicated bounds registers, as well as bounds checking and management instructions that provided generic hardware acceleration for pointer-based bounds checking schemes [120]. Several papers additionally explored using MPX as a fast, coarse-grained intra-process isolation mechanism [15, 55, 60], for which it was arguably better suited.

MPX has architectural support for a centralized in-memory metadata structure that contains bounds entries for the location of every pointer in the program. In that regard, typical MPX-accelerated bounds checkers, such as those implemented by the GCC and ICC toolchains in the past [19, 81], are very similar to SoftBound, which itself was inspired by a hardware implementation of the same idea [25]. Indeed, Oleksenko et al. analyzed the performance characteristics of Intel MPX when used for its intended bounds checking purpose [81], and included a performance comparison with, among others, SoftBound. They also evaluated a store-only working mode of such an MPX-based bounds checker and found that it reduces the performance overhead by a factor of 2.

A key difference between MPX's design and SoftBound is that MPX redundantly stores the pointer's value in the bounds entry that describes its intended referent. The goal is to allow the detection of external uninstrumented code that overwrites pointers in memory without updating their associated metadata entries, e.g., by re-assigning it to a different object. During loads of pointers, using the `BNDLDX` instruction [47], the processor checks whether the bounds table entry is present and holds a pointer value that matches its disjointly stored copy as a way to verify whether the metadata is still up to date. If it is not, MPX can take one of two implementation-defined actions. On the one hand, MPX can update the bounds table entry to cover the entire address space [47], i.e., the loaded pointer value can point to any object in the program, as a security concession that prioritizes compatibility with external code [77]. This compatibility mechanism allows MPX to gracefully handle calls to uninstrumented libraries, dynamically *unbounding* pointers when external code changes them instead of terminating the program. On the other hand, MPX can simply terminate the program. This latter option prioritizes security over compatibility.

From a store-only bypass perspective, the aforementioned compatibility option makes *pointer crafting* much easier than it is with SoftBound, which strictly distinguishes between valid pointer-holding locations and non-pointer data (cfr. Section 3.2). In its store-only mode, MPX would then graciously interpret any attacker-crafted pointer as a valid pointer for any object in the program, which bypasses previous pointer crafting limitations with SoftBound, yielding the single most expressive store-only bypass primitive we have observed in our review of the literature.

7 Discussion & Related Work

Until now, we described several attack vectors against store-only bounds checkers that go beyond information disclosure, in the hope of recalibrating the community's expectations about the security guarantees of such defenses. In this section, we take a broader look at other types of store-only hardening, and the impact of our findings on other areas of memory safety enforcement.

Write Integrity Testing (WIT) [5] WIT is a notable memory safety hardening that solely provides store-only validity checks. However, its enforcement mechanism fundamentally differs from that of bounds checkers. At compile time,

WIT assigns the same *color* to all memory writes that may alias. This creates disjoint alias sets [53], each identified with a unique color, that hold all objects in the points-to sets of the aliasing writes. At run time, WIT tags each object with the color of its alias set, and queries the color of the actually-accessed object on memory writes. WIT's store-only validity check verifies that the looked-up color matches the statically-assigned color of the write. This validity check ensures that the actually-accessed object is within the statically-computed points-to set of the memory write. Within that set, the memory write can corrupt all objects. Naturally, this permits clear memory safety violations, and has been regarded as strictly *weaker* than precise bounds checking for that reason. However, because WIT establishes the set of accessible objects statically, its validity checks cannot be fooled by our store-only bypass. Contrary to bounds checkers, WIT does not propagate any bounds or metadata information dynamically. As such, its security guarantees are not affected by any memory-unsafety from which the memory write operand originates; the same, statically-determined set of objects will be enforced regardless. WIT shows increased resilience over bounds checkers in the face of arbitrary memory reads, which makes it more suitable as a store-only hardening mechanism.

Impact on Static Analysis Bounds checkers typically include a range of compiler optimizations to suppress overhead [12, 18, 44, 45, 52, 66, 100, 108, 112, 117, 122]. A popular optimization is to check whether pointer operands of memory accesses are always in bounds of any object they could refer to [5]; if so, they are provably safe and do not require a dynamic bounds check. This in-bounds analysis typically requires statically tracing pointers backward to determine their origin, accumulating any offsets they garner along the way. Many pointers are loaded from memory eventually (Section 4), at which point thorough analyses perform a Reaching Definitions Analysis (RDA) [4] to determine the possible values of the loaded pointer. The in-bounds analysis can then continue investigating all these possible loaded pointer values. If all possible loaded values are in bounds, the analysis will consider the original memory access as safe, and leave it uninstrumented.

Again, a problem appears when the loaded pointer value originates from an exploitable memory read. Attackers can invalidly load a different pointer, and, due to the optimization, there will not even be a bounds check left to bypass. The underlying problem here is that many static analyses do not account for the memory-unsafety of C and C++ [69], but are still used to prove its safety properties. To avoid this specific issue, we recommend only performing RDA on memory loads which themselves are also provably in bounds.

Store-Only Testing In this paper, we have primarily discussed the weaknesses of bounds checkers as exploit mitigations, facing a sophisticated adversary that is motivated to break the program's protection through any means necessary. However, some bounds checkers simply aim to catch memory safety violations that are triggered during development or (fuzz) testing [17, 36, 67, 94]. The latter are commonly referred to as "sanitizers" [97], and tend to use less secure methods of catching memory errors, that nevertheless detect violations more precisely, e.g., at object bounds instead of allocation bounds [28]. Performance can still be important here, e.g., to improve throughput during automated fuzz testing [36, 48, 119, 122]. Indeed, the original AddressSanitizer (ASan) paper, now integrated into popular compilers [20, 21], included an evaluation of a "writes-only" instrumentation mode, which reduced the run-time overhead threefold. However, since ASan is not meant to run in production, despite a stint in the Tor browser [24], the impact of our attack is limited. Still, our work undermines the assumption that when a program is thoroughly sanitized/fuzzed for invalid write bugs, attackers will not be able to corrupt program memory or achieve arbitrary code execution.

Selective Bounds Checking Apart from store-only deployments, researchers have also proposed using bounds checkers to protect only a security-critical, sensitive part of the data space [3, 15, 60, 101]. These defenses generally include a coarse-grained isolation mechanism in the non-sensitive part to prevent access to the sensitive part, e.g. using SFI [114] or Intel MPK [47]. Typically, a pointer analysis determines which memory accesses are allowed to access the sensitive region and which are not. Depending on the way the analysis computes sensitivity, we believe that such selective bounds checkers carry a similar vulnerability to their store-only siblings. Consider the snippet below:

```
1 ptrToSens = nonSensArray[i];
2 *ptrToSens = ...;
```

The `nonSensArray` is non-sensitive, and it contains non-sensitive pointers to sensitive objects. The load from the array on line 1 is only instrumented with coarse-grained bounds checks, since the pointer analysis correctly determined that it accesses a non-sensitive object (`nonSensArray`). The store on line 2 is bounds checked in a fine-grained way, since it is supposed to access sensitive data. When the load on 1 is exploitable, however, attackers can load any valid pointer to the sensitive region from the non-sensitive region, which will pass the validity check on line 2, in true store-only bypass fashion. Hence, attackers can choose which sensitive object gets written to on line 1, by abusing a memory error they were permitted to exploit (coarse-grained bounds check). Note that we bypass two layers of defense-in-depth at once here: attackers are not supposed to write to the sensitive region (inter-sensitive isolation), and sensitive memory accesses are not supposed to be exploitable, because they are bounds checked (intra-sensitive isolation).

One option to address this issue is to include pointers to sensitive objects in the sensitive region as well [15, 96], recursively. However, this can quickly lead to a very large sensitive region, with almost all memory accesses instrumented, and the associated performance overhead.

8 Conclusion

In this work, we uncovered fundamental weaknesses of store-only bounds checking, directly caused by the lack of protection on memory reads. In particular, we demonstrated that invalid loads of pointers give attackers control over *hardened* memory writes. We empirically validated our attack on a prominent bounds checker prototype, and characterized the same weakness in other bounds checker designs. Through automated code analysis, we showed that a large corpus of real software exhibits the vulnerable patterns that enable our store-only bypass.

Looking ahead, we discussed potential avenues to rebalance the security and overhead advantages of store-only hardening. To this end, we recognized the resilience of Data Flow Integrity (DFI) against malicious pointer loads. Given the broader importance of efficient memory safety enforcement, we encourage new research into store-only hardening, keeping in mind the subversive effects of attacker-controlled memory loads.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback. In addition, we thank Silviu Vlasceanu and Mahmoud Ammar from Huawei Trusted System Security Lab Munich for the interesting conversations that led to this work, and Dairo de Ruck for providing access to much-needed computation resources. This research is partially funded by the Research Fund KU Leuven, and by the Cybersecurity Research Program Flanders.

Availability

Our attack experiments and code analysis queries are available at https://github.com/ku-leuven-msec/not-quite-write-experiments.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595932267. doi: 10.1145/1102120.1102165.

[2] Masab Ahmad, Syed Kamran Haider, Farrukh Hijaz, Marten van Dijk, and Omer Khan. Exploring the performance implications of memory safety primitives in many-core processors executing multi-threaded workloads. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334839. doi: 10.1145/2768566.2768572. URL https://doi.org/10.1145/2768566.2768572.

[3] Salman Ahmed, Hans Liljestrand, Hani Jamjoom, Matthew Hicks, N. Asokan, and Danfeng (Daphne) Yao. Not all data are created equal: Data and pointer prioritization for scalable protection against Data-Oriented attacks. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1433–1450, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL https://www.usenix.org/conference/usenixsecurity23/presentation/ahmed-salman.

[4] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[5] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing Memory Error Exploits with WIT. In *2008 IEEE Symposium on Security and Privacy (S&P 2008)*, pages 263–277, 2008. doi: 10.1109/SP.2008.30.

[6] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, volume 10, page 96, 2009.

[7] Anit Anubhav and Manish Sardiwal. The journey and evolution of god mode in 2016: Cve-2016-0189, 2017. URL https://www.virusbulletin.com/virusbulletin/2017/01/journey-and-evolution-god-mode-2016-cve-2016-0189/.

[8] Arm Ltd. *Arm Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile*, 2022.

[9] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 290–301, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 089791662X. doi: 10.1145/178243.178446. URL https://doi.org/10.1145/178243.178446.

[10] Roberto Avanzi. The QARMA block cipher family. *IACR Transactions on Symmetric Cryptology*, pages 4–44, 2017.

[11] Yanis Belkheyar, Joan Daemen, Christoph Dobraunig, Santosh Ghosh, and Shahram Rasoolzadeh. Bipbip: A low-latency tweakable block cipher with small dimensions. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(1): 326–368, Nov. 2022. doi: 10.46586/tches.v2023.i1.326-368. URL https://tches.iacr.org/index.php/TCHES/article/view/9955.

[12] Lukas Bernhard, Michael Rodler, Thorsten Holz, and Lucas Davit. xTag: Mitigating Use-After-Free Vulnerabilities via Software-Based Pointer Tagging on Intel x86-64. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 502–519, 2022. doi: 10.1109/EuroSP53844.2022.00038.

[13] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305648. doi: 10.1145/1966913.1966919.

[14] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. CUP: Comprehensive User-Space Protection for C/C++. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 381–392, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355766. doi: 10.1145/3196494.3196540.

[15] Scott A. Carr and Mathias Payer. DataShield: Configurable Data Confidentiality and Integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 193–204, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349444. doi: 10.1145/3052973.3052983. URL https://doi.org/10.1145/3052973.3052983.

[16] Shuo Chen, Jun Xu, and Emre C. Sezer. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, 7 2005. USENIX Association. URL https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats.

[17] Xingman Chen, Yinghao Shi, Zheyu Jiang, Yuan Li, Ruoyu Wang, Haixin Duan, Haoyu Wang, and Chao Zhang. MTSan: A feasible and practical memory sanitizer for fuzzing cots binaries. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 841–858, 2023.

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

[19] GCC Developers. Intel MPX support in the GCC compiler, June 2018. URL https://gcc.gnu.org/wiki/Intel%20MPX%20support%20in%20the%20GCC%20compiler.

[20] GCC Developers. Program instrumentation options. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html#index-fsanitize_003daddress, 2024.

[21] LLVM Developers. Addresssanitizer. https://clang.llvm.org/docs/AddressSanitizer.html, 2024.

[22] Rust Developers. Rust programming language, 2024. URL https://www.rust-lang.org/.

[23] SoftBoundCETS developers. softboundcets-34, 2014. URL https://github.com/santoshn/softboundcets-34.

[24] Tor Developers. Tor browser 5.5a4-hardened is released, November 2015. URL https://blog.torproject.org/tor-browser-55a4-hardened-released/.

[25] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 103–114, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939586. doi: 10.1145/1346281.1346295. URL https://doi.org/10.1145/1346281.1346295.

[26] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933751. doi:

10.1145/1134285.1134309. URL https://doi.org/10.1145/1134285.1134309.

[27] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 144–157, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933204. doi: 10.1145/1133981.1133999. URL https://doi.org/10.1145/1133981.1133999.

[28] Baozeng Ding, Yeping He, Yanjun Wu, Alex Miller, and John Criswell. Baggy bounds with accurate checking. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*, pages 195–200, 2012. doi: 10.1109/ISSREW.2012.24.

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

[30] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 132–142, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342414. doi: 10.1145/2892208.2892212.

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

[32] Gregory J. Duck, Yuntong Zhang, and Roland H. C. Yap. Hardening binaries against more memory errors. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pages 117–131, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391627. doi: 10.1145/3492321.3519580. URL https://doi.org/10.1145/3492321.3519580.

[33] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332132. doi: 10.1145/2663716.2663755. URL https://doi.org/10.1145/2663716.2663755.

[34] Ronald Gil, Hamed Okhravi, and Howard Shrobe. There's a hole in the bottom of the c: On the effectiveness of allocation protection. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 102–109, 2018. doi: 10.1109/SecDev.2018.00021.

[35] Amogha Udupa Shankaranarayana Gopal, Raveendra Soori, Michael Ferdman, and Dongyoon Lee. 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)*, 2023.

[36] Floris Gorter, Enrico Barberis, Raphael Isemann, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. FloatZone: Accelerating memory error detection using the floating point unit. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 805–822, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL https://www.usenix.org/conference/usenixsecurity23/presentation/gorter.

[37] Enes Göktas, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 227–242, 2018. doi: 10.1109/EuroSP.2018.00024.

[38] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. TypeSan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 517–528, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978405. URL https://doi.org/10.1145/2976749.2978405.

[39] Istvan Haller, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. Metalloc: Efficient and comprehensive metadata management for software security hardening. In *Proceedings of the 9th European Workshop on System Security*, EuroSec '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342957. doi: 10.1145/2905760.2905766. URL https://doi.org/10.1145/2905760.2905766.

[40] Mohamed Tarek Bnziad Mohamed Hassan. *Hardware-Software Co-design for Practical Memory Safety*. PhD thesis, Columbia University, 2022.

[41] Konrad Hohentanner, Florian Kasten, and Lukas Auer. 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*, pages 27–33, 2023.

[42] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. Cryptsan: Leveraging arm pointer authentication for memory safety in c/c++. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, pages 1530–1539, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450395175. doi: 10.1145/3555776.3577635. URL https://doi.org/10.1145/3555776.3577635.

[43] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 177–192, USA, 2015. USENIX Association. ISBN 9781931971232.

[44] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. The Taming of the Stack: Isolating Stack Data from Memory Errors. In *Proceedings of the 2020 ISOC Network and Distributed Systems Security Symposium (NDSS)*, February 2022.

[45] Kaiming Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. Top of the heap: Efficient memory error protection for many heap objects. *arXiv preprint arXiv:2310.06397*, 2023.

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

[47] Intel Inc. *Intel 64 and IA-32 Architectures. Software Developer's Manual*, 2021.

[48] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. FuZZan: Efficient sanitizer metadata design for fuzzing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 249–263. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/jeon.

[49] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, USA, 2002. USENIX Association. ISBN 1880446006.

[50] X. Jin, X. Xiao, S. Jia, W. Gao, H. Zhang, D. Gu, S. Ma, Z. Qian, and J. Li. Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK. In *2022 IEEE Symposium on Security and Privacy (S&P)*, pages 473–488, Los Alamitos, CA, USA, May 2022. IEEE Computer Society. doi: 10.1109/SP46214.2022.00028. URL https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00028.

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

[52] Tina Jung, Fabian Ritter, and Sebastian Hack. Pico: A presburger in-bounds check optimization for compiler-based memory safety instrumentations. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(4):1–27, 2021.

[53] Vineet Kahlon. Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. *SIGPLAN Not.*, 43(6):249–259, June 2008. ISSN 0362-1340. doi: 10.1145/1379022.1375613.

[54] Piyus Kedia, Rahul Purandare, Udit Agarwal, and Rishabh. Cguard: Scalable and precise object bounds protection for c. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1307–1318, 2023.

[55] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *European Conference on Computer Systems (EuroSys)*, 2017.

[56] Michael Kounavis, Sergej Deutsch, Santosh Ghosh, and David Durham. K-cipher: A low latency, bit length parameterizable cipher. In *2020 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7, 2020. doi: 10.1109/ISCC50000.2020.9219582.

[57] Taddeus Kroes, Koen Koning, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Fast and generic metadata management with mid-fat pointers. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349352. doi: 10.1145/3065913.3065920. URL https://doi.org/10.1145/3065913.3065920.

[58] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In

Proceedings of the Thirteenth EuroSys Conference, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355841. doi: 10.1145/3190508.3190553.

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

[60] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, USA, 2014. USENIX Association. ISBN 9781931971164.

[61] Michael Larabel. The first rust-written network phy driver set to land in linux 6.8, 12 2023. URL https://www.phoronix.com/news/Linux-6.8-Rust-PHY-Driver.

[62] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M. Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. Cryptographic capability computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, pages 253–267, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572. doi: 10.1145/3466752.3480076. URL https://doi.org/10.1145/3466752.3480076.

[63] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. PACMem: Enforcing spatial and temporal memory safety via arm pointer authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, pages 1901–1915, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394505. doi: 10.1145/3548606.3560598. URL https://doi.org/10.1145/3548606.3560598.

[64] Hans Liljestrand, Carlos Chinea, Rémi Denis-Courmont, Jan-Erik Ekberg, and N. Asokan. Color my world: Deterministic tagging for memory safety, 2022. URL https://arxiv.org/abs/2204.03781.

[65] Ankur Limaye and Tosiron Adegbija. A workload characterization of the spec cpu2017 benchmark suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 149–158, 2018. doi: 10.1109/ISPASS.2018.00028.

[66] Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing. Camp: Compiler and allocator-based heap memory protection. In *USENIX Security Symposium*, 2024. URL https://zplin.me/papers/CAMP.pdf. To appear in USENIX Security 2024.

[67] Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. Giantsan: Efficient memory sanitization with segment folding. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2024)*, 2024.

[68] Zhengyang Liu and John Criswell. Flexible and efficient memory object metadata. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2017, pages 36–46, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350440. doi: 10.1145/3092255.3092268. URL https://doi.org/10.1145/3092255.3092268.

[69] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhotak, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58:44–46, 2015. URL http://cacm.acm.org/magazines/2015/2/182650-in-defense-of-soundiness/abstract.

[70] The MITRE Corporation (MITRE). CWE-125: Out-of-bounds read. https://cwe.mitre.org/data/definitions/125.html, 2024.

[71] The MITRE Corporation (MITRE). CWE-416: Use after free. https://cwe.mitre.org/data/definitions/416.html, 2024.

[72] The MITRE Corporation (MITRE). CWE-787: Out-of-bounds write. https://cwe.mitre.org/data/definitions/787.html, 2024.

[73] Micah Morton, Jan Werner, Panagiotis Kintis, Kevin Snow, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Security risks in asynchronous web servers: When performance optimizations amplify the impact of data-oriented attacks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 167–182, 2018. doi: 10.1109/EuroSP.2018.00020.

[74] Yeoul Na. -fbounds-safety. enforcing bounds safety for production c code. *EuroLLVM Developers' Meeting*, May 2023. URL https://llvm.org/devmtg/2023-05/slides/TechnicalTalks-May11/01-Na-fbounds-safety.pdf.

[75] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542504.

[76] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 189–200, USA, 2012. IEEE Computer Society. ISBN 9781450316422.

[77] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Everything You Want to Know About Pointer-Based Checking. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 190–208, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-80-4. doi: 10.4230/LIPIcs.SNAPL.2015.190. URL http://drops.dagstuhl.de/opus/volltexte/2015/5026.

[78] Myoung Jin Nam, Periklis Akritidis, and David J Greaves. Framer: A tagged-pointer capability system with memory safety applications. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, pages 612–626, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450376280. doi: 10.1145/3359789.3359799.

[79] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, may 2005. ISSN 0164-0925. doi: 10.1145/1065887.1065892. URL https://doi.org/10.1145/1065887.1065892.

[80] White House Office of the National Cyber Director (ONCD). Back to the building blocks: A path toward secure and measurable software. Technical report, ONCD, February 2024. URL https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf.

[81] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

[82] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49), November 1996. URL http://www.phrack.com/issues.html?issue=49&id=14.

[83] PaX Team. Address space layout randomization (aslr). https://pax.grsecurity.net/docs/aslr.txt, 2001.

[84] Tribuvan Kumar Prakash and Lu Peng. Performance characterization of spec cpu2006 benchmarks on intel core 2 duo processor. *ISAST Trans. Comput. Softw. Eng*, 2(1):36–41, 2008.

[85] Marco Prandini and Marco Ramilli. Return-oriented programming. *IEEE Security and Privacy*, 10(6):84–87, November 2012. ISSN 1540-7993. doi: 10.1109/MSP.2012.152.

[86] Alex Rebert and Christoph Kern. Secure by design: Google's perspective on memory safety. Technical report, Google Security Engineering, 2024.

[87] Inc. Red Hat. CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160., December 3 2014. URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[88] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, et al. Address oblivious code reuse: On the effectiveness of leakage resilient diversity. In *NDSS*, 2017.

[89] Mark Russinovich, September 2022. URL https://twitter.com/markrussinovich/status/1571995117233504257.

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

[91] Gururaj Saileshwar, Rick Boivie, Tong Chen, Benjamin Segal, and Alper Buyuktosunoglu. Heapcheck: Low-cost hardware support for memory safety. *ACM Trans. Archit. Code Optim.*, 19(1), January 2022. ISSN 1544-3566. doi: 10.1145/3495152. URL https://doi.org/10.1145/3495152.

[92] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, S&P '15,

pages 745–762, USA, 2015. IEEE Computer Society. ISBN 9781467369497. doi: 10.1109/SP.2015.51.

[93] Jiwon Seo, Junseung You, Donghyun Kwon, Yeongpil Cho, and Yunheung Paek. ZOMETAG: Zone-based Memory Tagging for Fast, Deterministic Detection of Spatial Memory Violations on ARM. *IEEE Transactions on Information Forensics and Security*, 2023.

[94] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

[95] Fermin J Serna. The info leak era on software exploitation. *Black Hat USA*, 7, 2012.

[96] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS 2016*, 2016.

[97] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (S&P)*, pages 1275–1295, 2019. doi: 10.1109/SP.2019.00010.

[98] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections. *Black Hat USA*, 2008.

[99] Jeffrey Vander Stoep. Memory safe languages in android 13, 12 2022. URL https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html.

[100] Yulei Sui, Ding Ye, Yu Su, and Jingling Xue. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Transactions on Reliability*, 65(4):1682–1699, 2016. doi: 10.1109/TR.2016.2570538.

[101] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. Oat: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (S&P)*, pages 1433–1449, 2020. doi: 10.1109/SP40000.2020.00042.

[102] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013. doi: 10.1109/SP.2013.13.

[103] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W Keckler, and Mark Stephenson. cucatch: A debugging tool for efficiently catching memory safety violations in cuda applications. *Proceedings of the ACM on Programming Languages*, 7(PLDI):124–147, 2023.

[104] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection*, pages 121–141, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-23644-0.

[105] Martin Unterguggenberger, David Schrammel, Lukas Lamster, Pascal Nasahl, and Stefan Mangard. Cryptographically enforced memory safety. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, pages 889–903, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700507. doi: 10.1145/3576915.3623138. URL https://doi.org/10.1145/3576915.3623138.

[106] Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. Type-after-type: Practical and complete type-safe memory reuse. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 17–27, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365697. doi: 10.1145/3274694.3274705.

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

[108] Haojie Wang, Jidong Zhai, Xiongchao Tang, Bowen Yu, Xiaosong Ma, and Wenguang Chen. Spindle: Informed memory access monitoring. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 561–573, USA, 2018. USENIX Association. ISBN 9781931971447.

[109] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014. doi: 10.1109/ISCA.2014.6853201.

[110] Shengjie Xu, Wei Huang, and David Lie. In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *Proceedings of the 26th ACM International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '21, pages 224–240, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446761. URL https://doi.org/10.1145/3445814.3446761.

[111] Shengjie Xu, Eric Liu, Wei Huang, and David Lie. Mifp: Selective fat-pointer bounds compression for accurate bounds checking. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 609–622, 2023.

[112] Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. 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*, pages 413–426. Springer, 2017.

[113] Hengkai Ye, Song Liu, Zhechang Zhang, and Hong Hu. VIPER: Spotting Syscall-Guard variables for Data-Only attacks. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1397–1414, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL https://www.usenix.org/conference/usenixsecurity23/presentation/ye.

[114] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009. doi: 10.1109/SP.2009.25.

[115] Suan Hsi Yong and Susan Horwitz. Protecting c programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 307–316, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137435. doi: 10.1145/940071.940113.

[116] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. Paricheck: an efficient pointer arithmetic checker for c programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 145–156, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589367. doi: 10.1145/1755688.1755707. URL https://doi.org/10.1145/1755688.1755707.

[117] Yizhuo Zhai, Zhiyun Qian, Chengyu Song, Manu Sridharan, Trent Jaeger, Paul Yu, and Srikanth V Krishnamurthy. Don't waste my efforts: Pruning redundant sanitizer checks of developer-implemented type checks. 2024. To appear in USENIX Security 2024.

[118] Chao Zhang, Lei Duan, Tao Wei, and Wei Zou. Secgot: Secure global offset tables in elf executables. In *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013)*, pages 995–998. Atlantis Press, 2013/03. ISBN 978-90-78677-61-1. doi: 10.2991/iccsee.2013.250. URL https://doi.org/10.2991/iccsee.2013.250.

[119] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. SANRAZOR: Reducing redundant sanitizer checks in c/c++ programs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 479–494, 2021.

[120] Tong Zhang, Dongyoon Lee, and Changhee Jung. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 631–644, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304017.

[121] Yiyu Zhang, Tianyi Liu, Zewen Sun, Zhe Chen, Xuandong Li, and Zhiqiang Zuo. Catamaran: Low-overhead memory safety enforcement via parallel acceleration. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 816–828, 2023.

[122] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating address sanitizer. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-yuchen.