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
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 runtime 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 pointers1) [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

---

1For 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 trade-off [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.

```c
// exploitable pointer load
ptr = array[i];
bounds = lookup(&array[i])->bounds;
// ...
assert_in_bounds(ptr, bounds);
*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:

```c
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:

```c
int adminLvl = dangling_ptr->lvl;
if (adminLvl > 2)
    system("/bin/bash");
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:

```c
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-overhead 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 C3 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® 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
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 heapifying 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.

![Figure 1: Occurrences of the enabling code patterns in the 1,000 most popular C/C++ GitHub repositories. The top axis counts the lines of code of each repository. The bottom axis counts the unsafe data pointer loads, and unsafe function ptr loads respectively.](image-url)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Type</td>
<td>None</td>
<td>None</td>
<td>Commodity</td>
<td>Commodity</td>
</tr>
<tr>
<td>Per-Pointer Metadata</td>
<td>Pointer-based</td>
<td>Object-based</td>
<td>Pointer-based</td>
<td>Pointer-based</td>
</tr>
<tr>
<td>Per-Object Metadata</td>
<td>Disjoint</td>
<td>In-pointer</td>
<td>Disjoint</td>
<td>None</td>
</tr>
<tr>
<td>Pointer Reuse</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Pointer Crafting</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Illegitimate Targets</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

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 stunt 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:

```c
ptrToSens = nonSensArray[i];
*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


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


[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


