Diagnosing and Neutralizing Address-Sensitive Behavior in Multi-Variant Execution Systems

Anton Schelfhout
anton.schelfhout@kuleuven.be
DistriNet, KU Leuven
Ghent, Belgium

Adriaan Jacobs
adriaan.jacobs@kuleuven.be
DistriNet, KU Leuven
Ghent, Belgium

Jonas Vinck
jonas.vinck@kuleuven.be
DistriNet, KU Leuven
Ghent, Belgium

Stijn Volckaert
stijn.volckaert@kuleuven.be
DistriNet, KU Leuven
Ghent, Belgium

Abstract

Multi-Variant eXecution (MVX) systems are a potent building block for comprehensive memory corruption defenses. They run multiple diversified variants of the same program in parallel, feed them the same inputs, and monitor whether they produce the same outputs. If applied correctly, MVX systems ensure that (i) no single exploit payload can simultaneously compromise all variants and (ii) any exploit causes an observable divergence in the variants' behavior.

After repeatedly demonstrating their prowess in detecting control-flow hijacks, MVX systems are naturally keen to extend their protection to *data-only* attacks, which do not solely focus on corrupting code pointers. However, that would require diversifying the variants' data layout as well, which vastly exacerbates known compatibility issues of existing systems. Due to Address-Sensitive Behavior (ASB), benign programs can behave in dissimilar but functionally equivalent ways depending on memory layout and address values. This causes variants to diverge in ways that are indistinguishable from the effects of an attack to the MVX monitor.

In this paper, we explore the practical implications of adopting data diversification in MVX systems. For the first time, we characterize and quantify the issue of ASB across a wide range of real-world software, and find that it is a significant hurdle towards support for data diversification in MVX. To help address this issue, we developed a new *variant diffing* technique that allows us to recognize and, at times, even neutralize different classes of ASB by locating address-related data in the variants without compiler support.

CCS Concepts

• **Security and privacy** → *Operating systems security*; **Software security engineering**.

EuroSec'25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1563-1/2025/03
https://doi.org/10.1145/3722041.3723094

Keywords

Address-Sensitive Behavior, Multi-Variant Execution, Software Non-Determinism, Software Diversity

1 Introduction

Memory safety issues are a long-standing source of exploitation in the system-level software stack that underpins our digital infrastructure [1–7] . Despite numerous attempts to eliminate these vulnerabilities through testing [8–10], fuzzing [11, 12], or migration to memory-safe languages [13–17], they continue to present a flourishing exploitation avenue for attackers to this day [18] .

Among the landscape of defenses proposed over many decades [19–24], Multi-Variant eXecution (MVX) stands out as a particularly strong approach to defeat memory error exploitation at a low run-time cost [25–42]. MVX systems run multiple variants of the same program in parallel, supply them with the same inputs, and deduplicate their outputs [25]. The variants are semantically equivalent but diversified so that exploit payloads cannot simultaneously compromise all variants [43, 44]. Instead, they make the variants' behavior diverge [31]. The MVX system detects such divergences by running the variants in lock-step while comparing their system call sequences and arguments [32].

Uniquely, MVX systems present an escape from the classical trade-off between security and run-time overhead that characterizes typical enforcement-based exploit mitigations [45, 46] , since the overhead of monitoring and synchronizing variants at system calls is relatively low (< 5% on syscall-intensive benchmarks [47]) and indifferent to the set of diversity techniques applied to the variants. Instead, the primary downside of MVX is its increased resource consumption, often requiring additional system memory and CPU cores to function efficiently [32]. Fortunately, these resources tend to be easier to scale than the single-threaded performance of modern processors [48]. This makes MVX an attractive solution for highly security-critical yet performance-conscious applications.

This is especially true in the data-only attack era. Unlike control-flow hijacks, which corrupt code pointers to achieve arbitrary code execution, data-only attacks need not rely on corrupting specific types of data [49–55] to achieve similarly insidious exploits [56–58]. Instead, they can target a wide variety of hard-to-isolate program data [59], such as system call arguments [55, 60], decision-making data [59, 61, 62], or application-specific security-sensitive data [7, 63, 64]. This has led many current mitigation proposals to

revert to enforcing complete [19, 20, 65–75] or partial [49–51, 53, 76–78] memory safety, which requires frequent checks on most memory accesses, causing excessive run-time overhead. MVX systems could instead inhibit the corruption or leakage of attack targets through diversified data layouts in all variants, making the effects of data-only exploits unpredictable and unequal among the different variants [26], and leading to divergences [34, 79].

However, the broader adoption of data layout diversification in MVX systems faces a major obstacle, as many programs behave non-deterministically when their memory layout is not fixed. This phenomenon, referred to as "Address-Sensitive Behavior" (ASB) in prior work [29, 31, 80], is a known issue for reproducibility and checkpoint/restore use cases [81–83], but is fatal for MVX systems [80]. Listing 1 shows an address-dependent sorting operation, a typical example of ASB. As the order of objects differs across variants, different objects will be printed. These divergences, while benign, are indistinguishable from the effects of an attack on the system. As such, ASB inhibits current-generation MVX systems from adopting the data diversification necessary to counter modern data-only attacks at a time when comparably efficient defenses for these attacks are as sought-after as they are rare.

```
int compare(const void* a, const void* b) {
    return (*(intptr_t*)a - *(intptr_t*)b);
}
int main() {
    int arr[] ={object1, object2, ...};
    qsort(arr, sizeof(sometype), compare);
    print_array(arr,n);
}
```

Listing 1: Example of an address-dependent iteration order, with differing syscall behavior as a result.

Studying, characterizing, and exploring a path towards neutralizing Address-Sensitive Behavior (ASB) in the context of MVX is the central focus of this paper. For the first time, we examine the presence and effects of ASB in a broad range of software. Through manual experimentation with diversified data layouts in a state-of-the-art MVX system, we readily find numerous examples in common, moderately complex programs, such as the SPEC CPU suite [84]. We show that previous neutralization efforts do not suffice to handle more complex cases of ASB, and argue that the challenge of complete ASB neutralization is much greater than previously understood. In search of a pragmatic solution, we develop a new *variant diffing* technique that leverages data layout diversification to let the monitor automatically discover the location of pointers in the variants' address spaces. For diverging system calls where contents of buffer arguments differ, this technique allows for determining if the differing bytes are indeed pointers and if they are equivalent among the variants. This automatically alleviates the compatibility issue observed in previous work [47]. For the more common type of ASB-originated divergences where the variants have taken a different control-flow path, we use the variant diffing technique to help diagnose the original location of ASB in the program. While we cannot confidently distinguish benign from malicious behavior in this case, our diagnostic information helps bridge the gap between the opaque divergence and its subtle ASB-related cause, and lets developers know where to manually refactor their code to eliminate the non-deterministic behavior.

In summary, we contribute the following. First, we demonstrate the problem ASB causes for data layout diversification inside MVX systems, analyze common examples of ASB, and identify limitations of previous ASB neutralization attempts. Then, we present a new technique called variant diffing that allows us to automatically understand the memory layout of diversified variants in an MVX system. We implemented this technique in a diagnostic tool that helps us link divergences back to the source code location of the ASB that caused them. Finally, we present an automatic neutralization technique that uses our variant diffing infrastructure to distinguish benign from malicious divergences for certain ASB classes.

2 Background

MVX systems were originally conceived as a mechanism for turning probabilistic software diversity-based defenses into deterministic defenses [26–31, 33, 34, 40, 79]. The core idea is that benign program functionality is oblivious to low-level details such as memory layout [31, 85] or instruction set architecture [42, 79], while successful exploits must be specifically tailored to these aspects of each variant [43]. Due to the input replication in MVX, the input payload cannot be specialized to each variant, and any successful exploitation of a single variant would trigger diverging behavior from the other variants. Since their initial conception, MVX systems have also been adapted to increase software reliability or to safely deploy software updates [25, 35–37]. Almost all security-focused MVX systems verify that the variants perform the same system calls [28–30, 32, 79], with the same arguments, in the same order, to ensure equivalent behavior. This monitoring granularity strikes an attractive balance between the overhead of synchronizing and monitoring variant operations, and preventing successful exploitation from going unnoticed. After all, to have a real impact on the system, exploits eventually have to perform system calls [29].

Prior work on MVX has primarily focused on defenses against code-injection and control-flow hijacking attacks [25, 26, 28, 31, 86]. However, in response to improving code pointer defenses, attackers have increasingly shifted toward advanced exploit techniques that do not rely on corrupting code pointers at all, so-called *data-only* attacks [56–60, 87]. Direct Data Manipulation (DDM) attacks, like DirtyCred [61] and Heartbleed [7], directly access security-critical data without hijacking the control flow first [56, 59, 60, 62]. More recently, Hu et al. showed that Data-Oriented Programming (DOP) attacks can grant expressive code execution capabilities [57], by overwriting function arguments and conditional expressions that determine the control flow of the program [57, 58, 88].

Previous work has recognized the shortcomings of code-level diversification alone [89–91], also in the context of MVX [26, 34, 79], and has taken to data layout diversification in response [90]. As was also observed by previous work [34, 80], this may cause slight variations in the variants' order of operations, which can cause benign divergences and prompt a shutdown. One proposed solution to this problem is to relax monitoring granularity, as implemented by BUDDY [34] and DieHard [26], which only check for equivalence of I/O syscalls. As Lu et al. state, this helps avoid cases where the order or arguments of syscalls is affected by the variants' memory

layout [34]. Indeed, many cases of previously reported ASB primarily affect memory-related syscalls, such as `mmap` or `mprotect`, which are not included for divergence checking under an I/O-only policy [32, 34]. However, the effects of ASB are not fundamentally limited to syscalls that implement memory management operations (as we show in Section 3.1). Worse yet, relaxing the monitoring granularity to I/O syscalls alone also forfeits the MVX system's unique ability to comprehensively stop both existing and new attack vectors, irrespective of a specific attacker interface to the application, such as the network. Many more syscalls, including those that implement memory-management operations, can be abused by attackers [32, 92], which may not always involve the specific I/O interface targeted by *relaxed* MVX systems.

We believe that the effects of ASB and benign divergences impact broader MVX adoption, necessitating a conscious approach that preserves the security advantages of MVX in the data-only era instead of sacrificing them for niche ASB issues.

ASB is not the sole cause of benign divergences in MVX. However, it is the only cause that has not been thoroughly investigated and addressed in prior research. Other causes include asynchronous signal delivery [93], inter-process [47] and inter-thread [94] communication via shared memory, and virtual system call pages [31]. All of these causes represent sources of input the variants can read directly from user space, thus bypassing the monitor. Without the monitor's mediation, variants could receive different inputs from these input sources and start to diverge.

3 Address-Sensitive Behavior

The fact that programs behave non-deterministically with respect to their memory layout is not a new finding and has frustrated development efforts beyond MVX alone. GDB [95] disables Address-Space Layout Randomization (ASLR) by default to help improve the consistency of bug reproduction across runs. Developers of the KLEE symbolic execution engine [96] recently proposed an approach to neutralize the non-deterministic effects of the heap allocator during path exploration [83], which significantly affected bug finding times, by enforcing deterministic heap layouts across program runs. Similarly, Riot Games developers encountered non-determinism when analyzing League of Legends game replays using their Chronobreak infrastructure [81], which was, in part, caused by ASB. MVX systems have long included workarounds for ASB too, since it already caused issues for code layout diversification alone. For instance, GHUMVEE [29] exposes a binary patching interface which allows developers to replace address-sensitive operations with variant-agnostic and address-insensitive operations [29].

None of these prior ASB encounters fully demonstrate the issues it causes for fine-grained data layout diversification in MVX. Unlike bug-finding or game replay use cases, MVX systems cannot completely disable randomization since they depend on it for their security benefits. On the other hand, targeted and manual neutralization, such as that supported by GHUMVEE's patching infrastructure [29], presupposes that developers already know where the address-sensitive operations are and how they can be rewritten into semantically equivalent address-insensitive operations. The former is especially difficult to determine based on opaque and possibly hard-to-reproduce divergences reported by the MVX monitor.

3.1 Common Examples of ASB

To better understand the impact of ASB on MVX, we examined previous mentions of ASB-induced divergences in existing MVX research [28, 29, 34, 40]. We then carried out experiments running data-diversified variants of various desktop utilities [97–103] and the SPEC CPU2006 suite [84] in the ReMon [32] MVX system to get a picture of relevant ASB cases in real-world software. We describe the most relevant causes of ASB below. All listed ASB occurrences can be eliminated by manually refactoring the code. The main difficulty is finding which code constructs cause ASB. Later in this paper, we introduce variant diffing, a novel technique that makes the search for ASB causes easier. Using this variant diff, we can mitigate certain ASB occurrences without manual refactoring.

C1 *Uninitialized memory.* The effective value of uninitialized memory depends on the contents previously allocated at those memory addresses, potentially making it variant-specific. In many cases, it is desirable to detect uninitialized reads [104], as they represent bugs or vulnerabilities in the program [26, 105, 106]. However, uninitialized memory can also inadvertently appear as padding bytes of structs in buffer arguments to system calls [80], where they could be used to leak data [40], leading to unrecoverable divergences between the variants. We found such cases in X11 programs that communicate structured data via socket calls to the X11 server, e.g., in *libmotif*'s [103] `WriteTargetsTable` function, used by various X11 programs such as *xpdf* [102] and *xedit* [101].

Österlund et al. invariably zero out diverging bytes [40], without stopping execution. Alternatively, all memory can be zero-initialized on allocation [104].

C2 *Pointers passed in untyped buffer arguments.* Pointers regularly appear as syscall arguments, in which case typical MVX systems compare the pointed-to contents of the pointers for equivalence [28]. For typed arguments, this is not an issue; if diverging bytes are typed as pointers, the MVX system will ignore the different pointers and compare the pointed-to content itself. However, they can also appear unexpectedly in untyped buffer arguments [29], e.g., when serializing binary data or printing out pointer values [47]. We found that graphical X11 applications, once again, communicate raw pointer values as part of structured data in buffers to the X11 server. The *mplayer* [107] video player also printed out string representations of pointer values during its warning logging.

Current MVX monitors cannot distinguish these buffers from malicious attempts to leak data or pointers, and flag a divergence.

C3 *Pointers as keys in hash map.* Widely used system libraries often use an address-sensitive value (often the address of an object) as keys in hash maps. Since the keys differ across variants, the distribution of objects within the hash map will also differ. This could cause one variant to experience more bucket collisions, resulting in one variant requesting more memory. Many real-world applications contain this pattern. The LLVM compiler framework contains many standard data types that use it [108], as do common systems-level libraries like *GLib* [98], *libhunspell* [100], and *libgtk+2.0* [99].

Due to its prominence, GHUMVEE manually interposed offending hash functions with a custom, MVX-aware version, that returned the leader variant's hash results in all variants [29].

C4 *Address-dependent iteration order.* Many programs contain sorted collections of pointers or address-related data, including the hash map case previously mentioned. The iteration order over these collections is entirely determined by the pointer values, which are likely different in the diversified variants. This can make them execute wildly different code, with different system calls, benignly. Lidbury and Donaldson [82] mention that SQLite [109] and SpiderMonkey [110] iterate through sorted containers of pointers, causing issues for their record/replay mechanism. We also found many standard data types in LLVM that implement this pattern [108].

C5 *Alignment.* Some programs over-align pointers for performance reasons or to satisfy the requirements of particular hardware [111]. For instance, `ptmalloc` aligns new arenas on a 1MB boundary [112]. To achieve this, it maps a 2MB region, finds a suitable 1MB-aligned pointer within it, and then unmaps the unneeded pages at both ends, as illustrated in Figure 1. Because the variants will only guarantee the 2MB region to be page-aligned, and its base address is otherwise randomized, the 1MB-aligned address will likely be found at different offsets in different variants. These offsets are non-equivalent address-related data, which can later cause divergences [80], e.g., in the `ptmalloc` example, where it is used as an argument to the `sys_munmap` syscall.

Figure 1: Alignment in glibc's `ptmalloc` implementation [80].

3.2 Challenges in ASB Neutralization

Control-Flow-Altering ASB. The alignment case (**C5**) already illustrates a trivial pitfall of the approach that neutralizes ASB by supplying variant-agnostic values to address-sensitive operations (e.g., returning the leader variant's hash results as a hash function's return value in all variants) [29]. Even if we can force the variants to take the same control-flow path during the alignment check, e.g., by injecting one variant's alignment result in all variants [29], the offset of the 1MB-aligned pointer would be incorrect for some of the variants, causing bugs down the line. In other programs, an instruction requiring specific alignment may crash on unaligned values [111], or simply cause performance degradation. In contrast, GHUMVEE's approach works flawlessly to neutralize all ASB in the hash map case (**C3**) on numerous real-world programs. There, the *one-way* nature of the hashing function gives ASB neutralization the leeway to "lie" about the hash result with impunity since its result is never validated against the actual pointer value. Similarly, in the container iteration case (**C4**), the iteration order typically does not affect the program's functionality; most programs primarily use this pattern to efficiently obtain a unique set of pointers, which happens to have an address-dependent order.

Evidently, some address-sensitive operations cause the program to have more assumptions about the underlying pointer value than others. We generally refer to these assumptions as the "feedback" of the address-sensitive operation on the pointer value. More feedback imposes more constraints on ASB neutralization and even variant diversification because more properties about the variants' pointers have to match to ensure equivalent behavior. For instance, if the variants' 2MB buffers have to contain a 1MB-aligned address at the same offset, their least significant 20 (!) bits must be equal, limiting entropy to the remaining more significant bits.

This *feedback* remains unaccounted for in GHUMVEE's patching solution, and likely requires semantic insight of application developers to properly neutralize. But to do so, a second ASB issue must be overcome. As described by Volckaert et al. in the past [29], ASB tends to originate as minor control-flow divergences among the variants, which then lead to increasingly divergent behavior due to non-equivalent program state, after which all the variants inevitably arrive at different system call invocations. At this point, the MVX monitor first becomes aware of the divergence, which may have initially occurred in an entirely different part of the program, and it becomes hard for application developers to track down the original ASB-related cause.

Identifying the origin of ASB-induced divergences is, in itself, an unsolved problem. The MVX monitor has no insight into the structure or equivalence of the diversified variants' data, frustrating diagnostic efforts. Worse yet, there may exist many more address-sensitive operations in the program that do not lead to *MVX-observable* divergences in the current software version, or on the current input workload, but may well cause problems for newer versions after a software update, requiring developers to repeat the tedious diagnostic process.

Control-Flow-Preserving ASB. In more straightforward ASB cases, such as **C2**, the ASB origin coincides with the system call divergence, which helps to identify the ASB origin and potentially neutralize ASB by injecting equivalent memory or register value into all variants. However, manually refactoring or interposing these locations still incurs a lot of developer effort, which may not scale well to frequently changing code bases with third-party dependencies [113]. In the case of unintentionally-diverging uninitialized padding (**C1**), we could zero out the differing bytes, as implemented by kMVX [40]. When the diverging bytes are intentional pointer values, this risks causing bugs on the recipient side, e.g., when communicating structured data through socket calls in X11 (**C2**).

Ideally, the MVX monitor would automatically be able to recognize untyped address-related data in the syscall arguments, verify it for equivalence between the variants, and allow execution to continue if safe. In Section 4.1, we present a neutralization approach that successfully implements this idea.

4 Diagnosing ASB: Variant Diffing

Data layout diversification complements code layout diversification in existing MVX systems [32] to complete the memory layout diversification picture. Under Disjoint Memory Layouts (DML), the variants contain the same set of memory objects at disjoint locations. This means that (i) every object in the leader variant will have an equivalent object in the followers, and (ii) the storage address range for these equivalent objects is guaranteed not to overlap with the range of the leader object. Throughout this section, we consider two memory objects equivalent if they are semantically the same object, but they are not byte-for-byte identical. Without ASB, and assuming that the MVX system neutralizes other sources of non-determinism [28, 47, 94], the in-memory representation of equivalent objects only differs when they contain pointers.

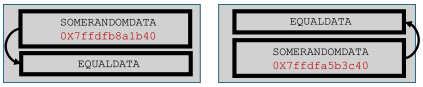

Figure 2: Equivalent objects containing unequal but equivalent pointers in two variants.

With ASB, however, there might be other differences between equivalent objects. We find that a *variant diff* that iterates over all equivalent objects and that examines their differences can prove useful in detecting ASB. Differing bytes that do not represent pointer values represent the effects of address-dependent operations that, if caught early enough, can help us understand the source of ASB-induced divergences (see Section 4.2). Similarly, differing bytes that *do* represent equivalent pointer values can help us distinguish between legitimate pointers that refer to equivalent objects, versus the effects of attacks that maliciously leak pointer values to defeat the MVX system's diversification [34].

To evaluate the potential benefits of variant diffing, we implemented the technique inside the ReMon MVX system [32], adding 1998 LoC. ReMon offers the support to run complex applications [47, 94] and already neutralizes most sources of non-determinism [28, 29]. We implemented Disjoint Memory Layouts (DML) by extending ReMon's Disjoint Code Layouts (DCL) infrastructure [31]. With our extension, ReMon returns non-overlapping address ranges for all `sys_mmap` and `sys_brk` calls executed by the variants. Unless otherwise specified, we did not apply additional fine-grained object-level layout diversification, such as random inter-object padding or shuffling [26, 34]. DML suffices to generate unequal addresses for equivalent objects, which allows the variant diff to work.

Tracking equivalent objects. To compare equivalent objects, we must keep track of their location across the different variants. In our implementation, this is trivial for global and stack objects, since ReMon only diversifies the base addresses of their respective memory regions, but does not shuffle or pad individual objects. We treat these static regions as if they are a single, large object. Under more fine-grained diversification, the MVX monitor should be informed of the diversification seed [26] or the locations of equivalent objects in these regions, which can reasonably be expected when diversification is applied in the compiler or program loader [44].

For the heap, we implemented much more fine-grained tracking of individual object boundaries by forcing the variants to use a custom memory allocator we implemented based on *mimalloc* [114], adding 300 LoC. This custom memory allocator executes a new hypercall, `register_obj`. The hypercall synchronizes the variants, verifies that they are in equivalent states, and returns a unique Equivalent Object ID for the new allocation. Our underlying presupposition is that non-diverged variants allocate equivalent objects with the same size, in the same order. Within each variant, our custom allocator keeps track of the object ID and the boundaries of each object. The allocator also makes the ID-to-boundary mapping available to the monitor so that it can easily locate and compare equivalent objects while performing the variant diff.

To catch ASB-induced divergences as early as possible, we perform an equivalence check on the synchronized `register_obj` call, where we compare not just the hypercall arguments but also the invocation context based on a stack trace we generate for each variant. This helps us avoid situations where ASB leads to undetected but non-equivalent allocation behavior in the variants, which would cause the variants to track non-equivalent objects under the same ID, and cause noise in the variant diff. Finally, to avoid differing bytes due to uninitialized data (cfr. Section 3.1), we ensure that all memory returned by the heap allocator is properly zeroed out [104].

Discovering pointers. As shown in Figure 2, pointers appear as unequal parts of otherwise byte-for-byte identical objects. However, they do not necessarily appear at aligned locations [115], and not all 64 bits are necessarily unequal. Hence, for every diverging byte, we check with any eight-byte sliding window containing this diverging byte if the window translates to an equivalent address in all variants.

4.1 Detecting Untyped Pointers in Arguments

The variant diff enables us to reliably discover address-related data in the program, which directly helps to solve the primary issue in distinguishing benign from malicious system calls in case **C2**, where the variants' control flow has not diverged, but the contents of their buffer arguments to a system call like `write` still differ.

We have implemented a prototype solution that performs a *partial* variant diff, only on the offending buffer arguments of syscalls that would otherwise cause a shutdown. We scan differing buffers for pointers, again using the eight-byte window on differing bytes. If we detect equivalent pointers, i.e., pointers to equivalent objects, at the same offset in all buffers, we can be sure that the programmer intended to place specific pointer data in the buffer, and we can allow the system call to continue. To exploit this leniency, attackers would have to discover and place equivalent pointer values in the same location in both variants, which would mean they defeated the MVX' diversification and replication already.

To handle the common debugging and error logging practice of printing out pointer values, we pragmatically attempt to interpret differing bytes as hexademical string representations of pointers too, which we then check for equivalence.

If none of these attempts work to explain the differing bytes as benign ASB, we still assume the divergence is malicious and terminate the program. From our experiments, we expect that the current prototype suffices to handle most similar cases of pointer data in untyped buffer arguments to syscalls. However, most importantly, the partial variant diff gives future work the tools to handle new cases, should they come up.

4.2 Finding ASB Origins at Divergences

The most common case of MVX-defeating ASB manifests when the variants take different control-flow paths after an address-dependent conditional evaluation, e.g., whether to grow hash map storage [80], or the order in which they iterate over a sorted container of pointers [82]. In this case, the variants may perform non-equivalent system calls, which are observed as a divergence by the MVX monitor, which then terminates execution. To better understand the presence of address-dependent values in the variants' address spaces, we perform a complete diff of the variants' address spaces before they terminate, recursively comparing equivalent objects and collecting the observed divergences, i.e., unequal object contents that do not represent pointers to equivalent objects.

We ran our tool on an Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz with 64 GiB DDR4 RAM. The machine runs Ubuntu 20.04 LTS with Linux kernel version 5.15.0. We tested 13 C programs in the SPEC CPU2006 [84] suite and found that two of the benchmarks diverged due to ASB: `403.gcc` and `456.hmmer` showed divergent allocation behavior among the variants. We manually confirmed in the source code that both originated from address-sensitive operations, `403.gcc` due to a case similar to pointer hashing (**C3**), and `456.hmmer` due to pointer alignment, which later resulted in both variants requesting different amounts of extra memory. However, `456.hmmer` only caused an observable divergence on our own `register_obj` hypercall, meaning that, even though the allocation behavior technically diverges, it usually goes unnoticed. Catching it early gives us a better insight into where ASB first alters control flow. This slight divergence is crucially important to detect for the functionality of the variant diff, as we require fully equivalent allocation behavior to correctly link allocations in different variants together. This highlights a dual purpose in our checking off allocation behavior through `register_obj`: keeping the variant diff correct and catching divergences due to ASB early.

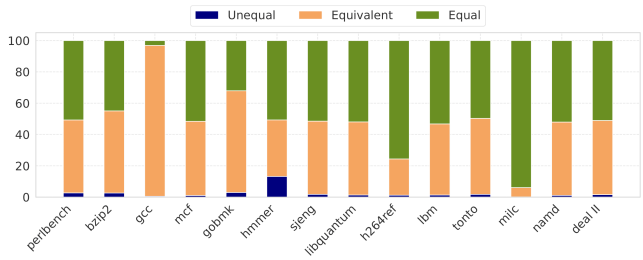

Figure 3: Result of variant diff on SPEC CPU2006 suite.

However, our goal reaches beyond detecting system call divergences due to ASB, which MVX systems already do inadvertently. Our variant diffing mechanism also allows us to detect ASB that did not cause MVX-observable divergences but may still do so in the future after software updates or different inputs. We conducted an experiment where we ran the diff when the variants terminated. We labeled all of the variant's equivalent allocations as either equal when they were byte-for-byte identical, equivalent when they differed only because they contained equivalent pointers, and unequal when they differed for another reason. Figure 3 shows what percentage of allocations for each benchmark falls into which category.

We see that the vast majority of allocations are either equal or equivalent. `403.gcc` terminates rather quickly due to the divergence, likely explaining why only a small percentage of allocations, 0.53%, are unequal. On the other hand, `456.hmmer` shows a much higher percentage, suggesting that non-crashing ASB runs rampant before finally triggering a divergence. In all other benchmarks, though they did not show divergences, we still observe some unequal allocations, which could lead to detectable divergences as the codebase evolves. With the granularity of the variant diff as it is, we cannot determine where or when these allocations became unequal. Still, we can show where these allocations were first made using the backtrace from when they were made. This gives developers valuable insights into which objects are influenced by ASB.

Our study leads us to believe that (i) many ASB-divergences are caused by a single code construct, and (ii) ASB is very rarely an intended program construct for performance or other reasons. As a result, we believe that it might be possible to remove ASB through (limited) refactoring without substantially changing the application's semantics or performance characteristics.

5 Discussion and Future Work

Enforcing a program's determinism with respect to its memory layout naturally has benefits beyond MVX systems alone, as mentioned in Section 3. In addition, we expect that our variant diffing technique could also be of interest to other application domains. Being able to locate pointers in memory images of arbitrary systems-level programs precisely is an extremely coveted ability in itself, pursued by a plethora of previous work for a wide variety of reasons, such as dangling pointer nullification [66, 116–121], garbage collection [115, 122], online program re-randomization [123–126], and reverse engineering [127, 128], among others [129–132]. In the past, previous work relied on heuristics [115, 122], compiler [66, 129, 130] and even hardware support [131] to distinguish pointers from other program data. We can see how MVX systems could provide such info as a simple byproduct of redundant execution and memory layout diversification. However, a high-quality variant diff would still require all ASB to be neutralized in the variants, to ensure that differing bytes in equivalent objects are faithfully pointer values, and not second-degree effects of earlier address-sensitive operations.

Our current variant diffing implementation prioritizes effectiveness over run-time efficiency, since we primarily aim to *sanitize* programs for ASB issues. The largest source of overhead are the variant synchronization points at every heap allocation. In ASB-free programs, we could relax these restrictions to speed up the equivalent object tracking. However, computing the variant diff will still be costly; variants must be in equivalent stopped states, and both variants' process images must still be parsed. Future work could explore whether the variant's inherent parallelism [33, 133], or asynchronous scanning techniques from the garbage collection world [134], can help generate efficient variant diffs.

6 Conclusion

In this paper, we explored the challenges that Address-Sensitive Behavior (ASB) poses for data layout diversification within MVX systems. We examined various real-world examples of ASB and formulated challenges for solving ASB based on their characteristics. This study uncovered shortcomings of existing solutions as well as new insights. We introduced a novel technique called variant diffing, which enables automatic analysis of the memory layout across diversified variants in MVX. We have developed a diagnostic tool based on variant diffing and integrated it into the advanced MVX framework, ReMon. Additionally, we proposed an automatic neutralization strategy that can distinguish between benign and malicious divergences for control-flow preserving ASB.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their helpful feedback. This research is partially funded by the Research Fund KU Leuven, and by the Cybersecurity Research Program Flanders. We thank D. Poetzsch-Heffter, T. Bourguenolle, P. Larsen, and M. Franz for running ASB experiments.

References

[1] 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.

[2] Victor Van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In *Research in Attacks, Intrusions, and Defenses: 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings 15*, pages 86–106. Springer, 2012.

[3] Eugene H. Spafford. The internet worm program: an analysis. *SIGCOMM Comput. Commun. Rev.*, 19(1):17–57, jan 1989. ISSN 0146-4833. doi: 10.1145/66093.66095. URL https://doi.org/10.1145/66093.66095.

[4] CVE-2020-10713. Available from MITRE, CVE-ID CVE-2020-10713., 2020. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-10713.

[5] CVE-2023-0179. Available from MITRE, CVE-ID CVE-2023-0179., 2023. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-0179.

[6] CVE-2019-13720. Available from MITRE, CVE-ID CVE-2019-13720., 2019. URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13720.

[7] CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160., 2014. URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Kostya Serebryany. OSS-Fuzz - google's continuous fuzzing service for open source software. Vancouver, BC, August 2017. USENIX Association.

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

[14] Jeff Vander Stoep and Stephen Hines. Rust in the android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html, 2021.

[15] Jaemin Hong and Sukyoung Ryu. Concrat: An automatic c-to-rust lock api translator for concurrent programs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 716–728, 2023. doi: 10.1109/ICSE48619.2023.00069.

[16] Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. Ownership guided c to rust translation. In *International Conference on Computer Aided Verification*, pages 459–482. Springer, 2023.

[17] Mehmet Emre, Peter Boyland, Aesha Parekh, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Aliasing limits on translating c to safe rust. 7(OOPSLA1), April 2023. doi: 10.1145/3586046. URL https://doi.org/10.1145/3586046.

[18] Ben Hawkes. News and updates from the project zero team at google. https://googleprojectzero.blogspot.com/p/0day.html, 2019.

[19] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Everything You Want to Know About Pointer-Based Checking. In *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 für Informatik. ISBN 978-3-939897-80-4. doi: 10.4230/LIPIcs.SNAPL.2015.190. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SNAPL.2015.190.

[20] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. CC '16, page 132–142, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342414. doi: 10.1145/2892208.2892212. URL https://doi.org/10.1145/2892208.2892212.

[21] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[22] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. {Code-Pointer} integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, 2014.

[23] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, Seattle, WA, November 2006. USENIX Association. URL https://www.usenix.org/conference/osdi-06/securing-software-enforcing-data-flow-integrity.

[24] Sandeep Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, page 1–22, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 9783540705413. doi: 10.1007/978-3-540-70542-0_1. URL https://doi.org/10.1007/978-3-540-70542-0_1.

[25] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *USENIX Security Symposium*, 2006.

[26] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. *SIGPLAN notices*, 41(6):158–168, 2006. ISSN 0362-1340.

[27] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Diversified process replicæ for defeating memory error exploits. In *IEEE Performance, Computing, and Communications Conference (IPCCC)*, 2007.

[28] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 33–46, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584829. doi: 10.1145/1519065.1519071.

[29] Stijn Volckaert, Bjorn De Sutter, Tim De Baets, and Koen De Bosschere. Ghumvee: efficient, effective, and flexible replication. In *Foundations and Practice of Security: 5th International Symposium, FPS 2012, Montreal, QC, Canada, October 25-26, 2012, Revised Selected Papers 5*, pages 261–277. Springer, 2013.

[30] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2016.

[31] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2016.

[32] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and efficient application monitoring and replication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 167–179, Denver, CO, June 2016. USENIX Association. ISBN 978-1-931971-30-0. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/volckaert.

[33] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. Bunshin: compositing security mechanisms through diversification. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[34] Kangjie Lu, Meng Xu, Chengyu Song, Taesoo Kim, and Wenke Lee. Stopping memory disclosures via diversification and replicated execution. *IEEE Transactions on Dependable and Secure Computing*, 18(1):160–173, 2021. doi: 10.1109/TDSC.2018.2878234.

[35] Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013.

[36] Petr Hosek and Cristian Cadar. Varan the unbelievable: An efficient n-version execution framework. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[37] Matthew Maurer and David Brumley. TACHYON: Tandem execution for efficient live patch testing. In *Proceedings of the USENIX Security Symposium*, 2012.

[38] Dohyeong Kim, Yonghwi Kwon, William N Sumner, Xiangyu Zhang, and Dongyan Xu. Dual execution for on the fly fine grained execution comparison. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[39] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. LDX: Causality inference by lightweight dual execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[40] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. kMVX: Detecting kernel information leaks with multi-variant execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[41] Alexios Voulimeneas, Dokyung Song, Per Larsen, Michael Franz, and Stijn Volckaert. dmvx: Secure and efficient multi-variant execution in a distributed setting. In *Proceedings of the 14th European Workshop on Systems Security*, EuroSec '21, page 41–47, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383370. doi: 10.1145/3447852.3458714. URL https://doi.org/10.1145/3447852.3458714.

[42] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. A framework for software diversification with ISA heterogeneity. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[43] Michael Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 New Security Paradigms Workshop*, NSPW

'10, page 7–16, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450304153. doi: 10.1145/1900546.1900550. URL https://doi.org/10.1145/1900546.1900550.

[44] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*, pages 276–291. IEEE, 2014. ISBN 1479946869.

[45] Adriaan Jacobs and Stijn Volckaert. Not quite write: On the effectiveness of Store-Only bounds checking. In *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*, pages 171–187, Philadelphia, PA, August 2024. USENIX Association. ISBN 978-1-939133-43-4. URL https://www.usenix.org/conference/woot24/presentation/jacobs.

[46] 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.

[47] Jonas Vinck, Bert Abrath, Bart Coppens, Alexios Voulimeneas, Bjorn De Sutter, and Stijn Volckaert. Sharing is caring: Secure and efficient shared memory support for mvees. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 99–116, 2022.

[48] Karl Rupp. 42 years of microprocessor trend data, February 2018. URL https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

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

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

[51] Juhee Kim, Jinbum Park, Yoochan Lee, Chengyu Song, Taesoo Kim, and Byoungyoung Lee. Petal: Ensuring access control integrity against data-only attacks on linux. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 2919–2933, 2024.

[52] Jinmeng Zhou, Jiayi Hu, Ziyue Pan, Jiaxun Zhu, Wenbo Shen, Guoren Li, and Zhiyun Qian. Beyond control: Exploring novel file system objects for data-only attacks on linux systems. *arXiv preprint arXiv:2401.17618*, 2024.

[53] 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, 2023.

[54] Munir Geden and Kasper Rasmussen. Truvin: Lightweight detection of data oriented attacks through trusted value integrity. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 174–181, 2020. doi: 10.1109/TrustCom50675.2020.00035.

[55] 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, 2023.

[56] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of {Data-Oriented} exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, 2015.

[57] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986, 2016. doi: 10.1109/SP.2016.62.

[58] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1868–1882, 2018.

[59] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX security symposium*, volume 5, page 146, 2005.

[60] Brian Johannesmeyer, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical {Data-Only} attack generation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1401–1418, 2024.

[61] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1963–1976, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394505. doi: 10.1145/3548606.3560585. URL https://doi.org/10.1145/3548606.3560585.

[62] 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/.

[63] Xuancheng Jin, Xuangan Xiao, Songlin Jia, Wang Gao, Dawu Gu, Hang Zhang, Siqi Ma, Zhiyun Qian, and Juanru Li. Annotating, tracking, and protecting cryptographic secrets with cryptompk. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 650–665. IEEE, 2022.

[64] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. Dynpta: Combining static and dynamic analysis for practical selective data protection. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1919–1937. IEEE, 2021.

[65] 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*, pages 1–14, 2018.

[66] Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing. CAMP: Compiler and allocator-based heap memory protection. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4015–4032, Philadelphia, PA, August 2024. USENIX Association. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/lin-zhenpeng.

[67] Zheng Yu, Ganxiang Yang, and Xinyu Xing. ShadowBound: Efficient heap memory protection through advanced metadata management and customized compiler optimization. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7177–7193, Philadelphia, PA, August 2024. USENIX Association. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/yu-zheng.

[68] 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)*, pages 535–552, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/gopal.

[69] 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, page 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.

[70] 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*, pages 381–392, 2018.

[71] 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.

[72] 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.

[73] 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.

[74] 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*, pages 612–626, 2019.

[75] Floris Gorter, Taddeus Kroes, Herbert Bos, and Cristiano Giuffrida. Sticky tags: Efficient and deterministic spatial memory error mitigation using persistent memory tags. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 217–217. IEEE Computer Society, 2024.

[76] 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*, pages 193–204, 2017.

[77] Sharjeel Khan, Bodhisatwa Chatterjee, and Santosh Pande. Pythia: Compiler-guided defense against non-control data attacks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 850–866, 2024.

[78] Christopher Jelesnianski, Mohannad Ismail, Yeongjin Jang, Dan Williams, and Changwoo Min. Protect the system call, protect (most of) the world with bastion. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 528–541, 2023.

[79] Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz, and Stijn Volckaert. Distributed heterogeneous n-variant execution. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2020.

[80] Stijn Volckaert. *Advanced Techniques for multi-variant execution*. PhD thesis, Ghent University, 2015.

[81] Rick Hoskinson. Determinism in league of legends: Fixing divergences. https://technology.riotgames.com/news/determinism-league-legends-fixing-divergences, 2018.

[82] Christopher Lidbury and Alastair F. Donaldson. Sparse record and replay with controlled scheduling. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 576–593, New York, NY, USA, 2019. ACM. ISBN 1450367127.

[83] Daniel Schemmel, Julian Büning, Frank Busse, Martin Nowack, and Cristian Cadar. Kdalloc: The klee deterministic allocator: Deterministic memory allocation during symbolic execution and test case replay. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA

2023, page 1491–1494, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702211. doi: 10.1145/3597926.3604921.
[84] Standard Performance Evaluation Corporation. Spec cpu® 2006. https://www.spec.org/cpu2006/, 2018.
[85] Babak Salamat, Andreas Gal, and Michael Franz. Reverse stack execution in a multi-variant execution environment. In *Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, pages 1–7, 2008.
[86] 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. IEEE, 2018.
[87] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N. Asokan, and Danfeng (Daphne) Yao. Exploitation techniques for data-oriented attacks with existing and potential defense approaches. *ACM Trans. Priv. Secur.*, 24(4), September 2021. ISSN 2471-2566. doi: 10.1145/3462699. URL https://doi.org/10.1145/3462699.
[88] Jannik Pewny, Philipp Koppe, and Thorsten Holz. Steroids for doped applications: A compiler for automated data-oriented programming. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 111–126. IEEE, 2019.
[89] 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.
[90] Felix Berlakovich and Stefan Brunthaler. R2c: Aocr-resilient diversity with reactive and reflective camouflage. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 488–504, 2023.
[91] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*, pages 763–780. IEEE, 2015.
[92] André Rösti, Stijn Volckaert, Michael Franz, and Alexios Voulimeneas. I'll be there for you! perpetual availability in the a8 mvx system. In *Proceedings of the 40th Annual Computer Security Applications Conference*, ACSAC '24, New York, NY, USA, 2024. Association for Computing Machinery.
[93] Babak Salamat, Todd Jackson, Gregor Wagner, Christian Wimmer, and Michael Franz. Runtime defense against code injection attacks using replicated execution. *IEEE Transactions on Dependable and Secure Computing*, 8(4):588–601, 2011. doi: 10.1109/TDSC.2011.18.
[94] Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, and Michael Franz. Taming parallelism in a multi-variant execution environment. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 270–285, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349383. doi: 10.1145/3064176.3064178. URL https://doi.org/10.1145/3064176.3064178.
[95] GDB Developers. Gdb: The gnu project debugger, 2025. URL https://sourceware.org/gdb/.
[96] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.
[97] LLVM Developers. The llvm compiler infrastructure. https://llvm.org/, 2025.
[98] GLib Development Team. Glib. https://gitlab.gnome.org/GNOME/glib/, 2025.
[99] The GTK Team. The gtk project. https://www.gtk.org/, 2025.
[100] Hunspell Developers. Hunspell. https://github.com/hunspell/hunspell, 2025.
[101] Chris D. Peterson and Paulo Cesar Pereira de Andrade. xedit - simple text editor for x. https://www.x.org/archive/X11R7.6/doc/man/man1/xedit.1.xhtml, 2025.
[102] Glyph & Cog LLC. xpdf - portable document format (pdf) file viewer. https://www.xpdfreader.com/xpdf-man.html, 2025.
[103] Integrated Computer Solutions Inc. Motif. https://motif.ics.com/motif, 2025.
[104] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. Safeinit: Comprehensive and practical mitigation of uninitialized read vulnerabilities. In *NDSS*, volume 17, pages 1–15, 2017.
[105] CVE-2008-0166. Available from MITRE, CVE-ID CVE-2008-0166., 2008. URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166.
[106] CVE-2016-4569. Available from MITRE, CVE-ID CVE-2016-4569., 2016. URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4569.
[107] MPlayer Developers. Mplayer - the movie player. http://www.mplayerhq.hu/design7/news.html, 2025.
[108] LLVM Developers. Llvm programmer's manual. https://llvm.org/docs/ProgrammersManual.html, 2025.
[109] SQLite Developers. Sqlite. https://www.sqlite.org/, 2025.
[110] SpiderMonkey Developers. Spidermonkey. SpiderMonkey, 2025.
[111] Intel Inc. *Intel 64 and IA-32 Architectures. Software Developer's Manual*, 2021.
[112] fishermen. Hustfisher/ptmalloc, December 2024.
[113] DORA. Accelerate state of devops, 2023. URL https://services.google.com/fh/files/misc/2023_final_report_sodr.pdf.
[114] Daan Leijen, Ben Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. Technical Report MSR-TR-2019-18, Microsoft, June 2019. URL https://www.microsoft.com/en-us/research/publication/mimalloc-free-list-sharding-in-action/.
[115] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988.
[116] Zekun Shen and Brendan Dolan-Gavitt. Heapexpo: Pinpointing promoted pointers to prevent use-after-free vulnerabilities. In *Proceedings of the 36th Annual Computer Security Applications Conference*, ACSAC '20, page 454–465, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450388580. doi: 10.1145/3427228.3427645. URL https://doi.org/10.1145/3427228.3427645.
[117] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 405–419, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349383. doi: 10.1145/3064176.3064211. URL https://doi.org/10.1145/3064176.3064211.
[118] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.
[119] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.
[120] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G Neumann, Simon W Moore, Robert NM Watson, et al. Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 545–557, 2019.
[121] Nathaniel Wesley Filardo, Brett F Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, et al. Cornucopia: Temporal safety for cheri heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 608–625. IEEE, 2020.
[122] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. Precise garbage collection for c. In *Proceedings of the 2009 international symposium on Memory management*, pages 39–48, 2009.
[123] SengMing Yeoh, Xiaoguang Wang, Jae-Won Jang, and Binoy Ravindran. smvx: Multi-variant execution on selected code paths. In *Proceedings of the 25th International Middleware Conference*, pages 62–73, 2024.
[124] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Codarr: Continuous data space randomization against data-only attacks. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 494–505, 2020.
[125] Zihao Chang, Jihan Lin, Haifeng Sun, Runkuang Li, Ying Wang, Bin Hu, Xiaofang Zhao, Dejun Jiang, Ninghui Sun, and Sa Wang. Chaos: Function granularity runtime address layout space randomization for kernel module. In *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 23–30, 2024.
[126] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.
[127] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T King. Digging for data structures. In *OSDI*, volume 8, pages 255–266, 2008.
[128] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.
[129] Brian Suchy, Simone Campanoni, Nikos Hardavellas, and Peter Dinda. Carat: A case for virtual memory through compiler-and runtime-based address translation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–345, 2020.
[130] Brian Suchy, Souradip Ghosh, Drew Kersnar, Siyuan Chai, Zhen Huang, Aaron Nelson, Michael Cuevas, Alex Bernat, Gaurav Chaudhary, Nikos Hardavellas, et al. Carat cake: Replacing paging via compiler/kernel cooperation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 98–114, 2022.
[131] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, 2015. doi: 10.1109/SP.2015.9.
[132] Abhishek Bapat, Jaidev Shastri, Xiaoguang Wang, Abilesh Sundarasamy, and Binoy Ravindran. Dapper: A lightweight and extensible framework for live program state rewriting. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, pages 738–749. IEEE, 2024.
[133] Luís Pina, Anastasios Andronidis, and Cristian Cadar. Freeda: Deploying incompatible stock dynamic analyses in production via multi-version execution. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 1–10, 2018.
[134] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 70–83, New

York, NY, USA, 1994. Association for Computing Machinery. ISBN 0897916360. doi: 10.1145/174675.174673. URL https://doi.org/10.1145/174675.174673.

A ASB in Real-World Software

	C1	C2	C3	C4	C5
x11	✓	✓			
libmotif	✓				
xpdf 3.03	✓				
xedit 7.7+2	✓				
glibc 2.19	✓				
mplayer 1.4		✓			
glib 2.40.2			✓		
libhunspell			✓		
libgtk+2.0			✓		
SQLite 3.24.0				✓	
SpiderMonkey				✓	
ptmalloc					✓
Mimalloc 1.8/2.1					✓
LLVM Memory Sanitizer 3.7.0					✓

Table 1: Real-world software affected by ASB case. If no version is shown, the issue persists over multiple versions.